C++ 11：走近一点

迈克尔·乔达基斯

2012 年 3 月 25 日 警察

评价我： 4.88/5 (26 票)

了解 C++11 中的一些有趣功能。

下载 c11_2.zip - 1.5 KB

介绍

本文旨在帮助有经验的 C++ 用户理解和使用 C++ 11 中一些有趣的增强功能。我将讨论一些有趣的 C++ 11 特性，包括它们在 Visual Studio 11 Beta 和 g++ 中的可用性。

背景

需要非常好的C++知识。在这里，我们将只讨论新事物，它们大多是先进的。如果您不是专业的 C++ 开发人员，最好将本文留待以后使用。

靠近点

汽车

auto 关键字允许从函数返回中推断出变量的类型。这样您就不必显式指定类型。例如，这段代码：

复制代码

```
int foo() { return 5; }
int j = foo();
```

可以写成：

复制代码

```
int foo() { return 5; }
auto j = foo();
```

并且编译器会自动将 j 视为 int。

你可能会说大不了。不是这样。猜猜当你有一个时会发生什么`typename::user::object::foodump::longlongyum::iterator`：

复制代码

```
name::user::object::foodump::longlongyum::iterator foo() { return ... }
name::user::object::foodump::longlongyum::iterator j = foo();
for(name::user::object::foodump::longlongyum::iterator x = some.begin() ; x < some.end() ; x++)
{ ... }
```

所有这些都非常无聊且容易出错。使用 auto 使它变得更加简单和干净：

复制代码

```
name::user::object::foodump::longlongyum::iterator foo() { return ... }
auto j = foo();
for(auto x = some.begin() ; x < some.end() ; x++) { ... }
```

我的评价： 很有帮助。允许避免错别字。

Visual C++ 11：可用。

G++：可用。

常量表达式

该**constexpr**关键字允许您指定始终具有恒定的返回值的函数。考虑这个问题：

复制代码

```
int foo() { return 15; }
int j[foo()];
```

编译器错误。编译器无法知道 foo() 返回值是否为常量。

复制代码

```
constexpr int foo() { return 15; }
int j[foo()];
```

并且编译器知道这`foo()`将返回一个常量。

我的评价：不确定。`constexpr`函数可以做什么有很多限制，而且实际上 `int j[15]`比`int j[foo()]`. 然而，**constexpr**可以提示编译**器用**编译时计算替换运行时计算——尽管我还没有找到一个实际的例子。

Visual C++ 11：不可用。

G++：可用。

显式覆盖、最终、删除和默认

这些修饰符允许您在成员函数中显式：

- **覆盖**告诉编译器一个函数必须覆盖一个父类函数
- **final**告诉编译器一个函数不能被后代类函数覆盖
- **default**告诉编译器显式生成默认构造函数
- **delete**告诉编译器不能调用类的成员。

如果您想覆盖父函数，但不小心更改了函数签名，则**覆盖**很有用，因此会生成第二个函数：

复制代码

```
class a
  {
  public:
```

```
   virtual void foo(int) {...}
  };
class b : public a
  {
  public:
   virtual void foo() override; // Compiler error. void foo() has not replaced void foo(int)
  }
```

这里程序员缩进覆盖`a::foo()`有`b::foo()`，但没有覆盖指示创建一个新的虚拟函数来代替，因为程序员忘记了原来`foo()`接受int作为参数。

如果你想确保一个函数永远不会被覆盖，**final**很有用：

复制代码

```
class a
  {
  public:
   virtual void foo(int) final {...}
  };
class b : public a
  {
  public:
   virtual void foo(int) {}; // Compiler error. void foo(int) is final.
  }
```

default允许程序员告诉编译器应该自动生成默认构造函数，尽管可能还有其他构造函数：

复制代码

```
class a
  {
  public:
   a() = default;
   a(int) { ... }
  };
```

delete允许程序员明确限制对成员的调用，例如由于不需要的强制转换或复制构造函数而被调用的函数：

复制代码

```
class a
  {
  public:
   a(const a&) = delete;
   void x(int) = delete;
   void x(float);
  };
```

在这里，类复制构造函数不能被调用，并且调用`a::x()`必须显式传递一个浮点数（例如你不能调用`x(0)`）。

我的评价：有用。它有助于避免不必要的错误。

Visual C++ 11：不可用。

外部模板

模板中的关键字**extern**告诉编译器不要实例化模板：

复制代码

```
template <typename X> class a
  {
  public:
   void test(X);
  };
```

```
template class a<int>; // C++ 03 instantiation
extern template class a<int> // C++ 11 won't instantiate.
```

