用于大型程序的工具

C++ Primer第五版

第18章

异常处理

noexcept异常说明

```
#include <iostream>
using namespace std;
// will compile, even though it clearly violates its exception specification
void f() noexcept       // promises not to throw any exception
{
      throw exception();  // violates the exception specification
}

void g() { }
void h() noexcept(noexcept(f())) { f(); }
void i() noexcept(noexcept(g())) { g(); }
int main()
{
      try {
            cout << "f: " << std::boolalpha << noexcept(f()) << endl;
            cout << "g: " << std::boolalpha << noexcept(g()) << endl;
            cout << "h: " << std::boolalpha << noexcept(h()) << endl;
            cout << "i: " << std::boolalpha << noexcept(i()) << endl;
            f();
      }
      catch (exception & e) {
            cout << "caught " << e.what() << endl;
      }
}
```

noexcept有两层含义：
当跟在函数参数列表后面时，它是异常说明符；
当作为noexcept异常说明的bool实参出现时，它是一个运算符；

boolalpha的作用是使bool型变量按照false、true的格式输出。如不使用该标识符，那么结果会按照1、0的格式输出。

一个异常如果没有被捕获，则它将终止当前的程序

```
f: true
g: false
h: true
i: false
```

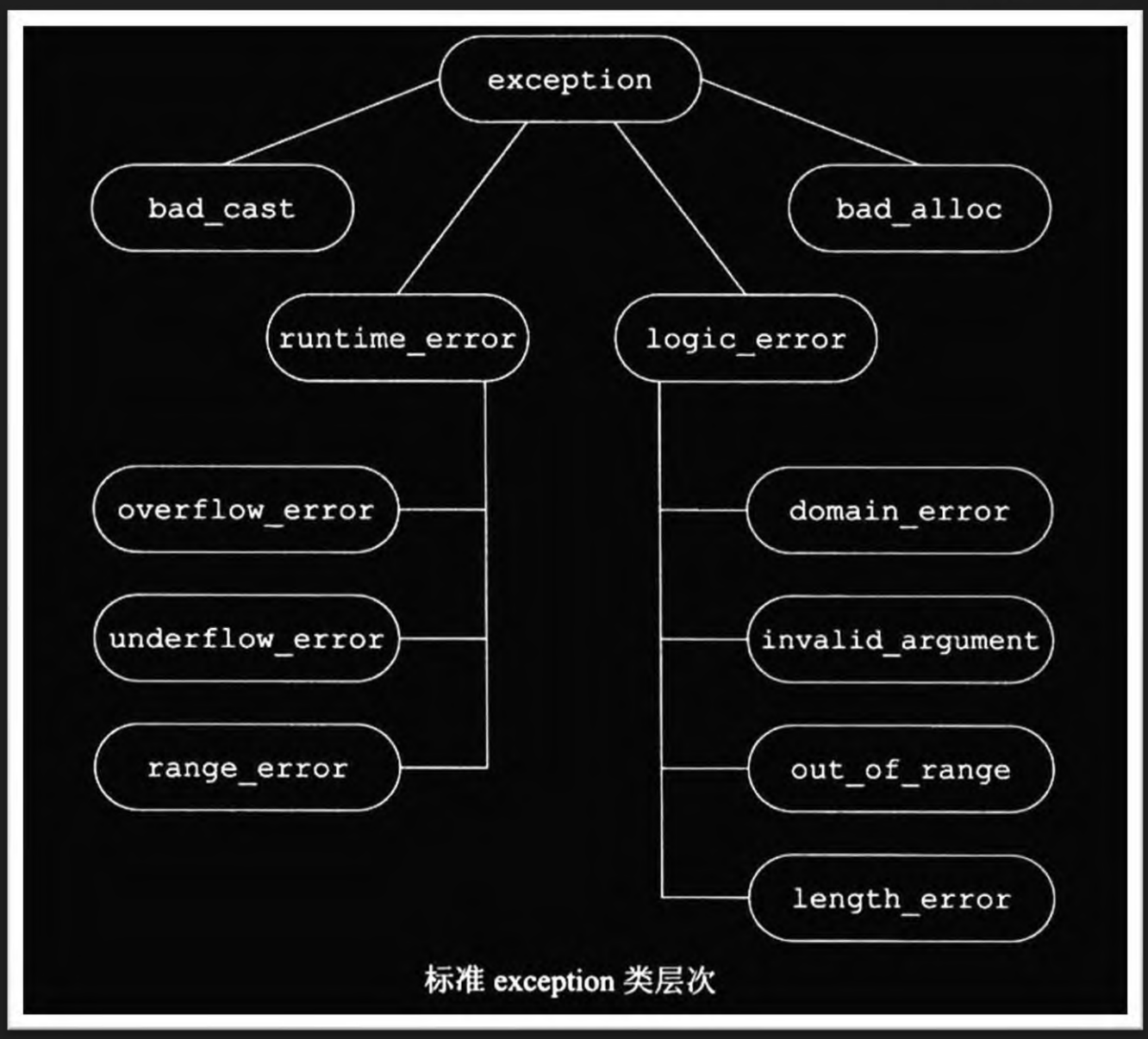
函数指针声明为noexcept，那么指向的函数必须一致

```
void alloc(int) noexcept(false); //alloc可能抛出异常
void recoup(int) noexcept(true); //alloc不会抛出异常
//recoup和pf1都承诺不会抛出异常
void (*pf1)(int) noexcept = recoup;
//正确：recoup不会抛出异常，pf2可能抛出异常，二者之间互不干扰
void (*pf2)(int) = recoup;

pf1 = alloc; //错误：alloc可能抛出异常，但是pf1已经说明了它不会抛出异常
pf2 = alloc; //正确：pf2和alloc都可能抛出异常
```

