C++11 – 概览 [n 的第 1 部分]

拉卡姆拉朱拉古拉姆

2019 年 9 月 22 日 警察

C++11 标准概览，涵盖了 VS2010 支持的特性

介绍

C++0x 现在是一个正式的标准，以后将被称为**C++11**。它于 2011 年被 ISO C++ 批准。

这篇文章的目的是鸟瞰大部分C++11特性，对于那些已经发布到VS2010中的特性，给出了深入的分析。这篇文章可以作为一个平台，开始对个别特征进行综合研究。

文章被分成几部分，所以长度是在海湾。我自己害怕阅读冗长的东西。此外，一口气抓住所有东西会很无聊。

由于这是我关于 CodeProject 的第一篇文章，请注意格式和拼写错误（如果有）。

背景

查尔斯·巴贝奇的差分机用了将近一个世纪的时间才演变成一台电动计算机。在 1940 年代，由于当时的计算机速度和内存容量低，只使用汇编级语言。十年后情况开始发生转变，在 1950 年至 1970 年期间，许多编程语言蓬勃发展，其中许多语言幸存至今。

1979 年，为贝尔实验室工作的 Bjarne Stroustrup 开始增强“C”语言，首先添加类，然后添加虚拟函数、运算符重载、多重继承、模板和异常处理等功能。他最初称它为“C with Classes”。1983年改名为C++[++可能是说增加了C]。

C++ 地标/迭代时间线

- 1983 - 第一个商业 C++ 编译器
- 1998 - C++ 标准委员会标准化 C++ [C++98]
- 2003 - 一个没有新特性的错误补丁 [C++03]
- 2005 - 发布了名为“图书馆技术报告”（简称TR1）的技术报告
- **2011 - 引入了大量功能并增强了 C++ 标准库。**

我们可以看到，这次迭代是最大的一次（好吧……好吧……STL 的添加也可能很大）。

我们必须知道这个新标准吗？？

是的！越早越好。抵制变化是人类。但是，当每种语言或项目达到静态阶段时，我们程序员/开发人员就会失业。我们喜欢参与动态项目，语言也是如此。

改变是不可避免的，当专家委员会集思广益近十年时，结果显然是美好而富有成果的。

即使我们对在代码中加入这些新的迭代不感兴趣，快速浏览一下这些特性将帮助我们避免甚至在使用旧编译器编码某些场景之前进行思考。此外，只需切换到支持 C++11 的编译器，我们就会受益匪浅，因为标准模板库已针对性能进行了增强和改进。因此，如果您的项目正在使用 STL 容器/算法，那么请尽早切换。

C++11 特性

这是一个总结核心 C++11 特性及其在 VS2010 中的实现状态的表格。

特征	意图	VS2010 状态
• 汽车	可用性改进	是的
• 声明类型	可用性改进	是的
• 尾随回报类型	可用性和性能改进	是的
• 右尖括号	可用性改进	是的
• 静态断言	可用性改进	是的
• 空指针	可用性和性能改进	是的
• 强类型枚举	类型安全改进	部分的
• 右值引用	性能改进	是的
• 移动语义	性能改进	是的
• 长长的	性能改进	是的
• 覆盖控制	可用性改进	是的
• Lambda 表达式	可用性和性能改进	是的
• 防止变窄	性能改进	不
• 基于范围的 for 循环	可用性和性能改进	不
STL 增强		
• 大批		是的
• 真正智能的指针 [unique_ptr、shared_ptr、weak_ptr]		是的
• 绑定和函数		是的
• 元组		是的
• unordered_map, unordered_set		是的
• forward_list		是的

..... 还有很多

个人特色

自动关键字

引入此功能是为了使 C++ 成为一种更有用的语言。委员会给' auto'关键字赋予了新的含义[只是为了提醒读者，旧auto关键字用于声明具有局部作用域的变量，所有未指定存储类为static、extern或register的局部变量都被隐式转换到auto存储类说明符]。

根据新的解释，auto通过检查 RHS 表达式或其初始值设定项，有助于推断定义对象的类型。

C++ 复制代码

```
auto i = 5      // i will be of type int

int n=3;
double pi=3.14;
auto j=pi*n;    // j will be of type double
```