C++ 11 不会实例化“a”，但 a 将对当前编译单元可见。

我的评价：有用。它有助于避免在多个 cpp 文件中进行不必要的实例化。

Visual C++ 11：不可用。

哈希表

C++ 11 具有无序关联容器，称为**哈希表**。这些容器使用散列函数来存储元素。

复制代码

```
void test_hash()
    {
    unordered_map<string,int> days(7);
    const char* str[] = {"Sun","Mon","Tue","Wed","Thu","Fri","Sat"};
    for(int i = 0 ; i < 7 ; i++)
       days[str[i]] = i;
    auto j = days["Wed"]; // j = 3;
    }
```

我的评价：如果你需要它很有用。

Visual C++ 11：可用。

G++：可用。

初始化列表

最后，C++ 缺少的一件事。使用 {} 初始化非 POD 对象的能力。例如，这有效：

复制代码

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

但这不是：

复制代码

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

必须以一种麻烦的方式分配值，例如使用 for 循环。C++ 11 允许一个特殊的模板调用`initializer_list`，它可以通过 {} 获取它的值，并且这可以用作任何函数中的成员参数，构造函数与否：模板`<typename X>`类 A

复制代码

```
template <typename X> class A
    {
    public:
        A(int x);
        A(std::initializer_list<int> y);
        void foo(std::initializer_list<X> list);
    };
A<int> a(5); // construct using A(int);
A<float> b({1,2,3,4,5,6}); // construct using A(std::initializer_list<int> y);
b.foo({0.5f,0.6f}); // call foo with parameters.
```

我的评价：非常有用。

Visual C++ 11：不可用 :(。

G++：可用。

Lambda 函数

甲**lambda函数**是能够另一个函数内使用的匿名函数。

语法：[capture][arguments]{body}(<调用参数>)

捕获是将在函数外部的主体（例如函数）中使用的标识符列表。如果标识符不在捕获列表中，则任何在 lambda 函数之外使用标识符的尝试都会产生编译器错误。

复制代码

```
vector<int> flist(15);
int sum = 0;
for(int i= 0 ; i < 15 ;i++)
    flist[i] = i;
for(int i = 0 ; i < 15 ; [&sum](int x) { sum += x;}(flist[i++]));
// sum = 105.
```

[&sum](int x) { sum += x;} 是 lambda 函数。它通过引用捕获外部变量的“总和”，并将作为参数传递的“x”添加到其中。在 for 循环中使用 flist[i] 立即调用它。

您不需要立即调用它：

复制代码

```
auto f = [] (int x, int y) { return x + y; };
int j = f(5,6); // j = 11;
```

我的评价：有用，但要小心，因为它很容易导致难以维护的代码。

Visual C++ 11：可用。

G++：可用。

空指针

所述**nullptr**（的类型恒定**nullptr_t**）允许用户指定必须不从零被隐式转换指针类型。

考虑这个问题：

复制代码

```
int foo(int x) { ... }
int foo(char* a) { ... }
foo(NULL);
```

调用`foo(NULL)`将调用`foo(int)`，而不是`foo(char*)`，因为 NULL 被定义为 0。

复制代码

```
int foo(int x) { ... }
int foo(char* a) { ... }
foo(nullptr);
```

现在，`foo(char*)`将被调用。nullptr 可以隐式转换为任何其他指针类型，但不能转换为任何整型 int 类型（bool 除外）。int j = nullptr 生成编译器错误。

我的评价：几乎没用。使用整数类型和指针的重载函数是非常罕见的，如果有，则您的设计（通常）存在缺陷。

Visual C++ 11：可用。

G++：可用。

多态函数指针

它们类似于函数指针，不同之处在于它们允许隐式转换它们的参数。例如：

复制代码

```
int foo(int x) { return x + 1;}
typedef int (*foop)(short y);
foop f = foo; // Error. It is int foo(int), not int foo(short);
```

使用多态指针可以完成这样的转换：

复制代码

```
std::function<bool (int)> foo;
bool f2(short a) { ... }
foo = f2; // Valid. short can be converted to int
bool A = foo(5);
```

我的评价：有用，如果你知道自己在做什么。C++ 是强类型的，弱化该特性会导致问题。

Visual C++ 11：不可用。

G++：可用。

常用表达

您可能从**IRegExp**知道的函数现在位于标准头文件 <regex> 中：

复制代码

```
char* str = "<h2>Header</h2>";
regex rx("<h(.)>([^<]+)");
cmatch res;
eegex_search(str, res, rx); // isolate "Header"
```

