揭开 C++ 的面纱
—— C++ 的新特性

New Features
For C++

作者：Amir Kirsch

INCREDIBUILD

目录

近年来 C++ 的发展历程主要围绕着两大重要领域：C++ 语言自身的不断更新以及标准库的扩展。

C++ 语言的革新内容包括：右值引用、lambda 表达式、新版上下文关键字(比如 override 和 final)、基本范围的 for 循环、自动变量声明、操作符（decltype）、可变参数模板和一些 C++20 的概念及更多其他内容。

这些新增的 C++ 语言特性，要求编译器有能力解析并且应对 C++ 语言的新功能。

declval、threads、 std::optional std::any

提高使用自己公司的产品比重
We Eat Our Own Dog Food

在 standard documents 中，C++ 标准库这一部分在 [library] 标签下有自己的分支，例如，可参考草案：http://eel.is/c++draft/library

几乎所有的库都基于 C++ 语言，不出意外的话，你可以自己实现这一操作。我倾向于将其称为"we eat our own dog food （特指：使用自己的产品）"，因为所有的库都依赖于 C++ 核心(这个有趣的俚语原文可参考链接：

https://en.wikipedia.org/wiki/Eating_your_own_dog_food#Origin_of_the_term

标准本身并没有提到任何如何成功运行库的细节（比如运行方式），但提及了统一 API 和相关行为的需求，例如并发性需求。举个例子来说，对于关联式容器：

Insert 和 emplace 成员变量应该不影响迭代器和 C++ 容器的引用性和有效性，erase 成员变量后迭代器会因为被擦除的元素而失效。

详情请见：http://eel.is/c++draft/associative.reqmts#9

本文将重点放于 C++ 语言中的特定库部分，并展示如何具体操作。具体来说，我们会详细讨论这些新特性中有没有一些不为人所知的细节，又或是我们是否可以运行具有自身个性化的版本。需要特别注意的是，运行个性化版本，主要是为了更好地理解实现过程，而不是为了取代标准库的某些部分。我们不建议此类做法。

下文中会探讨：

-std::any
-std::enable_if
-std::common_type
-std::declval

什么是 STD::ANY?

在 Java 脚本中，和其他脚本语言一样，可以写成：

```
var a = 3;  // Number
a = "hello"; // String
a = {x: "foo", y: 3.5}; // Object
a = function(msg){alert(msg);}; // function
```

示例中变量"a"是 Java 中特定的 type，但运行过程中会一直变化。

我们能不能在 C++ 中也做到这样呢?

好吧，如果你认为可以使用 "auto" 工具达到这一目标，你就大错特错了。

```
auto a = 3;  // int
a = "hello"; // compilation error
             // cannot bind const char* to int(不能将 const char* 连结到整数（int）上)
```

不过，在 C++ 中你仍然可以通过使用 std::any 达到差不多的效果。

凯尔文.海尼（Kevlin Henney）在2000年7月至8月刊的《C++ 报告》这一杂志上介绍了可以容纳任何情况的想法。

论文详情链接：
http://www.sdg.demon.co.uk/curbralan/papers/valuedconversions.pdf

有传言说海尼希望将这一新理论用他自己的名字"海尼（Henney）"命名。但最终人们认为 "any" 更合适。

在 2001 年，这一概念以 boost::any: 被加入到 Boost 中。

详情请见：
https://scicomp.ethz.ch/public/manual/Boost/1.55.0/any.pdf

接着又以 std::any 的形式被加入 C++17 中，

有了 any 你就可以做以下事情，例如：
```
std::any a = 3; // holding int
a = "hello";      // holding const char*
a = []{std::cout << "I'm a lambda"}; // now holding a lambda!
```

是的，变量"a"在运行时看似一直在改变，而这就是完全有效的 C++.

虽然我说变量"a"看起来一直在变化它的 type，但其实"a"的 type 从未改变-它的本质一直是 std::any.

