每个C++开发者都应该使用的十个C++11特性

作者 Marius Bancila, 2013年4月2日

这篇文章讨论了一系列所有开发者都应该学习和使用的C++11特性，在新的C++标准中，语言和标准库都加入了很多新属性，这篇文章只会介绍一些皮毛，然而，我相信有一些特征用法应该会成为C++开发者的日常用法之一。你也许已经找到很多类似介绍C++11标准特征的文章，这篇文章可以看成是那些常用特征描述的一个集合。

目录：

- auto关键字
- nullptr关键字
- 基于区间的循环
- Override和final
- 强类型枚举
- 智能指针
- Lambdas表达式
- 非成员begin()和end()
- static_assert宏和类型萃取器
- 移动语义

auto关键字

在C++11标准之前，auto关键字就被用来标识临时变量语义，在新的标准中，它的目的变成了另外两种用途。auto现在是一种类型占位符，它会告诉编译器，应该从初始化式中推断出变量的实际类型。当你想在不同的作用域中（例如，命名空间、函数内、for循环中中的初始化式）声明变量的时候，auto可以在这些场合使用。

```
auto i = 42;        // i is an int
auto l = 42LL;      // l is an long long
auto p = new foo(); // p is a foo*
```

使用auto经常意味着较少的代码量（除非你需要的类型是int这种只有一个单词的）。当你想要遍历STL容器中元素的时候，想一想你会怎么写迭代器代码，老式的方法是用很多typedef来做，而auto则会大大简化这个过程。

```
std::map<std::string, std::vector<int>> map;
for(auto it = begin(map); it != end(map); ++it)
{
}
```

你应该注意到，auto并不能作为函数的返回类型，但是你能用auto去代替函数的返回类型，当然，在这种情况下，函数必须有返回值才可以。auto不会告诉编译器去推断返回值的实际类型，它会通知编译器在函数的末段去寻找返回值类型。在下面的那个例子中，函数返回值的构成是由T1类型和T2类型的值，经过+操作符之后决定的。

```
template <typename T1, typename T2>
auto compose(T1 t1, T2 t2) -> decltype(t1 + t2)
{
   return t1+t2;
}
auto v = compose(2, 3.14); // v's type is double
```

nullptr关键字

0曾经是空指针的值，这种方式有一些弊端，因为它可以被隐式转换成整型变量。nullptr关键字代表值类型std::nullptr_t，在语义上可以被理解为空指针。nullptr可被隐式转换成任何类型的空指针，以及成员函数指针和成员变量指针，而且也可以转换为bool(值为false)，但是隐式转换到整型变量的情况不再存在了。

```
void foo(int* p) {}

void bar(std::shared_ptr<int> p) {}

int* p1 = NULL;
int* p2 = nullptr;
if(p1 == p2)
{
}

foo(nullptr);
bar(nullptr);

bool f = nullptr;
int i = nullptr; // error: A native nullptr can only be converted to bool or, using r
einterpret_cast, to an integral type
```

为了向下兼容，0仍可作为空指针的值来使用。

基于区间的循环

C++11加强了for语句的功能，以更好的支持用于遍历集合的“foreach”范式。在新的形式中，用户可

以使用for去迭代遍历C风格的数组、初始化列表，以及所有非成员begin()和end被重载的容器。

当你仅仅想获取集合/数组中的元素来做一些事情，而不关注索引值、迭代器或者元素本身的时候，这种for的形式非常有用。

```
std::map<std::string, std::vector<int>> map;
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
map["one"] = v;

for(const auto& kvp : map)
{
  std::cout << kvp.first << std::endl;

  for(auto v : kvp.second)
  {
     std::cout << v << std::endl;
  }
}

int arr[] = {1,2,3,4,5};
for(int& e : arr)
{
  e = e*e;
}
```

Override和final

我经常会发现虚函数在C++中会引起很多问题，因为没有一个强制的机制来标识虚函数在派生类中被重写了。virtual关键字并不是强制性的，这给代码的阅读增加了一些困难，因为你可能不得不去看继承关系的最顶层以确认这个方法是不是虚方法。我自己经常鼓励开发者在派生类中使用virtual关键

