程序设计实习

C++ 面向对象程序设计

张勤健
zqj@pku.edu.cn

北京大学信息科学技术学院

2025 年 4 月 30 日

统一的初始化方法

```
int x{5};
int y{};// 默认初始化为 0（不同于 int y; 未初始化）
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;
  int 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}};
  return 0;
}
```

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
char 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 });
```

Most Vexing Parse（MVP）问题

Most Vexing Parse（MVP） 是 C++ 中一个经典的语法歧义问题，源于编译器将对象初始化语句错误解析为函数声明，导致代码行为与预期不符。该问题在 C++11 之前尤其常见，而统一初始化 {} 是解决它的关键手段。

问题的本质

C++ 的语法规则允许函数声明与对象初始化在某些情况下形式相同，编译器会优先解析为函数声明，而非对象构造。

```cpp
class Timer { /* ... */ };

Timer t();  // 你以为：调用默认构造函数创建对象 t
            // 实际：声明一个函数 t，返回 Timer 类型，无参数
```

```cpp
struct Widget {
  Widget(int a, double b);  // 构造函数
};

Widget w(int(42), double(3.14));  // 你以为：构造对象 w，参数 42 和 3.14
                                  // 实际：声明函数 w，参数为 int 和 double，返回 Widget
```

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，编译起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&
/*
decltype 推导三规则
1. 如果 e 是一个没有带括号的标记符表达式或者类成员访问表达式（上例中的 (2) 和 (3)），
那么的 decltype(e) 就是 e 所代表的实体的类型。如果没有这种类型或者 e 是一个被重载的函数，则会导致编译错误。
2. 如果 e 是一个函数调用或者一个重载操作符调用，那么 decltype(e) 就是该函数的返回类型（上例中的 (1)）。
3. 如果 e 不属于以上所述的情况，则假设 e 的类型是 T：当 e 是一个左值时，decltype(e) 就是 T&；
否则（e 是一个右值），decltype(e) 是 T。
上例中的 (4) 即属于这种情况。在这个例子中，e 实际是 (a->x)，由于有这个括号，因此它不属于前面两种情况，
所以应当以本条作为判别依据。而 (a->x) 是一个左值，因此会返回 double &。
*/
//decltype 与 const，引用，指针的结合，可以参考
//https://www.cnblogs.com/cauchy007/p/4966485.html
```

基于范围的for循环

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  struct A {
5  int n;
6  A(int i):n(i) { }
7  };
8  int main()  {
9  int ary[] = {1,2,3,4,5};
10 for (int & e: ary)
11   e*= 10;
12 for (int e : ary)
13   cout << e << ",";
14 cout << endl;
15 vector<A> st(ary, ary+5);
16 for (auto & it: st)
17   it.n *= 10;
18 for (A it: st)
19   cout << it.n << ",";
20 return 0;
21 }
```

基于范围的for循环

```
1  #include <iostream>
2  #include <vector>
3  using namespace std;
4  struct A {
5  int n;
6  A(int i):n(i) { }
7  };
8  int main()  {
9  int ary[] = {1,2,3,4,5};
10 for (int & e: ary)
11   e*= 10;
12 for (int e : ary)
13   cout << e << ",";
14 cout << endl;
15 vector<A> st(ary, ary+5);
16 for (auto & it: st)
17   it.n *= 10;
18 for (A it: st)
19   cout << it.n << ",";
20 return 0;
21 }
```

输出：

10,20,30,40,50,

100,200,300,400,500,

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

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

```
1  int main() {
2  int x = 100, y=200, z=300;
3  cout << [ ](double a, double b) { return a + b; }(1.2, 2.5) << endl;
4  auto ff  = [=, &y, &z](int n) {
5    cout << x << endl;
6    y++; z++;
7    return n*n;
8  };
9  cout << ff(15) << endl;
10 cout << y << "," << z << endl;
11 return 0;
12 }
```

Lambda 表达式

```
1  int main() {
2  int x = 100, y=200, z=300;
3  cout << [ ](double a, double b) { return a + b; }(1.2, 2.5) << endl;
4  auto ff  = [=, &y, &z](int n) {
5    cout << x << endl;
6    y++; z++;
7    return n*n;
8  };
9  cout << ff(15) << endl;
10 cout << y << "," << z << endl;
11 return 0;
12 }
```

输出:

3.7

100

225

201,301

Lambda 表达式

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

Lambda 表达式

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

输出:

11 2 33 4

Lambda 表达式

```cpp
#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;
}
```

Lambda 表达式

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  using namespace std;
5  int main() {
6  vector<int>  a { 1,2,3,4};
7  int total = 0;
8  for_each(a.begin(),a.end(),[&](int & x) { total += x; x *= 2; });
9  cout << total << endl;  //输出 10
10 for_each(a.begin(),a.end(),[ ](int x) { cout << x << " ";});
11 return 0;
12 }
```

