**C++11 的新特性总结**

**1 变量和基本类型**

**1.1 long long 类型**

扩展精度浮点数，10位有效数字

**1.2 列表初始化**

初始化的几种不同形式，其中用**花括号**来初始化变量称为列表初始化；

比如：

**int i = 0;**

**int i = {0};**

**int i{0};**

**int i(0);**

需要注意的是，当用于内置类型的变量时，这种初始化形式有一个重要的特点：如果我们使用初始化且初始值存在丢失信息的风险，则编译器报错；

例如：

**long double ld = 3.1414141414;**

**int a{ld}, b = {ld}; //报错**

**int c(ld), d = ld; //正确**

**cout << "a:" << a << "b:" << b << "c:" << c << "d" << d << endl;**

运行的时候，a,b则会提示报错信息：error: type 'long double' cannot be narrowed to 'int' in initializer list [-Wc++11-narrowing]，这是因为使用long double的值初始化int变量时可能会丢失数据，所以拒绝a和b的初始化请求；虽然c,d虽然没有报错，但是确实丢失了数据；

[这是为什么？]

**1.3 nullptr 常量**

有几种生成空指针的方法：

**int \*p1 = nullptr; // 等价于int \*p1 = 0;**

**int \*p2 = 0;**

**int \*p3 = NULL; // 等价于int \*p3 = 0;**

在新标准下，建议尽量使用新标准nullptr，nullptr是一种特殊类型的字面值，它可以被转换成任意其它的指针类型，虽然其它的方式也是可以的；

**1.4 constexpr 变量**

将变量声明为constexpr类型以便由编译器来验证变量的值是否是一个常量表达式；

声明为constexpr的变量一定是一个常量，而且必须用常量表达式来初始化，比如说下面的情况则是不正确的：

**int t = 10;**

**constexpr int q = t + 20;**

**cout << "q" << q << endl;**

需要将t声明为 const 才是正确的；

一般来说，如果你认定变量是一个常量表达式，那就把它声明为constexpr类型；

**1.5 类型别名声明**

使用类型别名可以使复杂的类型名字变得更简单明了，易于理解和使用；

现在有两种方法可以用来定义类型别名，一种是 typedef ，另一种则是新标准中的 using；

**#include <iostream>**

**using namespace std;**

**int add(int val) {**

**return 10 + val;**

**}**

**int main() {**

**typedef double dnum;**

**// 字符指针**

**typedef char \*pstring;**

**// 函数**

**// 返回值类型为int，参数类型为int的函数**

**typedef int func(int);**

**// 函数指针，指向返回值类型为int，参数类型为int的函数**

**typedef int (\*pfunc)(int);**

**// 函数引用，指向返回值类型为int，参数类型为int的函数**

**typedef int (&tfunc)(int);**

**pfunc pfunc\_add = nullptr;**

**pfunc\_add = add;**

**cout << "函数指针，result is " << pfunc\_add(10) << endl;**

**tfunc tfunc\_add = add;**

**cout << "函数引用，result is " << tfunc\_add(10) << endl;**

**func &func\_add = add; //这里使用指针或者引用都可以**

**cout << "函数，result is " << func\_add(10) << endl;**

**// 数组**

**// 元素类型为int，个数为10的数组**

**typedef int arr[10];**

**// 数组指针，指向元素类型为int，个数为10的数组**

**typedef int (\*parr)[10];**

**// 数组引用，绑定到元素类型为int，个数为10的数组**

**typedef int (&tparr)[10];**

**using dnum2 = double;**

**using pstring2 = char\*;**

**using func2 = int(int);**

**using pfunc2 = int(\*)(int);**

**using arr2 = int[10];**

**using parr2 = int(\*)[10];**

**using tparr2 = int(&)[10];**

**std::cout << "Hello, World!" << std::endl;**

**return 0;**

**}**

但是需要注意的是，如果某个类型别名指代的是复合类型或者常量，那么就会产生意想不到的后果；

比如说

**typedef char \*pstring;**

**const pstring cstr = 0;**

按照我们正常的理解就是，将char\*替换掉pstring，得到 const char\* cstr；然而事实是pstring是一个字符指针，其基本数据类型是个指针，此时用此字符指针去声明cstr，得到的是一个常量的字符指针，但是按照本意是指向char的常量指针，其基本类型是char，也就是说两者修饰的东西是不一样的！

但是如果是引用变量则没有关系。

参考文章

[C++ - 类型别名以及类型的自动推断](http://cuckootan.me/2016/05/06/C/C++/C++%20-%20%E7%B1%BB%E5%9E%8B%E5%88%AB%E5%90%8D%E4%BB%A5%E5%8F%8A%E7%B1%BB%E5%9E%8B%E7%9A%84%E8%87%AA%E5%8A%A8%E6%8E%A8%E6%96%AD/)

