

C++11 常用新特性

目录

- 1.自动类型推导
- 2.右值引用
- 3.列表初始化
- 4.For each
- 5.Lambda表达式
- 6.智能指针
- 7.可变参数模板
- 8.默认成员函数控制
- 9.新增容器

自动类型推导 auto

在C++11之前，auto关键字用来指定存储期。在新标准中，它的功能变为类型推断。

auto现在成了一个类型的占位符，通知编译器去根据初始化代码推断所声明变量的真实类型。

各种作用域内声明变量都可以用到它。例如，名空间中，程序块中，或是for循环的初始化语句中。

自动类型推导 auto

在没有auto以前，遍历一个容器需要这样来书写一个迭代器：

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
int main() {
```

```
    vector<int> vc = { 1,2,3,4,5,6 };
```

```
    for (vector<int>::iterator it = vc.begin(); it != vc.end(); it++) {
```

```
        cout << *it << " ";
```

```
    }
```

```
    return 0;
```

```
}
```

自动类型推导 auto

有了auto之后，可以写出如下代码：

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
int main() {
```

```
    vector<int> vc = { 1,2,3,4,5,6 };
```

```
    for (auto it= vc.begin(); it != vc.end(); it++) {
```

```
        cout << *it << " ";
```

```
    }
```

```
    return 0;
```

```
}
```

auto 与 const

先看一段代码：

```
int x = 0;
```

```
const auto n = x;
```

```
auto f = n;
```

```
const auto& r1 = x;
```

```
auto& r2 = r1;
```

auto 与 const

```
int x = 0;  
const auto n = x;  
auto f = n;  
const auto& r1 = x;  
auto& r2 = r1;
```

- 1.第 2 行代码中，n 为 const int，auto 被推导为 int。
- 2.第 3 行代码中，n 为 const int 类型，但是 auto 却被推导为 int 类型，这说明当=右边的表达式带有 const 属性时，auto 不会使用 const 属性，而是直接推导出 non-const 类型。
- 3.第 4 行代码中，auto 被推导为 int 类型，这个很容易理解，不再赘述。
- 4.第 5 行代码中，r1 是 const int & 类型，auto 也被推导为 const int 类型，这说明当 const 和引用结合时，auto 的推导将保留表达式的 const 类型。

总结：

- 1.当类型不为引用时，auto 的推导结果将不保留表达式的 const 属性；
- 2.当类型为引用时，auto 的推导结果将保留表达式的 const 属性。

auto 的高级用法

auto 除了可以独立使用，还可以和某些具体类型混合使用，这样 auto 表示的就是“半个”类型，而不是完整的类型：

```
int x = 0;
```

```
auto *pt1 = &x; //pt1 为 int * , auto 推导为 int
```

```
auto pt2 = &x; //pt2 为 int* , auto 推导为 int*
```

```
auto &r1 = x; //r1 为 int& , auto 推导为 int
```

```
auto r2 = r1; //r2 为 int , auto 推导为 int
```