现在，让我们来看一个类型**很难写的情况**：

C++ 复制代码

```
// take a hypothetical Map of ( int and an map(int,int) )
map< int, map<int,int> > _Map;
// see the verbose for defining a const iterator of this map
map<int, map<int,int>>::const_iterator itr1 = _Map.begin();
// now with auto our life gets simplified
const auto itr2 = _Map.begin();
```

现在，举一个**很难知道**类型的情况：

C++ 复制代码

```
template<class U, class V>
void Somefunction(U u, V v)
{
    ??? result = u*v; // now what would be the type of result ???

    // with auto we leave the compiler to determine the type
    auto result = u*v;
 }
```

我将为下一个功能扩展此功能，以了解auto. 的auto关键字的发现更多的使用，同时声明和初始化到一个lambda表达式一个变量[我们将lambda表达式很快覆盖。

关于此功能的几点：

1. 我们可以在关键字上使用const, volatile, 指针 (*), 引用 (&),rvalue引用 (&& - 我们很快就会知道这一点) 说明符auto：

   C++ 复制代码

   ```
   auto k = 5;
   auto* pK = new auto(k);
   auto** ppK = new auto(&k);
   const auto n = 6;
   ```

2. 声明为的变量auto必须有一个初始值设定项：

   C++ 复制代码

   ```
   auto m; // m should be initialized
   ```

3. 一个auto关键字不能与另一种类型被接合：

   C++ 复制代码

```
auto int p; // no way
```

4. 方法/模板参数不能声明为 auto：

C++ 复制代码

```
void MyFunction(auto parameter){} // no auto as method argument

template<auto T>                  // utter nonsense - not allowed
void Fun(T t){}
```

5. 在堆上用auto关键字 using 表达式声明的变量必须有一个初始值设定项：

C++ 复制代码

```
int* p = new auto(0);  //fine
int* pp = new auto();  // should be initialized

auto x = new auto();   // Hmmm ... no initializer

auto* y = new auto(9); // Fine. Here y is a int*
auto z = new auto(9);  //Fine. Here z is a int* (It is not just an int)
```

6. 由于auto关键字是类型的占位符，但不是类型本身，因此auto不能用于类型转换或运算符，例如sizeofand typeid。

C++ 复制代码

```
int value = 123;
auto x2 = (auto)value; // no casting using auto

auto x3 = static_cast<auto>(value); // same as above
```

7. 使用auto关键字声明的声明符列表中的所有变量必须解析为相同的类型：

C++ 复制代码

```
auto x1 = 5, x2 = 5.0, x3='r';  // This is too much....we cannot combine like this
```

8. 除非声明为引用，否则 Auto 不会推断 CV 限定符（常量和易失性限定符）：

C++ 复制代码

```
const int i = 99;
auto j = i;       // j is int, rather than const int
j = 100           // Fine. As j is not constant

// Now let us try to have reference
auto& k = i;      // Now k is const int&
k = 100;          // Error. k is constant

// Similarly with volatile qualifer
```

9. auto 除非声明为引用，否则将数组衰减为指针：

C++ 复制代码

```
int a[9];
auto j = a;
cout<<typeid(j).name()<<endl; // This will print int*

auto& k = a;
cout<<typeid(k).name()<<endl; // This will print int [9]
```

decltype 类型说明符

return_value 声明类型（表达式）

[*return_value*是表达式参数的类型]

这可用于确定表达式的类型。正如 Bjarne 所暗示的那样，如果我们只需要我们将要初始化的变量的类型，`auto`通常是一个更简单的选择。但是如果我们需要一个不是变量的类型，比如返回类型，那么`decltype`这就是我们应该尝试的事情。

现在让我们回顾一下我们之前工作过的示例：

C++ 复制代码

```
template<class U, class V>
void Somefunction(U u, V v)
{
   result = u*v;               // now what would be the type of result ???

   decltype(u*v) result = u*v; // Hmm ....we got what we want
}
```

在下一节中，我会让你熟悉结合 auto 和`decltype`声明模板函数的概念，模板函数的返回值类型取决于它的模板参数。

几点`decltype`：

1. 如果表达式是函数，则`decltype`给出函数返回的类型：

   C++ 复制代码

   ```
   int add(int i, int j) { return i+j; }

   decltype( add(5,6) ) var = 5; // Here the type of var is return of add( ) -> which is int
   ```