[C/C++ 函数指针的用法](http://www.cnblogs.com/TenosDoIt/p/3164081.html)

**1.6 auto 类型指示符**

auto让编译器通过初始值来推算变量的类型，所以，其定义的变量必须要有初始值；

使用auto也能在一条语句中声明多个变量；因为一条声明语句只能有一个基本数据类型，所以该语句中所有的变量的初始基本数据类型都必须是一样的；

* **顶层const**：指针本身是一个常量；
* **底层const**：指针指向的对象是一个常量；

**int main() {**

**int i = 0;**

**const int ci = i;**

**auto b = &i; // b是一个整形指针（整数的地址就是指向整数的指针）**

**auto c = &ci; // c是一个指向整数常量的指针（对常量对象取地址是一种底层const）**

**return 0;**

**}**

**1.7 decltype 类型指示符**

作用：选择并返回操作数的数据类型；

**1.7.1 decltype 和 auto 的区别**

* 处理顶层const和引用的方式

**const int ci = 0, &cj = ci;**

**decltype(ci) x = 0;**

**decltype(cj) y = x;**

**decltype(cj) z; //报错，因为cj是一个引用，因此作为引用的 z 必须要进行初始化**

引用从来都是作为其所指对象的同义词出现，也就是根据其所指的对象决定，只有在decltype处是个例外；

* decltype的结果类型与表达式形式密切相关

**int i = 0;**

**decltype((i)) a; //报错，因为a类型为 int&，必须进行初始化**

**decltype(i) b; //正确**

需要注意的是，decltype((variable))的结果永远是引用，而decltype(variable)结果只有当variable本身就是一个引用时才是引用，其实就是根据它的类型决定；

**1.8 类内初始化**

建议初始化每一个内置类型的变量，为了保证初始化后程序安全

**2 字符串、向量和数组**

**2.1 使用 auto 或 decltype 缩写类型**

注意的是，如果表达式中已经有了size函数就不要再使用int了，因为size函数返回的是unsigned，如果其和有符号数进行比较的时候，有符号数会自动的转换成一个比较大的无符号数。

**2.2 范围 for 语句**

**string str("hello world");**

**for (auto c : str) {**

**cout << c;**

**}**

**2.3 定义vector对象的vector（向量的向量）**

编译器根据模版vector生成了三种不同的类型，分别是：

vector<int>，vector<vector<int>>, vector<classname>

新的版本已经不需要再添加一个空格

vector<vector<int> >

**2.4 vector对象的列表初始化**

放在花括号里

**2.5 容器的cbegin 和 cend 函数**

**2.6 标准库begin 和 end 函数**

**2.7 使用auto和decltype简化声明**

**3 表达式**

**3.1 除法的舍入规则**

新标准中，一律向0取整（直接切除小数部分）

**double a = 12/5;**

**cout << a << endl;**

输出结果为2，删掉了小数部分；

**3.2 用大括号包围的值列表赋值**

**3.3 将sizeof用于类成员**

使用作用域运算符来获取类成员的大小是许可的；

* 对数组执行sizeof运算得到整个数组所占空间的大小，等价于对数组中  
  所有的元素各执行一次sizeof运算并将所得结果求和；
* 对string对象或vector对象执行sizeof运算只返回该类型固定部分的大小，不会计算对象中的元素占用多少空间；

**4 语句**

**4.1 范围for语句**

**5 函数**

**5.1 标准库 initializer\_list类**

可以处理不同数量实参的函数，**前提是实参类型要相同**；

其提供的操作包括：

* 默认初始化
* 列表初始化
* 拷贝对象
* size
* begin
* end

**5.2 列表初始化返回值**

函数可以返回花括号包围的值的列表，如果列表为空，临时量执行初始化，否则，返回的值由函数的返回类型决定；

**vector<string> getAddress(int num) {**

**if (num == 0) {**

**return {};**

**}**

**return {"address1", "address2", "address3"};**

**}**

**5.3 定义尾置返回类型**

任何函数的定义都能够使用尾置来返回，最适用于返回类型比较复杂的情况；

比如：

// 返回类型为指针，该指针指向含有10个整数的数组

**auto func(int i) -> int(\*)[10] {**

**int arr[10] = {0};**

**return &arr;**

**}**

**5.4 使用decltype简化返回类型定义**

**int odd[] = {1, 3, 5, 7, 9};**

**int even[] = {2, 4, 6, 8, 10};**

**// 返回一个指针，指向数组**

**decltype(odd) \*getArr(int i) {**

**return (i % 2) ? &odd : &even;**

**}**

decltype并不会因为获取的是数组，则返回的是指针，还是需要在函数名前面加上 \*；

**5.5 constexpr函数**

是指能用于常量表达式的函数；

需要遵从几项约定：

* 函数的返回类型以及所有形参的类型都是字面值类型（只能用它的值来称呼它）；
* 函数体中必须有且只有一条return语句（C++14不再做要求）；
* 必须非virtual

