### 8.4 SFINAE (Substitution Failure Is Not An Error)

### 8.4 SFINAE(替换失败并不是错误)

In C++ it is pretty common to overload functions to account for various argument types. When a compiler sees a call to an overloaded function, it must therefore consider each candidate separately, evaluating the arguments of the call and picking the candidate that matches best (see also Appendix C for some details about this process).

在C++中，重载函数以支持不同类型的参数是一种很常见的现象。当编译器看到对重载函数的调用时，它必须分别考虑每个候选函数，并评估每个调用参数，然后从中挑出最佳匹配的那一个（有关此过程的详细信息，请参阅附录C）

In cases where the set of candidates for a call includes function templates, the compiler first has to determine what template arguments should be used for that candidate, then substitute those arguments in the function parameter list and in its return type, and then evaluate how well it matches (just like an ordinary function).

如果候选的函数集中包含函数模板，那么编译器首先必须确定应为候选的函数模板使用哪些模板参数，然后将这些参数替换为函数参数列表及其返回值类型，再评估其匹配程度（就像普通函数一样）。

However, the substitution process could run into problems: It could produce constructs that make no sense. Rather than deciding that such meaningless substitutions lead to errors, the language rules instead say that candidates with such substitution problems are simply ignored.

但是这个替换过程可能会遇到问题：替换产生的结果可能没有意义，但语言规则并不会把这种无意义的替换当成错误，而是说具有此类替换问题的候选者将被忽略。

We call this principle SFINAE (pronounced like sfee-nay), which stands for “substitution failure is not an error.”

我们称这个原则为SFINAE(发音像sfee-nay)，它表示“替换失败并不是一个错误”。

Note that the substitution process described here is distinct from the on-demand instantiation process (see Section 2.2 on page 27): The substitution may be done even for potential instantiations that are not needed (so the compiler can evaluate whether indeed they are unneeded). It is a substitution of the constructs appearing directly in the declaration of the function (but not its body).

注意，此处所说的替换过程与“按需实例化”过程(见27页的2.2节)是不同的：即使是那些不需要被真正实例化的模板，也可能会进行替换(这样，编译器就可以评估是否确实不需要他们)。但是它们只会替换直接出现函数声明(不是函数体)的那些部分

Consider the following example:

考虑如下例子：

```
// number of elements in a raw array:
template<typename T, unsigned N>
std::size_t len(T(&)[N])
{
    return N;
}

// number of elements for a type having size_type:
template<typename T>
typename T::size_type len(T const& t)
{
    return t.size();
}
```

Here, we define two function templates len() taking one generic argument:

此处，我们定义了两个函数模板len()，它们都带有一个泛型的参数：

1. The first function template declares the parameter as T(&)[N], which means that the parameter has to be an array of N elements of type T.

第1个函数模板将参数声明为T(&)[N]，这意味着参数必须是一个具有N个元素、类型为T的数组。

2. The second function template declares the parameter simply as T, which places no constraints on the parameter but returns type T::size_type, which requires that the passed argument type has a corresponding member size_type.

第2个函数模板将参数简单地声明为T，它对参数没有任何约束。但返回类型为T::size_type，这要求传入的参数必须要有相应的size_type成员。

When passing a raw array or string literals, only the function template for raw arrays matches:

当传入一个原生数组或字符串字面量时，只有那个为原生数组定义的函数模板能够匹配：

```
int a[10];

std::cout << len(a); // OK: only len() for array matches
std::cout << len("tmp"); //OK: only len() for array matches
```

According to its signature, the second function template also matches when substituting (respectively) int[10] and char const[4] for T, but those substitutions lead to potential errors in the return type T::size_type. The second template is therefore ignored for these calls.

根据函数签名，第2个函数模板在(分别)用int[10]和char const[4]替换T后，也能够匹配。但是这些替换会导致在处理返回类型T::size_type时出现错误。因此，对于这两个调用，第2个模板会被忽略。

When passing a std::vector<>, only the second function template matches:

当传入std::vector<>时，只有第2个函数模板能够匹配：

```
std::vector<int> v;
std::cout << len(v); // OK: only len() for a type with size_type matches
```

When passing a raw pointer, neither of the templates match (without a failure). As a result, the compiler will complain that no matching len() function is found:

当传入裸指针时，两个模板都不匹配(但是不会因此而报错)。此时编译器会抱怨没有找到合适的len()函数：

```
int* p;
std::cout << len(p); // ERROR: no matching len() function found
```

Note that this differs from passing an object of a type having a size_type member, but no size() member function, as is, for example, the case for std::allocator<>:

注意，这和传递一个具有size_type成员而没有size()成员函数的对象是不同的。例如，如果传递的是std::allocator<>：

```
std::allocator<int> x;
std::cout << len(x); // ERROR: 匹配第2个len()模板len(), 但不能调用size()成员函数。
```

When passing an object of such a type, the compiler finds the second function template as matching function template. So instead of an error that no matching len() function is found, this will result in a compile-time error that calling size() for a std::allocator<int> is invalid. This time, the second function template is not ignored.