那么问题是 std::any 如何让包含一切呢？是内部成员变量起了作用吗？这到底是黑魔法，还是简单易行的事情呢？

神秘面纱下的 STD::ANY

最初的想法是实现属于自己的 any 就可以有一个无符号类型 void* 来管理其余所有不同的 type.不论在哪种情况下，any 都会复制得到的参数来达到有一个被 void* 来表示的动态分配。

但是，有一个问题。

我们需要销毁某一个变量时（比如在析构函数中）或者获得另外一个值时（例如在赋值运算符中）释放 any 类包含的值。你不能在 void* 上执行删除。所以我们需要记住这些 type.

可能模板可以帮助改善这个情况？
有没有可能 any 可以被模板化记录下他的 type，像这样：any<T>。

好吧，这也是行不通的。因为这样的话不能分派 const char* 到 any<int>.然而我们的目的是为了 any 中不包含 type 和任何东西。

那么我们如何包含实际的 type，既可以释放，但仍然不将 any 类型的数据呢当作专门的某一类呢？

答案在这儿：

a.将 any 当作一种常见的类，例如：不是模板类。
b.在 any 中有一个内联类-让我们称之为 "holder-当这个"holder" 成员变量包含实际类，把他藏在 any 中。

c.在 any 中有一个模板构造函数，允许 any 获得一切参数。

```
template < typename T >
any(T t) : ptr(new holder<T>(std::move(t))) {}
```

所以我们希望的是这样的：

```
class any {
   template < typename T >
   struct holder {

      T value;
      holder(T&& t) : value(std::move(t)) {}
   };


    holder * ptr = nullptr; // problem here, holder is templated（这里的问题是，holder 被模板化了）

public:
   any()  {}
   ~any() { delete ptr; }

   template < typename T >
   any(T t) : ptr(new holder<T>(std::move(t))) {}

   any& operator=(any a) {
      std::swap(ptr, a.ptr);
      return *this;
   }
     // ...
};
```

以上代码显然不是最好的。我们不能像这样包含成员变量：
holder *
Holder 类被模板化所以我们必须提供模板参数。
但是我们不能像这样表示，
holder<T> *
因为 type T 不能完全定义 any，它只存在于 any 的构造函数中，而不存在于 any 本身中。

解决方案是使用虚拟析构函数，添加一个未模板化的基本类 base_holder，指向 base_holder 的含有 any 的指针，实际是指向 holder<T>。

所以看起来应该是这样的：

```
class any {

   struct base_holder {
      virtual ~base_holder() {}
         // ...
   };
   template < typename T >
   struct holder: base_holder {
      T value;
      holder(T&& t) : value(std::move(t)) {}
   };

   base_holder * ptr = nullptr; //现在这样就可以了

public:
   any()  {}
   ~any() { delete ptr; }
   template < typename T >
   any(T t) : ptr(new holder<T>(std::move(t))) {}

   any& operator=(any a) {
      std::swap(ptr, a.ptr);
      return *this;
   }
     // ...
};
```

因为 T 在 any 中是未知的，所以这被说成"被擦除"。

STD::ANY 的用法

很不幸，std::any 的用法是不简单的。为了获取出存储在内的值，你应该指示出 type，如下例所示：

```
std::any a = 3; // holding int
std::cout << a; // compilation error, std::cout cannot print std::any
std::cout << (int)a; // compilation error, almost..., but not yet
std::cout << std::any_cast<int>(a); // ok - prints 3
```

让 std::any 自动转换为它所包含的实际类型是非常有用的，但是这是不可能的，因为实际类型是"擦除的"，而且 std::any 是未知的，它需要程序员执行 std::any_cast.