**constexpr int func2() {**

**return 10;**

**}**

**int main() {**

**int arr[func2() + 10] = {0};**

**return 0;**

**}**

**6 类**

**6.1 使用=default生成默认构造函数**

如果实现了默认的构造函数，编译器则不会自动生成默认版本；可以通过使用关键字 default 来控制默认构造函数的生成，显示的指示编译器生成该函数的默认版本；

**class MyClass{**

**public:**

**MyClass()=default; //同时提供默认版本和带参版本**

**MyClass(int i):data(i){}**

**private:**

**int data;**

**};**

比如说如果想要禁止使用拷贝构造函数，则使用关键字 delete；

**class MyClass**

**{**

**public:**

**MyClass()=default;**

**MyClass(const MyClass& )=delete;**

**};**

**int main() {**

**MyClass my;**

**MyClass my2(my); //报错**

**return 0;**

**}**

但需要注意的是，析构函数是不允许使用delete的，否则无法删除；

**6.2 类对象成员的类内初始化**

当提供一个类内初始值时，必须以符号＝或者花括号表示

**6.3 委托构造函数**

一个委托构造函数使用它所属类的其它构造函数执行它自己的初始化过程，或者说它把它自己的一些（或者全部）指责**委托**给了其它构造函数；

**class MyClass**

**{**

**public:**

**MyClass() : MyClass(10) {}**

**MyClass(int d) : data(d){}**

**inline int getData() {return data;}**

**private:**

**int data;**

**};**

**int main() {**

**MyClass my;**

**cout << my.getData() << endl;**

**return 0;**

**}**

**6.4 constexpr构造函数**

如果想要使得函数拥有编译时计算的能力，则使用关键字 constexpr

同样的编译时使用对象：

**class Square {**

**public:**

**constexpr Square(int e) : edge(e){};**

**constexpr int getArea() {return edge \* edge;}**

**private:**

**int edge;**

**};**

**int main() {**

**Square s(10);**

**cout << s.getArea() << endl;**

**return 0;**

**}**

如果成员函数标记为 constexpr，则默认其是内联函数，如果变量声明为  
constexpr，则默认其是 const；

参考文章

[constexpr指定符(C++11 起)](http://zh.cppreference.com/w/cpp/language/constexpr)

**7 IO库**

**7.1 用string对象处理文件名**

新版本中增加了使用库类型string对象作为文件名，之前只能使用C风格的字符数组；

ifstream infile("/hello.sh");

**8 顺序容器**

补充：

顺序容器的类型：

| **容器** | **描述** |
| --- | --- |
| vector | **可变大小数组**。支持快速随机访问。在尾部之外的位置插入或者删除元素时可能很慢 |
| deque | **双端队列**。支持快速随机访问。在头尾位置插入／删除速度很快 |
| list | **双向链表**。只支持双向双向顺序访问。在list中任何位置进行插入／删除操作 |
| forward\_list | **单向链表**。只支持单向顺序访问。在链表任何位置进行插入／删除操作都很快 |
| array | 固定大小数组。支持快速随即访问。不能添加或删除元素 |
| string | 与vector相似的容器，但专门用于保存字符（字符数组，封装了char而已）。随机访问速度快，在尾部插入、删除速度快 |

总而言之，数组随机访问速度快，链表插入删除速度快；

**8.1 array和forward\_list容器**

* **array**

**#include <string>**

**#include <iterator>**

**#include <iostream>**

**#include <algorithm>**

**#include <array>**

**int main()**

**{**

**// 使用聚合初始化来构造**

**std::array<int, 3> a1{ {1,2,3} }; // C++11中需要使用双重花括号（而14中不需要）**

**std::array<int, 3> a2 = {1, 2, 3}; // 不再需要等号了**

**std::array<std::string, 2> a3 = { {std::string("a"), "b"} };**

**// 支持基本的容器操作**

**std::sort(a1.begin(), a1.end());**

**std::reverse\_copy(a2.begin(), a2.end(), std::ostream\_iterator<int>(std::cout, " "));**

**// 支持范围for**

**for(auto& s: a3)**

**std::cout << s << ' ';**

**}**

* forward\_list

和 list 的使用类似，就不贴代码了；

template < class T, class Alloc = allocator<T> > class forward\_list;

