C++11 标准

一、目录

二、新的语言特性

1、泛型的 Lambda 函数

在 C++11 中，lambda 函数的形式参数需要被声明为具体的类型。C++14 放宽了这一要求，允许 lambda 函数的形式参数声明中使用类型说明符 auto。

```
auto lambda = [](auto x, auto y) {return x + y;}
```

泛型 lambda 函数遵循模板参数推导的规则。以上代码的作用与下面的代码相同：

```
struct unnamed_lambda
{
   template<typename T, typename U>
      auto operator()(T x, U y) const {return x + y;}
};
auto lambda = unnamed_lambda();
```

2、Lambda 捕获表达式

C++11 的 lambda 函数通过值拷贝（by copy）或引用（by reference）捕获（capture）已在外层作用域声明的变量。这意味着 lambda 的值成员不可以是 move-only 的类型。C++14 允许被捕获的成员用任意的表达式初始化。这既允许了 capture by value-move，也允许了任意声明 lambda 的成员，而不需要外层作用域有一个具有相应名字的变量。

这是通过使用一个初始化表达式完成的：

```
auto lambda = [value = 1] {return value;}
```

lambda 函数 lambda 的返回值是 1，说明 value 被初始化为 1。被声明的捕获变量的类型会根据初始化表达式推断，推断方式与用 auto 声明变量相同。

使用标准函数 std::move 可以使之被用以通过 move 捕获：

```
auto ptr = std::make_unique<int>(10); //See below for std::make_unique
auto lambda = [ptr = std::move(ptr)] {return *ptr;}
```

声明 ptr = std::move(ptr)使用了两次 ptr。第一次使用声明了一个新的变量，但在捕获部分，这个变量还不在作用域内。所以第二个 ptr 表示之前在 lambda 之外声明的变量。

3、函数返回类型推导

C++11 允许 lambda 函数根据 return 语句的表达式类型推断返回类型。C++14 为一般的函数也提供了这个能力。C++14 还拓展了原有的规则，使得函数体并不是{return expression;}形式的函数也可以使用返回类型推导。

为了启用返回类型推导，函数声明必须将 auto 作为返回类型，但没有 C++11 的后置返回类型说明符：

```
auto DeduceReturnType();     //返回类型由编译器推断
```

如果函数实现中含有多个 return 语句，这些表达式必须可以推断为相同的类型。[9]

使用返回类型推导的函数可以前向声明，但在定义之前不可以使用。它们的定义在使用它们的翻译单元（translation unit）之中必须是可用的。

这样的函数中可以存在递归，但递归调用必须在函数定义中的至少一个 return 语句之后：

```
auto Correct(int i) {
   if (i == 1)
      return i;                    // 返回类型被推断为 int
   else
      return Correct(i-1)+i;   // 正确，可以调用
}

auto Wrong(int i)
{
   if(i != 1)
      return Wrong(i-1)+i;   // 不能调用，之前没有 return 语句
   else
      return i;                    // 返回类型被推断为 int
}
```

4、另一种类型推断

C++11 中有两种推断类型的方式。auto 根据给出的表达式产生具有合适类型的变量。decltype 可以计算给出的表达式的类型。但是，decltype 和 auto 推断类型的方式是不同的。特别地，auto 总是推断出非引用类型，就好像使用了 std::remove_reference 一样，而 auto&& 总是推断出引用类型。然而 decltype 可以根据表达式的值类别（value category）和表达式的性质推断出引用或非引用类型：

```
int i;
```

```
int&& f();
auto x3a = i;                    // x3a 的类型是 int
decltype(i) x3d = i;             // x3d 的类型是 int
auto x4a = (i);                  // x4a 的类型是 int
decltype((i)) x4d = (i);         // x4d 的类型是 int&
auto x5a = f();                  // x5a 的类型是 int
decltype(f()) x5d = f();         // x5d 的类型是 int&&
```