2. 如果表达式是 an `lvalue`，则`decltype`给出`lvalue`对表达式类型的引用。

   C++ 复制代码

   ```
   struct M { double x; };

   double pi = 3.14;
   const M* m = new M();
   decltype( (m->x) ) piRef = pi;

       // Note: Due to the inner brackets the inner statement is evaluated as expression,
       // rather than member 'x' and as type of x is double and as this is lvale
       // the return of declspec is double& and as 'm' is a const pointer
       // the return is actually const double&.
       // So the type of piRef is const double&
   ```

3. 重要的是要注意，`decltype`它不会像`auto`那样计算表达式，而只是推断表达式的类型：

   C++ 复制代码

   ```
   int foo(){}
   decltype( foo() ) x; // x is an int and note that
                        // foo() is not actually called at runtime
   ```

尾随回报类型

这对于 C++ 开发人员来说是一个全新的特性。到目前为止，函数返回值的类型应该放在函数名之前。从 C++11 开始，我们也可以将返回类型放在函数声明的末尾，当然只有在将返回类型的名称替换为 auto 之后。现在我们为什么要这样做。让我们来了解一下：

C++ 复制代码

```
template<class U, class V>
??? Multiply(U u, V v)    // how to specifiy the type of the return value
{
   return u*v;
}
```

我们显然不能这样做：

C++ 复制代码

```
template<class U, class V>
decltype(u*v) Multiply(U u, V v)    // Because u & v are not defined before Multiply.
                    //  What to do...what to do !!!
{
   return u*v;
}
```

在这种情况下，我们可以使用auto然后后者一旦u&v的定义已知，我们可以使用指定返回类型decltype。
很酷，不是吗？

C++ 复制代码

```
template<class U, class V>
auto Multiply(U u, V v) -> decltype(u*v)    // Note -> after the function bracket.
{
   return u*v;
}
```

直角括号

看看这个声明：

C++ 复制代码

```
map<int, vector<int>> _Map;
```

这是早期编译器的错误，因为>s之间没有空格，编译器会将其视为右移运算符。

但是 C++11 编译器会将这些多个右尖括号解析为接近模板参数列表，从而使我们无需在>.

与其他功能相比，这不是一个很好的功能，但是当我们 C++ 开发人员寻求完美时，这里是一个值得关注的功能。

静态断言

此宏可用于检测和诊断编译时错误。编译期。这与 CRT-assert 宏形成对比，后者是运行时的断言。这个好东西可用于在编译时检查程序不变量。

这需要一个可以评估为的表达式bool和一个string文字。如果表达式的计算结果为false，则编译器会发出包含指定string文字的错误并且编译失败。如果true，则声明无效。

我们可以static_assert在：

1. 命名空间/全局范围：

   C++ 复制代码

   ```
   static_assert(sizeof(void *) == 4, "Oops...64-bit code generation is not supported.");
   ```

2. 班级范围：

   C++ 复制代码

   ```
   template<class T, int _n>

   class MyVec
   {
       static_assert( _n > 0 , "How the hell the size of a vector be negative");
   };

   void main()
   {
       MyVec<int, -2> Vec_;
       // The above line will throw error as shown below ( in VS2010 compiler):
   ```

```
    //   > \main_2.cpp(120) : error C2338: How the hell the size of a vector be negative
    //   > main_2.cpp(126) : see reference to class template instantiation 'MyVec<t,_n />'
    //     being compiled
    //   > with
    //   > [
    //         > T=int,
    //       > _n=-2
    //   > ]

    // This is fine
        MyVec<int, 100> Vec_;
}
```

3. 块范围：

C++ 复制代码

```
template<typename T, int div>
void Divide( )
{
   static_assert(div!=0, "Bad arguments.....leading to division by zero");
}

void main()
{
   Divide<int,0> ();
   // The above line will generate
   // error C2338: Bad arguments.....leading to division by zero
}
```

请记住，由于static_assert是在编译时计算的，因此不能用于检查依赖于运行时值（如函数参数）的假设：

C++ 复制代码

```
void Divide(int a, int b)
{
   static_assert(b==0, "Bad arguments.....leading to division by zero");
   // sorry mate! the above check is not possible via static_assert...use some other means
}
```

该static_assert声明对于调试模板特别有用。如果常量表达式参数不依赖于模板参数，则编译器会立即对其求值。否则，编译器会在实例化模板时评估常量表达式参数。