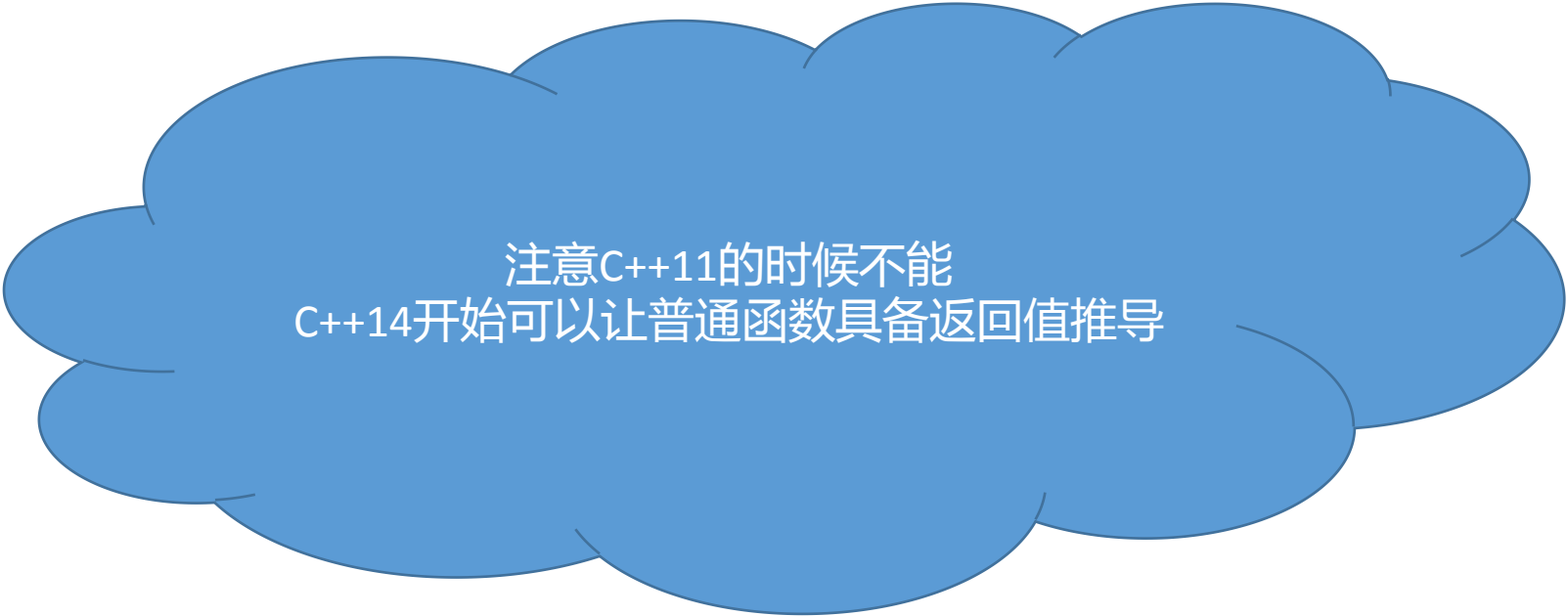

auto 的限制

1. auto 不能在函数的参数中使用
2. auto 不能作用于类的非静态成员变量
3. auto 不能作用于模板参数
4. auto 不能用于推导数组类型

auto 的限制

1. auto 不能在函数的参数中使用

```
auto funcName(int v1,int v2)
{
    return v1 + v2;
}
```



注意C++11的时候不能
C++14开始可以让普通函数具备返回值推导

auto 的限制

2.auto 不能作用于类的非静态成员变量

```
class Student {  
protected:  
    auto name;  
    auto age;  
};
```

就是没有static关键字修饰的成员变量

auto 的限制

3.auto 不能作用于模板参数

```
template <typename T>
```

```
class Student{
```

```
    //dosomething:
```

```
};
```

```
int main(){
```

```
    Student<int> s1;
```

```
    Student<auto> s2 = s1; //错误
```

```
    return 0;
```

```
}
```

auto 的限制

4.auto 不能用于推导数组类型

```
int arr[100] = {0};
```

```
auto auto_arr = arr;
```

```
auto auto_arr2[1] = arr;
```

decltype

decltype，在C++中，作为操作符，用于查询表达式的数据类型。

decltype在C++11标准制定时引入，主要是为泛型编程而设计，以解决泛型编程中，由于有些类型由模板参数决定，而难以（甚至不可能）表示的问题。

decltype 关键字是为了解决 auto 关键字只能对变量进行类型推导的缺陷而出现的。它的用法和 sizeof 很相似。

在此过程中，编译器分析表达式并得到它的类型，却不实际计算表达式的值。有时候，我们可能需要计算某个表达式的类型

lambda表达式如果我们想要使用它的类型我们就需要使用decltype

decltype

```
auto num1 = 100 ;  
auto num2 = 200;  
decltype(num1 + num2) num3;
```

num3的类型就是num1 + num2 最终的结果的类型。

decltype

在泛型编程中，可能需要通过参数的运算来得到返回值的类型：

```
#include <iostream>
using namespace std;
template <typename R, typename T, typename U>
R add(T t, U u)
{
    return t + u;
}
int main() {
    int a = 1; float b = 2.0;
    auto c = add<decltype(a + b)>(a, b);
    cout << c << endl;
    return 0;
}
```


decltype

下面的代码正常吗？

```
template <typename T, typename U>  
decltype(t + u) add(T t, U u)  
{  
    return t + u;  
}
```

decltype

下面的代码正常吗？

```
template <typename T, typename U>
decltype(t + u) add(T t, U u)
{
    return t + u;
}
```

C++ 的返回值是前置语法，在返回值定义的时候参数变量还不存在。
所以报错！

decltype

```
template <typename T, typename U>
auto add(T t, U u)->decltype(t+u) //尾随返回类型
{
    return t + u;
}
```

C++14开始可以直接写成：

```
template <typename T, typename U>
auto add(T t, U u)
{
    return t + u;
}
```

右值引用&&

C++98 中提出了引用的概念，引用即别名，引用变量与其引用实体公共同一块内存空间，而引用的底层是通过指针来实现的，因此使用引用，可以提高程序的可读性。

比如交换两个变量的值，消除两个对象交互时不必要的对象拷贝，节省运算存储资源，提高效率。

```
void change(int& n1, int& n2)
{
    int temp = n1;
    n1 = n2;
    n2 = temp;
}
```

右值引用&&

为了提高程序运行效率，C++11中引入了右值引用，右值引用也是别名，但其只能对右值引用。

```
int func1(int v1, int v2){
    return v1 + v2;
}

int main()
{
    const int&& num1 = 10;//右值示例
    // 引用函数返回值，返回值是一个临时变量，为右值
    int&& num2 = func1(10, 20);

    cout << num1 << endl;
    cout << num2 << endl;
    return 0;
}
```

右值与左值

- 1.一般认为：左值可放在赋值符号的左边，右值可以放在赋值符号的右边；或者能够取地址的称为左值，不能取地址的称为右值
- 2.左值也能放在赋值符号的右边，右值只能放在赋值符号的右边