C++14 增加了 decltype(auto)的语法。这允许不必显式指定作为 decltype 参数的表达式，而使用 decltype 对于给定表达式的推断规则。

decltype(auto)的语法也可以用于返回类型推导，只需用 decltype(auto)代替 auto。

5、放松的 constexpr 限制

C++11 引入了 constexpr 函数的概念，这样的函数可以在编译期执行。它们的返回值可以用于期望常量表达式的场合（如模板的非类型参数）。然而 C++11 要求 constexpr 函数只含有一个将被返回的表达式（也可以还含有 static_assert 声明等其它语句，但允许的语句类型很少）。

C++14 将放松这些限制。声明为 constexpr 的函数将可以含有以下内容：

- 任何声明，除了：
  - static 或 thread_local 变量。
  - 没有初始化的变量声明。
- 条件分支语句 if 和 switch。goto 是不允许的。
- 所有的循环语句，包括基于范围的 for 循环。
- 表达式可以改变一个对象的值，只需该对象的生命期在常量表达式函数内开始。包括对有 constexpr 声明的任何非 const 非静态成员函数的调用。

调用非 constexpr 函数仍然是受限的。所以如果使用基于范围的 for 循环，begin 和 end 的重载形式必须自身被声明为 constexpr。值得注意的是，std::initializer_list 在本地和全局都具有 constexpr 声明的 begin/end 函数。

此外，C++11 指出，所有被声明为 constexpr 的非静态成员函数也隐含声明为 const（即函数不能修改*this 的值）。这点已经被删除，非静态成员函数可以为非 const。然而，只有当对象的生命期在常量表达式求值中开始，非 const 的 constexpr 成员函数才可以修改类成员。

6、变量模板

在 C++之前的版本中，模板可以是函数模板或类模板（C++11 引入了类型别名模板）。C++14 现在也可以创建变量模板。包括特化在内，通常的模板的规则都适用于变量模板的声明和定义。

7、聚合体成员初始化

C++11 新增 member initializer，这是一个表达式，被应用到类作用域的成员上，如果构造函数没有初始化这个成员。聚合体的定义被改为明确排除任何含有 member initializer 的类，因此，他们不允许使用聚合初始化。

C++14 将放松这一限制，这种类型也允许聚合初始化。如果花括号初始化列表不提供该参数的值，member initializer 会初始化它。

8、二进制字面值

C++14 的数字可以用二进制形式指定。其格式使用前缀 0b 或 0B。这样的语法也被 Java、Python、Perl 和 D 语言使用。

9、数字分位符

C++14 引入单引号（'）作为数字分位符号，使得数值型的字面量可以具有更好的可读性。

Ada、D 语言、Java、Perl、Ruby 等程序设计语言使用下划线（_）作为数字分位符号，C++之所以不和它们保持一致，是因为下划线已被用在用户自定义的字面量的语法中。

```
auto integer_literal = 100'0000;
auto floating_point_literal = 1.797'693'134'862'315'7E+308;
auto binary_literal = 0b0100'1100'0110;
auto silly_example = 1'0'0'000'00;
```

三、新的标准库特性

1、共享的互斥体和锁

C++14 增加了一类共享的互斥体和相应的共享锁。起初选择的名字是 std::shared_mutex，但由于后来增加了与 std::timed_mutex 相似的特性，std::shared_timed_mutex 成为了更适合的名字。

2、元函数的别名

C++11 定义了一组元函数，用于查询一个给定类型是否具有某种特征，或者转换给定类型的某种特征，从而得到另一个类型。后一种元函数通过成员类型 type 来返回转换后的类

型，当它们用在模板中时，必须使用 typename 关键字，这会增加代码的长度。

```
template <class T>
type_object<
   typename std::remove_cv<
      typename std::remove_reference<T>::type
   >::type
>get_type_object(T&);
```

利用类型别名模板，C++14 提供了更便捷的写法。其命名规则是：如果标准库的某个类模板（假设为 std::some_class）只含有唯一的成员，即成员类型 type，那么标准库提供 std::some_class_t<T>作为 typename std::some_class::type 的别名。

在 C++14，拥有类型别名的元函数包括：remove_const、remove_volatile、remove_cv、add_const、add_volatile、add_cv、remove_reference、add_lvalue_reference、add_rvalue_reference、make_signed、make_unsigned、remove_extent、remove_all_extents、remove_pointer、add_pointer、aligned_storage、aligned_union、decay、 enable_if、conditional、common_type、underlying_type、result_of、tuple_element。

```
template <class T>
type_object<std::remove_cv_t<std::remove_reference_t<T>>>
get_type_object(T&);
```