在 std::any 中包含的实际类可以在运行时使用 std::any type 方法来检索，该方法返回到 type_info，但是这样的用法只适用于 if-else 语句，不适用于超载。

用法总结：

"在之前你用 void* 的地方用 std::any.所以基本就是无处可用。"

——Richard Hodges

https://stackoverflow.com/questions/52715219/when-should-i-use-stdany

编辑附注：
总结事项一

C++ 中的新特性可以分为两类。第一类是 C++ 语言中自身语法的革新，比如:右值引用、lambda 表达式、新的上下文关键字(如 override 和 final 等)。这些新增加的新特性需要编译器可以适应。第二类包括对标准库的添加，比如:智能指针、declval、线程、原子变量等等。第二组的特殊之处在于，它几乎总是基于 C++ 语言本身，并没有其他需要特别注意的地方。

编辑附注：
总结事项二

标准本身并没有为库部分指示任何的实现细节(即如何实现)，但是它提出了规定统一 API 和行为的需求。本文的目标是回顾几种语言属性，并了解如何实现它们。

本文的目标就是为了盘点一下几种语言的属性，并了解如何实施运行这些语言。

接下去要探讨的一些属性是与模板 SFINAE 一起使用的工具——std::enable_if、std::common_type和std::declval。

在我们开始前，先了解一下什么是 SFINAE

SFINAE 的全称是：Substitution Failure is Not An Error（替换失败并不是一个错误）

SFINAE 不是新兴事物，从 C++98 中就有了。所以你可能听说过 SFINAE .自从 C++11 以来，SFINAE 又重新受到关注，因为它添加了一些新的元素来支持SFINAE.

SFINAE 背后的概念很简单。当编译器将模板视为针对类的解决方案或这是一种方法时，调用模板签名可能与提供的实际模板参数不匹配。在这种情况下，模板被认为是"没有成功匹配"。编译器也在继续寻找更合适的。

举例：

让我们假设我们有以下模板来适用于所有整型变量 (short int, int, long int)：

```
template<class T>
bool print(const T& t){
    std::cout "integer number: " << t;
}
```

这个方法有两个问题：
a.这个方法太贪心了，它什么都接受，不仅仅只有整型数据。
b.如果它得到没有提取操作符到输出流的T。例如重载操作符
c.<<(std:: ostream&out, const t&t)——最终会导致编译错误。

假设我们有另外一个模板法来应对浮点型数据：

```
template<class T>
bool print(const T& t){
    std::cout "floating point number: " << t;
}
```

我们希望能够调用我们的输出方法，它实际上不是一个单独的方法，而是两个独立的模板方法。然而，他们共用同一个签名，这就很容易引起歧义，造成错误。

在 C++20 中，我们有可以解决这个问题的概念。但在 C++20 之前，需要对模板参数添加一些限制，以便正确解析所需的方法。

稍后我们再来讨论这两种情况。

让我们来看另一个需要 SFINAE 的例子：

假设我们想要使用关联容器来映射一些通用算法中的键和值。我们通常更喜欢使用 unordered_map，但是我们不确定所提供的键是否具有哈希函数。有一些键已实现了哈希函数，但是还有一些则没有。因此，我们希望创建自己的类，就将其命名为 associative_map 吧，如果键实现了哈希函数，其实就是一个 unordered_map，如果没有实现哈希函数，则为 map，在这种情况下，键需要有一个操作符 <.

为了实现这点，我们来看看另一个古老的语言工具：template specialization

它允许我们声明模板类(调用基础模版，而不是继承该基础模版），具体情况可以专门定制。

在我们的例子中，我们将使用一个"基础模板"的通用情况:

```
template <typename Key, typename Value, typename = void>
struct associative_map: std::map<Key, Value> {
  // inheriting all public constructors from base
  using std::map<Key, Value>::map;
};
```

注意第三个模板语句：typename = void 有一点奇怪-他没有命名，没有被使用(和那些没有名称的一起使用)，并且它有一个默认值"void"。使用者不会发送它。它是为接下去出现的特化而存在的。

