Calculating integer factorials in constant time, taking advantage of overflow behavior

devblogs.microsoft.com/oldnewthing/20151214-00

December 14, 2015

Raymond Chen

For some reason, people on StackOverflow calculate factorials a lot. (Nevermind that it's not necessarily the best way to evaluate a formula.) And you will see factorial functions like this:

```
int factorial(int n)
{
 if (n < 0) return -1; // EDOM
 int result = 1;
 for (int i = 2; i <= n; i++) result *= i;
 return result;
}
```

But you can do better than this by taking advantage of undefined behavior.

Since signed integer overflow results in undefined behavior in C/C++, you can assume that the result of the factorial does not exceed `INT_MAX`, which is 2147483647 for 32-bit signed integers. This means that *n* cannot be greater than 12.

So use a lookup table.

```
int factorial(int n)
{
 static const int results[] = {
    1,
    1,
    1 * 2,
    1 * 2 * 3,
    1 * 2 * 3 * 4,
    1 * 2 * 3 * 4 * 5,
    1 * 2 * 3 * 4 * 5 * 6,
    1 * 2 * 3 * 4 * 5 * 6 * 7,
    1 * 2 * 3 * 4 * 5 * 6 * 7 * 8,
    1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9,
    1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10,
    1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 * 11,
    1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10 * 11 * 12,
 };
 if (n < 0) return -1; // EDOM
 return results[n]; // undefined behavior if n > 12
}
```

(If you have 64-bit signed integers, then the table needs to go up to 20.)

The fact that you have undefined behavior if $n > 12$ is hardly notable, because the original code also had undefined behavior if $n > 12$. You just replaced one undefined behavior with another.

If you want to simulate two's complement overflow, for example, to preserve bug-for-bug compatibility, or because the function was defined to compute unsigned instead of signed factorial,[1] then you can do that by extending the table just a little bit more. You will need only entries up to 33! because because 34! is [an exact multiple of 2^{32}]. The result of `factorial(n)` is zero for $n \geq 34$, assuming 32-bit integers.

```
int factorial(int n)
{
 static const int results[] = {
          1, // 0!
          1, // 1!
          2, // 2!
          6, // 3!
         24, // 4!
        120, // 5!
        720, // 6!
       5040, // 7!
      40320, // 8!
     362880, // 9!
    3628800, // 10!
   39916800, // 11!
  479001600, // 12!
 1932053504, // 13!
 1278945280, // 14!
 2004310016, // 15!
 2004189184, // 16!
 4006445056, // 17!
 3396534272, // 18!
  109641728, // 19!
 2192834560, // 20!
 3099852800, // 21!
 3772252160, // 22!
  862453760, // 23!
 3519021056, // 24!
 2076180480, // 25!
 2441084928, // 26!
 1484783616, // 27!
 2919235584, // 28!
 3053453312, // 29!
 1409286144, // 30!
  738197504, // 31!
 2147483648, // 32!
 2147483648, // 33!
 };
 if (n < 0) return -1; // EDOM
 if (n > 33) return 0; // overflowed to zero
 return results[n];
```

I didn’t calculate all those numbers myself. I wrote a program to do it.

```
class Program
{
 public static void Main() {
  uint result = 1;
  uint n = 0;
  while (result != 0) {
   System.Console.WriteLine(" {0,10}, // {1}!", result, n);
   result *= ++n;
  }
 }
}
```

Extending the above algorithm to 64-bit integers is left as an exercise.

[1] Unsigned arithmetic is defined by the C/C++ standards to be modulo 2^n for some n.

Raymond Chen

Follow