**十大必掌握C++11新特性**

本文主要罗列C++11相比较旧版本有重大改变并且我们今后可能会频繁接触的一些功能。  
　　**1、新增基于范围的for循环**  
　　类似Java中foreach语句，为遍历数组提供了很大方便。

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

for(int &x : nArr)

{

x \*=2; *//数组中每个元素倍乘*

}

**2、自动类型推断 auto**  
　　它的作用就是当编译器在一个变量声明的时候，能够根据变量赋的值推断该变量的数据类型。这样就有些逼近Python中定义变量的功能，无需提前声明定义的变量的数据类型。例如：

auto i = 1; *//编译器自动推断i为int类型*

　　这个功能在各种标准模板库容器中使用，作用更突出。如下：

vector<int> vec(6,10);

vector<int>::iterator iter = vec.iterator();

auto iterAuto = vec.iterator(); *//相比较上一句方便很多*

**3、匿名函数 Lambda**  
　　如果代码里面存在大量的小函数，而这些函数一般只被一两处调用，那么不妨将它们重构成Lambda表达式，也就是匿名函数。作用就是当你想用一个函数，但是又不想费神去命名一个函数。  
　　该功能函数实际上在其他面向对象语言中早就存在，例如Java，Python都定义了该功能。C++中Lambda表达式格式如下：

[capture](params)->ret { body};

　　① [capture]指定在可见域范围内lambda表达式代码内可见的参数。例如：  
- [a, &b]，前文定义的a以值方式被表达式捕获，b则是以引用的方式；  
- [this] 以值的方式捕获 this 指针。  
- [&] 以引用的方式捕获所有的外部自动变量。  
- [=] 以值的方式捕获所有的外部自动变量。  
- [] 不捕获外部的任何变量。  
【注】：const 类型的 lambda 表达式，该类型的表达式不能改捕获(“capture”)列表中的值。  
　　② (params)指定lambda表达式内部变量定义。  
　　③ ->ret是返回类型，如果 lambda 代码块中包含了 return 语句，则该 lambda 表达式的返回类型由 return 语句的返回类型确定。如果没有 return 语句，则类似 void f(…) 函数。  
　　④ {body}是Lambda表达式主题结构。  
　　例句：

auto func = [](int i){ return i+4};*// 可以体会auto的好处了*

cout<< func(10) << endl; *//输出为14*

**4、后置返回类型（tailng-return-type）**

template Ret adding\_func(const Lhs &lhs, const Rhs &rhs) {return lhs + rhs;}

　　这不是合法的C++，因为lhs和rhs还没定义；解析器解析完函数原型的剩余部分之前,它们还不是有效的标识符。  
　　为此, C++11引入了一种新的函数声明语法，叫做后置返回类型(trailing-return-type)。

template auto adding\_func(const Lhs &lhs, const Rhs &rhs) -> decltype(lhs+rhs) {return lhs + rhs;}

　　这种语法可以用到更普通的函数声明和定义上:

struct SomeStruct {

auto func\_name(int x, int y) -> int;

};

auto SomeStruct::func\_name(int x, int y) -> int {

return x + y;

}

　　关键字auto的这种用法与在自动类型推导中有所不同。  
　　**5、显示重写(覆盖)override和final**  
　　在C++03中，很容易让你在本想重写基类某个函数的时候却意外地创建了另一个虚函数。例如：

struct Base {

virtual void some\_func(float);

};

struct Derived : Base {

virtual void some\_func(int);

};

　　本来Derived::some\_func函数是想替代Base中那个函数的。但是因它的接口不同，又创建了一个虚函数。这是个常见的问题，特别是当用户想要修改基类的时候。  
　　C++11引入了新的语法来解决这个问题：

struct Base {

virtual void some\_func(float);

};

struct Derived : Base {

virtual void some\_func(int) override; *// 病态的,不会重写基类的方法*

};

　　override 这个特殊的标识符意味编译器将去检查基类中有没有一个具有相同签名的虚函数，如果没有，编译器就会报错！  
　　C++11还增加了防止基类被继承和防止子类重写函数的能力。这是由特殊的标识符final来完成的，例如：

struct Base1 final { };

struct Derived1 : Base1 { }; *// 病态的, 因为类Base1被标记为final了*

struct Base2 {

virtual void f() final;

};

struct Derived2 : Base2 {

void f(); *// 病态的, 因为虚函数Base2::f 被标记为final了.*

};

　　在这个例子中, virtual void f() final；语句声明了一个虚函数却也阻止了子类重写这个函数。它还有一个作用，就是防止了子类将那个特殊的函数名与新的参数组合在一起。  
　　需要注意的是，override和final都不是C++语言的关键字。他们是技术上的标识符，只有在它们被用在上面这些特定的上下文在才有特殊意义。用在其它地方他们仍然是有效标识符。  
　　**6、空指针常量 nullptr**  
　　NULL通常在C语言中预处理宏定义为(void\*)0或者0，这样0就有int型常量和空指针的双重身份。但是C++03中只允许0宏定义为空指针常量，这就会造成如下的错误：

void foo(int n);

void foo(char\* cArr);

