# C++

|  |  |  |  |
| --- | --- | --- | --- |
| 日期 | 版本 | 修订内容 | 修改人 |
| 2018-09-30 | V0.1 | 初稿 | 朱灿基 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

# 一、概述

“凡事豫则立，不豫则废”

C++11相比C++0x有较大的改动，添加了很多新特性。

# 二、C++基础

## 2.1、内存

### 2.2.1、new/delete

free/delete怎么知道有多少内存要释放？[[2]、[3]](#_参考文献)

实际上在申请内存的时候，申请到的地址会比你实际的地址大一点点，它包含了一个存有申请空间大小的结构体。

比如你申请了20byte的空间，实际上系统申请了48byte的空间。

|  |
| --- |
| 16-byte header containing size, special marker, checksum, pointers to next/previous block and so on.  32 bytes data area (your 20 bytes padded out to a multiple of 16)) |

The address then given to you is the address of the data area. Then, when you free the block, free will simply take the address you give it and, assuming you haven't stuffed up that address or the memory around it, check the accounting information immediately before it. Graphically, that would be along the lines of:

|  |
| --- |
| \_\_\_\_ The allocated block \_\_\_\_  / \  +--------+--------------------+  | Header | Your data area ... |  +--------+--------------------+  ^  |  +-- The address you are given |

Keep in mind the size of the header and the padding are totally implementation defined (actually, the entire thing is implementation-defined (a) but the in-line accounting option is a common one).

The checksums and special markers that exist in the accounting information are often the cause of errors like "Memory arena corrupted" or "Double free" if you overwrite them or free them twice.

The padding (to make allocation more efficient) is why you can sometimes write a little bit beyond the end of your requested space without causing problems (still, don't do that, it's undefined behaviour and, just because it works sometimes, doesn't mean it's okay to do it).

这样就可以不需要提供任何其他信息，而正确的释放内存。

### 2.2.2、new

|  |
| --- |
| new int; //创建一个int大小的内存  new int(100); //创建一个int型的内存，并赋值为100  new int[100]; //定义int数组变量，创建100个int型的内存 |

例子：

|  |
| --- |
| int\* p = new int;  int\* p = new int(100);  int\* p[] = new int[100]; |

## 2.2、指针

### 2.2.1、指针变量的定义

在定义指针变量时必须指定基类型。为什么要指定基类型呢？

因为不同类型的数据在内存中占用的字节和存放方式是不一样的（例如整型占8个字节，字符型占1个字节）。如果想通过指针引用一个变量，只知道一个地址是不够的，因为无法判断从地址处取多少个字节的数据。必须知道数据的类型，才能按照存储单元的长度以及数据的存储形式正确的取出该数据。

指针的加减运算是改变多少个数据类型的长度。

比如，“使指针值加1”，对于整型指针（int \*）这个1意味着使地址值加8个字节。（假设是64位机器）

通过指针引用数组。当指针指向数组元素的时候，指针变量p指向数组元素a[0]，用p+1表示指向下一个元素a[1]。

### 2.2.2、指向函数的指针

函数指针：如果在程序中定义了一个函数，在编译时，编译系统位函数代码分配了一段存储空间，这段存储空间的起始地址（又称为入口地址）称为这个函数的指针。

可以定义一个指向函数的指针变量，用来存放某一函数的起始地址，这就意味着此指针变量指向该函数。例如

|  |
| --- |
| int (\* p)(int , int); |

定义p是一个指向函数的指针变量，它可以指向函数的类型位整型且有两个整型参数的函数。p的类型用int (\*)(int , int)表示。

用函数指针变量调用函数

|  |
| --- |
| int max(int, int);  int (\*p)(int, int);//定义指向函数的指针变量  **p = max;**//使指针变量指向该函数  int a, b, c;  **c = (\*p)(a, b);**//用函数指针变量调用函数，用(\*p)代替函数名。 |

### 2.2.3、指针数组

一个数组，若其元素均为指针类型数据，称之为指针数组。也就是说，指针数组中的每一个元素都存放一个地址，相当于一个指针变量。

|  |
| --- |
| int \* p[4]; |

由于[]比\*优先级高，因此p先与[4]结合，形参p[4]形式，这显然是数组形式，表示p数组有4个元素。然后再与p前面的”\*”结合，“\*”表示此数组是指针类型的，每个数组元素都可指向一个整形变量。

指针数组比较适合用来指向若干个字符串，使字符串处理更加方便灵活。

指向指针数据的指针：指向指针数据的指针变量，简称为指向指针的指针。

## 2.3、值传递和引用传递

当类类型作为函数参数进行值传递和引用传递时，构造函数、析构函数、拷贝构造函数调用次数及资源问题。

### 2.3.1、函数参数中值传递和引用传递

### 2.3.2、函数返回值的值传递和引用传递

### 2.3.3、小结

## 2.4、深复制和浅复制

浅复制：同一类型的对象进行按位拷贝。执行类似于memcpy的按位拷贝。

深复制：当一个对象中有指针或其他资源时，按位拷贝会将指针的地址拷贝到新的对象中，那就是两个对象的指针都指向了同一块堆内存。如果一个对象执行析构的时候，另一个对象的指针就成了“悬挂指针”，因为其不在指向有效的内存了。为了解决这个问题，所以需要深复制的方式，对资源进行重新分配堆内存。使两个对象的指针分别指向不同的堆内存。实现方法就是在拷贝构造函数从堆中再分配内存，将该分配来的内存的指针交给新对象，就使用原来的对象对该内存进行初始化。

简单来说，深复制就是拷贝构造函数中为指针成员分配新的内存再进行内容拷贝。

# 三、深入理解C++11：C++11新特性解析与应用

C++11相对于C++98/03有那些显著的增强呢？

1、通过内存模型、线程、原子操作等来支持本地并行编程（Native Concurrency）。

2、通过统一初始化表达式、auto、declytype、移动语义等来统一对泛型编程的支持。

3、通过constexpr、POD（概念）等更好地支持系统编程。

4、通过内联命名空间、继承构造函数和右值引用等，以更好地支持库的构建。

C++11就像是一门新的语言。如果将C++98标准中的特性和C++11放到一起，C++11就像是个恐怖的“编程语言泛型联盟”。利用它不仅仅可以写出面向对象语言的代码，也可以写出过程式编程语言代码、泛型编程语言代码、函数式编程语言代码、元编程编程语言代码，或者其它。多泛型的支持使得C++11语言的“硬能力”几乎在编程语言中”无出其右“。

C++标准委员会更专注于以下理念：

1、保持语言的稳定性和兼容性（Maintain stability and compatibility）

2、更倾向于使用库而不是扩展语言来实现特性（Prefer libraries to language extensions）。

3、更倾向于通用的而不是特殊的手段来实现特性（Prefer generality to specialization）。

4、专家新手一概支持（Support both experts and novices）。

5、增强类型的安全性（Increase type safety）。

6、增强代码执行性能和操作硬件的能力（Improve performance and ability to work directly）。

7、开发能够改变人们思维方式的特性（Make only changes that change the way people think）。

8、融入编程现实（Fit into the real world）.