如果虚函数承诺不会抛出异常，则派生出来的虚函数也必须做出相同的承诺

```
class Base{
public:
    virtual double f1(double) noexcept;     //不会抛出异常
    virtual int f2() noexcept(false);       //可能会抛出异常
    virtual void f3();                      //可能抛出异常
};
class Derived: public Base{
public:
    double f1(double);      //错误:Base::f1承诺不会抛出异常
    int f2() noexcept(false); //正确：与Base::f2的异常说明一致
    void f3() noexcept;     //正确：Derived的f3做了更严格的限定，是允许的
};
```

异常类层次



标准 exception 类层次

```
class out_of_stock: public std::runtime_error {
public:
   explicit out_of_stock(const std::string &s):
                  std::runtime_error(s) { }
};

class isbn_mismatch: public std::logic_error {
public:
   explicit isbn_mismatch(const std::string &s):
                  std::logic_error(s) { }
   isbn_mismatch(const std::string &s,
   const std::string &lhs, const std::string &rhs)
      std::logic_error(s), left(lhs), right(rhs) { }
   const std::string left, right;
};
```

```
int main()
{
   Sales_data item1, item2, sum;
   while (cin >> item1 >> item2) {
   try {
         sum = item1 + item2;
} catch (const isbn_mismatch &e) {
        cerr << e.what() << ": left isbn(" << e.left
          <<") right isbn("<<e.right << ")"<<endl;
      }
   }
    return 0;
}
```

```
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
   if (isbn() != rhs.isbn())
      throw isbn_mismatch("wrong isbns", isbn(), rhs.isbn());
   units_sold += rhs.units_sold;
   revenue += rhs.revenue;
   return *this;
}
```

命名空间

每个命名空间都是一个作用域

```
namespace cplusplus_primer{
        class Sales_data { /*...*/};
        Sales_data operator+(const Sales_data&,
                             const Sales_data&);
        class Query{/*...*/};
        class Query_base{/*...*/};
} //命名空间结束后无须分号，这一点与块类似

cplusplus_primer::Query q = cplusplus_primer::Query("hello");
//假设还有另一个命名空间AddisonWesley也提供了一个Query类
AddisonWesley::Query q = AddisonWesley::Query("hello");
```