右值与左值

```
int num = 100;
// 函数的返回值结果为引用
int& returnNum(){
    return num;
}
int main()
{
    int num2 = 10;
    int num3 = num2;
    int* p = new int(0);
    // num2和num3,p和*p都是左值，说明：左值既可放在=的左侧，也可放在=的右侧
    const int num4 = 30;
    //num4 = num1;
    // 特例：num4 虽然是左值，但是为const常量，只读不允许被修改
    cout << &num4 << endl;
    // num4可以取地址，所以num4严格来看也是左值
    //num3 + 2 = 200;
    // 编译失败：因为num3+2的结果是一个临时变量，没有具体名称，也不能取地址，因此为右值
    returnNum() = 111;

    return 0;
}
```

移动语义

转移语义可以将资源(堆、系统对象等)从一个对象转移到另一个对象，这样可以减少不必要的临时对象的创建、拷贝及销毁。移动语义与拷贝语义是相对的，可以类比文件的剪切和拷贝。在现有的C++机制中，自定义的类要实现转移语义，需要定义移动构造函数，还可以定义转移赋值操作符。

以string类的移动构造函数为例：

```
MyString(MyString&& str) {  
    cout << "move ctor source from " << str.data << endl;  
    len = str.len;  
    data = str.data;  
    str.len = 0;  
    str.data = NULL;  
}
```


移动语义

和拷贝构造函数类似，有几点需要注意：

参数(右值)的符号必须是&&

参数(右值)不可以是常量，因为我们需要修改右值

参数(右值)的资源链接和标记必须修改，否则，右值的析构函数就会释放资源。转移到新对象的资源也就无效了。

标准库函数`std::move` 可以将左值变成一个右值。

编译器只对右值引用才能调用移动构造函数，那么如果已知一个命名对象不再被使用，此时仍然想调用它的移动构造函数，也就是把一个左值引用当做右值引用来使用，该怎么做呢？用`std::move`，这个函数以非常简单的方式将左值引用转换为右值引用。

列表初始化

统一列表初始化的使用

在 C++98 中，标准允许使用花括号 {} 对数组元素进行统一的列表初始值设定。比如：

```
int arr1[] = {1,2,3,4,5};
```

```
int arr2[100] = {0};
```

但对于一些自定义类型却不行，例如：

```
vector<int> vc{1,2,3,4,5};
```

在c++98中是无法编译成功的，只能够定义vector对应之后通过循环进行插入元素达到这个目的。

C++11扩大了用大括号括起的列表(初始化列表)的使用范围，使其可用于所有的内置类型和用户自定义的类型，使用初始化列表时，**可添加等号(=)，也可不添加。**

列表初始化

```
#include<iostream>
#include<vector>
#include<map>
using namespace std;
class ClassNum {
public:
    ClassNum(int n1 = 0, int n2 = 0) : _x(n1), _y(n2)
    {}
private:
    int _x;
    int _y;
};
int main() {
    int num1 = { 100 };//定于内置类型
    int num2{ 3 };//也可以不加=
    //数组
    int arr1[5] = { 1,3,4,5,6 };
    int arr2[] = { 4,5,6,7,8 };

    //STL中的容器
    vector<int>v{ 12,2 };
    map<int, int>mp{ {1,2},{3,4} };
    //自定义类型初始化
    ClassNum p{ 1, 2 };
    return 0;
}
```

For each

C++11 引入了基于范围的迭代写法，拥有了能够写出像 Python 类似的简洁的循环语句。

最常用的 vector 遍历将从原来的样子：

```
int main()
{
    vector<int> v1={ 1,2,3,4,5,6 };
    for (auto i : v1)//范围for
    {
        cout << i << " ";
    }
    return 0;
}
```

Lambda表达式

lambda表达式实际上是一个匿名类函数，在编译时会将表达式转换为匿名类函数。

语法：

[capture-list] (parameters) mutable -> return-type { statement}

[捕获列表](参数)->返回值{ 函数体 };

Lambda表达式

`[capture-list] (parameters) mutable -> return-type { statement }`

- ① `[capture-list]` : 捕捉列表，该列表总是出现在lambda函数的开始位置，编译器根据[]来判断接下来的代码是否为lambda函数，捕捉列表能够捕捉上下文中的变量供lambda函数使用。
(不能省略)
- ② `(parameters)` : 参数列表。与普通函数的参数列表一致，如果不需要参数传递，则可以连同()一起省略 (没有参数可以省略)
- ③ `mutable` : 默认情况下，lambda函数总是一个const函数，mutable可以取消其常量性。使用该修饰符时，参数列表不可省略(即使参数为空)。mutable放在参数列表和返回值之间
- ④ `->returntype` : 返回值类型。用追踪返回类型形式声明函数的返回值类型，没有返回值时此部分可省略。返回值类型明确情况下，也可省略，由编译器对返回类型进行推导。
(又没有返回值都可以省略)
- ⑤ `{statement}` : 函数体。在该函数体内，除了可以使用其参数外，还可以使用所有捕获到的变量。(不能省略)