设计理念中“更倾向于使用库而不是扩展语言来实现特性“的部分。库有一个很大的优势，就是其改动不需要编译器实现新特性（只要接口保持一致即可），当然，更重要的是库可以用于支持不同领域的编程。这样一来，通常读者不需要非常精通C++就能使用它们。

C++语言可以尽量保持较少的核心语言特性的同时，通过标准库扩大其功能。

下面列出了属于该设计理念下的库特性：

1、算法增强Algorithm improvements

2、容器增强Container improvements

3、分配算符Scoped allocators

4、std::array

5、std::forward\_list

6、无序容器Unordered containers

7、std::tuple

8、类型特性Type traits

9、std::function, std::bind

10、unique\_ptr

11、shared\_ptr

12、weak\_ptr

13、线程Threads

14、互斥Mutex

15、锁Locks

16、条件变量Condition variables

17、时间工具Time utilities

18、std::future,std::promises

19、std::async

20、随机数Random numbers

21、正则表达式regex

## 3.1、通用为本，专用为末

C++11的设计者总是希望从各种方案中抽象出更为通用的方法来构建新的特性。这意味着C++11中的新特性往往具有广泛的可用性，可以为其他已有的，或者新增的语言特性结合起来进行自由组合，或者提升已有特性的通用性。看到这些经过反复斟酌制定的新特性，并体会其“普适”的特性。

### 3.1.1、继承构造函数

类具有派生性，派生类可以自动获得基类的成员变量和接口（虚函数和纯虚函数，这里我们指的都是public派生）。不过基类的非虚函数则无法再被派生类使用了。这条规则对于类中最为特别的构造函数也不例外，如果派生类要使用基类的构造函数，通常需要在构造函数中显示声明。如：

|  |
| --- |
| struct A { A(int i) };  struct B:A { B(int i):A(i) {} }; |

B派生于A，B又在构造函数中调用A的构造函数，从而完成构造函数的”传递”。这在C++代码中非常常见。

但是，有的时候，我们的基类可能拥有数量众多的不同版本的构造函数。那么当基类中有大量的构造函数，而派生类却只有一些成员函数时，那么对于派生类而言，其构造就等同于构造基类。这个时候，在派生类中我们写的构造函数完完全全就是为了构造基类。为了语法规则，我们需要写很多的”透传”的构造函数。这无疑是相当不方便的。如：

|  |
| --- |
| struct A  {  A(int i){}  A(double d, int i){}  A(float f, int i, const char\* c){}  };  struct B:A  {  B(int i):A(i){}  B(double d, int i):A(d, i){}  B(float f, int i, const char\* c ):A(f, i ,c){}  virtual void ExtraInterface(){}  }; |

事实上，在C++中已经有了一个好用的规则，如果派生类要使用基类的成员函数，可以通过using声明。如：

|  |
| --- |
| #include <iostream>  using namespace std;  struct Base  {  void f(double i){cout << "Base:" << i << endl;}  };  struct Derived:Base  {  using Base::f;  void f(int i){cout << "Derived:"<< i<< endl;}  };  int main()  {  Base b;  b.f(4.5);// Base:4.5  Derived d;  d.f(4.5);// Base:4.5  } |

使用using声明，声明派生类Derived也可以使用基类版本的函数f。这样，派生类中实际上就拥有两个f函数的版本。

在C++11中，这个想法扩展到了构造函数上。子类可以通过使用using声明来声明继承基类的构造函数。如：

|  |
| --- |
| struct A  {  A(int i){}  A(double d, int i){}  A(float f, int i, const char\* c){}  };  struct B:A  {  using A::A;//继承构造函数  virtual void ExtraInterface(){}  }; |

C++11标准继承构造函数被设计为跟派生类中的各种类默认构造函数（默认构造、析构、拷贝构造等）一样，是隐式声明的。如果一个继承构造函数不被相关代码使用，编译器不会为其产生真正的函数代码。

不过继承构造函数只会初始化基类中成员变量，不能初始化派生类中的成员变量。

不过配合类成员的初始化表达式（C++11特性），为派生类成员变量设定一个默认值是没有问题。

|  |
| --- |
| struct A  {  A(int i){}  A(double d, int i){}  A(float f, int i, const char\* c){}  };  struct B:A  {  using A::A;//继承构造函数  int d{0};  };  int main()  {  B b(356);//b.d被初始化为0  } |

基类构造函数的参数会有默认值。对于继承构造函数来说，参数的默认值是不会被继承的。事实上，默认值会导致基类产生多个构造函数的版本，这些函数版本都会被派生类继承。必须小心使用带参数默认值的构造函数的基类。

继承构造函数”冲突”的情况。这通常发生在派生类拥有多个基类的时候。多个基类中的部分构造函数可能导致派生类中的继承构造函数的函数名、参数都相同，那么继承类中的冲突的继承构造函数将导致不合法的派生类代码。如：

|  |
| --- |
| struct A { A(int) {} };  struct B { B(int) {} };  struct C:A,B  {  using A::A;  using B::B;  } |

此时会导致C中重复定义相同类型的继承构造函数。这种情况可以显式定义继承类的冲突的构造函数，阻止隐式生成相应的继承构造函数。编译器目前不支持，编译不通过。

|  |
| --- |
| struct A { A(int) {} };  struct B { B(int) {} };  struct C:A,B  {  using A::A;  using B::B;  C(int){}  }; |

### 3.1.2、委派构造函数

与继承构造函数类似，委派函数也是C++11对C++的构造函数的一项改进，其目的也是为了减少程序员书写构造函数的时间。通过委派其他构造函数，多构造函数的类编写将更加容易。

构造函数代码冗余的例子。如

|  |
| --- |
| class Info  {  public:  Info() : type(1),name('a') { InitReset(); }  Info(int i) : type(i),name('a') { InitReset(); }  Info(char e) : type(1),name(e) { InitReset(); }  private:  void InitReset() {}  int type;  char name;  }; |

除了初始化列表有不同，其他部分基本相似，因此代码存在很多重复。

C++11中，委派构造函数是在构造函数的初始化列表位置进行构造的、委派的。

|  |
| --- |
| class Info  {  public:  Info() { InitReset(); }//目标构造函数  Info(int i) : Info() { type = i; }//委派构造函数  Info(char e) : Info() { name = e; }//委派构造函数  private:  void InitReset() {}  int type {1};  char name {'a'};  }; |

我们将一个构造函数设定为“基准版本”，比如本例的Info()版本构造函数，而其他构造函数可以通过委派“基准版本”进行初始化。

称在初始化列表中调用了“基准版本”的构造函数为委派构造函数（delegating constructor），而被调用的“基准版本”则称为目标构造函数（target constructor）。在C++11中，所谓委派构造，就是指委派函数将构造的任务委派给了目标构造函数来完成这样一种类构造的方式。

委派构造函数不能有初始化列表。在C++中，构造函数不同同时“委派”和使用初始化列表，所以如果委派函数要给变量赋初值，初始化代码必须放在函数体中。

因为初始化列表的初始化方式总是先于构造函数完成的。可以改造以下目标构造函数，使得委派构造函数依然可以在初始化列表中初始化所有成员。如：

|  |
| --- |
| class Info  {  public:  Info() : Info(1, 'a') { }  Info(int i) : Info(i, 'a') { }  Info(char e) : Info(1, e) { }  private:  void InitReset();  Info(int i, char e):type(i), name(e) { InitReset()}  int type;  char name;  }; |

