Constexpr - Generalized Constant Expressions in C++11

By Alex Allain

There are several improvements in C++11 that promise to allow programs written using C++11 to run faster than ever before. One of those improvements, generalized constant expressions, allows programs to take advantage of compile-time computation. If you're familiar with template metaprogramming, constexpr will seem like a way of making your life much easier. If you're not familiar with template metaprogramming, that's ok--constexpr makes the benefits of compile-time programming much more widely accessible.

The basic idea of constant expressions is to allow certain computations to take place at compile time--literally while your code compiles--rather than when the program itself is run. This has an obvious performance benefit: if something can be done at compile time, it will be done once, rather than every time the program runs. Need to compute the value of a known constant like the sine or cosine of a specific, known-at-compile number? Sure, you could use the library sin or cos function, but you'll pay the price of a function call at runtime. With constexpr, you can create a function that, at compile time, computes the value for you. Your user's CPU will never need to do any work at all for it!

Now, it's certainly true that if the operation can be computed at compile time, you could hard-code a constant. But doing that means that you lose flexibility if you later want to change the constant value--you have to recompute it, rather than just change an argument to a function.

Constexpr in action

In order to get compile time processing, you need to use the constext keyword on the function that you want to be able to compute at compile time.

```
1  constexpr int multiply (int x, int y)
2  {
3      return x * y;
4  }
5
6  // the compiler may evaluate this at compile time
7  const int val = multiply( 10, 10 );
```

Another benefit of constexpr, beyond the performance of compile time computation, is that it allows functions to be used in all sorts of situations that previously would have called for macros. For example, let's say you want to have a function that computes the the size of an array based on some multiplier. If you had wanted to do this in C++ without a constexpr, you'd have needed to create a macro or used template metaprogramming since you can't use the result of a function call to declare an array. But with constexpr, you can now use a call to a constexpr function inside an array declaration:

```
1  constexpr int getDefaultArraySize (int multiplier)
2  {
3      return 10 * multiplier;
4  }
5
6  int my_array[ getDefaultArraySize( 3 ) ];
```

Restrictions on constexpr functions

A constexpr function has some very rigid rules it must follow:

- It must consist of single return statement (with a few exceptions)
- It can call only other constexpr functions
- It can reference only constexpr global variables

Notice that one thing that isn't restricted is recursion. How can you do recursion if the function can only have a single return statement? By using the ternary operator (sometimes known as the question mark colon operator). For example, here's a function that computes the value of a specific factorial:

```
1  constexpr factorial (int n)
2  {
3      return n > 0 ? n * factorial( n - 1 ) : 1;
4  }
```

Now you can use factorial(2) and when the compiler sees it, it can optimize away the call and make the calculation entirely at compile time. In this way, by allowing more sophisticated calculations, constexpr behaves differently than a mere inline function. You can't inline a recursive function! In fact, any time the function argument is itself a constexpr, it can be computed at compile time.

What else can go in a constexpr function?

A constexpr function can have only a single line of executable code, but it may contain typedefs, using declarations and directives, and static_asserts.

Constexpr and runtime

A function declared as constexpr can also be called at runtime if the argument to the function is a non-constant--for example:

```
1  int n;
2  cin >> n;
3  factorial( n );
```

This means that you do not need to create separate functions for compile time and run time.

Using objects at compile time

Since any object that is referenced by a constexpr function must be constexpr, what if you want to use an object in that function? For example, what if you had a circle object?

```
1  class Circle
2  {
3      public:
4      Circle (int x, int y, int radius) : _x( x ), _y( y ), _radius( radius ) {}
5      double getArea () const
6      {
7          return _radius * _radius * 3.1415926;
8      }
9      private:
10         int _x;
11         int _y;
12         int _radius;
13 };
```

And you wanted to construct a circle at compile time and get its area?

```
1  constexpr Circle c( 0, 0, 10 );
2  constexpr double area = c.getArea();
```

It turns out that you can do this with a few small modifications to the Circle class. First, we need to declare the constructor as constexpr, and second, we need to declare the getArea function as constexpr. Declaring the constructor as a constexpr allows it to be run at compile time as long as it consists only of member initializations using other constexpr constructors (a compiler-generated default constructor can also be treated as constexpr, assuming all its members have constructors that are constexpr). Declaring the method getArea as constexpr allows it to be called at compile time:

```
1  class Circle
2  {
3      public:
4      constexpr Circle (int x, int y, int radius) : _x( x ), _y( y ), _radius( radius ) {}
5      constexpr double getArea ()
6      {
7          return _radius * _radius * 3.1415926;
8      }
9      private:
```

```
10          int _x;
11          int _y;
12          int _radius;
13  };
```

constexpr vs const

If you declare a class member function to be constexpr, that marks the function as 'const' as well. (Clearly it must be const if it is constexpr, because a constexpr function cannot modify the object in any way.) If you declare a variable as constexpr, that in turn marks the variable as const. However, it doesn't work the other way--a const function is not a constexpr, nor is a const variable a constexpr.

Constexpr and Floating Point Numbers

So far everything we've seen with constexpr could have been achieved--much more verbosely--using template metaprogramming. There is, however, one completely new piece of functionality enabled const constexpr: compile time computation of floating point values. Because double and float are not valid template parameter types, you can't easily use template metaprogramming to compute these values at compile time (for example, computing arbitrary values of sine and cosine). You'd have to fall back to using fixed point arithmetic. On the other hand, constexpr does allow the use of floating point values. Think about the possibility of compile time computation of values that are likely to show up in game or graphics programming. For example, the trigonometric functions sine and cosine are often used for computing object rotation (among other things). You can use the Taylor series for sine to compute sine values at compile time for constant arguments to sine.

Tradeoffs of constexpr

C++ already suffers from relatively slow compilation due to the need to recompile any code after changing a header file. Constexpr is sufficiently powerful that it risks introducing additional compile-time overhead. However, there are some built-in advantage sto constexpr that limit this risk. First, because constexpr functions always return the same output value for the same input, they can be memoized, and in fact GCC already supports memoization.

Due to the ability to memoize constexpr functions, in cases where constant expressions replace template metaprogramming, the performance impact shouldn't be worse than today, while the code will be much clearer. In fact, by eliminating the need to do a large number of template instantiations, the compilation time may actually be much faster.

Finally, the standard also allows compilers to limit the levels of nesting allowed for recursive constexpr methods (although it does require at least 512 levels to be allowed). This limitation may limit performance penalties for extremely high-overhead compile time calculations by preventing too much reliance on deep recursion. It's also useful to know about in case you do plan to write highly nested calculations.

Constexpr compiler availability

Constexpr requires the compiler to be able to do recursive function call processing at compile time, so it may not be a surprise that C++ compiler support for constexpr is pretty limited. In fact, the only compiler version I know of that fully supports constexpr is GCC 4.7. (Earlier versions of GCC allowed the syntax, but did not provide compile time processing.) If you want to try it out, you can either build GCC from source or pick up a pre-built Cygwin-compatible binary at this page.

Next: Faster Code with Rvalue References and Move Semantics Learn how C++11 lets you write faster code in yet another way, by avoiding slow copies in favor of fast moves
Previous: Range-Based For Loops Range-based for loops make iterating over vectors and other containers very easy