Lambda表达式

捕获列表说明

捕捉列表描述了上下文中那些数据可以被lambda使用，以及使用的方式传值还是传引用。

- ① [a,&b] 其中 a 以复制捕获而 b 以引用捕获。
- ② [this] 以引用捕获当前对象 (*this)
- ③ [&] 以引用捕获所有用于 lambda 体内的自动变量，并以引用捕获当前对象，若存在
- ④ [=] 以复制捕获所有用于 lambda 体内的自动变量，并以引用捕获当前对象，若存在
- ⑤ [] 不捕获，大部分情况下不捕获就可以了

Lambda表达式

在lambda函数定义中，参数列表和返回值类型都是可选部分，而捕捉列表和函数体可以为空。

C++11中最简单的lambda函数为：`[]{};` 该lambda函数不能做任何事情。没有意义！

Lambda表达式

```
#include<iostream>
using namespace std;
void(*FP)(); //函数指针
int main()
{
    // 最简单的lambda表达式, 该lambda表达式没有任何意义
    [] {};

    // 省略参数列表和返回值类型, 返回值类型由编译器推导为int
    int num1 = 3, num2 = 4;

    // 省略了返回值类型, 无返回值类型
    auto fun1 = [&num1, &num2](int num3) {num2 = num1 + num3; };
    fun1(100);
    cout << num1 << " " << num2 << endl;

    //捕捉列表可以是lambda表达式
    auto fun = [fun1] {cout << "great" << endl; };
    fun();

    // 各部分都很完善的lambda函数
    auto fun2 = [=, &num2](int num3)->int {return num2 += num1 + num3; };
    cout << fun2(10) << endl;
```

Lambda表达式

```
// 复制捕捉x
```

```
int x = 10;
```

```
auto add_x = [x](int a) mutable { x *= 2; return a + x; };
```

```
cout << add_x(10) << endl;
```

```
// 编译失败--->提示找不到operator=()
```

```
//auto fun3 = [&num1,&num2](int num3) {num2 = num1 + num3;};
```

```
//fun1 = fun3;
```

```
//允许使用一个lambda表达式拷贝构造一个新的副本
```

```
auto fun3(fun);
```

```
fun();
```

```
//可以将lambda表达式赋值给相同类型的函数指针
```

```
auto f2 = [] {};
```

```
FP = f2;
```

```
FP();
```

```
return 0;
```

```
}
```

Lambda表达式

函数对象与lambda表达式

函数对象，又称为仿函数，即可以想函数一样使用的对象，就是在类中重载了operator()运算符的类对象

从使用方式上来看，函数对象与lambda表达式完全一样。

Lambda表达式

```
#include <iostream>
using namespace std;
class Rate{
public:
    Rate(double rate) : _rate(rate){}
    double operator()(double money, int year)
    {
        return money * _rate * year;
    }
private:
    double _rate;
};
int main()
{
    // 函数对象
    double rate = 0.6;
    Rate r1(rate);
    double rd = r1(20000, 2);
    cout << rd << endl;
    // lambda
    auto r2 = [=](double monty, int year)->double {return monty * rate * year;};
    double rd2 = r2(20000, 2);
    cout << rd2 << endl;
    return 0;
}
```

Lambda表达式

在C++98中，对一个数据集中的元素进行排序，可以使用sort方法。

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;
int main()
{
    int array[] = { 3,6,9,5,4,7,0,8,2,1 };
    // 默认按照小于比较，排出来结果是升序
    sort(array, array + sizeof(array) / sizeof(array[0]));
    // 如果需要降序，需要改变元素的比较规则
    sort(array, array + sizeof(array) / sizeof(array[0]), greater<int>());
    return 0;
}
```

Lambda表达式

如果待排序元素为自定义类型，需要用户定义排序时的比较规则

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;
struct Goods{
    string name;
    double price;
};
struct Compare{
    bool operator()(const Goods& gl, const Goods& gr)
    {
        return gl.price <= gr.price;
    }
};
int main()
{
    Goods gds[] = { { "苹果", 5.1 }, { "橙子", 9.2 }, { "香蕉", 3.6 }, { "菠萝", 9.6 } };
    sort(gds, gds + sizeof(gds) / sizeof(gds[0]), Compare());
    for (int i = 0; i < 4; i++) {
        cout << gds[i].name << " " << gds[i].price << endl;
    }
    return 0;
}
```

Lambda表达式

有了lambda表示，代码就可以写成如下：

```
#include <iostream>
#include <algorithm>
#include <functional>
using namespace std;
struct Goods{
    string name;
    double price;
};
int main()
{
    Goods gds[] = { { "苹果", 5.1 }, { "橙子", 9.2 }, { "香蕉", 3.6 }, {"菠萝",9.6} };
    sort(gds, gds + sizeof(gds) / sizeof(gds[0]), [](const Goods& l, const Goods& r)->bool
        {
            return l.price < r.price;
        });
    for (int i = 0; i < 4; i++) {
        cout << gds[i].name << " " << gds[i].price << endl;
    }
    return 0;
}
```