定义一个私有的目标构造函数，这个构造函数接受两个参数，并将参数在初始化列表中初始化。

委托构造的一个很实际的应用就是使用构造模板函数产生目标构造函数。如：

|  |
| --- |
| class TDConstructed  {  template<typename T> TDConstructed(T first, T last):l(first, last) {}  list<int> l;  public:  TDConstructed(vector<short> & v):  TDConstructed(v.begin(), v.end()) {}  TDConstructed(deque<int> & d):  TDConstructed(d.begin(), d.end()) {}  }; |

我们定义了一个构造函数模板。而通过两个委派构造函数的委托，构造函数模板会被实例化。T分别会被推导为vector<short>::iterator和deque<int>::iterator两种类型。这样，TDConstructed类就可以很容易接受多种容器对其进行初始化。这比罗列不同类型的构造函数方便很多。

### 3.1.3、快速初始化成员变量

C++98支持在类声明中，通过“=”来初始化类中的静态成员变量。这种声明方式成为“就地”声明。但是C++98对非静态成员变量的初始化必须在构造函数中进行。

C++11标准允许非静态成员变量的初始化有多种形式。除了初始化列表外，标准还允许使用等号=或者花括号{ }进行就地的非静态成员变量的初始化。如：

|  |
| --- |
| struct init{ int a= 1; double b {1.2} }; |

### 3.1.4、右值引用：移动语义和完美转发

#### 1、移动语义

**背景：**

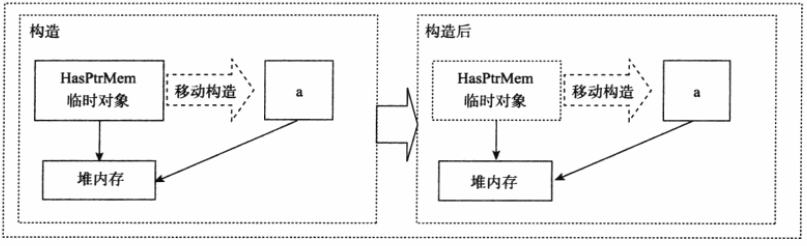
（1）、C++程序，如果在类中包含了一个指针成员的话，要特别小心拷贝构造函数的编写，因为会出现内存泄漏。

（2）、将类类型的对象传递给函数时、从函数返回类类型的对象时，都会调用拷贝构造函数。拷贝构造会在变量上拷贝构造出一个临时值，用以作为函数返回值。

（3）、如果对象指针指向非常大的堆内存数据的话，那么拷贝构造的过程代价就非常大。

**解决思路：**

在构造时，**使得新对象的指针指向临时对象的堆内存资源**。**同时保证临时对象不释放所指向的堆内存**，那么在构造完成后，临时对象被析构，新对象就从中“偷”到了临时对象所拥有的堆内存资源。



在C++11中，这样的“偷走”临时变量中资源的构造函数，就被称为“移动构造函数“。而这样的”偷“的行为，则称之为”移动语义“（move semantics）

移动构造函数如下：

|  |
| --- |
| HasPtrMem(HasPtrMem && h):d(h.d) //移动构造函数  {  h.d = nullptr; //将临时值得指针成员置为空  } |
|  |

移动构造函数使用参数h得成员d初始化了本对象得成员d（而不是像拷贝构造函数一样需要分配内存，然后将内容依次拷贝到新分配的内存中），而h得成员d随后被置为指针空值nullptr。这就完成了移动构造得全过程。

这里所谓得“偷“堆内存，就是指将对象d指向h.d所指得内存这一条语句，相应地，我们还将h得成员d置为指针空值。在移动构造完成之后，临时对象会立即被析构。如果不改变h.d（临时对象得指针成员）的话，则临时对象会析构掉本是我们”偷“来得堆内存。

如果堆内存不是一个int长度的数据，而是以Mbyte为单位的堆空间，那么这样的移动带来的性能提升将非常惊人。

**对比：**

如果直接传一个**引用**或者**指针**到函数参数中去，效果应该也不差。

其实从性能上来讲，这样的做法毫无问题，甚至只好不差。

不过从使用的方便性上来讲效果不好。代码编写效率和可读性存在这明显差别。

#### 2、左值、右值与右值引用

在C++中有一个被广泛认同得说法，那就是可以取地址的、有名字得就是**左值**，反之，不能取地址的、没有名字的就是**右值**。

比如:a=b+c; &a是允许的操作，但&(b+c)这样的操作则不会通过编译。

在C++11中，右值是由两个概念构成的，一个是将亡值（xvalue, eXpiring Value），一个是纯右值(prvalue, Pure Rvalue)。

**纯右值**是C++98标准中右值的概念，是用于辨识临时变量和一些不跟对象关联的值。比如非引用返回的函数返回的临时变量值就是一个纯右值。一些运算表达式，比如1+3产生的临时变量值，也是纯右值。此外，类型转换函数的返回值、lambda表达式等，也都是右值。

**将亡值**是C++11新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为他用），比如返回右值引用T&&的函数返回值、std::move的返回值，或者转换为T&&的类型转换函数的返回值。

在C++11中，**右值引用**就是对一个右值进行引用的类型。事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。通常情况下，我们只能是从右值表达式获得其引用。比如：

|  |
| --- |
| T && a = ReturnRvalue(); |

假设ReturnRvalue返回一个右值，我们就声明一个名为a的右值引用，其值等于ReturnRvalue函数返回的临时变量的值。

我们称C++98中的引用为“**左值引用**”。

左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名。

ReturnRvalue函数返回的右值在表达式语句结束后，其生命也就终结了（具有表达式生命期），而通过右值引用的声明，该右值又“重获新生”，其生命周期将与右值引用类型变量a的生命周期一样。

相比于以下声明方式:

|  |
| --- |
| T b = ReturnRvalue(); |

因为a是右值引用，直接绑定了ReturnRvalue()返回的临时量，而b只是由临时量构造而成，而临时量在表达式结束后会析构，因此会多一次析构和构造的开销。

需要指出的是，能够声明右值引用a的前提是ReturnRvalue返回的是一个右值。通常情况下，右值引用不能绑定到任何的左值的。

常量左值引用在C++98标准中就是个“万能”的引用。它可以接受非常量左值、常量左值、右值对其进行初始化。而且在使用右值对其初始化的时候，常量左值引用还可以像右值引用一样将右值的生命期延长。

为了语义的完整，C++11中还存在着常量右值引用，比如：

|  |
| --- |
| const T && crvalueref = ReturnRvalue(); |

但是，一来右值引用主要是为了移动语义，而移动语义需要右值是可以被修改的，那么常量右值引用在移动语义中就没有用武之地；二来如果要引用右值且让右值不可以更改，常量左值引用往往就足够了。

我们列出C++11中各种引用类型可以引用的值的类型。值得注意的是，只要能够绑定右值的引用类型，都能够延长右值的生命期。



问题：

我使用以下代码验证右值引用，结果不像预期的那样，它们都是构造一次、析构依次，并没有多一次析构和构造的过程。不知道哪里出了问题，这个问题暂时留在这里。

|  |
| --- |
| #include <iostream>  using namespace std;  class T  {  public:  T():m(0)  {  cout << "T construct :" << m << endl;  }  T(int i):m(i)  {  cout << "T construct with para :" << m << endl;  }  T & operator=(const T & rhs)  {  cout << "T assign construct :" << m << endl;  }  ~T()  {  cout << "T destruct :" << m << endl;  }    private:  int m;  };  T ReturnRvalue()  {  T t(9);  return t;  }  int main()  {  //c++98  T a = ReturnRvalue();  T b = a;    //c++11  //T && b = ReturnRvalue();  } |

