8.2 Computing with constexpr
8.2 使用constexpr计算

C++11 introduced a new feature, constexpr, that greatly simplifies various forms of compile-time computation. In particular, given proper input, a constexpr function can be evaluated at compile time. While in C++11 constexpr functions were introduced with stringent limitations (e.g., each constexpr function definition was essentially limited to consist of a return statement), most of these restrictions were removed with C++14. Of course, successfully evaluating a constexpr function still requires that all computational steps be possible and valid at compile time: Currently, that excludes things like heap allocation or throwing exceptions.
constexpr是C++11引入的一个新特性，它极大地简化了各种形式的编译期计算。特别是给定一个合适的输入，constexpr函数就可以在编译期完成相应的计算。然而，在C++11中constexpr函数有诸多的严格限制（例如：每个constexpr函数的定义本质上都只能有一个return语句），但是这些限制在C++14中大多数都被取消了。当然，成功计算constexpr函数仍然需要所有的计算步骤在编译期都是可行且有效的：当前，在堆上内存分配和异常抛出都是不被支持的。
Our example to test whether a number is a prime number could be implemented as follows in C++11:
在C++11中，判断一个数是否是质数的例子，可以实现如下：

```
constexpr bool doIsPrime(unsigned p, unsigned d) // p: 被检测数, d:当前除数
{
                  若除数d≠2, 则递归
    return d != 2 ? (p % d != 0) && doIsPrime(p, d - 1) //检测当前的d, 然后减小除数
                  : (p % 2 != 0);                       // 当除数为2时终止递归
                  若d=2, 则看p能否不被2整除
}

constexpr bool isPrime(unsigned p)
{
    return p < 4 ? !(p < 2)             //处理特殊情况
                 : doIsPrime(p, p / 2); // 开始递归, 从p/2开始。
}
```

因为小于等于3的自然数只有2和3是质数。

```
var = (y < 10) ? 30 : 40;
```

在这里，如果 y 小于 10，则 var 被赋值为 30，如果 y 不小于 10，则 var 被赋值为 40。

Due to the limitation of having only one statement, we can only use the conditional operator as a selection mechanism, and we still need recursion to iterate over the elements. But the syntax is ordinary C++ function code, making it more accessible than our first version relying on template instantiation.
由于C++11中要求只能有一条语句，此处我们只能使用条件运算符(:?)来进行条件选择，而且仍然需要用递归来遍历元素。但是这代码是一种普通的C++函数语法，它比依赖于模板实例化的第一个版本更容易理解。
With C++14, constexpr functions can make use of most control structures available in general C++ code. So, instead of writing unwieldy template code or somewhat arcane one-liners, we can now just use a plain for loop:
使用C++14，constexpr函数可以使用常规C++代码中的大部分控制结构。因此，现在可以使用普通的for循环来替代那种笨拙的模板代码和晦涩的单行代码方式：

```
constexpr bool isPrime(unsigned int p)
{
    for (unsigned int d = 2; d <= p / 2; ++d) {
        if (p % d == 0) {
            return false; // 可以整除d
        }
    }
    return p > 1; // 所有的除数都不能整除
}
```

稍微改进的算法

- 稍微注意一下，只需检测2,3,4,5...n/2是否能整除n。如果不能那么n就是素数。算法效率只是稍微提高了一点，它的时间复杂度依旧是O(n)

```
public static boolean isPrimeNumber1(int n) {
        if (n < 2) {
            return false;
        }
        for (int i = 2; i <= n / 2; i++) {
            if (n % i == 0) {
                return false;
            }
        }
        return true;
    }
```

With both the C++11 and C++14 versions of our constexpr isPrime() implementations, we can simply call
在C++11和C++14中实现的constexpr isPrime()，都可以直接地调用：

```
isPrime(9)
```

to find out whether 9 is a prime number. Note that it can do so at compile time, but it need not necessarily do so. In a context that requires a compile-time value (e.g., an array length or a nontype template argument), the compiler will attempt to evaluate a call to a constexpr function at compile time and issue an error if that is not possible (since a constant must be produced in the end). In other contexts, the compiler may or may not attempt the evaluation at compile time but if such an evaluation fails, no error is issued and the call is left as a run-time call instead.
来判断9是否为质数。注意，它可以在编译期求值，但不一定必须要这样做。在需要编译期数值的上下文中（如数组长度或非类型模板参数），编译器将尝试在编译期对constexpr函数进行求值，如果无法求值则会报错（因为最后必须要产生一个常量）。在其他上下文中，编译器可能会也可能不会在编译期尝试求值。但是如果这样的求值失败，则不会报错，而是将调用延迟到运行期执行。
For example:
例如：

```
constexpr bool b1 = isPrime(9); // 在编译期进行求值
```

will compute the value at compile time. The same is true with
将在编译期进行求值。下例情况也是如此

```
const bool b2 = isPrime(9); // 如果在namespace作用域，也会在编译期求值
```

provided b2 is defined globally or in a namespace. At block scope, the compiler can decide whether to compute it at compile or run time. This, for example, is also the case here:
如果b2定义在全局或命名空间中，那么也会在编译期进行计算。而如果在定义在块作用域({}内)，那么将由编译器决定是否在编译期进行计算。下面的例子也是属于这种情况：

```
bool fiftySevenIsPrime() {
    return isPrime(57); // 在编译期或运行期求值
}
```

the compiler may or may not evaluate the call to isPrime at compile time.
此时编译器可能会也可能不会在编译期对isPrime()求值。
On the other hand:
另一方面：

```
int x;
…
std::cout << isPrime(x); // 在运行期求值
```

will generate code that computes at run time whether x is a prime number.
无论x是不是一个质数，都将产生运行期的计算代码。

分类: C++模板编程

« 上一篇： 第8章 编译期编程：8.1 模板元编程
» 下一篇： 第8章 编译期编程：8.3 偏特化的执行路径选择