智能指针

1.shared_ptr

2.unique_ptr

3.weak_ptr

智能指针

垃圾回收机制已经大行其道，得到了诸多编程语言的支持，例如 Java、Python、C#、PHP 等。而 C++ 虽然从来没有公开得支持过垃圾回收机制，但 C++98/03 标准中，支持使用 `auto_ptr` 智能指针来实现堆内存的自动回收；C++11 新标准在废弃 `auto_ptr` 的同时，增添了 `unique_ptr`、`shared_ptr` 以及 `weak_ptr` 这 3 个智能指针来实现堆内存的自动回收。

所谓智能指针，可以从字面上理解为“智能”的指针。具体来讲，智能指针和普通指针的用法是相似的，不同之处在于，智能指针可以在适当时机自动释放分配的内存。

也就是说，使用智能指针可以很好地避免“忘记释放内存而导致内存泄漏”问题出现。由此可见，C++ 也逐渐开始支持垃圾回收机制了。

智能指针shared_ptr

智能指针都是以类模板的方式实现的，shared_ptr也不例外。
shared_ptr<T>（其中T表示指针指向的具体数据类型）的定义位于<memory>头文件，并位于std命名空间中，因此在使用该类型指针时，程序中应包含如下2行代码：

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

和unique_ptr、weak_ptr不同之处在于，多个shared_ptr智能指针可以共同使用同一块堆内存。并且，由于该类型智能指针在实现上采用的是引用计数机制，即便有一个shared_ptr指针放弃了堆内存的“使用权”（引用计数减1），也不会影响其他指向同一堆内存的shared_ptr指针（只有引用计数为0时，堆内存才会被自动释放）。

智能指针shared_ptr

shared_ptr智能指针的创建

shared_ptr<T> 类模板中，提供了多种实用的构造函数

```
shared_ptr<int> p1;      //不传入任何实参
```

```
shared_ptr<int> p2(nullptr); //传入空指针 nullptr
```

空的 shared_ptr 指针，其初始引用计数为 0，而不是 1。

智能指针shared_ptr

shared_ptr智能指针的创建

在构建 shared_ptr 智能指针，也可以明确其指向

```
shared_ptr<int> p(new int(5));
```

C++11 标准中还提供了 std::make_shared<T> 模板函数，其可以用于初始化 shared_ptr 智能指针

```
shared_ptr<int> p = make_shared<int>(5);
```

智能指针shared_ptr

shared_ptr智能指针的创建

shared_ptr<T> 模板还提供有相应的拷贝构造函数和移动构造函数

```
shared_ptr<int> p3;
```

```
//调用拷贝构造函数
```

```
shared_ptr<int> p4(p3); //或者 shared_ptr<int> p4 = p3;
```

```
//调用移动构造函数
```

```
shared_ptr<int> p5(move(p4)); //或者 shared_ptr<int> p5 = move(p4);
```

智能指针shared_ptr

shared_ptr智能指针的创建

在初始化 shared_ptr 智能指针时，还可以自定义所指堆内存的释放规则，这样当堆内存的引用计数为 0 时，会优先调用自定义的释放规则。

在某些应用场景中，自定义释放规则是很有必要的。比如，对于申请的动态数组来说，shared_ptr 指针默认的释放规则是不支持释放数组的，只能自定义对应的释放规则，才能正确地释放申请的堆内存。

对于申请的动态数组，释放规则可以使用 C++11 标准中提供的 default_delete<T> 模板类，我们也可以自定义释放规则：

智能指针shared_ptr

```
#include <iostream>
using namespace std;
//自定义释放规则
void deleteInt(int* p) {
    delete[]p;
}
int main()
{
    //指定 default_delete 作为释放规则
    shared_ptr<int> p1(new int[3], default_delete<int[]>());

    //初始化智能指针，并自定义释放规则
    shared_ptr<int> p2(new int[3], deleteInt);
    return 0;
}
```

借助lambda, p2还可以写成这样：

```
shared_ptr<int> p2(new int[2], [](int* p) {delete[]p; });
```

智能指针unique_ptr

unique_ptr 指针自然也具备“在适当时机自动释放堆内存空间”的能力。和 shared_ptr 指针最大不同之处在于，unique_ptr 指针指向的堆内存无法同其它 unique_ptr 共享，也就是说，每个 unique_ptr 指针都独自拥有对其所指堆内存空间的所有权。