### 3.1.5、std::move 强制转化为右值

std::**move**唯一的功能是**将一个左值强制转化为右值引用**，继而我们可以通过右值引用使用该值，以用于移动语义。从实现上讲，std::move基本等同于一个类型转换：

|  |
| --- |
| static\_cast<T&&>(lvalue); |

被转化的左值，其生命期并没有随着左右值得转化而改变。

看move使用的例子：

|  |
| --- |
| #include <iostream>  using namespace std;  class HugeMem  {  public:  HugeMem(int size):sz(size > 0?size:1)  {  c = new int[sz];  }  ~HugeMem() {delete []c;}  HugeMem(HugeMem && hm):sz(hm.sz), c(hm.c)  {  hm.c = nullptr;  }  int \* c;  int sz;  };  class Moveable  {  public:  Moveable():i(new int(3)),h(1024){}  ~Moveable(){delete i;}  Moveable(Moveable && m):i(m.i),h(move(m.h))  {  m.i = nullptr;  }  int\* i;  HugeMem h;  };  Moveable GetTemp(){  Moveable tmp = Moveable();  cout << hex <<"Huge Mem from"<<\_\_func\_\_<<"@"<<tmp.h.c<<endl;  return tmp;  }  int main(){  Moveable a(GetTemp());  cout<<hex<<"Huge Mem from"<<\_\_func\_\_<<"@"<<a.h.c<<endl;  } |

输出结果：

|  |
| --- |
| Huge Mem fromGetTemp@0x2139030  Huge Mem frommain@0x2139030 |

在Moveable的移动构造函数中，使用std::move函数。该函数将m.h强制转化为右值，以迫使Moveable中的h能够实现移动构造。

这里使用std::move是因为m.h是m的成员，既然m将在表达式结束后被析构，其成员也自然会被析构。通过move转换为右值引用，延长了m.h的生命期，同h一样的生命期。

为了保证移动语义的传递，程序员在编写移动构造函数的时候，应该总是记得使用std::move转换拥有形如堆内存、文件句柄等资源的成员为右值，这样，如果成员支持移动构造的话，就可以实现其移动语义。而即使成员没有移动构造函数，那么接受常量左值的构造函数版本也会轻轻松松地实现拷贝构造，因此不会引起大的问题。

### 3.1.6、移动语义补充

移动语义一定是要修改临时变量的值。

如果声明移动构造函数：

|  |
| --- |
| Moveable(const Moveable &&)  const Moveable ReturnVal(); |

都会使得临时变量常量化，成为一个常量右值，那么临时变量的引用也就无法修改，从而导致无法实现移动语义。

在C++11中，拷贝/移动构造函数实际上有以下3个版本：

|  |
| --- |
| T object(T &)  T object(const T &)  T object(T &&) |

其中常量左值引用（第2个）的版本是一个拷贝构造版本，而右值引用版本（第3个）是一个移动构造版本。

只实现一种语义在类的编写中也是非常常见。而只有移动语义的类型时，因为只有移动语义表明该类型的变量所拥有的资源只能被移动，而不能被拷贝。那么这样的资源必须是唯一的。因此，只有移动语义构造的类型往往都是“资源型”的类型，比如说智能指针，文件流等，都可以视为“资源型”的类型。

### 3.1.7、完美转发

perfect forwarding是指在函数模板中，完全依照模板的参数的类型，将参数传递给函数模板中调用的另外一个函数。比如：

|  |
| --- |
| template <typename T>  void IamForwarding(T t) { IrunCodeActually(t); } |

IamForwarding是一个转发函数模板。而函数IrunCodeActually则是真正执行代码的目标函数。

对于目标函数IrunCodeActually，它总是希望转发函数将参数按照传入IamForwarding时的类型传递，而不产生额外的开销，就好像转发者不存在一样。

通常，使用引用类型，引用类型不会有拷贝的开销。其次，则需要考虑转发函数对类型的接受能力。因为目标函数可能需要能够即接受左值引用，又接受右值引用。

C++11引入了一条“引用折叠”（reference collapsing）的新语言规则，并结合新的模板推导规则来完成完美转发。

|  |
| --- |
| typedef const int T;  typedef T& TR;  TR& v = 1; |

TR& v = 1;在C++98中不合法，但在C++11中，一旦出现这样的表达式，就会发生引用折叠，即将复杂的位置表达式折叠为已知的简单表达式。



一旦定义中出现了左值引用，引用折叠总是优先将其折叠为左值引用。

这里不过分追求实现原理，仅需要学会使用该特性即可。

应用：

|  |
| --- |
| template <typename T>  void IamForwarding(T && t) { IrunCodeActually(static\_cast<T &&>(t)); } |

这里，不仅在参数部分使用了T &&这样的标识，在目标函数传参的强制类型转换中也使用了这样的形式。比如我们调用转发函数时传入了一个X类型的左值引用。转发函数将被实例化为如下形式：

|  |
| --- |
| void IamForwarding(X& && t){  IrunCodeActually(static\_cast<X& &&>(t));  } |

应用引用折叠规则：

|  |
| --- |
| void IamForwarding(X & t){  IrunCodeActually(static\_cast<X&>(t));  } |

如果调用转发函数时传入一个X类型的右值引用的话，转发函数将会被实例化为：

|  |
| --- |
| void IamForwarding(X&& && t){  IrunCodeActually(static\_cast<X&& &&>(t));  } |

应用引用折叠规则：

|  |
| --- |
| void IamForwarding(X && t){  IrunCodeActually(static\_cast<X&&>(t));  } |

static\_cast的作用：对于一个右值而言，当它使用右值引用表达式引用的时候，该右值引用却是个不折不扣的左值，那么在函数调用中继续传递右值，就需要使用std::move来进行左右值的转换。而std::move实质上就是一个static\_cast。不过在C++11中，用于完美转发的函数却不叫作move，而是另一个名字：forward。

所以可以把转发函数写成这样：

|  |
| --- |
| template<typename T>  void IamForwarding(T && t){  IrunCodeActually(forward(t));  } |

完美转发的一个作用就是做包装函数。可以用很少的代码记录单参数函数的参数传递状况。

### 3.1.8、初始化列表

自动变量和全局变量的初始化在C++11中被丰富了，可以通过以下几种形式完成初始化工作：

1、等号“=”加上赋值表达式。int a = 3+4

2、等号“=”加上花括号式的初始化列表。int a = {3+4}

3、圆括号式的表达式列表。int a(3+4)