无论如何，上面的操作允许目前创建 associative_map，但只是 std::map 的另一种说法。

但如果存在 std::hash<Key>() 这种方法，我们希望我们的 associative_map 成为一个 unordered_map！为了实现这一点，我们将添加专门版本:

```
template <typename Key, typename Value>
struct associative_map<Key, Value,
std::void_t<decltype(std::hash<Key>())>> : std::unordered_map<Key, Value> {
  using std::unordered_map<Key, Value>::unordered_map;
};
```

associative_map 模板类的专门化版本只有两个模板参数: Key 和 Value，为了让基本模板所需的三个模板参数齐全，我们使用std::void_t<decltype(std::hash<Key>())> 作为第三个模板语句，但是它有时候并不存在。

如果存在，则自动选择专门化版本。如果没有存在，那也没错——这是 SFINAE 而且使用的是基本模板。

使用关键字 decltype (添加在C++ 11中)是为了在专用版本中获取std::hash<Key> 的类型，但它可能不存在。如果它真不存在，则模板替换失败，但没有出现错误 (SFINAE)，在这个例子中我们将返回基本模板。

关于在上面的例子中使用 std::void_t 要说一下。当我们使用associative_map<Key, Value>时，第三个模板参数首先从基本模板中完成，就像我们发送了void一样。然后，当考虑专门版本时，如果第三个参数不是void，那么它就不是一个匹配项，专门版本就会被忽略。因此，我们希望使用std::void_t 将 decltype(std::hash<Key>()) 返回的值"强制转换"为 void.

STD::ENABLE_IF

在有些时候，我们希望仅在编译表达式为真时启用模板。例如，只有当T是整数时。
表达式 enable_if 一开始在 boost 中诞生，然后被加入了 C++11 的标准库。这里有一个输出用法的举例：

```
template<class T>
typename std::enable_if<std::is_integral<T>::value>::type
print(const T& t) {
   std::cout << "integer: " << t << std::endl;
}

template<class T>
typename std::enable_if<std::is_floating_point<T>::value>::type
print(const T& t) {
   std::cout << "floating: " << t << std::endl;
}
```

正如您所看到的，我们这里有两个非常相似的方法，因为 SFINAE 和std::enable_if 的存在，它们可以一起编译而不会产生歧义。在这个例子中，我们使用SFINAE作为返回类型，它不是 Void 就是无效。C++ 语言需要在从属类型之前添加关键字typename，这使得返回值的类型声明有点长：

```
typename std::enable_if<std::is_integral<T>::value>::type
```
和
```
typename std::enable_if<std::is_floating_point<T>::value>::type
```

表达式 std::enable_if 是一个模板类，它有两个模板参数，第一个是布尔表达式，必须在编译时求值，第二个参数默认为 void，也可以是其他 Type:

```
template< bool B, class T = void >
struct enable_if;
```

如果现有的布尔表达式被赋值为 true，则 enable_if::type 被赋值为 T（如果没有特别告知，则 T 被 void）。

但是，当布尔表达式被赋值为 false 时，则不定义enable_if::type，这会导致替换失败，然而对 SFINAE 很有用。

在我们的例子中：

std::is_integral<T>::value
如果 T 是整型时，则为真，否则为假。
std::is_floating_point<T>::value
如果 T 是浮点型时，则为真，否则为假。

举个例子来说：

typename std::enable_if<std::is_floating_point<T>::value>::type
void-只有当 T 是浮点型时，这是有效的 type

STD::ENABLE_IF 的真相

在模板特化的基础上 std::enable_if的实现非常简单。

```
template<bool B, class T = voi d>
struct enable_if {};

template<class T>
struct enable_if<true, T> {
   using type = T;
};
```

如果提供的布尔表达式为真，则选择 std::enable_if 的专门版本，它可以定义 type.如果布尔表达式为假，则返回到没有定义字段 type 的基本模板

有一个选项可以放弃基本模板的实际实现，并让其不完整：

```
template<bool B, class T = void>
struct enable_if;
```