Alloc，容器内部用来管理内存分配以及释放的内存分配器的类型，默认使用的是 std::allocator<T>，；

参考文章

[std::array](http://zh.cppreference.com/w/cpp/container/array)

[std::forward\_list](http://zh.cppreference.com/w/cpp/container/forward_list)

**8.2 容器的cbegin和cend函数**

返回的是const的迭代器，当不需要写访问时，应使用cbegin和cend；

**8.3 容器的列表初始化**

**8.4 容器的非成员函数swap**

除了 array 外，swap不对任何元素进行拷贝、删除或者插入操作，因此可以保证常数时间内完成；swap 只是交换了容器内部数据结构，不会交换元素，因此，除string 外，指向容器的迭代器、引用和指针在 swap 操作后都不会失效；

但是，array会真正的交换它们的元素。

**8.5 容器insert成员的返回类型**

在新标准下，接受元素个数或范围的insert版本返回的是－指向第一个新加入元素的迭代器；

如果范围为空，不插入任何元素，insert 操作会将第一个参数返回；

**int main() {**

**list<string> lst;**

**auto itr = lst.begin();**

**string word;**

**while (cin >> word) {**

**itr = lst.insert(itr, word); //等价于push\_front**

**}**

**return 0;**

**}**

**8.6 容器的emplace成员**

当调用 push 或 insert 成员函数时，我们将元素类型的对象传递给它们，这些对象被拷贝到容器里面，但是，当我们调用一个 emplace 成员函数时，则是将函数传递给元素类型的构造函数，会使用这些参数在容器管理的内存中直接构造元素；包括三个函数：emplace\_front，emplace，emplace\_back，分别对应着 push\_front，insert，push\_back 为头部，指定位置，尾部；

**class MyClass**

**{**

**public:**

**MyClass(int d) : data(d){}**

**inline int getData() {return data;}**

**private:**

**int data;**

**};**

**int main() {**

**vector<MyClass> mys;**

**mys.emplace\_back(10);**

**mys.push\_back(MyClass(20));**

**for (auto itr = mys.begin(); itr != mys.end(); itr++) {**

**cout << (\*itr).getData() << endl;**

**}**

**return 0;**

**}**

**8.7 shrink\_to\_fit**

调用该函数要求 deque，vector，string 退回不需要的内存空间；

**int main() {**

**vector<int> nums;**

**nums.push\_back(10);**

**nums.push\_back(10);**

**nums.push\_back(10);**

**nums.push\_back(10);**

**nums.push\_back(10);**

**cout << nums.capacity() << endl;**

**nums.shrink\_to\_fit();**

**cout << nums.capacity() << endl;**

**return 0;**

**}**

返回结果为8，5

区别：

* reserve：预分配存储区大小，即capacity的值
* resize：容器大小，即size的值

**8.9 string的数值转换函数**

新标准中，引入多个函数实现数值数据和标准库string之间的转换：

| **函数** | **描述** |
| --- | --- |
| to\_string(val) | 返回任意算术类型val的字符串 |
| stoi(s, p, b) | int类型 |
| stol(s, p, b) | long类型 |
| stoul(s, p, b) | unsigned long类型 |
| stoll(s, p, b) | long long类型 |
| stoull(s, p, b) | unsigned long long类型 |
| stof(s, p, b) | float类型 |
| stod(s, p, b) | double类型 |
| stold(s, p, b) | long double类型 |

**9 泛型算法**

**9.1 lambda表达式**

一个lambda具有一个返回类型、一个参数列表和一个函数体；

[capture list](parameter list) -> return type { function body }

**9.2 lambda表达式中的尾置返回类型**

**9.3 标准库bind函数**

auto newCallable = bind(callable, arg\_list);

例子（绑定类成员函数）

**#include <iostream>**

**#include <functional>**

**using namespace std;**

**using namespace std::placeholders;**

**class MyClass**

**{**

**public:**

**void fun1(void)**

**{**

**cout << "void fun1(void)" << endl;**

**}**

**int fun2(int i)**

**{**

**cout << "int fun2(int i)" << " i = " << i << endl;**

**return i;**

**}**

**};**

**int main()**

**{**

**MyClass my;**

**//使用类对象绑定**

**auto fun1 = bind(&MyClass::fun1, my);**

**fun1();**

**MyClass \*p;**

**//使用类指针绑定，-1为占位符**

**auto fun2 = bind(&MyClass::fun2, p, \_1);**

**int i = fun2(1);**

**cout << "i = " << i << endl;**

**cin.get();**

**return 0;**

**}**

**10. 关联容器**

关联器类型

* map 关联数组，保存关键字－值对
* set 关键字即值，即只保存关键字的容器
* multimap 关键字可重复出现的map
* multiset 关键字可重复出现的set
* unordered\_map 用哈希函数组织的map
* unordered\_set 用哈希函数组织的set
* unordered\_multimap 用哈希函数组织的map，关键字可重复
* unordered\_multiset 用哈希函数组织的set，关键字可重复