4、花括号式的初始化列表。int a {3+4}

使用初始化列表对vector、map等非内置的复杂的数据类型也可以进行初始化。

在C++11中，标准总是倾向于使用更为通用的方式来支持新的特性。

#include <initializer\_list>

然后生命一个以initialize\_list<T>模板类为参数的构造函数，同样可以使得自定义的类使用列表初始化。

### 3.1.9、POD类型

Plain Old Data

POD在C++中是非常重要的一个概念，通常用于说明一个类型的属性，尤其是用户自定义类型的属性。POD属性在C++11中往往又是构建其他C++概念的基础。

Plain表示POD是个普通的类型，在C++中常见的类型都有这样的属性，而不像一些存在着虚函数虚继承的类型那么特别。

Old体现了其与C的兼容性，比如可以用最老的memcpy进行复制，使用memset进行初始化等。

C++11将POD划分为两个基本的概念的合集，即：平凡的（trivial）和标准布局的（standard layout）。

通常情况下，一个平凡的类或结构体应符合以下定义：

1、拥有平凡的默认构造函数（trivial constructor）和析构函数（trivial destructor）

不定义类的构造函数，编译器就会生成一个平凡的默认构造函数。而一旦定义了构造函数，即使构造函数不包含参数，函数体没有任何的代码，那么该构造函数也不再是“平凡”的。

可以使用=default关键字显式地声明缺省版本的构造函数从而使得类型恢复“平凡化”。

2、拥有平凡的拷贝构造函数（trivial copy constructor）和移动构造函数（trivial move constructor）

3、拥有平凡的拷贝赋值运算符（trivial assignment operator）和移动赋值运算符（trivial move operator）

4、不能包含虚函数以及虚基类

POD的另一个概念是标准布局。

1、所有非静态成员有相同的访问权限。

2、在类或者结构体继承时，满足以下两种情况之一：

2.1、派生类中有非静态成员，且只有一个仅包含静态成员的基类。

2.2、基类有非静态成员，而派生类没有非静态成员。

3、类中第一个非静态成员的类型与其基类不同。

4、没有虚函数和虚基类

5、所有非静态数据成员均符合标准布局类型，其基类也符合标准布局

以上5点构成标准布局的含义前两条最为重要。

使用POD的优点：

1、字节赋值，代码中可以安全地使用memset和memcpy对POD类型进行初始化和拷贝等操作。

2、提供对C内存布局兼容。C++程序可以与C函数进行互相操作，因为POD类型的数据在C与C++间的操作总是安全的。

3、保证了静态初始化的安全有效。静态初始化在很多时候能够提高程序的性能，而POD类型的对象初始化往往更加简单。

### 3.1.10、一般化的SFINEA规则

SFINEA – Substitution failure is not an error，中文直译“匹配失败不是错误”。这条规则表示的是对重载的模板的参数进行展开的时候，如果展开导致了一些类型不匹配，编译器并不会报错。

## 3.2、新手易学，老兵易用

### 3.2.1、auto类型推导

静态类型：类型检查主要发生在编译阶段

动态类型：类型检查主要发生在运行阶段。

## 3.3、提高类型安全

C++更强调类型，目的时为了在构建复杂的软件系统时，能够尽可能地在编译期找到错误并提醒程序员。

### 3.3.1、强类型枚举

### 3.3.2、堆内存管理：智能指针与垃圾回收

#### 3.3.2.1、显式内存管理

C/C++中的显式堆内存管理。由于没有正确处理堆内存的分配与释放，在语言层面上，可以归纳为以下一些问题。

1、野指针：一些内存单元已被释放，之前指向它的指针却还在被使用。这些内存有可能被运行时系统重新分配给程序使用，从而导致了无法预测的错误。

2、重复释放：程序试图去释放已经被释放过的内存单元，或者释放已经被重新分配过的内存单元，就会导致重复释放错误。通常重复释放内存会导致C/C++运行时系统打印出大量错误及诊断信息。

3、内存泄漏：不再需要使用的内存单元如果没有被释放就会导致内存泄漏。如果程序不断地重复进行这类操作，将会导致内存占用剧增。

显式的管理内存在性能上有一定的优势，但也被广泛地认为是容易出错的。

#### 3.3.2.2、C++11的智能指针

在C++98中，智能指针通过一个模板类型“auto\_ptr”来实现。auto\_ptr以对象的方式管理对分配的内存，并在适当的时间（比如析构），释放所获得的堆内存。这种堆内存管理的方式只需要将new操作返回的指针作为auto\_ptr的初始值即可，不用再显式地调用delete。比如：

|  |
| --- |
| auto\_ptr(new int); |

C++11标准中改用unique\_ptr、shared\_ptr及weak\_ptr等智能指针来自动回收堆内存分配的对象。

unique\_ptr与所指对象的内存绑定紧密，不能与其他unique\_ptr类型的指针对象共享所指对象的内存。因为每个unique\_ptr都是唯一地“拥有”所指的对象内存，其他指针无法共享其“所有权”。事实上，这种“所有权”仅能够通过标准库的move函数来转移，一旦“所有权”转移成功，原来的unique\_ptr指针就失去了对象内存的“所有权”。

从实现上讲，unique\_ptr是一个删除了拷贝构造函数、保留了移动构造函数的指针封装类型。所以仅可以使用右值对unique\_ptr对象进行构造，而且一旦构造成功，右值对象中的指针即被“窃取”，因此该右值对象即刻失去了对指针的“所有权”。

shared\_ptr允许多个该智能指针共享地“拥有”同一堆分配对象的内存。与unique\_ptr不同的是，由于在实现上采用了引用计数，所以一旦一个shared\_ptr指针放弃了“所有权”（失效），其他的shared\_ptr对对象内存的引用并不会收到影响。

如果多个shared\_ptr指针共享了new分配的堆内存，其中一个指针调用reset函数只会导致引用计数的降低，而不会导致堆内存的释放。只有在引用计数归零的时候，shared\_ptr才会真正释放所占有的堆内存的空间。

weak\_ptr它可以指向shared\_ptr指针所指向的对象内存，却并不拥有该内存。而使用weak\_ptr成员lock，则可以返回其所指内存的一个shared\_ptr对象，且在所指对象内存已经无效时，返回指针空值（nullptr）。这在验证shared\_ptr智能指针的有效性上会有很有作用。

#### 3.3.2.3、垃圾回收的分类

这里，把之前使用过，现在不再使用或者没有任何指针再指向的内存空间就成为“垃圾”。而将这些“垃圾”收集起来以便再次利用的机制，就被成为“垃圾回收”（Carbage Collection）。

垃圾回收的两大分类：

1、基于引用计数（reference counties garbage collector）的垃圾回收器

引用计数主要是使用系统记录对象被引用（引用、指针）的次数。当对象被引用的次数变为0时，该对象即可被视为“垃圾”而回收。

2、基于跟踪处理（tracing garbage collector）的垃圾回收器

跟踪处理的基本方法是产生跟踪对象的关系图，然后进行垃圾回收。

使用跟踪方式的垃圾回收算法主要有以下几种：

（1）、标记-清除（Mark-Sweep）

（2）、标记-整理（Mark-Compact）

（3）、标记-拷贝（Mark-Copy）

#### 3.3.2.4、C++11与最小垃圾回收支持

指针的灵活性使用可能是C/C++中的一大优势，而对于垃圾回收来说，却会带来很大的困扰。被隐藏的指针会导致编译器在分析指针的可达性（生命周期）时出错。而即使编译器开发出了隐藏指针分析的手段，其带来的编译开销也不会让程序员对编译时间的显著增长视而不见。历史上，解决这种问题的方法通常是新接口。C++11和垃圾回收的解决方案也不例外，就是让程序员利用这样的接口来提示编译器代码中存在指针不安全的区域。

C++11为了做到最小的垃圾回收支持，首先对“安全”的指针进行了定义，或者使用C++11中的术语说，安全派生（safely derived）的指针。安全派生的指针是指向由new分配的对象或其子对象的指针。