字，我自己也是这么做的，这可以让代码更易读。然而，有一些不明显的错误仍然会出现，下面这段代码就是个例子。

```
class B
{
public:
   virtual void f(short) {std::cout << "B::f" << std::endl;}
};

class D : public B
{
public:
   virtual void f(int) {std::cout << "D::f" << std::endl;}
};
```

D::f本应该重写B::f，但是这两个函数的签名并不相同，一个参数是short，另一个则是int，因此，B::f仅仅是另外一个和D::f命名相同的函数，是重载而不是重写。你有可能会通过B类型的指针调用f()，并且期盼输出D::f的结果，但是打印出来的结果却是B::f。

这里还有另外一个不明显的错误：参数是相同的，但是在基类中的函数是const成员函数，而在派生类中则不是。

```
class B
{
public:
   virtual void f(int) const {std::cout << "B::f " << std::endl;}
};

class D : public B
{
public:
   virtual void f(int) {std::cout << "D::f" << std::endl;}
};
```

又一次，这两个函数的关系是重载而非重写，因此，如果你想通过B类型的指针来调用f()，程序会打印出B::f，而不是D::f。

幸运的是，有一种方法可以来描述你的意图，两个新的、专门的标识符（不是关键字）添加进了C++11中：override，可以指定在基类中的虚函数应该被重写；final，可以用来指定派生类中的函数不会重写基类中的虚函数。第一个例子会变成：

```
class B
{
public:
   virtual void f(short) {std::cout << "B::f" << std::endl;}
};

class D : public B
{
public:
   virtual void f(int) override {std::cout << "D::f" << std::endl;}
};
```

这段代码会触发一个编译错误（如果你使用override标识符尝试第二个例子，也会得到相同的错误。）：

> 'D::f': 有override标识符的函数并没有重写任何基类函数

另一方面，如果你想要一个函数永远不能被重写（顺着继承层次往下都不能被重写），你可以把该函数标识为final，在基类中和派生类中都可以这么做。如果实在派生类中，你可以同时使用override和final标识符。

```
class B
{
public:
   virtual void f(int) {std::cout << "B::f" << std::endl;}
};
```

```
class D : public B
{
public:
   virtual void f(int) override final {std::cout << "D::f" << std::endl;}
};

class F : public D
{
public:
   virtual void f(int) override {std::cout << "F::f" << std::endl;}
};
```

用'final'声明的函数不能被'F::f'重写。

强类型枚举

“传统”的C++枚举类型有一些缺点：它会在一个代码区间中抛出枚举类型成员（如果在相同的代码域中的两个枚举类型具有相同名字的枚举成员，这会导致命名冲突），它们会被隐式转换为整型，并且不可以指定枚举的底层数据类型。

通过引入一种新的枚举类型，这些问题在C++11中被解决了，这种新的枚举类型叫做强类型枚举。这种类型用enum class关键字来标识，它永远不会在代码域中抛出枚举成员，也不会隐式的转换为整形，同时还可以具有用户指定的底层类型（这个特征也被加入了传统枚举类型中）。

```
enum class Options {None, One, All};
Options o = Options::All;
```

智能指针

有大量的文章介绍过智能指针，因此，我仅仅想提一提智能指针的引用计数和内存自动释放相关的东西：

- unique_ptr：当一块内存的所有权并不是共享的时候（它并不具有拷贝构造函数），可以使用，但是，它可以被转换为另外一个unique_ptr（具有移动构造函数）。
- shared_ptr：当一块内存的所有权可以被共享的时候，可以使用（这就是为什么它叫这个名）。
- weak_ptr：具有一个shared_ptr管理的指向一个实体对象的引用，但是并没有做任何引用计数的工作，它被用来打破循环引用关系（想象一个关系树，父节点拥有指向子节点的引用（shared_ptr），但是子节点也必须持有指向父节点的引用；如果第二个引用也是一个独立的引用，一个循环就产生了，这会导致任何对象都永远无法释放）。