智能指针weak_ptr

shared_ptr是采用引用计数的智能指针，多个shared_ptr实例可以指向同一个动态对象，并维护了一个共享的引用计数器。

对于引用计数法实现的计数，总是避免不了循环引用的问题，shared_ptr也不例外。

看案例

智能指针weak_ptr

```
#include <iostream>
using namespace std;
class CB;
class CA
{
public:
    CA() { cout << "CA() called! " << endl; }
    ~CA() { cout << "~CA() called! " << endl; }
    void set_ptr(shared_ptr<CB>& ptr) { m_ptr_b = ptr; }
    void b_use_count() { cout << "b use count : " << m_ptr_b.use_count() << endl; }
    void show() { cout << "this is class CA!" << endl; }
private:
    shared_ptr<CB> m_ptr_b;
};

class CB
{
public:
    CB() { cout << "CB() called! " << endl; }
    ~CB() { cout << "~CB() called! " << endl; }
    void set_ptr(shared_ptr<CA>& ptr) { m_ptr_a = ptr; }
    void a_use_count() { cout << "a use count : " << m_ptr_a.use_count() << endl; }
    void show() { cout << "this is class CB!" << endl; }
private:
    shared_ptr<CA> m_ptr_a;
};
```

智能指针weak_ptr

```
int main(){
    shared_ptr<CA> ptr_a(new CA());
    shared_ptr<CB> ptr_b(new CB());

    cout << "a use count : " << ptr_a.use_count() << endl;
    cout << "b use count : " << ptr_b.use_count() << endl;

    ptr_a->set_ptr(ptr_b);
    ptr_b->set_ptr(ptr_a);

    cout << "a use count : " << ptr_a.use_count() << endl;
    cout << "b use count : " << ptr_b.use_count() << endl;

    return 0;
}
```

Microsoft Visual Studio 调试控制台

CA() called!

CB() called!

a use count : 1

b use count : 1

a use count : 2

b use count : 2

E:\VStudioCode\Test002\x64\Debug\T

按任意键关闭此窗口. . .

智能指针weak_ptr

可以看到最后两个类都没有被析构。

这个时候，可以用weak_ptr来解决这个问题，可以把两个类中的一个成员变量改为weak_ptr对象即可。weak_ptr不会增加应用计数，所以引用就构不成环。

private:

```
weak_ptr<CB> m_ptr_b;
```

可变参数模板

C++11之前，类模版和函数模版中只能含固定数量的模版参数

C++11的新特性可变参数模板能够让您创建可以接受可变参数的函数模板和类模。

```
#include <iostream>
```

```
using namespace std;
```

```
//函数模板的参数个数为0到多个参数，每个参数的类型可以各不相同
```

```
template<class...T>
```

```
void funcName(T...args) { //args一包形参，T一包类型
```

```
    cout << sizeof...(args) << endl; //sizeof...固定语法格式计算获取到模板参数的个数
```

```
    cout << sizeof...(T) << endl; //注意sizeof...只能计算...的可变参
```

```
}
```

```
int main() {
```

```
    funcName(100, 200, 300, 400, 600);
```

```
    return 0;
```

```
}
```

参数包展开

1.递归函数

2.使用if constexpr

参数包展开

1.递归函数

2.使用if constexpr

参数包展开

//递归

```
#include <iostream>
using namespace std;
void funcName1() {
    cout << "递归终止函数" << endl;
}
template<class T, class ...U>
void funcName1(T frist, U...others) {
    cout << "收到的参数值:" << frist << endl;
    funcName1(others...); //注意这里传进来的是一包形参不能省略...
}
int main() {
    funcName1(1,2, 3, 4, 5, 6);
    return 0;
}
```


参数包展开

```
//if constexpr
```

```
#include <iostream>
```

```
using namespace std;
```

```
//if constexpr...()//constexpr代表的是常量的意思或者是编译时求值
```

```
//c++17中新增一个语句叫做编译期间if语句 ( constexpr if )
```

```
template<class T, class...U>
```

```
void funcName2(T frist, U...args) {
```

```
    cout << "收到参数：" << frist << endl;
```

```
    if constexpr (sizeof...(args) > 0) { //constexpr必须有否则无法编译成功，圆括号里面是常量表达式
```

```
        funcName2(args...);
```

```
    }
```

```
}
```

```
int main() {
```

```
    funcName2(1,2, 3, 4, 5, 6);
```

```
    return 0;
```

```
}
```

默认成员函数控制

在C++中对于空类编译器会生成一些默认的成员函数，比如：构造函数、拷贝构造函数、运算符重载、析构函数和&和const&的重载、移动构造、移动拷贝构造等函数。

如果在类中显式定义了，编译器将不会重新生成默认版本。

有时候这样的规则可能被忘记，最常见的是声明了带参数的构造函数，必要时则需要定义不带参数的版本以实例化无参的对象。而且有时编译器会生成，有时又不生成，容易造成混乱，于是C++11让程序员可以控制是否需要编译器生成。

默认成员函数控制

```
#include <iostream>
using namespace std;
class ClassTest
{
public:
    ClassTest(int num):n(num){}
    ClassTest() = default; // 显式缺省构造函数 让编译器生成不带参数的默认构造函数，改成delete就不会生成了
    ClassTest(const ClassTest&) = delete; // 禁止编译器生成默认的拷贝构造函数
    ClassTest& operator=(const ClassTest& a); // 在类中声明，在类外定义时，让编译器生成默认赋值运算符重载

private:
    int n;
};
ClassTest& ClassTest::operator=(const ClassTest& a) = default; // 在类外让编译器默认生成
int main() {
    ClassTest c1;
    return 0;
}
```