空指针

引入此功能主要是为了解决使用（臭名昭著且令人讨厌的）NULL宏产生的陷阱。众所周知， NULL它只不过是0在编译时扩展为的预处理器，而这种扩展通常会导致歧义。举个例子：

C++ 复制代码

```
void SomeFunction(int i){ }

void SomeFunction(char* ch) { }
```

现在像: 这样的调用SomeFunction(NULL)将始终解析为SomeFunction(int i)，即使我们想SomeFunction(char* ch)使用null指针参数进行调用。

为了强制，我们必须像这样调用： SomeFunction((char*) NULL) // yak ..ugly

为了避免这些不便，nullptr最后引入了。的nullptr字面意思null指针不是整数。因此，这可以安全地用于指示对象句柄、内部指针或本机指针类型不指向对象。

本系列的第二部分将介绍一些功能。请参阅“ C++11 – 概览 [n 的第 2 部分] ”。

其余功能将在接下来的部分中介绍。

感谢您阅读这篇文章。如果您评价/发送反馈会很有帮助，这样我就可以在处理剩余部分或用新信息更新这部分时进行改进。

其他来源

由于该标准仅在 3 个月前冻结，因此没有描述新功能的书籍。以下是一些有助于深入了解所有功能的参考资料。

- http://www.open-std.org/jtc1/sc22/wg21/docs/papers/

致谢

感谢 Clement Emerson 的观点和评论。

历史

- 2012 年 1 月 9 日：添加了简介和第 1 部分
- 2012 年 1 月 14 日：添加了第 2 部分的链接，“ C++11 – 概览 [n 的第 2 部分] ”
- 2012 年 1 月 15 日：更正了拼写错误
- 2012 年 1 月 21 日：更正了几个断开的链接 [无附加信息]
- 2012 年 1 月 25 日：向`auto`和添加了几点`decltype`（如用户“ephisino”所述）
- 2012 年 2 月 3 日：更正了几个错别字 [无附加信息]

执照

分享

关于作者

拉卡姆拉朱拉古拉姆
技术负责人
印度

手表
该会员

我从 IIT-M 获得了先进制造技术的硕士学位，从 2004 年起主要从事 CAD 领域的 C++ 工作。从 2015 年开始使用 Angular 7.0 及更高版本、HTML5、CSS3 研究 Web 技术。

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

我的5票
steppeman 26-Sep-19 2:35

我的5票
Mallesh Kumar 23-Sep-19 17:08

返回类型
Сергій Ярошко 19-Jun-18 15:42

回复：返回类型
Lakamraju Raghuram 22-Sep-19 1:21

Nice
Manikandan10 22-May-14 15:53

My vote of 5
Member 4289613 20-Aug-13 16:21

Re: My vote of 5
Lakamraju Raghuram 26-Aug-13 12:21

Re: My vote of 5
Member 4289613 26-Aug-13 18:36

My vote of 5
aydinsahin 29-Jun-13 5:35

My vote of 5
François DAGUER 15-Feb-13 16:46

Should use cbegin() to get const_iterator
qzq 26-Jun-12 11:33

Re: Should use cbegin() to get const_iterator
Lakamraju Raghuram 3-Jul-12 19:01

My vote of 5
Manoj Kumar Choubey 29-Feb-12 11:54

My vote of 5
Ștefan-Mihai MOGA 9-Feb-12 0:29

Re: My vote of 5
Lakamraju Raghuram 9-Feb-12 1:05

My vote of 5
Ashish Tyagi 40 8-Feb-12 18:11

Re: My vote of 5
Lakamraju Raghuram 8-Feb-12 18:24

My vote of 5
Sridhar Patnayak 6-Feb-12 18:52

Re: My vote of 5
Lakamraju Raghuram 6-Feb-12 18:58

typo
alxxl 27-Jan-12 23:28

Re: typo
Lakamraju Raghuram 28-Jan-12 9:47

My vote of 5
geoyar 23-Jan-12 23:53

Re: My vote of 5
Lakamraju Raghuram 24-Jan-12 8:54

My vote of 5
Shafi Mohammad 21-Jan-12 18:22

My vote of 5
jvardhan 21-Jan-12 17:34

Refresh 1 2 3 4 Next ▷

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。