

2024年春

程序设计实习：C++程序设计

第十二讲 C++11特性

贾川民
北京大学



课前提醒

- 课程安排：4月10日 C++11及高阶
- 课程安排：4月17日 QT内容介绍，理教108
- 期中考试安排：4月24日上午，机房待定
- 关于作业：
 - 前六次作业答案已在教学网发布
 - poj题目汇总

课前提醒

- 作业题目练习合集 (仅用于复习, 不计分)

<http://cxsjsx.openjudge.cn/2023hwall/>

往年期中填空题汇总

<http://cxsjsx.openjudge.cn/practise2023cpp/>

- 知识点回看, 围观郭神MOOC:

<https://www.icourse163.org/course/PKU-1002029030>

统一的初始化方法

// 使用 **大括号** 对数组或是容器进行统一的初始化方式

```
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};
```

```
    A a[] = {{1,2},{3,4},{5,6}};    // 使用圆括号不可以
```

```
}
```

统一的初始化方法

/* 初始化列表语法可防止缩窄, 即禁止将数值赋给无法存储它的数值变量
常规初始化运行程序执行可能没有意义的操作

如果使用初始化列表语法, 编译器将禁止进行这样的类型转换, 即数值存储到比它“窄”的变量中 */

```
char c1 = 3.14e10;
```

```
int x1 = 3.14;
```

```
char c2 {3.14e10}; // 此操作会出现编译错误
```

```
int x2 = {459585821}; // 此操作会出现编译错误
```

// C++11 提供了模板类 `initializer_list`, 可将其用作构造函数的参数

```
double SumByIntialList(std::initializer_list<double> il) {  
    double sum = 0.0;  
    for (auto p = il.begin(); p != il.end(); p++) {  
        sum += *p;  
    }  
    return sum;  
}
```

```
double total = SumByIntialList( { 2.5,3.1,4 } );
```

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()); // k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(newT);` // T 可以是 int, char, 类名等各种类型

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(newT); // T 可以是 int, char, 类名等各种类型

- 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

智能指针shared_ptr

```
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;
// sp5.reset(q); 不妥, 会导致程序出错
sp1.reset();    //sp1放弃托管 A(3)
cout << "before end main" << endl;
sp4.reset();    //sp4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
end main
2 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

正确的做法: 不要使用同一个原始指针构造多个shared_ptr
创建多个shared_ptr的正常方法是使用一个已存在的
shared_ptr进行创建, 而不是使用同一个原始指针进行创建

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; // 利用引用做修改
    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（代表一个在内存中占有确定位置的对象）
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：

https://zh.cppreference.com/w/cpp/language/value_category

```

#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);
        }
}

```

```
// 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
```

无序容器 (哈希表)

```
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

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, 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, 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, 匹配失败
}
```

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

利用 **Lambda表达式**可以编写内嵌的匿名函数，用以替换独立函数或者函数对象，并且使代码更可读，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; //(1.2, 2.5)是参数
    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:

```
#include <functional>

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

Thanks !