安全派生指针的操作包括：

1、在解引用基础上的引用，比如：&\*p

2、定义明确的指针操作，比如：p+1

3、定义明确的指针转换，比如：static\_cast<void \*>(p)

4、指针和整型之间的reinterpret\_cast，比如：reinterpret\_cast<intptr\_t>(p)

## 3.4、提高性能及操作硬件的能力

### 3.4.1常量表达式

#### 3.4.1.1、运行时常量与编译时常量

常量表示该值不可修改，通常是通过const关键字来修饰的。比如：

|  |
| --- |
| const int I = 3; |

const还可以修饰函数参数、函数返回值、寒素本身、类等。大多数情况下，const描述的都是一些“运行时常量性”的概念，即具有运行时数据的不可更改性。不过有时候，我们需要的却是编译期的常量性，这是const关键字无法保证的。

|  |
| --- |
| const int GetConst() { return 1; } |

C++11中对编译时期常量的做法是constexpr，即常量表达式（constexpr）。比如使GetConst成为一个常量表达式，在函数表达式前加上constexpr即可。

|  |
| --- |
| constexpr int GetConst() { return 1; } |

有了常量表达式的声明，编译器就可以在编译时期对GetConst表达式进行值计算（evaluation），从而将其视为一个编译时期的常量。

#### 3.4.1.2、常量表达式函数

并不是所有的函数都有资格成为常量表达式函数。需要满足几点：

1、函数体只有单一的return返回语句。

2、函数必须返回值（不能是void函数）。

3、在使用前必须已有定义。

4、return返回语句表达式中不能使用非常量表达式的函数、全局数据，且必须是一个常量表达式。

只有一条语句，且该条语句必须是return语句。

必须有返回值。

常量表达式“使用”和“调用”的区别，前者讲的是编译时的值计算，而后者讲的是运行时的函数调用。

常量表达式中，不能使用非常量表达式的函数。比如：

|  |
| --- |
| int g = 3;  constexpr int h() { return g; }  //这是不能通过编译的。 |

#### 3.4.1.3、常量表达式值

由constexpr关键字修饰的变量就是所谓的常量表达式值（constant-expression value）。通常，常量表达式值必须被一个常量表达式赋值，而跟常量表达式函数一样，常量表达式值在使用前必须被初始化。

|  |
| --- |
| const int i = 1;  constexpr int j = 1; |

两者在大多数情况下是没有区别的。

对于自定义类型的数据，要使其成为常量表达式值，需要定义自定义常量构造函数（constant-expression constructor）。

|  |
| --- |
| struct MyType {  constexpr MyType(int x):i(x) {}  int I;  };  constexpr MyType mt = {0}; |

常量表达式的构造函数也有使用上的约束，主要有两点：

1、函数体必须为空

2、初始化列表只能由常量表达式来赋值。

### 3.4.2、变长模板

#### 3.4.2.1、变长函数和变长的模板参数

#### 3.4.2.1、变长模板：模板参数包和函数参数包

变长模板的语法，声明tuple是一个变长类模板：

|  |
| --- |
| template <typename... Elements> class tuple; |

在标示符Elements之前的使用了省略号来表示该参数是变长的。在C++11中，Elements被称作是一个“模板参数包”（template paramenter pack）。这是一种新的模板参数类型。类模板tuple可以接受任意多个参数作为模板参数。以下实例化的tuple模板类：

|  |
| --- |
| tuple<int, char, double> |

编译器则可以将多个模板参数打包成为“单个的”模板参数包Elements，即Element在进行模板推导的时候，就是一个包含了int、char和double三种类型类型集合。

### 3.4.3、原子类型与原子操作

#### 3.4.3.1、并行编程、多线程与C++

在C++11之前，C/C++一直是一种顺序的编程语言。顺序是指所有指令都是串行执行的，即在相同的时刻，有且仅有单个CPU的程序计数器指向可执行代码的代码段，并运行代码段中的指令。

常见的并行编程有多种模型，如共享内存、多线程、消息传递等。不过从实用性上讲，多线程模型往往具有较大的优势。多线程模型允许同一时间有多个处理器单元执行统一进程中的代码部分，而通过分离的栈空间和共享的数据区及堆栈空间，线程可以拥有独立的执行状态以及进行快速的数据共享。

在C++11之前，在C/C++程序中使用线程，这样的代码主要使用POSIX线程（pthread）和OpenMP编译器指令两种编程模型来完成程序的线程化。其中，POSIX线程是POSIX标准中关于线程的部分，可以通过一些Pthread线程的API来完成线程的创建、数据的共享、同步等功能。Pthread主要用于C语言，在类Unix系统上。

在C++11标准中，引入了多线程的支持。使得C/C++在进行线程编程时，不必依赖第三方库和标准。

#### 3.4.3.2、原子操作与C++11原子类型

原子操作，就是多线程程序中“最小的且不可并行化的”的操作。通常对一个共享资源的操作是原子操作的话，意味着多个线程访问该资源时，有且仅有唯一一个线程在对这个资源进行操作。那么从线程（处理器）的角度看，其他线程就不能够在本线程对资源访问期间对该资源进行操作，因此原子操作对于多个线程而言，不会发生有别于单线程程序的意外状况。

通常情况下，原子操作都是通过“互斥“（mutual exclusive）的访问来保证的。实现互斥通常需要平台相关的特殊指令，这在C++11标准之前，这常常意味着需要在C/C++代码中嵌入内联汇编代码。如果指向实现粗粒度的互斥，借助POSIX标准的pthread库中的互斥锁（mutex）也可以做到。

在C++11中，不需要为原子数据类型显式地声明互斥锁或调用加锁、解锁的API，线程就能够对变量互斥地进行访问。

相比于基于C以及过程编程的pthread“原子操作API“而言，C++11对于”原子操作“概念的抽象遵从了面向对象的思想——C++11标准定义的都是所谓的”原子类型“。而传统意义上所谓的”原子操作“，则抽象为针对于这些原子类型的操作。直观的看，编译器可以保证原子类型在线程间被互斥地访问。这样设计，从并行编程的角度看，是由于需要同步的总是数据而不是代码，因此C++11对数据进行抽象，会有利于产生更为良好的并行代码。而一些琐碎的概念，比如互斥锁、临界区则可以被C++11的抽象所掩盖，因而并行代码的编写也会变得更加简单。

由于C++11与C11标准都支持原子类型，因此可以简单的通过#include<cstdatomic>头文件中来使用对应于内置类型的原子类型定义。



更为普遍的做法，是使用atomic类模板。通过类模板，可以任意定义出需要的原子类型。比如：

|  |
| --- |
| std::atomic<T> t; |

就声明了一个类型为T的原子类型变量t。编译器会保证产生并行情况下行为良好的代码，以避免线程间对数据t的竞争。

对于线程而言，原子类型通常属于“资源型“的数据，这意味着多个线程通常只能访问单个原子类型的拷贝。因此在C++11中，原子类型只能从其模板参数类型中进行构造，标准不允许原子类型进行拷贝构造、移动构造，以及使用operator=等，以防止放生意外。

atomic类模板总是定义了从atomic<T>到T的类型转换函数，在需要时，编译器会隐式地完成原子类型到其对应的类型的转换。

|  |
| --- |
| atomic<T> af {1.2f};  float f = af;  float f1 {af}; |

#### 3.4.3.3、内存模型，顺序一致性与memory\_order

顺序一致性（sequential consistent）的内存模型（memory model）。

顺序一致只是属于C++11中多种内存模型中的一种。而在C++11中，并不是只支持顺序一致单个内存模型的原子变量，因为顺序一致往往意味着最低效的同步方式。

通常情况下，内存模型通常是一个硬件上的概念，表示的是机器指令是以什么样的顺序被处理器执行的。

对于并行编程来说，可能最根本的还是思考如何将大量计算的问题，按需分解成多个独立的、能够同时运行的部分，并找出真正需要在线程间共享的数据，实现为C++11的原子类型。