3、关联容器中的异构查找

C++标准库定义了四个关联容器类。set 和 multiset 允许用户根据一个值在容器中查找对应的的同类型的值。map 和 multimap 容器允许用户指定键（key）和值（value）的类型，根据键进行查找并返回对应的值。然而，查找只能接受指定类型的参数，在 map 和 multimap 中是键的类型，而在 set 和 multiset 容器中就是值本身的类型。

C++14 允许通过其他类型进行查找，只需要这个类型和实际的键类型之间可以进行比较操作。[17]这允许 std::set<std::string>使用 const char*，或任何可以通过 operator< 与 std::string 比较的类型作为查找的参数。

为保证向后兼容性，这种异构查找只在提供给关联容器的比较器允许的情况下有效。标准库的泛型比较器，如 std::less<>与 std::greater<>允许异构查找。

4、标准自定义字面值

C++11 增加了自定义字面量（user-defined literals）的特性，使用户能够定义新的字面量后缀，但标准库并没有对这一特性加以利用。C++14 标准库定义了以下字面量后缀：

- "s"，用于创建各种 std::basic_string 类型。
- "h"、"min"、"s"、"ms"、"us"、"ns"，用于创建相应的 std::chrono::duration 时间间隔。

```
using namespace std::literals;
std::string str = "hello world"s;
std::chrono::seconds dur = 60s;
```

- 两个"s"互不干扰，因为表示字符串的只能对字符串字面量操作，而表示秒的只针对数字。

5、通过类型寻址多元组

C++11 引入的 std::tuple 类型允许不同类型的值的聚合体用编译期整型常数索引。C++14 还允许使用类型代替常数索引，从多元组中获取对象。若多元组含有多于一个这个类型的对象，将会产生一个编译错误：

```
tuple<string, string, int> t("foo", "bar", 7);
int i = get<int>(t);              // i == 7
int j = get<2>(t);                 // Same as before: j == 7
string s = get<string>(t);    //Compiler error due to ambiguity
```

6、较小的标准库特性

std::make_unique 可以像 std::make_shared 一样使用，用于产生 std::unique_ptr 对象。

std::is_final，用于识别一个 class 类型是否禁止被继承。

std::integral_constant 增加了一个返回常量值的 operator()。

全局 std::begin/std::end 函数之外，增加了 std::cbegin/std::cend 函数，它们总是返回常量迭代器（constant iterators）。

四、已被移除或是不包含在 C++14 标准的特性

1、关于数组的扩展

在 C++11 和之前的标准中，在堆栈上分配的数组被限制为拥有一个固定的、编译期确定的长度。这一扩展允许在堆栈上分配的一个数组的最后一维具有运行期确定的长度。

运行期确定长度的数组不可以作为对象的一部分，也不可以具有全局存储期，他们只能被声明为局部变量。运行期确定长度的数组也可以使用 C++11 的基于范围的 for 循环。

同时还将添加 std::dynarray 类型，它拥有与 std::vector 和 std::array 相似的接口。代表一个固定长度的数组，其大小在运行期构造对象时确定。std::dynarray 类被明显地设计为当它被放置在栈上时（直接放置在栈上，或作为另一个栈对象的成员），可以使用栈内存而不是堆内存。

2、Optional 值

类似于 C#中的可空类型，optional 类型可能含有或不含有一个值。这一类型基于 Boost 的 boost::optional 类，而添加了 C++11 和 C++14 中的新特性，诸如移动和 in-place 构造。它不允许用在引用类型上。这个类被专门的设计为一个 literal type（如果模板参数本身是一个 literal type），因此，它在必要的情况下含有 constexpr 构造函数。

3、Concepts Lite

被 C++11 拒绝后，Concepts 受到彻底的修改。Concepts Lite 是 Concepts 的一个部分，仅包含类型约束，而不含 concept_map 和 axiom。它将在一个单独的 Technical Specification 中发展，并有可能加入 C++17。