换句话说，auto_ptr已经过时了，应该不再被使用了。

什么时候该使用unique_ptr，什么时候该使用shared_ptr，取决于程序对内存所有权的需求，我推荐你读一读这里的讨论。

下面第一个例子演示了unique_ptr的用法，如果你想要把对象的控制权转交给另一个unique_ptr，请使用std::move(我将会在最后一段讨论这个函数)。在控制权交接后，让出控制权的智能指针会变成null，如果调用get()，会返回nullptr。

```
void foo(int* p)
{
   std::cout << *p << std::endl;
}
std::unique_ptr<int> p1(new int(42));
std::unique_ptr<int> p2 = std::move(p1); // transfer ownership

if(p1)
  foo(p1.get());

(*p2)++;
```

```
if(p2)
  foo(p2.get());
```

第二个例子演示了shared_ptr的用法。尽管语义不同，因为所有权是共享的，但用法都差不多。

```
void foo(int* p)
{
}
void bar(std::shared_ptr<int> p)
{
   ++(*p);
}
std::shared_ptr<int> p1(new int(42));
std::shared_ptr<int> p2 = p1;

bar(p1);
foo(p2.get());
```

第一个声明等价于这个。

```
auto p3 = std::make_shared<int>(42);
```

make_shared是一个非成员函数，具有给共享对象分配内存，并且只分配一次内存的优点，和显式通过构造函数初始化的shared_ptr相比较，后者需要至少两次分配内存。这些额外的开销有可能会导致内存溢出的问题，在下一个例子中，如果seed()抛出一个异常，则表示发生了内存溢出。

```
void foo(std::shared_ptr<int> p, int init)
{
   *p = init;
}
foo(std::shared_ptr<int>(new int(42)), seed());
```

如果使用make_shared，则可以避开类似问题。第三个例子展示了weak_ptr的用法，注意，你必须通过调用lock()来获取shared_ptr中指向对象的引用，以此来访问对象。

```
auto p = std::make_shared<int>(42);
std::weak_ptr<int> wp = p;

{
  auto sp = wp.lock();
  std::cout << *sp << std::endl;
}

p.reset();

if(wp.expired())
  std::cout << "expired" << std::endl;
```

如果你试图在一个已经过期的weak_ptr上调用lock(被弱引用的对象已经被释放了)，你会得到一个空的shared_ptr。

Lambdas表达式

匿名的方法，也叫做lambda表达式，被加进了C++11标准里，并且立刻得到了开发者们的重视。这是一个从函数式语言中借鉴来的，非常强大的特征，它让一些其他的特征和强大的库得以实现。在任何函数对象、函数、std::function中出现的地方，你都可以用lambda表达式，你可以在这里阅读一下lambda的语法。

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);

std::for_each(std::begin(v), std::end(v), [](int n) {std::cout << n << std::endl;});
```

```
auto is_odd = [](int n) {return n%2==1;};
auto pos = std::find_if(std::begin(v), std::end(v), is_odd);
if(pos != std::end(v))
  std::cout << *pos << std::endl;
```

有一点复杂的是递归lambda表达式。想象一个代表斐波那契函数的lambda表达式，如果你试图用auto来写这个函数，你会得到编译错误：

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

```
error C3533: 'auto &': a parameter cannot have a type that contains 'auto'
error C3531: 'fib': a symbol whose type contains 'auto' must have an initializer
error C3536: 'fib': cannot be used before it is initialized
error C2064: term does not evaluate to a function taking 1 arguments
```

这个问题是由于auto会根据初始化式来推断对象类型，而初始化式却包含了一个引用自己的表达式，因此，仍然需要知道它的类型，这是一个循环问题。为了解决这个问题，必须打破这个无限循环，显式的用std::function来指定函数类型。

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

非成员begin()和end()

你也许已经注意到了，我在上面的例子中已经使用了非成员begin()和end()函数，这些是新加到STL中的东西，提升了语言的标准性和一致性，也使更多的泛型编程变成了可能，它们和所有的STL容器都是兼容的，但却不仅仅是简单的重载，因此你可以随意扩展begin()和end()，以便兼容任何类型，针对C类型数组的重载也一样是支持的。

让我们举一个前面写过的例子，在这个例子中，我试图打印输出一个vector，并且找到它的第一个奇数值的元素。如果std::vector用C风格数组来代替的话，代码可能会像如下这样：