#### 3.4.3.4、线程局部存储

线程局部存储（TLS，thread local storage）就是拥有线程生命期及线程可见性的变量。

线程局部存储实际上是由单线程程序中的全局/静态变量被应用到多线程程序中被线程共享而来。

通常情况下，线程会拥有自己的栈空间，但是堆空间、静态数据区（从可执行文件看，静态数据区对应的是可执行文件data、bss段的数据，从C/C++语言层面看，则对应的是全局/静态变量）则是共享的。全局、静态变量在多线程模型下就是在线程间共享的。

## 3.5、为改变思考方式而改变

### 3.5.1、指针空值nullptr

在C++11新标准中，出于兼容性的考虑，字面常量0的二义性并没有被消除。但是为二义性给出了新的答案，就是nullptr。它是一个所谓“指针空值类型“的常量。指针空值类型被命名为nullptr\_t。

### 3.5.2、默认函数的控制

#### 3.5.2.1、类与默认函数

在声明的自定义类中，编译器会默认生成一些它们未自定义的成员函数。这些函数版本被称为“默认函数“。包括以下自定义类型的成员函数。

构造函数

拷贝构造函数

拷贝赋值函数

移动构造函数

移动拷贝函数

析构函数

此外，C++编译器还会为以下这些自定义类型提供全局默认操作符函数。

|  |
| --- |
| operator,  operator &  operator &&  operator \*  operator ->  operator ->\*  operator new  operator delete |

在C++11中，标准是通过提供了新的机制来控制默认版本函数的生成来完成这个目标的。这个新机制重用了default关键字。在默认函数定义或声明时加上“=default“，从而显式地指示编译器生成该函数的默认版本。如果指定产生默认版本后，不再也不应该实现一份同名的函数。

在函数的定义或声明上加上“=delete“，会指示编译器不生成函数的缺省版本。一旦缺省版本被删除了，重载该函数也是非法的。

C++11标准称“=default“修饰的函数为显式缺省（explicit default）函数。

称“=delete“修饰的函数为删除（deleted）函数，称显式删除函数。

### 3.5.3、lambda函数

lambda（λ）在希腊字母表中。在数理逻辑或计算机科学领域中，lambda则是被用来表示一种匿名函数，这种匿名函数代表了一种所谓的λ演算（lambda calculus）。

从软件开发的角度看，lambda概念为基础的“函数式编程“（Functional Programming）是与命令式编程（Imperative Programming）、面向对象编程（Object-orientated Programming）等并列的一种编程泛型（Programming Paradigm）。

#### 3.5.3.1、C++11的lambda函数

观察代码：

|  |
| --- |
| int main()  {  int girls = 3, boys = 4;  auto totalChild = [](int x, int y)->int{ return x+y; };  return totalChild(girls, boys);  } |

定义一个lambda函数。该函数接受两个参数(int x, int y)，并且返回其和。直观地看，lambda函数跟普通函数相比不需要定义函数名，取而代之的多了一对方括号。此外，lambda函数还采用了跟踪返回类型的方式声明其返回值。

通常情况下，lambda函数的语法定义如下：

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

1、[capture]：捕获列表。捕获列表总是出现在lambda函数的开始处。事实上，[]是lambda引出符。编译器根据该引出符判断接下来的代码是否是lambda函数。捕获列表能够捕获上下文中的变量以供lambda函数使用。

2、(parameters)：参数列表。与普通函数的参数列表一致。如果不需要传递参数，则可以连同括号()一起省略。

3、mutable：mutable修饰符。默认情况下，lambda函数总是一个const函数，mutable可以取消其常量性。在使用该修饰符时，参数列表不可省略（即使参数为空）。

4、->return-type：返回类型。用追踪返回类型形式声明函数的返回类型。出于方便，不需要返回值的时候也可以连同符号->一起省略。此外，在返回类型明确的情况下，也可以省略该部分，让编译器对返回类型进行推导。

5、{statement}：函数。内容与普通函数一样，不过除了可以使用参数之外，还可以使用所有捕获的变量。

在lambda函数的定义中，参数列表和返还类型都是可选的部分，而捕获列表和函数体都可能为空。那么在极端情况下，C++11中最简略的lambda函数只需要声明为：

|  |
| --- |
| []{}; |

就可以了。不过，该lambda函数不能做任何事。

直观地讲，lambda函数与普通函数可见的最大区别之一，就是lambda函数可以通过捕获列表访问一些上下文中的数据。具体的，捕获列表描述了上下文中那些的数据可以被lambda使用，以及使用方式（以值传递的方式或引用传递的方式）。

|  |
| --- |
| int main()  {  int girls = 3, boys = 4;  auto totalChild = [girls, &boys]()->int{ return girls+boys; };  return totalChild();  } |

上述代码中，使用了捕获列表捕捉上下文中的变量girls、boys。totalChild不需要传递参数。此时的girls和boys可以视为lambda函数的一种初始状态，lambda函数的运算则是基于初始状态进行的运算。这与函数简单基于参数的运算是有所不同的。

语法上，捕捉列表由多个捕捉项组成，并以逗号分割。捕捉列表有以下几种形式：

1、[var]表示值传递方式捕捉变量var

2、[=]表示值传递方式捕捉所有父作用域的变量（包括this）

3、[&var]表示引用传递捕捉变量var

4、[&]表示引用传递捕捉所有父作用域的变量（包括this）

5、[this]表示值传递方式捕捉当前的this指针

父作用域：enclosing scope，这里指的是包含lambda函数的语句块。

通过一些组合，捕捉列表可以表示更复杂的意思。

1、[=,&a,&b]表示以引用传递的方式捕捉变量a和b，值传递方式捕捉其他所有变量。

2、[&,a,this]表示以值传递的方式捕捉变量a和this，引用传递方式捕捉其他所有变量。

依照现行的C++11标准，在块作用域（block scope，可以简单理解为在{}之内的任何代码都是块作用域的）以外的lambda函数捕捉列表必须为空。因此这样的lambda函数除去语法上的不同以外，跟普通函数区别不大。而在块作用域中的lambda函数仅能捕捉父作用域中的自动变量，捕捉任何非此作用域或者是非自动变量（如静态变量等）都会导致编译器报错。

#### 3.5.3.2、lambda与仿函数

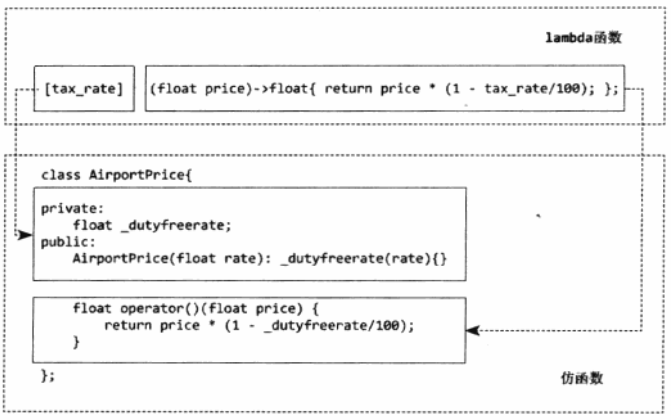
仿函数（函数对象）：简单讲就是重定义了成员函数operator()的一种自定义类型对象。这样的对象有个特点，就是其使用在代码层面感觉跟函数的使用并无二样，但究其本质却并非函数。