命名空间可以是不连续的

```
//----Sales_data.h----
//#include应该出现在打开命名空间的操作之前
#include <string>
namespace cplusplus_primer{
        class Sales_data {/*...*/};
        Sales_data operator+{const Sales_data&,const Sales_data&);
        //...
}
//----Sales_data.cc----
//确保#include出现在打开命名空间的操作之前
#include "Sales_data.h"
namespace cplusplus_primer{
        //Sales_data成员及重载运算符的定义
}
//---user.cc---
#include "Sales_data.h"
{
        using cplusplus_primer::Sales_data;
        Sales_data trans1, trns2;
        //...
        return 0;
}
```

嵌套的命名空间

```
namespace cplusplus_primer {
    //第一个嵌套的命名空间：定义了库的Query部分
    namespace QueryLib{
        class Query {/*...*/};
        Query operator&(const Query&, const Query&);
    //...
    }
    //第二个嵌套的命名空间：定义了库Sales_data部分
    namespace Bookstore{
        class Quote{/*...*/};
        class Disc_quote:public Quote{/*...*/};
    //...
    }
}

cplusplus_primer::QueryLib::Query
```

内联命名空间中的名字可以被外层命名空间直接使用

```
inline namespace FifthEd {
    //该命名空间表示本书第5版的代码
}
namespace FifthEd { //隐式内联
    class Query_base {/*...*/};
    //其他与Query有关的声明
}
namespace FourthEd{
    class Item_base {/*...*/};
    class Query_base {/*...*/};
    //本书第四版用到的其他代码
}

namespace cplusplus_primer {
    #include "FifthEd.h"
    #include "FourthEd.h"
}
```

关键字inline必须出现在命名空间第一次定义的地方

未命名的命名空间

```
int i; //i的全局声明
namespace {
      int i;
}
//二义性：i的定义出现在全局作用域、有出现在未命名的作用域
i = 10;

namespace local {
      namespace {
            int i;
      }
}
//正确：与全局作用域中的i不同
local::i = 42;
```

定义在未命名空间中的名字可以直接使用

命名空间的别名

```
namespace cplusplus_primer {/*...*/};
namespace primer = cplusplus_primer;
//指向一个嵌套的命名空间
namespace Qlib = cplusplus_primer::Querylib;
Qlib::Query q;
```

using声明：一次只引入命名空间的一个成员
using指示：与声明不同，所有名字都可见

```
//命名空间A和函数f定义在全局作用域中
namespace A {
	int i,j;
}
void f()
{
	using namespace A;	//把A中的名字注入到全局作用域中
	cout<<i*j<<endl;	//使用命名空间A中的i和j
	//...
}

namespace blip {
	int i = 16, j = 15; k = 23;
	//其他声明
}
int j = 0; //正确：blip的j隐藏在命名空间中
void manip()
{
	//using指示，blip中的名字呗“添加”到全局作用域中
	using namesapce blip; //如果使用了j，将在::j和blip::j之间产生冲突
	++i;		//将blip::i设定为17
	++j;		//二义性错误
	++::j;		//正确：将全局的j设定为1
	++blip::j;	//正确：16
	int k = 97;	//当前局部的k隐藏了blip::k
	++k;		//将当前局部的k设定为98
}
```

实参相关的查找与类类型形参

```
std::string s;
std::cin >>s;
//等价于operator>>(std::cin,s);
//为什么operator>>可以被直接调用
```

除了在常规的作用域查找外，还会查找实参类所属的命名空间

友元声明与实参相关的查找

```
namespace A{
	class C{
		//两个友元，在友元声明之外没有其他的声明
		//这些函数隐式地成为命名空间A的成员
		friend void f2();		//除非另有声明，否则不会被找到
		friend void f(const C&);	//根据实参相关的查找规则可以被找到
	};
}
int main()
{
	A::C cobj;
	f(cobj);		//正确：通过在A::C中的友元声明找到A::f
	f2();			//错误：A::f2没有被声明
}
```