```
int arr[] = {1,2,3};
std::for_each(&arr[0], &arr[0]+sizeof(arr)/sizeof(arr[0]), [](int n) {std::cout << n
<< std::endl;});

auto is_odd = [](int n) {return n%2==1;};
auto begin = &arr[0];
auto end = &arr[0]+sizeof(arr)/sizeof(arr[0]);
auto pos = std::find_if(begin, end, is_odd);
if(pos != end)
  std::cout << *pos << std::endl;
```

如果你使用非成员begin()和end()，代码可以这样写：

```
int arr[] = {1,2,3};
std::for_each(std::begin(arr), std::end(arr), [](int n) {std::cout << n << std::endl;
});

auto is_odd = [](int n) {return n%2==1;};
auto pos = std::find_if(std::begin(arr), std::end(arr), is_odd);
if(pos != std::end(arr))
  std::cout << *pos << std::endl;
```

这段代码基本上和使用std::vector那段代码一样，这意味着我们可以为所有支持begin()和end()的类型写一个泛型函数来达到这个目的。

```
template <typename Iterator>
void bar(Iterator begin, Iterator end)
{
   std::for_each(begin, end, [](int n) {std::cout << n << std::endl;});

   auto is_odd = [](int n) {return n%2==1;};
```

```
   auto pos = std::find_if(begin, end, is_odd);
   if(pos != end)
      std::cout << *pos << std::endl;
}

template <typename C>
void foo(C c)
{
   bar(std::begin(c), std::end(c));
}

template <typename T, size_t N>
void foo(T(&arr)[N])
{
   bar(std::begin(arr), std::end(arr));
}

int arr[] = {1,2,3};
foo(arr);

std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
foo(v);
```

static_assert宏和类型萃取器

static_assert会执行一个编译器的的断言，如果断言为真，什么都不会发生，如果断言为假，编译器则会显示一些特定的错误信息。

```
template <typename T, size_t Size>
class Vector
{
```

```
   static_assert(Size < 3, "Size is too small");
   T _points[Size];
};

int main()
{
   Vector<int, 16> a1;
   Vector<double, 2> a2;
   return 0;
}
```

```
error C2338: Size is too small
see reference to class template instantiation 'Vector<T,Size>' being compiled
   with
   [
      T=double,
      Size=2
   ]
```

当和类型萃取一起使用的时候，static_assert会变得更加有用，这些是一系列可以在编译期提供额外信息的类，它们被封装在了头文件里面，在这个头文件里，有若干分类：用来创建编译期常量的helper类，用来编译期获取类型信息的类型萃取类，为了可以把现存类型转换为新类型的类型转换类。

在下面那个例子里，add函数被设计成只能处理基本类型。

```
template <typename T1, typename T2>
auto add(T1 t1, T2 t2) -> decltype(t1 + t2)
{
   return t1 + t2;
}
```

然而，如果你这么写的话，并不会出现编译错误。

```
std::cout << add(1, 3.14) << std::endl;
std::cout << add("one", 2) << std::endl;
```

程序实际打印了4.14和“e”，但是如果我们添加一些编译器断言，这两行代码都会产生编译错误。

```
template <typename T1, typename T2>
auto add(T1 t1, T2 t2) -> decltype(t1 + t2)
{
   static_assert(std::is_integral<T1>::value, "Type T1 must be integral");
   static_assert(std::is_integral<T2>::value, "Type T2 must be integral");