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：

```cpp
#include <iostream>
#include <functional>
using namespace std;

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

auto fib2 = [](int n) {
return n <= 2 ? 1 : fib(n-1) + fib(n-2);
};

int main() {
cout << fib(5) << endl;   //输出 5
cout << fib2(5) << endl;   //输出 5
return 0;
}

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

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

右值引用和 move 语义

```
1  #include <iostream>
2  #include <string>
3  #include <cstring>
4  using namespace std;
5  class String {
6  public:
7  char * str;
8  String():str(new char[1]) { str[0] = 0;}
9  String(const char * s) {
10   str = new char[strlen(s)+1];
11   strcpy(str,s);
12 }
13 String(const String & s) {
14   cout << "copy constructor called" << endl;
15   str = new char[strlen(s.str)+1];
16   strcpy(str,s.str);
17 }
18 String & operator=(const String & s) {
19   cout << "copy operator= called" << endl;
20   if( str != s.str) {
21     delete [] str;
22     str = new char[strlen(s.str)+1];
23     strcpy(str,s.str);
24   }
25   return * this;
26 }
27 String(String && s):str(s.str) {  // move constructor
28   cout << "move constructor called"<<endl;
29   s.str = new char[1];
30   s.str[0] = 0;
31 }
```

右值引用和 move 语义

```
32  String & operator = (String &&s) {// move assigment
33    cout << "move operator= called"<<endl;
34    if (str!= s.str) {
35      delete [] str;
36      str = s.str;
37      s.str = new char[1];
38      s.str[0] = 0;
39    }
40    return *this;
41  }
42  ~String() { delete [] str; }
43  };
44  template <class T>
45  void MoveSwap(T& a, T& b)   {
46  T tmp(move(a)); // std::move(a) 为右值，这里会调用 move constructor
47  a = move(b);    // move(b) 为右值，因此这里会调用 move assigment
48  b = move(tmp);  // move(tmp) 为右值，因此这里会调用 move assigment
49  }
50  int main() {
51  //String & r = String("this"); // error
52  String s;
53  s = String("ok"); // String("ok") 是右值
54  cout << "******" << endl;
55  String && r = String("this");
56  cout << r.str << endl;
57  String s1 = "hello",s2 = "world";
58  MoveSwap(s1,s2);
59  cout << s2.str << endl;
60  return 0;
61  }
```

右值引用和 move 语义

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

可移动但不可复制的对象

```
1  struct A{
2  A(const A & a) = delete;
3  A(const A && a) { cout << "move" << endl; }
4  A() { };
5  };
6  A b;
7  A func()  {
8  A a;
9  return a;
10 }
11 void func2(A a) {  }
12 int main() {
13 A a1;
14 A a2(a1);    //compile error
15 func2(a1);   //compile error
16 func();
17 return 0;
18 }
```

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。

智能指针 shared_ptr

```cpp
#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;
  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)
}
```

智能指针 shared_ptr

```cpp
#include <iostream>
#include  <memory> // 需要包含这个头文件
int main() {
  std::shared_ptr<int> p1 = std::make_shared<int>();// 使用 make_shared 创建空对象
  *p1 = 78;
  std::cout << "p1 = " << *p1 << std::endl; // 输出 78
  std::cout << "p1 Reference count = " << p1.use_count() << std::endl;// 打印引用个数: 1
  std::shared_ptr<int> p2(p1); // 第 2 个 shared_ptr 对象指向同一个指针
  // 下面两个输出都是: 2
  std::cout << "p2 Reference count = " << p2.use_count() << std::endl;
  std::cout << "p1 Reference count = " << p1.use_count() << std::endl;
  // 比较智能指针, p1 等于 p2
  if (p1 == p2) {
    std::cout << "p1 and p2 are pointing to same pointer\n";
  }
  std::cout<<"Reset p1 "<<std::endl;
  // 无参数调用 reset, 无关联指针, 引用个数为 0
  p1.reset();
  std::cout << "p1 Reference Count = " << p1.use_count() << std::endl;
  // 带参数调用 reset, 引用个数为 1
  p1.reset(new int(11));
  std::cout << "p1  Reference Count = " << p1.use_count() << std::endl;
  // 把对象重置为 NULL, 引用计数为 0
  p1 = nullptr;
  std::cout << "p1  Reference Count = " << p1.use_count() << std::endl;
  if (!p1) {
    std::cout << "p1 is NULL" << std::endl; // 输出
  }
  return 0;
}
```

智能指针 shared_ptr