**10.1 关联容器的列表初始化**

和顺序容器差不多

**10.2 列表初始化pair的返回类型**

**10.3 pair的列表初始化**

**10.4 无序容器**

包括4个无序关联容器（使用哈希函数和关键字类型的 == 运算符）

* unordered\_map
* unordered\_set
* unordered\_multiset
* unordered\_multimap

**11 动态内存**

已总结，略

**11.1 智能指针**

**11.2 shared\_ptr类**

**11.3 动态分配对象的列表初始化**

**11.4 auto和动态分配**

**11.5 unique\_ptr类**

**11.6 weak\_ptr类**

**11.7 范围for语句不能应用于动态分配数组**

因为分配的内存并不是一个数组类型

**11.8 动态分配数组的列表初始化**

可以对数组中的元素进行值初始化，方法是在大小之后跟一对括号；

**int \*pia = new int[10]; //10个未初始化的int**

**int \*pia2 = new int[10](); //10个初始化为0的int**

**int \*pia3 = new int[10]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};**

**delete [] pia; //释放**

**delete [] pia2;**

**delete [] pia3;**

**11.9 auto不能用于分配数组**

虽然我们用空括号对数组中元素进行值初始化，但不能在括号中给出初始化器，这意味着不能用auto分配数组；

**11.10 allocator::construct可使用任意构造函数**

allocator分配的内存是未构造的，我们按需要在此内存中构造对象。在新标准库中，construct成员函数接受一个指针和零个或多个额外参数，在给定位置构造一个元素。

**int main() {**

**allocator<string> alloc;**

**string\* str = alloc.allocate(4); //分配10个未初始化的string**

**auto itr = str;**

**alloc.construct(itr, "chengdu"); //chengdu**

**cout << str[0] << endl;**

**itr++;**

**alloc.construct(itr, "beijing"); //beijing**

**cout << str[1] << endl;**

**itr++;**

**alloc.construct(itr, 10, 'c'); //10个c组成的字符串**

**cout << str[2] << endl;**

**itr++;**

**alloc.construct(itr); //空字符串**

**cout << str[3] << endl;**

**for (int i = 0; i < 4; i++) {**

**cout << str[i] << endl;**

**}**

**// 销毁**

**while (itr != str) {**

**alloc.destroy(itr--);**

**}**

**return 0;**

**}**

为了使用allocate返回的内存，我们必须用construct构造对象，使用未构造的内存，其行为是未定义的；我们只有对真正构造了的元素进行destroy操作；

**12 拷贝控制**

**12.1 将=default用于拷贝控制类对对象**

可以类内或者类外修饰成员函数，如果是类内，合成的函数将隐式地声明未内联的，如果不希望这样，则在类外声明；

我们只能对具有合成版本的成员函数使用=default（即，默认构造函数或拷贝控制成员）

**12.2 使用=delete用于拷贝控制成员**

对于析构函数已删除的类型，不能定义该类型的变量或释放指向该类型动态分配的指针；

**12.3 用移动类对象代替拷贝类对象**

为了解决之前存在的性能问题，避免不必要的内存拷贝，新标准中有两种方法来替代拷贝函数：移动函数和 move

移动构造函数

**A(A && h) : a(h.a){**

**h.a = nullptr;**

**}**

**12.4 右值引用**

右值引用就是必须绑定到右值的引用，通过 && 来获得右值引用；

**12.4.1 区别左值和右值**

* 左值：在赋值号左边，可以被赋值的值，可以取地址；
* 右值：在赋值号右边，取出值赋给其他变量的值；
* 左值引用：type & 引用名 = 左值表达式
* 右值引用：type && 引用名 = 右值表达式