   return t1 + t2;
}
```

```
error C2338: Type T2 must be integral
see reference to function template instantiation 'T2 add<int,double>(T1,T2)' being co
mpiled
   with
   [
      T2=double,
      T1=int
   ]
error C2338: Type T1 must be integral
see reference to function template instantiation 'T1 add<const char*,int>(T1,T2)' bei
ng compiled
   with
   [
      T1=const char *,
      T2=int
   ]
```

移动语义

这又是一个很重要，并且涉及到很多C++11技术特征的话题，关于这个话题不仅仅能写一段，更能写一系列文章。因此，我在这里并不会描述太多技术细节，如果你还没有对这个话题很熟悉，我会鼓励你去翻阅一些额外的资料。

为了区分指向左值的引用和指向右值的引用，C++11引入了右值引用（用&&来表示）的概念。左值是指一个有名字的对象，而右值则是一个没有名字的对象（临时对象）。移动语义允许修改右值（之前考虑到它的不可改变性，因此和const T& types的概念有些混淆）。

一个C++类/结构体有一些隐式成员函数：默认构造函数（当且仅当另外一个构造函数没有被显式的定义），拷贝构造函数，一个析构函数，以及一个拷贝赋值操作符。拷贝构造函数和拷贝赋值操作符一般会执行按位拷贝（或者浅拷贝），例如，逐一按位拷贝变量。这意味着如果你有一个包含指向某个对象的指针的类，它们只会把指针的地址进行拷贝，并不会拷贝指针指向的对象。这在某些情况下是可以的，但是对于绝大多数情况，你需要的是深拷贝，也就是对指针指向的对象进行拷贝，而不是指针本身的值，在这种情况下你不得不显式的写一个拷贝构造函数和拷贝赋值操作符来执行深拷贝。

那么，如果你想要初始化或者复制的源数据是个右值类型（临时的）会怎么样？你仍然不得不拷贝它的值，但是很快，这个右值就会消失，这意味着一些操作的开销，包括分配内存以及最后拷贝数据，这些都是不必要的。

我们引入了移动构造函数和移动赋值操作符，这两个特殊的函数接受一个T&&类型的右值参数，这两个函数可以修改对象，类似于把引用指向的对象“偷”来。举一个例子，一个容器的具体实现（例如vector或者queue）可能会包含一个指向数组元素的指针，我们可以为这些元素分配另一个数组空间，从临时空间中拷贝数据，然后当临时数据失效的时候再删除这段内存，我们也可以直接用这个临时的数据来实例化，我们只是拷贝指向数组元素的指针地址，于是，这节省了一次分配内存的开销，拷贝一系列元素并且稍后释放掉的开销。

下面这个例子展示了一个虚拟缓冲区的实现，这段缓冲区由一个名字标识（只是为了能更好的解释），有一个指针（用std::unique_ptr封装起来），指向一个类型为T的数组，也有一个存储数组大小的变量。

```
template <typename T>
```

```
class Buffer
{
   std::string          _name;
   size_t               _size;
   std::unique_ptr<T[]> _buffer;

public:
   // default constructor
   Buffer():
      _size(16),
      _buffer(new T[16])
   {}

   // constructor
   Buffer(const std::string& name, size_t size):
      _name(name),
      _size(size),
      _buffer(new T[size])
   {}

   // copy constructor
   Buffer(const Buffer& copy):
      _name(copy._name),
      _size(copy._size),
      _buffer(new T[copy._size])
   {
      T* source = copy._buffer.get();
      T* dest = _buffer.get();
      std::copy(source, source + copy._size, dest);
   }

   // copy assignment operator
   Buffer& operator=(const Buffer& copy)
   {
      if(this != ©)
      {
         _name = copy._name;
```

```
      if(_size != copy._size)
      {
         _buffer = nullptr;
         _size = copy._size;
         _buffer = _size > 0 > new T[_size] : nullptr;
      }

      T* source = copy._buffer.get();
      T* dest = _buffer.get();
      std::copy(source, source + copy._size, dest);
   }

   return *this;
}

// move constructor
Buffer(Buffer&& temp):
   _name(std::move(temp._name)),
   _size(temp._size),
   _buffer(std::move(temp._buffer))
{
   temp._buffer = nullptr;
   temp._size = 0;
}

// move assignment operator
Buffer& operator=(Buffer&& temp)
{
   assert(this != &temp); // assert if this is not a temporary

   _buffer = nullptr;
   _size = temp._size;
   _buffer = std::move(temp._buffer);

   _name = std::move(temp._name);
```

```
        temp._buffer = nullptr;
        temp._size = 0;

        return *this;
    }
};