当然，正则表达式是一个独立的章节，我们不能在这里讨论。但很高兴将其视为标准。

我的评价：有用，如果你需要正则表达式。

Visual C++ 11：可用。我确定他们使用旧的 VB 的 IRegExp ;)
G++：可用。

成员大小

在 C++ 11 中，**sizeof**可用于获取类成员的大小，而无需类实例：

复制代码

```
class a {
  public
   int b;
  };
int x = sizeof(a::b); // Valid in C++ 11. In C++ 03 you had to have an instance of a class.
```

我的评价：必填。

Visual C++ 11：不可用。

静态断言

该**断言**关键字可以在运行时被用来测试一个断言，并且#ERROR预处理指令可在预处理器中使用。新关键字**static_assert**可用于编译器：

复制代码

```
int j = 0;
static_assert(j,"error!"); // Compiler error: compilation failed.
```

我的评价：可能有用，但不是很重要。

Visual C++ 11：可用。

G++：可用。

字符串文字

到目前为止，您可以分别使用 "string" 和 L"string" 来使用普通字符串和宽字符串。C++ 11 添加了以下（非常有用的）文字：

- **u8"string"**定义一个 UTF-8 字符串。
- **u"string"**表示 UTF-16 字符串。
- **U"string"**表示 UTF-32 字符串。
- **R"string"**表示原始字符串。

使用**R"string"**允许您从源复制/粘贴字符串，而无需转义特殊字符，例如 \。

复制代码

```
char* a1 = "Hello there, you can use the \ and / characters to pass arguments"; // Ugh,
probably a bug, we forgot to escape \
char* a2 = R"Hello there, you can use the \ and / characters to pass arguments"; // Fine.
```

我的评价：必须的，尤其是R"string"。

Visual C++ 11：不可用。GRRRR。

G++：可用。

强枚举

不输入枚举，例如：

复制代码

```
enum test {a = 0,b,c,d,e}; // implementation of the enum and it's size is compiler-dependant
int j = e; // valid.
bool h = d; // valid.
```

现在可以输入：

复制代码

```
enum class test: unsigned long {a = 0,b,c,d,e}; // implementation of the enum now is with
unsigned long; Guaranteed size and type.
int j = -1;
if (j < test::c) //warning, comparing signed to unsigned.
```

我的评价：好。

Visual C++ 11：不可用。
G++：可用。

线程

C++ 11 中添加了线程类（包括像互斥锁这样的同步内容）。

复制代码

```
void test_thread_func(const char* message)
    {
```

```
    // Do something with parameter
    }
void test_thread()
    {
    thread t(bind(test_thread_func,"hi!"));
        t.join();
    }
```

这将在一个单独的线程中启动“test_thread_func”并等待它使用 join() 函数完成。还有许多其他功能，例如互斥锁、线程本地存储支持、原子操作等。

我的评价：很好，尽管您通常需要针对严重线程模型的特定于操作系统的扩展。幸运的是，有一个 native_handle() 成员函数返回本机句柄（在 Win32 中为 HANDLE），因此您可以在没有 C++ 替代方案时使用它。

Visual C++ 11：可用。
G++：不可用（至少我无法编译）。

Unique_Ptr 和更多智能指针

unique_ptr（C++ 11 中的新功能）、**shared_ptr**和**weak_ptr**可用。**unique_ptr**以可以移动指针的方式扩展（现已弃用）**auto_ptr**：

复制代码

```
unique_ptr<int> p1(new int(100));
unique_ptr<int> p2 = move(p1); // Now p2 owns the pointer.

p2.reset(); // memory is freed
p1.reset(); // does nothing.
```

我的评价：很好，虽然你不应该太依赖自动指针来摆脱困境。使用分配器类来避免 new 和 delete 很好，但不要过度使用，因为在我看来已经有太多 xxxx_ptr 类了。不要陷入依赖智能指针来增强设计薄弱的应用程序的陷阱。

另外，请注意，许多人投票支持使用智能指针，以便在发生异常时可以安全地销毁内存，这样您就可以继续而不会泄漏。记住什么是异常应该是：一个错误，你 99% 不应该正常继续。不要使用异常代替**if**或**switch**。不要在密码无效、文件未找到等错误时抛出异常。不要捕获“内存不足”异常。您的代码已经存在缺陷，或者 Windows 非常不稳定！

Visual C++ 11：可用。

G++：可用。

无限制工会

不受限制的工会允许在什么可以成为工会和什么不能成为工会方面有更多的自由：

复制代码

```
class A
   {
   int b;
   int c;
   A() { b =5; c = 4;}
   };
union B
   {
   int j;
   A a; // Invalid in C++ 03: A has a non trivial constructor. Valid in C++ 11:
   };
```