重载与命名空间

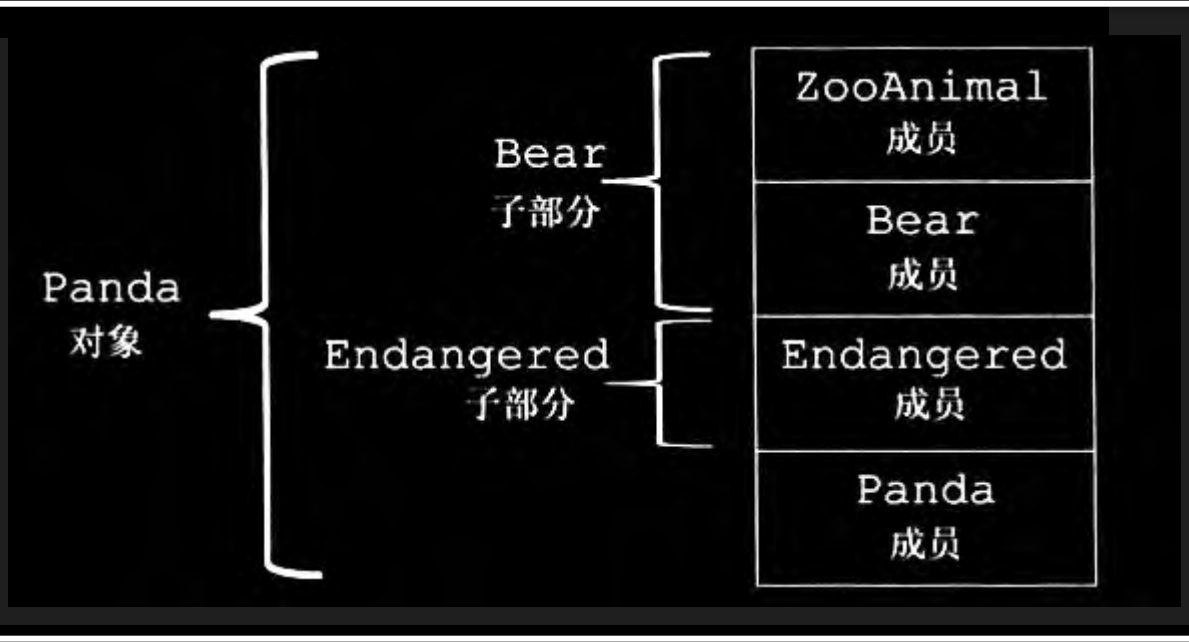

```
#include <string>
using std::string;

#include <iostream>

namespace libs_R_us {
        void print(int)
                { std::cout << "libs_R_us::print(int)" << std::endl; }
        void print(double)
                { std::cout << "libs_R_us::print(double)" << std::endl; }
}

// ordinary declaration
void print(const std::string &)
{
        std::cout << "print(const std::string &)" << std::endl;
}

// this using directive adds names to the candidate set for calls to print:
using namespace libs_R_us;

// the candidates for calls to print at this point in the program are:
//     print(int) from libs_R_us
//     print(double) from libs_R_us
//     print(const std::string &) declared explicitly

int main()
{
        int ival = 42;
        print("Value: "); // calls global print(const string &)
        print(ival);      // calls libs_R_us::print(int)
}
```

```
print(const std::string &)
libs_R_us::print(int)
```

多重继承与虚继承

多重继承的派生类从每个基类中继承状态

```
class Bear:public zooAnimal { };
class Panda:public Bear, public Endangred {/*...*/};
```



