

Chapter 8: Compile-Time Programming

第8章 编译期编程

C++ has always included some simple ways to compute values at compile time. Templates considerably increased the possibilities in this area, and further evolution of the language has only added to this toolbox.

C++一直以来都包含一些在编译期计算值的简单方法。模板极大地增加了这一领域的可能性，而语言的进一步发展通常也是在这一工具箱里进行的。

In the simple case, you can decide whether or not to use certain or to choose between different template code. But the compiler even can compute the outcome of control flow at compile time, provided all necessary input is available.

比较简单的情况是，你可以决定是否启用某个模板，或者在不同的模板代码之间进行选择。但是只要提供所有必要的输入条件，编译器甚至可以在编译期计算控制流的结果。

In fact, C++ has multiple features to support compile-time programming:

实际上，C++有很多特性可以支持编译期编程：

- Since before C++98, templates have provided the ability to compute at compile time, including using loops and execution path selection. (However, some consider this an “abuse” of template features, e.g., because it requires nonintuitive syntax.)

从C++98之前的版本开始，模板就提供了在编译期计算的能力，包括使用循环和执行路径选择（然而有些人认为这是对模板特性的“滥用”。例如，因为它的语法不够直观）

- With partial specialization we can choose at compile time between different class template implementations depending on specific constraints or requirements.

通过偏特化，可以在编译期根据特定的限制条件或需求进行不同类模板实现的选择。

- With the SFINAE principle, we can allow selection between different function template implementations for different types or different constraints.

利用SFINAE原则，可以根据不同类型或不同限制条件，在函数模板的不同实现之间做出选择。

- In C++11 and C++14, compile-time computing became increasingly better supported with the constexpr feature using “intuitive” execution path selection and, since C++14, most statement kinds (including for loops, switch statements, etc.).

在C++11和C++14中，可在constexpr中使用更直观的执行路径选择方式和大多数的语句类型（从C++14开始，包括循环和switch语句等），这使得编译期计算得到越来越好的支持。

- C++17 introduced a “compile-time if” to discard statements depending on compile-time conditions or constraints. It works even outside of templates.

C++17引入“编译期if”，通过它可以根据编译期的条件或限制来弃用某些语句。它甚至可以在模板之外使用。

This chapter introduces these features with a special focus on the role and context of templates.

本章将介绍这些特性，重点介绍模板及其相关方面的应用。

8.1 Template Metaprogramming


8.1 模板元编程

Templates are instantiated at compile time (in contrast to dynamic languages, where genericity is handled at run time). It turns out that some of the features of C++ templates can be combined with the instantiation process to produce a sort of primitive recursive “programming language” within the C++ language itself. For this reason, templates can be used to “compute a program.” Chapter 23 will cover the whole story and all features, but here is a short example of what is possible.

模板是在编译期实例化的（而动态语言的泛型是在运行期处理）。事实证明，C++模板的一些特性可以与实例化过程相结合，从而在C++语言中产生一种原始递归的“编程语言”。因此，模板可以用来“计算一个程序的结果”。第23章将会对这些特性进行全面的讨论，但这里给出一个简短的示例来说明它可能的情况。

The following code finds out at compile time whether a given number is a prime number:

下例代码可在编译期判断一个给定的数字是否为质数：

template<unsigned p, unsigned d> // p: 待检查数字, d: 当前除数
struct DoIsPrime {
 static constexpr bool value = (p % d != 0) && DoIsPrime<p, d - 1>::value;
};

template<unsigned p> //如果除数为2则停止递归
struct DoIsPrime<p, 2> {
 static constexpr bool value = (p % 2 != 0);
};

template<unsigned p> // 主模板
struct IsPrime {
 // 开始递归，除数从p/2开始：
 static constexpr bool value = DoIsPrime<p, p / 2>::value;
};

// 特殊情况（避免模板实例化陷入无限递归）：
template<>
struct IsPrime<0> { static constexpr bool value = false; };

template<>
struct IsPrime<1> { static constexpr bool value = false; };

template<>
struct IsPrime<2> { static constexpr bool value = true; };

template<>
struct IsPrime<3> { static constexpr bool value = true; };



The IsPrime<> template returns in member value whether the passed template parameter p is a prime number. To achieve this, it instantiates DolsPrime<>, which recursively expands to an expression checking for each divisor d between p/2 and 2 whether the divisor divides p without remainder.

IsPrime<>模板将参数p是否为质数的结果保存在value成员中。为了计算出结果，它会实例化DolsPrime<>，而模板又会递归地展开，以检查p除以p/2到2之间的每一个数是否有余数。

For example, the expression IsPrime<9>::value expands to

例如，表达式IsPrime<9>::value展开为

DoIsPrime<9,4>::value

which expands to

然后展开为

9%4!=0 && DoIsPrime<9,3>::value

which expands to

继续展开为

9%4!=0 && 9%3!=0 && DoIsPrime<9,2>::value

which expands to

继续展开为

9%4!=0 && 9%3!=0 && 9%2!=0

which evaluates to false, because 9%3 is 0.

这个求值结果为false，因为9%3==0。

As this chain of instantiations demonstrates:

正如这个链式实例化所展现的那样：

- We use recursive expansions of DolsPrime<> to iterate over all divisors from p/2 down to 2 to find out whether any of these divisors divide the given integer exactly (i.e., without remainder).

我们通过递归地展开为DolsPrime<>来遍历所有介于p/2到2之间的数，以检查p是否可以被这之间的每一个数整除（即没有余数）。

- The partial specialization of DolsPrime<> for d equal to 2 serves as the criterion to end the recursion.

d为2的偏特化模板DoIsPrime<2>是模板终止递归的条件。

Note that all this is done at compile time. That is,

注意，所有这些都是在编译期完成的，也就是说，

IsPrime<9>::value

expands to false at compile time.

在编译期被展开为false。

The template syntax is arguably clumsy, but code similar to this has been valid since C++98 (and earlier) and has proven useful for quite a few libraries.

这个模板语法可以说是很笨拙的，但是类似的代码从C++98（以及更早的版本）开始，就已经出现了，并且已经证明对于一些库的开发者是有帮助的。

See Chapter 23 for details.

更多细节请参阅第23章。

分类: C++模板编程

好文要顶

关注我

收藏该文





浅墨浓香
关注 - 0
粉丝 - 243

+加关注

« 上一篇：[第7章 按值传递或按引用传递：7.6 推荐的模板参数声明方法](#)

» 下一篇：[第8章 编译期编程：8.2 使用constexpr计算](#)