template <typename T>
Buffer<T> getBuffer(const std::string& name)
{
    Buffer<T> b(name, 128);
    return b;
}
int main()
{
    Buffer<int> b1;
    Buffer<int> b2("buf2", 64);
    Buffer<int> b3 = b2;
    Buffer<int> b4 = getBuffer<int>("buf4");
    b1 = getBuffer<int>("buf5");
    return 0;
}
```

默认拷贝构造函数和复制赋值操作符应该看起来很类似，对于C++11标准来说，新的东西是根据移动语义设计的移动构造函数和移动赋值操作符。如果你运行这段代码，你会看到，当b4被构造的时候，调用了移动构造函数。而当b1被分配一个值的时候，移动赋值操作符被调用了，原因则是getBuffer()返回的值是一个临时的右值。

你可能注意到了一个细节，当初始化name变量和指向buffer的指针的时候，我们在移动构造函数中使用了std::move。name变量是一个字符串类型，std::string支持移动语义，unique_ptr也是一样的，然而，如果我们使用_name(temp._name)，复制构造函数将会被调用，但对于_buffer来说，这却是不可能的，因为std::unique_ptr并没有拷贝构造函数，但是为什么std::string的移动构造函数在这种情况下没有被调用?因为即使为Buffer调用移动构造函数的对象是一个右值类型，在构造函数的内部却实际是个左值类型，为什么？因为他有一个名字“temp”，而一个有名字的对象是左值类型。为了让它再一

次变成右值类型（也为了可以恰当的调用移动构造函数），我们必须使用std::move。这个函数的作用只是把一个左值类型的引用转换成右值类型引用。

更新：虽然这个例子的目的是展示下如何实现移动构造函数和移动赋值操作符，但实现的具体细节可能会有所不同，另外一个实现的方案是7805758成员在评论中提到的方法，为了能让大家更容易看到，我把它写在了正文中。

```
template <typename T>
class Buffer
{
   std::string          _name;
   size_t               _size;
   std::unique_ptr<T[]> _buffer;

public:
   // constructor
   Buffer(const std::string& name = "", size_t size = 16):
      _name(name),
      _size(size),
      _buffer(size? new T[size] : nullptr)
   {}

   // copy constructor
   Buffer(const Buffer& copy):
      _name(copy._name),
      _size(copy._size),
      _buffer(copy._size? new T[copy._size] : nullptr)
   {
      T* source = copy._buffer.get();
      T* dest = _buffer.get();
      std::copy(source, source + copy._size, dest);
   }

   // copy assignment operator
   Buffer& operator=(Buffer copy)
```

```
    {
        swap(*this, copy);
        return *this;
    }

    // move constructor
    Buffer(Buffer&& temp):Buffer()
    {
       swap(*this, temp);
    }

    friend void swap(Buffer& first, Buffer& second) noexcept
    {
        using std::swap;
        swap(first._name  , second._name);
        swap(first._size  , second._size);
        swap(first._buffer, second._buffer);
    }
};
```

结论

C++11包含了很多内容，以上内容只是一部分初步介绍，这篇文章文章展示了一系列C++核心技术以及标准库特征的用法，但是，我推荐你至少对其中一些特征去做一些额外、深入的阅读。

出处：Ten C++11 Features Every C++ Developer Should Use

c++ c++11 移动语义 智能指针

const_cast

发表于 2013-05-03 00:25

评论

时间 推荐

推荐 0

感觉C + + 越来越臃肿了。

zealotwjr

05-20 16:59

推荐 0

光 auto 那一段就毗漏不少，比如storage duration, trailing return type，这些都翻译错了或者被忽略了。请参考：http://blogs.ejb.cc/archives/7190/top-10-new-features-you-should-know-about-c-11

ray_linn

01-20 18:05

翻译中很多细节都经不住推敲。比如memory leak 不是内存溢出，是内存泄露。 – ray_linn 01-21 14:43

”之前考虑到它的不可改变性，因此和const T& types的概念有些混淆 “ 这完全是不懂C++的人瞎翻译 – ray_linn 01-22 14:35

推荐 0

越来越复杂，啰嗦，碎片化。

白龙

2013-05-03 22:49

C++标准化确实时间太长了。。总是需要新的编译器不断的兼容。 前两天我改一个SDK的bug，用了一个刚学到的C++11技巧，但code review的时候让一个老美的architect给拒绝了，原因是考虑到编译器兼容问题。。 – const_cast 2013-05-03 23:32

推荐

0

nullptr关键字下第一行 “被隐式转换成阵型变量” =》 整型变量

王腾超

2013-05-03 15:43

推荐

0

有些特征已经测试过了

lt

2013-05-03 09:22

特征-〉特性 – lt 2013-05-03 09:23

已更正。感谢。 – const_cast 2013-05-03 09:42

我要评论

需要登录后才能发言

记住我 登录