派生类的派生列表中可以包含多个基类

```
//显式地初始化所有基类
Panda::Panda(std::string name,bool onExhibit)
        :Bear(name,onExhibit,"Panda"),
        Endangered(Endangered::critical) { }
//隐式地使用Bear的默认构造函数初始化Bear子对象
Panda::Panda()
        :Endangered(Endangered::critical) { }
```

需要同时构造并初始化它的所有基类子对象

只能初始化它的直接基类

panda对象按如下次序进行初始化：

- ZooAnimal --> Bear --> Endangered --> Panda

panda对象析构函数的调用顺序：

- Panda --> Endangered --> Bear --> ZooAnimal

继承的构造函数与多重继承

```
struct Base1 {
    Base1() = default;
    Base1(const std::string&);
    Base1(std::shared_ptr<int>);
};
struct Base2 {
    Base2() = default;
    Base2(const std::string);
    Base2(int);
};
//错误：D1试图从两个基类中都继承D1::D1(const string&)
struct D1:public Base1,public Base2 {
    using Base1::Base1; //从Base1继承构造函数
    using Base2::Base2; //从Base2继承构造函数
};
```

```
//正确：为构造函数定义它自己的版本
struct D2:public Base1,public Base2 {
    using Base1::Base1; //从Base1继承构造函数
    using Base2::Base2; //从Base2继承构造函数
    //D2必须自定义一个接受string的构造函数
    D2(const string &s):Base1(s),Base2(s) { }
    D2() = default;//D2定义了它自己的构造函数
                   //则必须出现
};
```

类型转换与多个基类

```
//接受Panda的基类引用的一系列操作
void print(const Bear&);
void highlight(const Endangered&);
ostream& operator<<(ostream&, const zooAnimal&);
Panda ying_yang("ying_yang");
print(ying_yang);              //把一个panda对象传递给一个Bear的引用
hightlight(ying_yang);         //把一个panda传递给一个Endangered的引用
cout<<ying_yang<<endl;         //把一个panda对象传递给zooAnimal的引用
```