**int a = 0; //2**

**int b = a; //1**

**一个对象被用作右值时，使用的是它的内容（值），比如1中的 a ，被当作左值时，使用的是它的地址，比如2中的 a，常规左值；**

**int main() {**

**int i = 1; //i为常规左值**

**int &r = i; //正确：r绑定到i上，r是一个引用**

**int &&rr = i; //错误：不能将一个右值引用绑定到左值i上**

**int &r2 = i \* 1; //错误：等号右边是一个右值，但左值引用只能绑定到左值上**

**int &&rr2 = i \* 1; //正确：右值引用绑定到右值上**

**const int &r3 = i \* 1; //正确：可以将一个const的左值引用绑定到右值上**

**return 0;**

**}**

* 返回左值引用的函数、赋值、下标、解引用和前置递增／递减运算符，都是返回左值的表达式的例子，我们可以将一个左值引用绑定到这类表达式的结果上；
* 返回非引用类型的函数，连同算术、关系、位以及后置递增／递减运算符，都生成右值；我们不能将左值引用绑定到这类表达式上，但是我们可以将一个const的左值引用或者一个右值引用绑定到这类表达式之上；

**12.5 标准库move函数**

我们可以销毁一个移后源对象，也可以赋予它新值，但不能使用一个移后源对象的值

**// 移动构造函数，参数 "arg.member" 是左值**

**A(A&& arg) : member(std::move(arg.member))**

**{**

**}**

**// 移动赋值函数**

**A& operator=(A&& other) {**

**member = std::move(other.member);**

**return \*this;**

**}**

**12.6 移动构造函数和移动赋值**

**12.7 移动构造函数通常是noexcept**

如果需要通知标准库这是一个不会抛出异常的移动操作，可以在函数后面指明 noexcept，如果在一个构造函数中，noexcept 出现在参数列表和初始化列表开始的冒号之间；

不抛出异常的移动构造函数和移动赋值运算符必须标记为noexcept

**12.8 移动迭代器**

新标准中可以通过调用标准库的 make\_move\_iterator 函数将一个普通迭代器转换为一个**移动迭代器**，这样可以避免拷贝操作带来的性能问题；

**#include <iostream> // std::cout**

**#include <iterator> // std::make\_move\_iterator**

**#include <vector> // std::vector**

**#include <string> // std::string**

**#include <algorithm> // std::copy**

**int main() {**

**std::vector<std::string> foo(3);**

**std::vector<std::string> bar{ "one","two","three" };**

**std::copy(make\_move\_iterator(bar.begin()),**

**make\_move\_iterator(bar.end()),**

**foo.begin());**

**// bar now contains unspecified values; clear it:**

**bar.clear();**

**std::cout << "foo:";**

**for (std::string& x : foo) std::cout << ' ' << x;**

**std::cout << '\n';**

**return 0;**

**}**

**12.9 引用限定成员函数**

引用限定符可以是 & 或者 &&，分别指出 this 可以指向一个左值或者右值，引用限定符只能用于成员函数，而且必须同时出现在函数的声明和定义中；

可以在参数列表之后使用引用限定符来指定this对象的左值与右值属性;

* 若引用限定符为&,则表明this对象指向着左值对象;
* 若引用限定符为&&,则表明this对象指向着右值对象；

参考文章

[C++-引用限定符](http://pp-qq.github.io/2016/04/16/6.3-%E5%8F%B3%E5%80%BC%E5%BC%95%E7%94%A8%E5%92%8C%E6%88%90%E5%91%98%E5%87%BD%E6%95%B0/)

**13 重载运算与类型转换**

**13.1 function类模版**

function模版提供一种通用、多态的函数封装。其实例可以对任何可调用的目标进行存储、赋值和调用操作，这些目标包括函数、lambda表达式、绑定表达式以及其它函数对象等；