传递此类对象时，编译器会匹配到第2个函数模板。因此不会出现“未找到匹配的len函数”的错误，而是会报一个编译期错误，提示对于 std::allocator<int>而言，调用size()是一个无效的操作。 这一次，第2个模板函数不会被忽略掉。

Ignoring a candidate when substituting its return type is meaningless can cause the compiler to select another candidate whose parameters are a worse match. For example:

忽略掉那些在替换之后(译注：即已经匹配成功,如上述的std::allocator<int>)返回类型无效(如调用size())的候选函数是没有意义的，因为这会导致编译器选择另一个参数匹配程度较差的函数。（译注：这种情况下，该函数仍然会做为候选函数被保留下来，不会被忽略）。例如：

```
// number of elements in a raw array:
template<typename T, unsigned N>
std::size_t len(T(&)[N])
{
    return N;
}

// number of elements for a type having size_type:
template<typename T>
typename T::size_type len(T const& t)
{
    return t.size();
}

// 其他类型的后备函数:
std::size_t len(…)
{
    return 0;
}
```

Here, we also provide a general len() function that always matches but has the worst match (match with ellipsis (...) in overload resolution (see Section C.2 on page 682).

这里还提供一个通用的len函数，它总会匹配所有的调用，但也是所有重载函数中匹配最差的一个（通过省略号...来匹配）（见第682页的C.2节）

So, for raw arrays and vectors, we have two matches where the specific match is the better match. For pointers, only the fallback matches so that the compiler no longer complains about a missing len() for this call. But for the allocator, the second and third function templates match, with the second function template as the better match. So, still, this results in an error that no size() member function can be called:

因此，对于原生数组和vector，都有两个函数可以匹配，其中特化为数组类型的那个匹配更好。对于指针类型，只有后备函数(fallback)可以匹配，编译器不再抱怨找不到本次调用的len函数。但是对于std::allocator<int>的调用，第2和第3个函数模板均可以匹配，但第2个函数模板依然是最佳匹配。因此，编译器还是会报错提示缺少size()函数。

```
int a[10];

std::cout << len(a); // OK: 数组版本的len()是最佳匹配
std::cout << len("tmp"); //OK: 数组版本的len()是最佳匹配
std::vector<int> v;
std::cout << len(v); // OK: size_type版本的len()是最佳匹配

int* p;

std::cout << len(p); // OK: 只有后备函数(fallback)可以匹配
std::allocator<int> x;
std::cout << len(x); // ERROR: 第2个len()函数模板是最佳匹配，但不能调用x的size()函数。
```

See Section 15.7 on page 284 for more details about SFINAE and Section 19.4 on page 416 about some applications of SFINAE.

请参阅第284页15.7节中更多关于SFINAE的细节，以及第416页19.4节中一些关于SFINAE的应用。

#### SFINAE and Overload Resolution

#### SFINAE和重载方案

Over time, the SFINAE principle has become so important and so prevalent among template designers that the abbreviation has become a verb. We say “we SFINAE out a function” if we mean to apply the SFINAE mechanism to ensure that function templates are ignored for certain constraints by instrumenting the template code to result in invalid code for these constraints. And whenever you read in the C++ standard that a function template “shall not participate in overload resolution unless...” it means that SFINAE is used to “SFINAE out” that function template for certain cases.

随着时间的推移，SFINAE原则在模板设计者中已经变得如此重要和流行，以至于这个缩写已经变成一个动词。如果我们利用SFINAE机制来确保在某些限制条件下，通过这些条件让模板产生无效代码来忽略掉该模板，我们称之为“我们SFINAE掉了一个函数”。当你在C++标准里读到函数模板“不应参与重载解析过程，除非...”时，它的意思是，在某些情况下，使用SFINAE原则“SFINAE掉”了这个函数模板。

For example, class std::thread declares a constructor:

例如，std::thread声明了一个构造函数：

```
namespace std {
    class thread {
    public:
        ...
        template<typename F, typename... Args>
        explicit thread(F&& f, Args&&... args);
        ...
    };
}
```

with the following remark:

并做了如下的注释：

Remarks: This constructor shall not participate in overload resolution if decay_t<F> is the same type as std::thread.

注释：如果decay_t<F>的类型和std::thread相同的话，该构造函数不应该参考重载解析过程。

This means that the template constructor is ignored if it is called with a std::thread as first and only argument. The reason is that otherwise a member template like this sometimes might better match than any predefined copy or move constructor (see Section 6.2 on page 95 and Section 16.2.4 on page 333 for details).