　　上面声明了两个重载函数，当我调用foo(NULL)，编译器将会调用foo(int)函数，而实际上我是想调用foo(char\*)函数的。为了避免这个歧义，C++11重新定义了一个新关键字nullptr，充当单独空指针常量。  
　　**7、long long int类型**  
　　C++03中，最大的整数类型是long int。它保证使用的位数至少与int一样。 这导致long int在一些实现是64位的，而在另一些实现上却是32位的。C++11增加了一个新的整数类型long long int来弥补这个缺陷。它保证至少与long int一样大,并且不少于64位。这个类型早在C99就引入到了标准C中， 而且大多数C++编译器都以扩展的形式支持这种类型了。  
　　**8、模板的别名**  
　　在进入这个主题前,先弄清楚“模板”和“类型”的区别。类型，是具体的数据类型，可以直接用来定义变量。 模板，是类型的模板，根据这个模板可以产生具体的类型；模板是不能直接定义变量的；当指定了所有的模板参数后，就产生了一个具体的类型，就可以用来定义变量了。  
　　在C++03中，只能为类型（包括完全特化的模板，也是一种类型）定义别名，而不能为模板定义别名：

template <typename First, typename Second, int Third>

class SomeType;

template <typename Second>

typedef SomeType<OtherType, Second, 5> TypedefName; *// 在C++03中, 这是非法的.*

　　C++11增加为模板定义别名的能力,用下面这样的语法：

template <typename First, typename Second, int Third>

class SomeType;

template <typename Second>

using TypedefName = SomeType<OtherType, Second, 5>;

*//这种using语法也可以用来定义类型的别名:*

typedef void (\*FunctionType)(double); *// 老式语法*

using FunctionType = void (\*)(double); *// 新式语法*

**9、允许sizeof运算符可以再类型数据成员上使用，无需明确对象。**

struct p {otherClass member;};

sizeof(p::member);

**10、线程支持**  
　　C++11虽然从语言上提供了支持线程的内存模型,但主要的支持还是来自标准库。  
　　新的标准库提供了一个线程类(std::thread)来运行一个新线程，它带有一个函数对象参数和一系列可选的传递给函数对象的参数。通过std::thread::join()支持的线程连接操作可以让一个线程直到另一个线程执行完毕才停止。std:thread::native\_handle()成员函数提供了对底层本地线程对象的可能且合理的平台相关的操作。  
　　为支持线程同步,标准库增加了互斥体(std::mutex, std::recursive\_mutex等)和条件变量(std::condition\_variable 和std::condition\_variable\_any)。这些都是通过RAII锁和加锁算法就可以简单使用的。  
　　有时为了高性能或底层工作，要求线程间的通信没有开销巨大的互斥锁。原子操作可以达到这个目的,这可以随意地为一个操作指定最小的内存可见度。显式的内存屏障也可以用于这个目的。  
　　C++11线程库还包含了futures和promises，用于在线程间传递异步结果。并且提供了std::packaged\_task来封装可以产生这种异步结果的函数调用。  
　　更高级的线程支持，如线程池，已经决定留待在未来的 Technical Report 加入此类支持。更高级的线程支持不会是 C++11 的一部份，但是其最终实现将建立在目前已有的线程支持之上。std::async 提供了一个简便方法来运行线程，并将线程绑定在 std::future上。用户可以选择一个工作是要在多个线程上异步的运行，还是在一个线程上运行并等待其所需要的数据。默认的情况，实现可以根据底层硬件选择前面两个选项的其中之一。另外在较简单的使用场景下，实现也可以利用线程池提供支持。  
　　**11、元组类型**  
　　元组（tuple）由预先确定数量的多种对象组成，元组可以看作是struct数据成员的泛化，在Python中是一个基本数据结构。TR1 tuple类型的C++11版本获益于像可变参数模板这样的C++11语言特性。TR1版本的元组需要一个由实现定义的包含的类型的最大数目，而且需要大量的宏技巧来实现。相比之下，C++11版本的不需要显式的实现定义的最大类型数目。尽管编译器有一个内部的模板实例化的最大递归深度，但C++11版的元组不会把它暴露给用户。  
　　用可变参数模板,元组类的定义看上去像下面这样：

template <class ...Types> class tuple;

*//下面是定义和使用元组的一个例子:*

typedef std::tuple <int, double, long &, const char \*> test\_tuple;

long lengthy = 12;

test\_tuple proof (18, 6.5, lengthy, "Ciao!");

lengthy = std::get<0>(proof); *// 把'lengthy' 赋值为18.*

std::get<3>(proof) = " Beautiful!"; *// 修改元组的第四个元素*

**其它新增标准程序库**  
　　正则化表达式库<regex>；字符串类<string>字新增与其他类型互换的方法，如to\_string()，stoi()，stol等；STL标准模板库新增unordered\_map以及unordered\_set，基于hash表的关联容器等等。这些详细使用可自行上网查找，也十分重要的知识点。

　　以上内容参考以下文章总结：

<http://en.wikipedia.org/wiki/C++11>  
<http://www.cnblogs.com/pzhfei/archive/2013/03/02/CPP_new_feature.html#section_6.3>  
<http://www.cnblogs.com/haippy/archive/2013/05/31/3111560.html>