新增容器

1.std::array

2.std::forward_list

3.unordered系列

std::array

1.std::array 保存在栈内存中，相比堆内存中的 std::vector，它能够灵活的访问容器里面的元素，从而获得更高的性能。

2.std::array 会在编译时创建一个固定大小的数组，std::array 不能够被隐式的转换成指针，使用 std::array 只需指定其类型和大小即可！

3.c++11 封装了相关的数组模板类，不同于 C 风格数组，它不会自动退化成 T* 类型，它能作为聚合类型聚合初始化。

std::array 是封装固定大小数组的容器，数组元素下标索引从 0 开始

定义和初始化 std::array<> 对象

```
std::array<int, 5> arr;
```

此处，std::array 对象 arr 表示一个固定大小为 5 且未初始化的 int 数组，因此所有 5 个元素都包含垃圾值。

```
std::array < std::string, 10 > arr1;
```

这里，std::array 对象 arr1 表示一个固定大小为 10 的字符串数组。

// 前 2 个值将被初始化，其他值为 0。

```
std::array < int , 10 > arr3 = { 1, 2 };
```

定义和初始化 std::array<> 对象

```
#include <iostream>
```

```
#include <array>
```

```
using namespace std;
```

```
int main() {
```

```
    array<int,3> arr = { 1,2,3 };
```

```
    int len = 3;
```

```
    array<int, len> arr2 = { 4,5,6 };
```

```
    return 0;
```

```
}
```

代码有什么问题吗？

定义和初始化 std::array<> 对象

```
#include <iostream>
```

```
#include <array>
```

```
using namespace std;
```

```
int main() {
```

```
    array<int,3> arr = { 1,2,3 };
```

```
    int len = 3;
```

```
    //非法, 数组大小参数必须是常量表达式
```

```
    array<int, len> arr2 = { 4,5,6 };
```

```
    return 0;
```

```
}
```


访问 std::array 中的元素

有 3 种方法可以访问 std::array 中的元素

// 创建并初始化一个大小为 10 的数组。

```
std::array<int, 10> arr = { 1,2,3,4,5,6,7,8,9,10 };
```

运算符 [] : 使用运算符 [] 访问 std::array 中的元素

```
int x = arr [ 2 ] ;
```

使用 [] 运算符访问超出范围的元素将导致未定义的行为。

at() : 使用 at() 成员函数访问 std::array 中的元素

```
// Accessing element using at() function
```

```
int x = arr.at(2);
```

使用 at() 函数访问任何超出范围的元素将抛出 out_of_range 异常。

std::tuple 的 get<>()

```
int x = std::get< 2 >( arr ) ;
```

使用 get<> 运算符访问超出范围的元素将导致编译时错误。

其他常用函数

元素访问

<code>at(C++11)</code>	访问指定的元素，同时进行越界检查 (公开成员函数)
<code>operator[] (C++11)</code>	访问指定的元素 (公开成员函数)
<code>front(C++11)</code>	访问第一个元素 (公开成员函数)
<code>back(C++11)</code>	访问最后一个元素 (公开成员函数)
<code>data(C++11)</code>	直接访问底层数组 (公开成员函数)

迭代器

<code>begin cbegin(C++11)</code>	返回指向起始的迭代器 (公开成员函数)
<code>end cend(C++11)</code>	返回指向末尾的迭代器 (公开成员函数)
<code>rbegin crbegin(C++11)</code>	返回指向起始的逆向迭代器 (公开成员函数)
<code>rend crend(C++11)</code>	返回指向末尾的逆向迭代器 (公开成员函数)

容量

<code>empty(C++11)</code>	检查容器是否为空 (公开成员函数)
<code>size(C++11)</code>	返回容纳的元素数 (公开成员函数)
<code>max_size(C++11)</code>	返回可容纳的最大元素数 (公开成员函数)

操作

<code>fill(C++11)</code>	以指定值填充容器 (公开成员函数)
<code>swap(C++11)</code>	交换内容 (公开成员函数)

std::forward_list

std::forward_list 是一个列表容器，使用方法和 std::list 基本类似。

与 std::list 的双向链表的实现不同，std::forward_list 使用单向链表进行实现，提供了 $O(1)$ 复杂度的元素插入，不支持快速随机访问，也是标准库容器中唯一一个不提供 size() 方法的容器

forward_list 具有插入、删除表项速度快、消耗内存空间少的特点，但只能向前遍历。与其它序列容器(array、vector、deque)相比，forward_list 在容器内任意位置的成员的插入、提取(extracting)、移动、删除操作的速度更快，因此被广泛用于排序算法。

std::forward_list创建

由于 forward_list 容器以模板类 forward_list<T>（ T 为存储元素的类型 ）的形式被包含在 <forward_list> 头文件中，并定义在 std 命名空间中。因此，在使用该容器之前，代码中需包含下面两行代码：

```
#include <forward_list>
```

```
using namespace std;
```