它的意思是，在调用该构造函数时，如果std::thread是其第1个也是唯一的一个参数的话，那么该构造函数应该被忽略。原因是，如果不忽略该模板的话，这样的成员模板有时可能会产生比任何预定义的拷贝或移动构造函数更好的匹配（有关详细信息请参阅第95页6.2节和第333页的16.2.4节）。

By SFINAE’ing out the constructor template when called for a thread, we ensure that the predefined copy or move constructor is always used when a thread gets constructed from another thread.

通过SFINAE掉该构造函数模板，就可以确保在用一个std::thread构造另一个std::thread时始终调用预定义的拷贝或移动构造函数。

Applying this technique on a case-by-case basis can be unwieldy. Fortunately, the standard library provides tools to disable templates more easily. The best-known such feature is std::enable_if<>, which was introduced in Section 6.3 on page 98. It allows us to disable a template just by replacing a type with a construct containing the condition to disable it.

使用该技术逐项禁用相关模板是很麻烦的。幸运的是，标准库提供了一些更容易禁用模板的工具。其中最著名的是第98页6.3节介绍的std::enable_if<>。它允许我们通过用包含限制条件的语句来替代类型，从而禁用模板。

As a consequence, the real declaration of std::thread typically is as follows:

因此，std::thread的实际声明，典型的代码如下：

```
namespace std {
    class thread {
    public:
        ...

        template<typename F, typename... Args,
                typename = std::enable_if_t<!std::is_same_v<std::decay_t<F>, thread>>>
        explicit thread(F&& f, Args&... args);

        ...
    };
}
```

See Section 20.3 on page 469 for details about how std::enable_if<> is implemented, using partial specialization and SFINAE.

关于std::enable_if<>实现的更多细节，请参阅第469页20.3节，它使用了偏特化和SFINAE。

#### 8.4.1 Expression SFINAE with decltype

#### 8.4.1 通过decltype来SFINAE掉表达式

It’s not always easy to find out and formulate the right expression to SFINAE out function templates for certain conditions.

对于某些限制条件，要找到并设计正确的表达式来SFINAE掉函数模板并不容易。

Suppose, for example, that we want to ensure that the function template len() is ignored for arguments of a type that has a size_type member but not a size() member function. Without any form of requirements for a size() member function in the function declaration, the function template is selected and its ultimate instantiation then results in an error:

假设，我们想要让函数模板len()在其参数有size_type成员但没有size()成员函数的情况下被忽略。如果没有在函数声明中以某种形式要求size()成员函数必须存在，那么该函数模板就会被选择，并在最终实例化时会导致错误。

```
template<typename T>
typename T::size_type len(T const& t)
{
    return t.size();
}

std::allocator<int> x;
std::cout << len(x) << '\n'; //ERROR: len() selected, but x has no size()
```

There is a common pattern or idiom to deal with such a situation:

处理这一情况有一种常用模式或者说习惯用法：

- Specify the return type with the trailing return type syntax (use auto at the front and -> before the return type at the end).

通过尾随返回类型语法来指定返回类型(在函数名称的前面加auto，并在函数名后面加->和返回类型)

- Define the return type using decltype and the comma operator.

通过decltype和逗号运算符定义返回类型。

- Formulate all expressions that must be valid at the beginning of the comma operator (converted to void in case the comma operator is overloaded).

将所有需要成立的表达式放在逗号运算符的前面（为了防止可能会发生运算符被重载的情况，需要将这些表达式的类型转换为void）。

- Define an object of the real return type at the end of the comma operator.

在逗号运算符的末尾定义一个类型为近回类型的对象。

For example:

例如：

```
template<typename T>
auto len(T const& t) -> decltype((void)(t.size()), T::size_type())
{
    return t.size();
}
```

Here the return type is given by

这里返回类型定义为

```
decltype( (void)(t.size()), T::size_type() )
```

The operand of the decltype construct is a comma-separated list of expressions, so that the last expression T::size_type() yields a value of the desired return type (which decltype uses to convert into the return type). Before the (last) comma, we have the expressions that must be valid, which in this case is just t.size(). The cast of the expression to void is to avoid the possibility of a userdefined comma operator overloaded for the type of the expressions.

decltype的操作数是一组由逗号隔开的表达式。因此，最后一个表达式T::size_type()会产生一个预期的返回类型（decltype会将其转换为返回类型）。在最后一个逗号之前的所有表达式都必须是有效的，如本例中只有一个t.size()。将表达式转换为void是为了避免用户重载了该表达式对应类型的逗号运算符而导致类型的不确定性。

Note that the argument of decltype is an unevaluated operand, which means that you, for example, can create “dummy objects” without calling constructors, which is discussed in Section 11.2.3 on page 166.

注意，decltype的操作数是不会被求值的。也就是说，可以不调用构造函数而直接创建“傀儡对象(dummy object，dummy虚假、傀儡的意思)”，相关内容将在第166页的11.2.3节中加以讨论。

分类: C++模板编程



浅墨凉香
关注 - 0
粉丝 - 243
+加关注