第十周 C++11 新特性和 C++高级主题

第一节 C++新特性（1）

1.统一的初始化方法

（1）int arr[3]{1, 2, 3};
（2）vector<int> iv{1, 2, 3};
（3）map<int, string> mp{{1,"a"},{2,"b"}};
（4）string str{"Hello World"};
（5）int * p = new int[20]{1,2,3};
（6）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}; }

2.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;
    return 0;
}
```

3.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

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

4.decltype 关键字

```
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&
```

5.智能指针 share_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。

例程 1：

```
#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)
    share_ptr<A> sp2(sp1);//sp2 也托管 A(2)
    cout<<"1)"<<sp1->n<<","<<sp2->n<<endl;
    //output: 1)2,2 可以像一个指针一样使用
    shared_ptr<A> sp3;
    A*p=sp1.get();///p 指向 A(2)，把所托管的对象提取出来
    cout<<"2)"<<p->n<<endl;
    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 destruct11or
```

例程 2：

```
#include <memory>
#include <iostream>
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;
}
```

在 ptr 中托管了 p，但是我在 ptr2 中也托管 p 的时候，并不增加 ptr 中对 p 的托管计数。因为 ptr 和 ptr2 认为他们所托管的 p 不是一个 p（虽然实际上是一个 p）。在输出 end 之后主程序结束，ptr 和 ptr2 都要进行执行析构函数，p 被执行了两次析构函数，系统会崩溃。所以输出结果如下：

```
end
~A
~A
```

（之后程序崩溃）

6.空指针 nullptr

nullptr 在有些地方类似于空指针 NULL。例程如下：

```
#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，p3 和 p2 类型不同
    if(p3 == NULL)
        cout<<"equal4"<<endl;
    bool b = nullptr;//b=false
    int i = nullptr;//error，nullptr 无法自动转换成整型
    return 0;
}
```

7.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> str(ary,ary+5);
    for(auto &it:st)
        it.n*=10;
    for(A it: st)
        cout<<it.n<<",";
    return 0;
}
```

类似于 Java 中的使用。

8.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。例如：

```
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。我们前面学习的**都是左值引用**！

第二节 C++新特性（2）

1.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;
}
```

2.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。

3.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
    return 0;
}
```

4.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 和 y 以引用形式使用，其余变量以传值形式使用；

[&,x,y]：x 和 y 以传值的形式使用，其余变量以引用形式使用。

如果以传值方式进行传递对象，就不能修改该对象的值。“->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。

例程：

```
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;
    int a[4] = {4,2,11,33};
    sort(a,a+4,[](int x,int y)->bool{return x%10<y%10;});//按照个位数进行排序，按照原来的
```

方法需要新建一个函数，浪费时间和空间

```
    for_each(a,a+4,[](int x){cout<<x<<" ";});//输出 11 2 33 4
    return 0;
}
```

要注意，[ ](double a,double b) { return a + b; }(1.2,2.5)中的(1.2,2.5)并不是 lambda 表达式一部分，而只是调用 lambda 表达式函数的赋值语句。

例程：实现递归求菲波那切数列第 n 项：

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

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

第三节 强制类型转换

共有四种类型：static_cast、reinterpret_cast、const_cast 和 dynamic_cast。

1.static_cast

static_cast 用来进用行比较**“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。

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。

例程：

```
#include <iostream>
using namespace std;
class A
{
    public:
    int i ,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 = 200，因为 r 是 4 个字节，所以上面的引用只引用了 a 的前面 4 个字节，也就是 a.i
    cout<<a.i<<","a.j<<endl;
    int n = 300;
    A * pa = reinterpret_cast<A*> ( & n); //强行让 pa 指向 n
    pa->i = 400;//n 编程 400
    pa->j =500;//不安全，因为不知道后面这部分内存地址是干什么的，所以可能导致程序崩溃
    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 逐个比特拷贝给 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
    return 0;
}
```

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。

例程如下：

```
#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)
        cout<< "unsafe dynamic_cast2" << endl; //不会执行
    return 0;
}
```

如下所示：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。

第四节 异常处理

程序运行中总难免发生错误，我们希望不只是简单地终止程序运行，还能够反馈异常情况的信息：哪一段代码发生的、什么异常，还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。

1.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。我们需要告知函数调用者发生了异常，让函数调用者处理比较好，但是用函数返回值告知异常不方便。

（1）用 try,catch 进行异常处理

例程：

```
#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 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;
```

```
}
```

表 4.1 trycatch 例程输入输出结果

程序输入	9 0	0 6
输出结果	before dividing catch(...) finished	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！

（2）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。

例程：

```
#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");//抛出异常
    cout << "in Devide" << endl;
    return x / y;
}
int CountTax(int salary){//异常自己处理掉了
    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);//在这个函数自己处理完了，try 里面就感知不到这个错误了
```

```
        f = Devide(3,0);//Devide 本身没有处理异常，所以抛给了这个 try 里面了
        cout << "end of try block" << endl;
    }
    catch(CException e) {
        cout << e.msg << endl; }
    cout << "f=" << f << endl;
    cout << "finished" << endl;
    return 0;
}
```

2.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 类派生而来的。

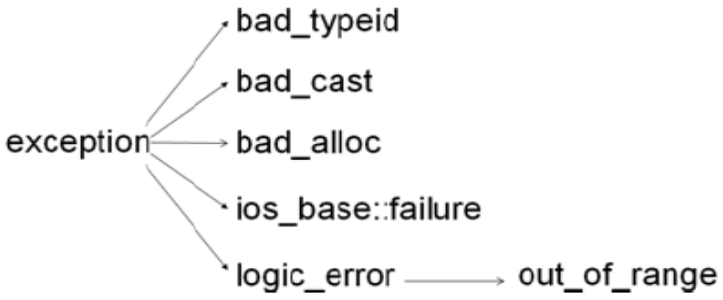

图 4.1 exception 派生出的异常类

（1）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。

（2）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。

（3）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。

3.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。

例程如下：

```
#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;
    //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
    return 0;
}
```