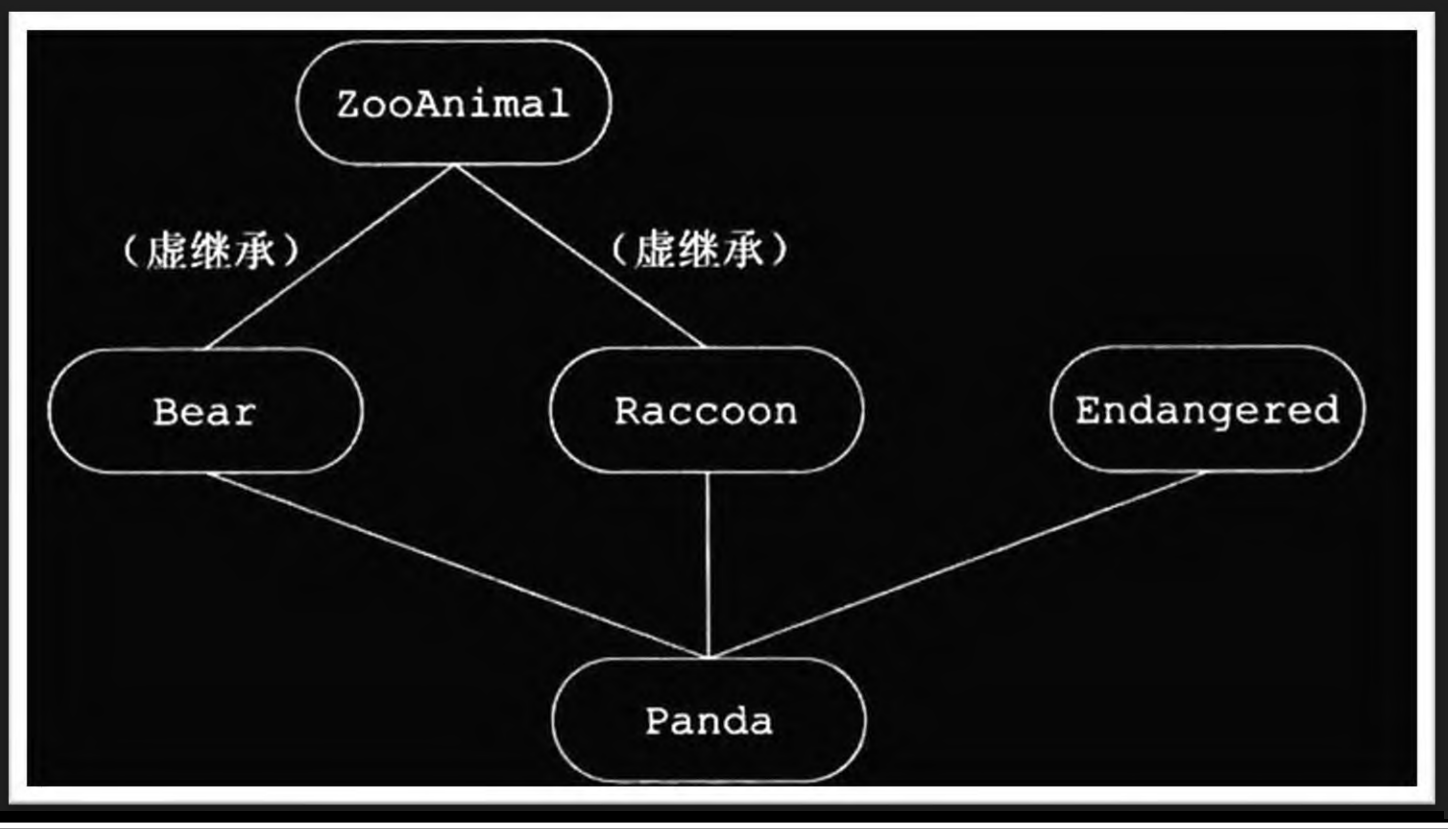
```
void print(const Bear&);
void print(const Endangered&);
Panda ying_yang("ying_yang");
print(ying_yang); //二义性错误
```

对编译器而言，转换到任意一种基类都一样好

多重继承下的类作用域

```
//如果zooAnimal和Endangered都定义了名为max_weight的成员
//并且Panda没有定义该成员，则下面的调用是错误的：
double d = ying_yang.max_weight();

//为了避免二义性错误，最好的办法是为派生类提供一个版本
double Panda::max_weight() const
{
    return std::max(zooAnimal::max_weight(),Endangered::max_weight());
}
```

虚继承：承诺愿意共享基类




```
//关键字public和virtual的顺序随意
class Raccoon:public virtual zooAnimal {/*...*/};
class Bear:virtual public zooAnimal {/*...*/};

class Panda:public Bear,public Raccoon, public Endangered{
}

void dance(const Bear&);
void rummage(const Raccoon&);
osgream& operator<<(ostream&,const zooAnimal&);
Panda ying_yang;
dance(ying_yang);          //正确：把一个Panda对象当成Bear传递
rummage(ying_yang);        //正确：把一个Panda对象当成Raccoon传递
cout<<ying_yang;           //正确：把一个Panda当成zooAnimal传递
```

构造函数与虚继承

```
Bear::Bear(std::string name,bool onExhibit):
        zooAnimal(name,onExhibit,"Bear") { }
Raccoon::Raccoon(std::string name,bool onExhibit):
        zooAnimal(name,onExhibit,"Raccoon") { }
Panda::Panda(std::string name,bool onExhibit) :
        ZooAnimal(name,onExhibit,"Panda"),
        Bear(name,onExhibit),
        Raccoon(name,onExhibit),
        Endangered(Endangered::critical),
        sleeping_flag(false) { }
```

由最底层的类直接初始化共享的基类

→ 虚基类总是先于非虚基类构造，与他们在继承体系中的次序和位置无关