创建 forward_list 容器的方式，大致分为以下 5 种。

std::forward_list创建

1) 创建一个没有任何元素的空 forward_list 容器：

```
std::forward_list<int> values;
```

由于 forward_list 容器在创建后也可以添加元素，因此这种创建方式很常见。

2) 创建一个包含 n 个元素的 forward_list 容器：

```
std::forward_list<int> values(10);
```

通过此方式创建 values 容器，其中包含 10 个元素，每个元素的值都为相应类型的默认值（int 类型的默认值为 0）。

3) 创建一个包含 n 个元素的 forward_list 容器，并为每个元素指定初始值。例如：

```
std::forward_list<int> values(10, 5);
```

如此就创建了一个包含 10 个元素并且值都为 5 个 values 容器。

4) 在已有 forward_list 容器的情况下，通过拷贝该容器可以创建新的 forward_list 容器。例如：

```
std::forward_list<int> value1(10);
```

```
std::forward_list<int> value2(value1);
```

注意，采用此方式，必须保证新旧容器存储的元素类型一致。

std::forward_list创建

5) 通过拷贝其他类型容器（或者普通数组）中指定区域内的元素，可以创建新的 forward_list 容器。例如：

//拷贝普通数组，创建forward_list容器

```
int a[] = { 1,2,3,4,5 };
```

```
std::forward_list<int> values(a, a+5);
```

//拷贝其它类型的容器，创建forward_list容器

```
std::array<int, 5>arr{ 11,12,13,14,15 };
```

```
std::forward_list<int>values(arr.begin()+1, arr.end());//拷贝arr容器中的{12,13,14,15}
```

其他函数

before_begin()	返回一个前向迭代器，其指向容器中第一个元素之前的位置。
begin()	返回一个前向迭代器，其指向容器中第一个元素的位置。
end()	返回一个前向迭代器，其指向容器中最后一个元素之后的位置。
cbefore_begin()	和 before_begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cbegin()	和 begin() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，不能用于修改元素。
empty()	判断容器中是否有元素，若无元素，则返回 true；反之，返回 false。
max_size()	返回容器所能包含元素个数的最大值。这通常是一个很大的值，一般是 $2^{32}-1$ ，所以我们很少会用到这个函数。
front()	返回第一个元素的引用。
assign()	用新元素替换容器中原有内容。
push_front()	在容器头部插入一个元素。
emplace_front()	在容器头部生成一个元素。该函数和 push_front() 的功能相同，但效率更高。
pop_front()	删除容器头部的一个元素。
emplace_after()	在指定位置之后插入一个新元素，并返回一个指向新元素的迭代器。和 insert_after() 的功能相同，但效率更高。
insert_after()	在指定位置之后插入一个新元素，并返回一个指向新元素的迭代器。

其他函数

<code>erase_after()</code>	删除容器中某个指定位置或区域内的所有元素。
<code>swap()</code>	交换两个容器中的元素，必须保证这两个容器中存储的元素类型是相同的。
<code>resize()</code>	调整容器的大小。
<code>clear()</code>	删除容器存储的所有元素。
<code>splice_after()</code>	将某个 <code>forward_list</code> 容器中指定位置或区域内的元素插入到另一个容器的指定位置之后。
<code>remove(val)</code>	删除容器中所有等于 <code>val</code> 的元素。
<code>remove_if()</code>	删除容器中满足条件的元素。
<code>unique()</code>	删除容器中相邻的重复元素，只保留一个。
<code>merge()</code>	合并两个事先已排好序的 <code>forward_list</code> 容器，并且合并之后的 <code>forward_list</code> 容器依然是有序的。
<code>sort()</code>	通过更改容器中元素的位置，将它们进行排序。
<code>reverse()</code>	反转容器中元素的顺序。

unordered系列

C++11 引入了两组无序容器：

`std::unordered_set`/`std::unordered_multiset`

`std::unordered_map`/`std::unordered_multimap`

无序容器中的元素是不进行排序的，内部通过 Hash 表实现，插入和搜索元素的平均复杂度为 $O(1)$ 。

头文件

```
#include <unordered_map>
```

```
#include <unordered_set>
```

练习

利用lambda完成，输入一个字符串，统计字符串中某个字符的个数。

输入案例：

abcstringabc

a

输出案例：

2

练习 代码

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

int getCharNum(string s, char c) {
    int count = 0;
    for_each(s.begin(), s.end(), [&](char ch) {
        if (c == ch) {
            count++;
        }
    });
    return count;
}

int main() {
    string s1;
    char ch;
    cin >> s1;
    cin >> ch;
    cout << getCharNum(s1, ch) << endl;

    return 0;
}
```

知识点总结

- 1.自动类型推导
- 2.右值引用
- 3.列表初始化
- 4.For each
- 5.Lambda表达式
- 6.智能指针
- 7.可变参数模板
- 8.默认成员函数控制
- 9.新增容器

作业

使用智能指针进行字符串 “hello world” 的右移位。