仿函数例子：

|  |
| --- |
| class \_functor {  public:  int operator()(int x, int y) {return x+y;}  };  int main(){  int girls = 3, boys = 4;  \_functor totalChild;  return totalChild(5, 6);  } |

class \_functor的operator()被重载，因此，在调用该函数的时候，就跟函数调用一样的形式，只不过这里的totalChild不是函数名称，而是对象名称。

事实上，仿函数是编译器实现lambda的一种方式。在现阶段，通常编译器都会把lambda函数转化为一个仿函数对象。因此，在C++11中，lambda可以视为仿函数的一种等价形式。



仿函数被广泛地用于STL中，同样的，在C++11中，lambda也在标准库中被广泛地使用。由于其书写简单，通常可以就地定义，因此用户常可以使用lambda代替仿函数来书写代码。

#### 3.5.3.3、lambda的基础使用

编写lambda函数非常容易。不过用得上lambda函数的地方比较特殊。最为简单的引用下，会利用lambda函数来封装一些代码逻辑，使其不仅具有函数的包装性，也具有就地可见的自说明性。

局部函数（local function，即在函数作用域中定义的函数），也称为内嵌函数（nested function）。局部函数通常仅属于其父作用域，能够访问父作用域的变量，且在其父作用域中使用。C/C++语言标准中不允许局部函数存在，C++11标准却用比较优雅的方式打破了这个规则。因为事实上，lambda可以像局部函数一样使用。

对于运算比较复杂的函数（比如实现一些很复杂的算法），通常函数中会有大量的局部状态（变量），这时候如果只需要一些“局部”的功能——比如打印一些内部状态，或者做一些固定的操作这些功能往往不能与其他任何的代码共享，却要在一个函数中多次重用。那么使用lambda的捕捉列表功能则相较于独立的全局静态函数或私有成员函数方便很多。设计一个仅适用捕获列表lambda函数不会像设计传统函数一样要关心大量细节：需要多少参数、那些参数需要按值传递、那些参数需要按引用或指针传递，统统不需要程序员来考虑。而且父函数结束后，该lambda函数也就不再可用了，不会污染任何名字空间。因此，对于复杂代码的快速开发而言，lambda的出现意义重大。

现有的C++11标准中的lambda等价的是有常量operator()的仿函数。因此在使用捕获列表的时候必须注意，按值传递方式捕捉的变量是lambda函数中不可更改的常量。标准这么设计可能源自早期STL算法一些设计的缺陷（对仿函数没有做限制，从而导致一些设计不算特别良好的算法出错）。而更一般地讲，这样的设计有其合理性，改变从上下文拷贝而来的临时变量通常不具有任何意义。绝大多数时候，临时变量只是用于lambda函数的输入，如果需要输出结果到上下文，可以使用引用，或者通过让lambda函数返回值来实现。

# 四、深入应用C++11：代码优化与工程级应用

In-Depth C++11 Code Optimization and Engineering Level Application

# 五、其他应用

## 5.1、事件驱动和消息驱动

**事件**：按下鼠标，按下键盘，将U盘插入USB接口，都将产生事件。比如说按下鼠标左键，将产生鼠标左键被按下的事件。

**消息**：当鼠标被按下，产生了鼠标按下事件，Windows侦测到这一事件的发生，随即发出鼠标被按下的消息到消息队列中，这消息附带了一系列相关的事件信息，比如鼠标那个键按了，在那个窗口被按的，等等。

1.要理解事件驱动和程序，就需要与非事件驱动的程序进行比较。实际上，现代的程序大多是事件驱动的，比如多线程的程序，肯定是事件驱动的。早期则存在许多非事件驱动的程序，这样的程序，在需要等待某个条件触发时，会不断地检查这个条件，直到条件满足，这是很浪费cpu时间的。而事件驱动的程序，则有机会释放cpu从而进入睡眠态（注意是有机会，当然程序也可自行决定不释放cpu），当事件触发时被操作系统唤醒，这样就能更加有效地使用cpu.

2.再说什么是事件驱动的程序。一个典型的事件驱动的程序，就是一个死循环，并以一个线程的形式存在，这个死循环包括两个部分，第一个部分是按照一定的条件接收并选择一个要处理的事件，第二个部分就是事件的处理过程。程序的执行过程就是选择事件和处理事件，而当没有任何事件触发时，程序会因查询事件队列失败而进入睡眠状态，从而释放cpu。

3.事件驱动的程序，必定会直接或者间接拥有一个事件队列，用于存储未能及时处理的事件。

4.事件驱动的程序的行为，完全受外部输入的事件控制，所以，事件驱动的系统中，存在大量这种程序，并以事件作为主要的通信方式。

5.事件驱动的程序，还有一个最大的好处，就是可以按照一定的顺序处理队列中的事件，而这个顺序则是由事件的触发顺序决定的，这一特性往往被用于保证某些过程的原子化。

6.目前windows,linux,nucleus,vxworks都是事件驱动的，只有一些单片机可能是非事件驱动的。

事件模式耦合高，同模块内好用；消息模式耦合低，跨模块好用。事件模式集成其它语言比较繁琐，消息模式集成其他语言比较轻松。事件是侵入式设计，霸占你的主循环；消息是非侵入式设计，将主循环该怎样设计的自由留给用户。如果你在设计一个东西举棋不定，那么你可以参考win32的GetMessage，本身就是一个藕合度极低的接口，又足够自由，接口任何语言都很方便，具体应用场景再在其基础上封装成事件并不是难事，接口耦合较低，即便哪天事件框架调整，修改外层即可，不会伤经动骨。而如果直接实现成事件，那就完全反过来了。

**线程驱动和事件驱动**

那么线程驱动编程和事件驱动编程之间的区别是什么呢？

线程驱动就是当收到一个请求的时候，将会为该请求开一个新的线程来处理请求。一般存在一个线程池，线程池中有空闲的线程，会从线程池中拿取线程来进行处理，如果线程池中没有空闲的线程，新来的请求将会进入队列排队，直到线程池中空闲线程。

事件驱动就是当进来一个新的请求的时，请求将会被压入队列中，然后通过一个循环来检测队列中的事件状态变化，如果检测到有状态变化的事件，那么就执行该事件对应的处理代码，一般都是回调函数。

对于事件驱动编程来说，如果某个时间的回调函数是计算密集型，或者是阻塞I/O,那么这个回调函数将会阻塞后面所有事件回调函数的执行。这一点尤为重要。

# 参考文献

1、《深入应用C++11 代码优化与工程级应用》，祈宇，机械工业出版社；

2、<https://stackoverflow.com/questions/1518711/how-does-free-know-how-much-to-free>

3、<http://gaoshen.site/2013-5-20-aboutfree/>

4、深入理解C++11：C++11新特性解析与应用，IBM XL编译器中国开发团队，机械工业出版社；