```
1  #include <iostream>
2  #include <memory>
3  using namespace std;
4  struct A{
5    ~A() { cout << "~A" << endl; }
6  };
7  int main() {
8    A * p = new A();
9    shared_ptr<A> ptr(p);
10   shared_ptr<A> ptr2;
11   ptr2.reset(p);   //并不增加 ptr 中对 p 的托管计数
12   cout << "end" << endl;
13   return 0;
14 }
15
```

智能指针 shared_ptr

```cpp
#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

智能指针 shared_ptr

```cpp
#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 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

```cpp
#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; // error, bool b(nullptr); ok
  int i = nullptr; //error,nullptr 不能自动转换成整型
  return 0;
}
```

空指针 nullptr

```
1  #include <memory>
2  #include <iostream>
3  using namespace std;
4  int main()   {
5    int* p1 = NULL;
6    int* p2 = nullptr;
7    shared_ptr<double> p3 = nullptr;
8    if(p1 == p2)
9      cout << "equal 1" <<endl;
10   if (p3 == nullptr)
11     cout << "equal 2" <<endl;
12   if (p3 == p2) ; // error
13   if (p3 == NULL)
14     cout << "equal 4" <<endl;
15   bool b = nullptr; // error, bool b(nullptr); ok
16   int i = nullptr; //error,nullptr 不能自动转换成整型
17   return 0;
18 }
```

去掉出错的语句后输出：

equal 1

equal 2

equal 4

空指针 nullptr

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

void func(void* t) {
  cout << "func ( void* ) " << endl;
}

void func(int i) {
  cout << "func ( int )  " << endl;
}

int main() {
  func(NULL);
  func(nullptr);
  return 0;
}
```

结果是什么?

空指针 nullptr

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

void func(void* t) {
  cout << "func ( void* ) " << endl;
}

void func(int i) {
  cout << "func ( int )  " << endl;
}

int main() {
  func(NULL);
  func(nullptr);
  return 0;
}
```

结果是什么?

```cpp
#define NULL ((void *)0) //C 语言

#define NULL 0
// C++11 起
#define NULL nullptr
```

override 和 final 关键字

```
1  class Base {
2    virtual void foo() final {}  // 禁止子类重写
3  };
4  class Derived : public Base {
5    void foo() override {}  // 显式标记重写（编译时报错，因为基类已 final）
6  };
```

无序容器 (哈希表)

```cpp
#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

正则表达式

```cpp
#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;
}
```

正则的细节可以参考：https://www.cnblogs.com/hesse-summer/p/10875487.html

https://cppinsights.io/

多线程

```cpp
#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";
}

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;
}
```

C++14 重要改进

- 泛型 Lambda:Lambda 参数支持 `auto`，实现泛型。

```
auto print = [](const auto& x) { std::cout << x; };
print(42);     // int
print("hello"); // const char*
```

- 返回类型推导 (`auto` 函数): 函数返回类型可由 `return` 语句推导。

```
auto add(int a, int b) { return a + b; } // 返回 int
```

- 二进制字面量和数字分隔符：提升代码可读性。

```
int bin = 0b1100'1010;  // 二进制表示
double pi = 3.1415'9265;
```

- `std::make_unique`：创建 `unique_ptr` 的标准方式。

```
auto ptr = std::make_unique<MyClass>(42, "example");
auto arr = std::make_unique<int[]>(10);  // 管理 10 个 int 的数组
```

C++17 结构化绑定

```
1 auto [var1, var2, ..., varN] = expression;
```

- auto 推导类型，变量数量必须与表达式结果成员数量一致。
- 支持引用修饰：auto&, const auto&, auto&&。

C++17 结构化绑定

```cpp
#include <iostream>
#include <tuple>
struct Point { int x; int y; };

int main() {
  // 结构体绑定
  Point p{3, 4};
  auto [a, b] = p;
  std::cout << a << ", " << b << "\n"; // 3, 4
  // 数组绑定
  int arr[] = {5, 6};
  auto& [c, d] = arr;
  c = 7;
  std::cout << arr[0] << ", " << arr[1] << "\n"; // 7, 6
  // 元组绑定
  auto t = std::make_tuple(8, 9.5, 'A');
  auto [e, f, g] = t;
  std::cout << e << ", " << f << ", " << g << "\n"; // 8, 9.5, A
  // 引用语义
  auto& [x, y] = p;
  x = 10;
  std::cout << p.x << "\n"; // 10
}
```

if 和 switch 初始化语句

```cpp
if (auto it = m.find(key); it != m.end()) {
    // it 仅在此作用域有效
}
```

```cpp
switch (auto code = fetch_status(); code) {
  case 200:
    std::cout << "OK (code: " << code << ")\n";
    break;
  case 404:
    std::cout << "Not Found (code: " << code << ")\n";
    break;
  default:
    std::cout << "Unknown code: " << code << "\n";
}
// code 超出作用域，无法访问
```