结果是相同的，在没有匹配的情况下会出现不太友好的编译错误。假设删除整型数据类的方法 print，但是仍然通过整型数据来调用该方法 print,实际上显示出有错误的代码看上去是下面这样子的：

```
template argument deduction/substitution failed:
In substitution of:
'template<class T> typename enable_if<std::is_floating_point<T>::value>::type
print(const T&) [with T = int]'
error: no type named 'type' in 'struct enable_if<false, void>'
```

如果基本模板是空的，那么有错误的代码看起来是这样的：

```
template argument deduction/substitution failed:
In substitution of 'template<class T> typename
enable_if<std::is_floating_point<_Tp>::value>::type print(const T&) [with T = int]':
error: invalid use of incomplete type 'struct enable_if<false, void>'
```

第一个显示出有错误的信息看上去有点长。

STD::ENABLE_IF_T

C++14 增加了下面的表达式：

```
template<bool B, class T = void>
using enable_if_t = typename enable_if<B,T>::type;
```

C++17 增加了下面的表达式：

```
template<class T>
inline constexpr bool is_floating_point_v = is_floating_point<T>::value;
```

is_integral 也是依次类推，同样如此。
这些表达式主要是为了美化代码避免 enable_if 和 is_floating_point 之后分别添加 ::type 或 ::value，例如下面的代码：

```
template<class T>
typename std::enable_if<std::is_floating_point<T>::value>::type
print(const T& t) {
  std::cout << "floating: " << t << std::endl;
}
```

可以用下面的代码替代：

```
template<class T>
typename std::enable_if_t<std::is_floating_point_v<T>>
print(const T& t) {
  std::cout << "floating: " << t << std::endl;
}
```

STD::IS_FLOATING_POINT

在模板特化的基础上，std::is_floating_point 的实现也比较简单。

最初的想法是：

```
template<class T>
struct is_floating_point {
  static constexpr bool value = false;
};

template<>
struct is_floating_point<double> {
  static constexpr bool value = true;
};

template<>
struct is_floating_point<float> {
  static constexpr bool value = true;
};

// and the same for `long double'
```

上文中的代码没什么错误，但是有更简单的一种方法。

```
template< class T >
struct is_floating_point
    : std::integral_constant<
        bool,
        std::is_same<float, typename std::remove_cv<T>::type>::value  ||
        std::is_same<double, typename std::remove_cv<T>::type>::value  ||
        std::is_same<long double, typename std::remove_cv<T>::type>::value
    > {};
```

详情请见：https://en.cppreference.com/w/cpp/types/is_floating_point
上面的写法是使用 C++ 语言的其他构建块：

```
std::integral_constant<bool, true>
std::integral_constant<bool, false>
```

同样：

```
std::is_same
```

我把 std::is_same 的实现就留给你们就当作为练习吧。

STD::COMMON_TYPE

另一个与模板相关的工具是 std::common_type，它支持实现一个未知返回类型的方法。比如常见的类型就是 int,bool 和 long int.

```
template<typename T, typename... Ts>
std::common_type_t<T, Ts...> sum(T t1, Ts... ts) {
 return t1 + sum(ts...);
}

template<typename T>
T sum(T t1) {
 return t1;
}
```

std::common_type_t<T, Ts...> 将因为存在模板参数而被返回到 common type 比如：int 的常见类就是 bool 和 long is long.