**#include <functional>**

**#include <iostream>**

**struct Foo {**

**Foo(int num) : num\_(num) {}**

**void print\_add(int i) const { std::cout << num\_+i << '\n'; }**

**int num\_;**

**};**

**void print\_num(int i)**

**{**

**std::cout << i << '\n';**

**}**

**struct PrintNum {**

**void operator()(int i) const**

**{**

**std::cout << i << '\n';**

**}**

**};**

**int main()**

**{**

**// 保存自由函数**

**std::function<void(int)> f\_display = print\_num;**

**f\_display(-9);**

**// 保存 lambda 表达式**

**std::function<void()> f\_display\_42 = []() { print\_num(42); };**

**f\_display\_42();**

**// 保存 std::bind 的结果**

**std::function<void()> f\_display\_31337 = std::bind(print\_num, 31337);**

**f\_display\_31337();**

**// 保存成员函数**

**std::function<void(const Foo&, int)> f\_add\_display = &Foo::print\_add;**

**Foo foo(314159);**

**f\_add\_display(foo, 1);**

**// 保存成员函数和对象**

**using std::placeholders::\_1;**

**std::function<void(int)> f\_add\_display2= std::bind( &Foo::print\_add, foo, \_1 );**

**f\_add\_display2(2);**

**// 保存成员函数和对象指针**

**std::function<void(int)> f\_add\_display3= std::bind( &Foo::print\_add, &foo, \_1 );**

**f\_add\_display3(3);**

**// 保存函数对象**

**std::function<void(int)> f\_display\_obj = PrintNum();**

**f\_display\_obj(18);**

**}**

**13.2 explicit类型转换运算符**

用来防止由构造函数定义的隐式转换；

看看构造函数的隐式转换，如果类的构造函数有一个参数，那么在编译的时候就会有一个缺省的转换操作，会将其参数转化成类的对象；

**class MyClass**

**{**

**public:**

**MyClass( int num );**

**}**

**// 编译器会将10转换成MyClass对象**

**MyClass obj = 10;**

避免这种自动转换的操作则需要 explicit：

**class MyClass**

**{**

**public:**

**explicit MyClass( int num );**

**}**

**MyClass obj = 10; //err,can't non-explict convert**

其只能用在类内部的构造函数声明上，不能用在类外部的函数定义上；

被声明为explicit的构造函数通常比其non-explicit兄弟更受欢迎。因为它们禁止编译器执行非预期（往往也不被期望）的类型转换。除非我有一个好理由允许构造函数被用于隐式类型转换，否则我会把它声明为explicit。---Effective c++

**14 面向对象程序设计**

**14.1 虚函数的override和final指示符**

override 可以帮助程序员的意图更加的清晰的同时让编译器可以为我们发现一些错误。其只能用于覆盖基类的**虚函数**；

final 使得任何尝试覆盖该函数的操作都将引发错误，并不特指虚函数；

均出现在形参列表（包括任何const或者引用限定符）以及尾置返回类型之后

**14.2 删除的拷贝控制和继承**

如果函数在基类中被定义为是删除的，则派生类对应的也是删除的；

**14.3 继承的构造函数**

委派和继承构造函数是由C++11引进为了减少构造函数重复代码而开发的两种不同的特性；

a. 通过特殊的初始化列表语法，**委派构造函数**允许类的一个构造函数调用其它的构造函数

**X::X(const string& name) : name\_(name) {**

**...**

**}**

**X::X() : X("") { }**

b. 继承构造函数允许派生类直接调用基类的构造函数，一如继承基类的其它成员函数，而无需重新声明，当基类拥有多个构造函数时这一功能尤其有用：

**class Base {**

**public:**

**Base();**

**Base(int n);**

**Base(const string& s);**

**...**

**};**

**class Derived : public Base {**

**public:**

**using Base::Base; // Base's constructors are redeclared here.**

**};**

using声明语句将令编译器产生代码，对于基类的每个构造函数，编译器都在派生类中生成一个形参列表完全相同的构造函数；如果派生类含有自己的数据成员，则这些成员将会被默认初始化；

**15 模版与泛型编程**

**15.1 声明模版类型形参为友元**

在新标准中，我们可以将模板类型参数声明为友元：

**template <typename Type> class Bar {**

**friend Type; //将访问权限授予用来实例化Bar的类型**

**};**

此处我们将用来实例化Bar的类型声明为友元。因此，对于某个类型名Foo，Foo 将成 Bar<Foo> 的友元。

**15.2 模版类型别名**

新标准中允许我们为模版定义一个类型别名：

**template<typename T> using twin = pair<T, T>;**

**twin<string> authors;**

其中，twin是一个 pair<T,T>；

**15.3 模版函数的默认模版参数**

**template <typename T, typename F = less<T>>**

**int compare(const T &v1, const T &v2, F f= F()) {**

**if (f(v1, v2)) return -1;**

**if (f(v2, v1)) return 1;**

**return 0;**

**}**

就像之前能为函数参数提供**默认实参**一样，compare有一个默认模版实参 less<T>和一个默认函数实参 F()

参考文章

[C++类模板与友元的关系](http://www.jianshu.com/p/6613dbd0827d)

**15.4 实例化的显示控制**

显示实例化：在不发生函数调用的时候将**函数模版实例化**或者在不适用类模版的时候将**类模版实例化**，**这可以避免实例化相同的模版所带来的额外开销**；

* 函数模版实例化

template函数返回类型 函数模板名<实际类型列表>（函数参数列表）

* 类模版实例化

template class 类模板名<实际类型列表>

参考文章

[C++模板之隐式实例化、显示实例化、显示调用、隐式调用和模板特化详解](http://www.voidcn.com/blog/k346k346/article/p-4834294.html)