我的评价：危险。联合主要用于较低级别的功能，如 C 位操作、汇编等。它们**应该**受到非常严格的限制。

Visual C++ 11：不可用。

变量对齐

到现在为止，您可以使用特定于编译器的 #pragmas 来指定对齐方式。C++ 11 允许 alignas 关键字：

复制代码

```
alignas(double) int d[8];  // d contains 8 integers, but it's size is enough to contain 8
doubles also.
```

此外，alignof 运算符返回标识符的对齐方式，例如 int j = alignof(d) 将返回 sizeof(double)。

我的评价：有用。

Visual C++ 11：不可用。
G++：不可用。

可变模板

我们文章中的最后一个条目讨论了可变参数模板。这些是可以接受任意数量和任意类型参数的模板：

复制代码

```
template <typename ...T> void foo(T...args)
```

这允许像 printf 这样的函数的类型安全实现。下面的示例检查参数的大小并使用适当的参数调用 func1()：

复制代码

```
void func1(int a1, int a2, int a3)
 { int j = a1 + a2 + a3;}
template<class...T> int funcV(A...args)
 {
 int size = sizeof...(A);
 if (size == 0) func1(1,2,3);
 if (size == 1) func1(1,2,args...);
 if (size == 2) func1(1,args...);
 if (size == 3) func1(args...);
 }
int main()
 {
 funcV(0);
 funcV(0,1,2);
 funcV(5,3);
 funcV("hello"); // error, func1 does not accept a char*.
 }
```

由于“args”对函数来说不容易获得（因为它们的类型和大小是未知的），它们通常被递归调用。

我的评价：有用，但它们真的很容易导致代码难以理解。谨慎使用。

Visual C++ 11：不可用。

G++：不可用。

代码

随附的 C11.cpp 包含所有 Visual C++ 11 Beta 和 G++ 可用的 C++ 11 内容的示例。如果您发现更多支持的功能和/或错误，请通知我！

玩得开心！

历史

- 24 - 03 - 2012：添加了 G++ 支持
- 2012 年 11 月 3 日：首次发布

执照

分享

关于作者

迈克尔·乔达基斯
软件开发人员
希腊
手表该会员

我正在使用 C++、PHP、Java、Windows、iOS、Android 和 Web（HTML/Javascript/CSS）。

我拥有数字信号处理和人工智能博士学位，专攻专业音频和人工智能应用。

我的主页：https://www.turbo-play.com

评论和讨论

添加评论或问题 电子邮件提醒 Search Comments

第一 页上一页 下一页

让我知道像 foreach 这样的 cx11 特性什么时候会成为 g++ 的一部分。
sharad sahu 25-Jul-12 14:52

我的4票
Gorpik 29-Mar-12 17:13

My vote of 5
Keith.Badeau 28-Mar-12 4:41

Re: My vote of 5
Michael Chourdakis 28-Mar-12 20:58

Re: My vote of 5
Keith.Badeau 29-Mar-12 11:32

Good article
giulicard 25-Mar-12 4:47

Re: Good article
Michael Chourdakis 25-Mar-12 5:00

Re: Good article
giulicard 25-Mar-12 5:12

Visual C++ 11
Dewey 25-Mar-12 4:36

Re: Visual C++ 11
Michael Chourdakis 25-Mar-12 4:55

Re: Visual C++ 11
AndreMK 27-Mar-12 3:03

Re: Visual C++ 11
Michael Chourdakis 27-Mar-12 3:12

Re: Visual C++ 11
AndreMK 28-Mar-12 1:01

My vote of 3
peterchen 25-Mar-12 0:15

Re: My vote of 3
Michael Chourdakis 25-Mar-12 0:29

Re: My vote of 3
giulicard 25-Mar-12 4:54

Re: My vote of 3
Michael Chourdakis 25-Mar-12 4:58

Re: My vote of 3
AndreMK 15-Feb-13 10:55

Re: My vote of 3
Nemanja Trifunovic 25-Mar-12 1:43

Re: My vote of 3
Michael Chourdakis 25-Mar-12 2:18

Re: My vote of 3
AndreMK 27-Mar-12 2:46

Re: My vote of 3
Michael Chourdakis 27-Mar-12 3:02

Re: My vote of 3
AndreMK 28-Mar-12 0:53

Re: My vote of 3
Dewey 25-Mar-12 4:34

About "Unique_Ptr and More Smart Pointers"
Cholo 13-Mar-12 23:42

一般 新闻 建议 问题 错误 答案 笑话 赞美 咆哮 管理员

使用Ctrl+Left/Right 切换消息，Ctrl+Up/Down 切换主题，Ctrl+Shift+Left/Right 切换页面。

永久链接
广告
隐私
Cookie
使用条款

布局：固定 | 体液