C++14 也支持自动返回类型，因此，我们可以用下列内容替换std::common_type 的用法：

```
template<typename T, typename... Ts>
auto sum(T t1, Ts... ts) {
 return t1 + sum(ts...);
}
```

但是这两者也是不一样的，因为 char 和 char 的常见类型显然是 char，而将两个 char 和自动返回类型相加的返回类型是 int！如果希望保留 common_type 而不是根据所进行的操作变化，那么 std::common_type 会是合适的工具。

同样，我们希望知道在 std::common_type 中是否存在变数，又或者是只使用 C++ 语言就可以实现而不是还需要其他的技巧。

std::common_type 实现背后的思想是三元运算符(缩写为if表达式:?:)。C++ 语言将三元运算符的返回类定义为 true 返回类和 false 返回类之间的常见类。举例来看这类的表达式：

```
    test_expression ? 3 : 2.5;
2个is。 (is double)
```

如下所示，接下去可以使用三元运算符实现 common_type:

```
template<typename T, typename... Ts>
struct common_type {
  using type = decltype( false ?
T() : typename common_type<Ts...>::type() );
};

template<typename T>
struct common_type<T> {
  using type = T;
};
```

请注意，上面的"false"只是一个随意的选择。它和下面的代码工作原理是一样的：

```
using type = decltype( true ?
                       T() : typename common_type<Ts...>::type() );
```

在上文的这些实现步骤中，我们假设所有涉及的 types 都有默认的构造函数，为了避免这个情况，我们可以使用 std::declval，接下去我们会详细讨论：

```
template<typename T, typename... Ts>
struct common_type;

template<typename... Ts>
using common_type_t = typename common_type<Ts...>::type;

template<typename T, typename... Ts>
struct common_type {
  using type = decltype( false ?
std::declval<T>() : std::declval<common_type_t<Ts...>>() );
};
```

前面的例子略过了 std::common_type 的一些细枝末节。欲知如何完整实现，请见 C++ 参考手册：

https://en.cppreference.com/w/cpp/types/common_type#Possible_implementation

STD::DECLVAL

关键词 decltype 其实是一种神奇的语言，仅靠我们自己是无法实现的。Decltype 对上文提及多次的元编程任务中非常有用，它能够在编译时获取表达式的类型。

但是，另一方面，std::declval 有库的特性。使用 std::declval 是为了声明类而不需要再去创建它们。

举个例子来说，假设我们希望获得 foo 方法的返回类，用来获取 Bar 类对象的参数。那么为了获取这类的表达式应是这样的：

```
decltype(foo(Bar()))
```

但是，如果 Bar 没有空构造函数，那么这个表达式将无法编译。尽管我们实际上并不需要 Bar 类对象，我们只是希望"想象"一个调用 foo 的参数类型为Bar.

为了能够让这样的表达式可以用在我们的例子中，但同时又不需要应对 Bar 如何被创建和 std::declval 如何被编写出来：

```
decltype(foo(std::declval<Bar>()))
```

现在我们不需要再假设 Bar 是如何被实体化的，它甚至可以是抽象的!

std::declval 操作的秘诀

很简单。它只是没有实现声明的一个方法，但声明返回值使它可以被 decltype所用：

```
template<class T>
typename std::add_rvalue_reference<T>::type declval() noexcept;
```

注意，如果我们只声明返回值为 T，就像这样：

```
template<class T>
T declval() noexcept;
```

如果不另外添加或者调用 declval，我们将无法使用不完整类或者是抽象类。std::add_rvalue_reference 的加入让所有用法都变得可行，包括不完整类型、抽象类、右值引用和左值引用。

例如，如果 foo 方法是我们想和decltype一起使用从而达到左值引用的：

```
int foo(Bar&);
```

我们可以根据 reference collapsing rules，使用 declval 时就非常简单了 decltype(foo(std::declval<Bar&>()))

欲知更多详见此链接：

https://stackoverflow.com/questions/20303250/is-there-a-reason-declval-returns-add-rvalue-reference-instead-of-add-lvalue-ref

请注意，我们只涉及了标准库的一小部分。但目的很明确，我们可以明确地说，C++"正在吃属于自己那份的狗粮"，因为库中没有魔法，所以一切都是在于 C++ 的自身语法基础上不断革新。