**15.5 模版函数与尾置返回类型**

例子：

//尾置返回允许我们在参数列表之后声明返回类型

**template <typename T>**

**auto func(T beg, T end) -> decltype(\*beg) {**

**return \*beg; //返回序列中一个元素的引用**

**}**

**15.6 引用折叠规则**

规则如下：

* 所有右值引用折叠到右值引用上仍然是一个右值引用，比如（A&& && 变成 A&&）
* 所有的其它引用类型之间的折叠都将会变成左值引用，比如（A& &变成A&，A& && 变成 A&，A&& & 变成 A&）

引用折叠只能应用于间接创建的引用的引用，如类型别名或者模版参数；

**15.7 用static\_cast将左值转换为右值**

可以通过类型转换 static\_cast<Type&&> 来将返回右值引用；

**int s=101;**

**int&& foo(){**

**return static\_cast<int&&>(s);**

**} //返回值为右值引用**

**int main() {**

**int i=foo(); //右值引用作为右值，在赋值运算符的右侧**

**int&& j=foo(); //j是具名引用。因此运算符右侧的右值引用作为左值**

**int\* p=&j; //取得j的内存地址**

**}**

参考文章

[右值引用](https://zh.wikipedia.org/wiki/%E5%8F%B3%E5%80%BC%E5%BC%95%E7%94%A8)

**15.8 标准库forward函数**

右值引用至少解决了两个问题：

1. 实现 move 语义
2. 完美转发

**完美转发**

有的时候，我们需要将一个函数中某一组参数原封不动的传递给另一个函数，这里不仅仅需要**参数的值**不变，而且需要**参数的类型属性（左值／右值）**保持不变－完美转发；

**使用forward**

* 原型：

**template< class T >**

**constexpr T&& forward( typename std::remove\_reference<T>::type& t );**

**template< class T >**

**constexpr T&& forward( typename std::remove\_reference<T>::type&& t );**

* 例子：

**#include <iostream>**

**#include <utility>**

**#include <memory>**

**class A {**

**public:**

**A(int && n) { std::cout << "rvalue constructor -> n=" << n << std::endl;}**

**A(int& n) { std::cout << "lvalue constructor -> n=" << n << std::endl;}**

**};**

**template<class T, class U>**

**std::unique\_ptr<T> make\_unique1(U&& u) {**

**return std::unique\_ptr<T>(new T(std::forward<U>(u)));**

**//return std::unique\_ptr<T>(new T(u));**

**}**

**int main() {**

**int i = 24;**

**auto p1 = make\_unique1<A>(666); // rvalue forwarding**

**auto p2 = make\_unique1<A>(i); // lvalue forwarding**

**auto p3 = make\_unique1<A>(std::move(i)); // rvalue forwarding**

**return 0;**

**}**

但是如果我们没有使用 forward 函数，结果则全部都调用的是 lvalue constructor；

参考文章

[Modern C++ | 移动语义与完美转发 | Universal Reference](http://www.sczyh30.com/posts/C-C/cpp-move-semantic/)

[移动语义（move semantic）和完美转发（perfect forward）](https://codinfox.github.io/dev/2014/06/03/move-semantic-perfect-forward/)

**15.9 可变参数模版与转发**

右值引用＋完美转发＋可变参数模版实现下面这个函数；

// args为右值引用，decltype用于返回值

**template<class Function, class... Args>**

**auto FuncWrapper(Function && f, Args && ... args) -> decltype(f(std::forward<Args>(args)...))**

**{**

**return f(std::forward<Args>(args)...);**

**}**

**void test0()**

**{**

**cout <<"void"<< endl;**

**}**

**int test1()**

**{**

**return 1;**

**}**

**int test2(int x)**

**{**

**return x;**

**}**

**string test3(string s1, string s2)**

**{**

**return s1 + s2;**

**}**

**int main() {**

**int num = 10;**

**int && nnum = num + 10;**

**int & nnum2 = num;**

**FuncWrapper(test0); // 没有返回值，打印1**

**cout << FuncWrapper(test1) << endl; // 没有参数，有返回值，返回1**

**cout << FuncWrapper(test2, 1) << endl; // 有参数，有返回值，返回1**

**cout << FuncWrapper(test2, std::move(num)) << endl; // 有参数，有返回值，返回左值10**

**cout << FuncWrapper(test2, std::move(nnum2)) << endl; // 有参数，有返回值，返回左值引用10**

**cout << FuncWrapper(test2, nnum) << endl; // 有参数，有返回值，返回右值引用20**

**cout << FuncWrapper(test3, "aa", "bb") << endl; // 有参数，有返回值，返回"aabb"**

**return 0;**

**}**

返回值分别为：

aabb

void

1

1

10

10

20

参考文章

[forward和完美转发](http://book.2cto.com/201507/53149.html)