Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Programming Techniques for Scientific Simulations Exercise 12

HS 15
Prof. M. Troyer

Problem 12.0 Template Meta-Programming 1, Factorial

This is a warmup exercise were we calculate the factorial $n! = n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1$ (note that $0! = 1$) at compile time. Use class templates to achieve the loop through recursion and the ending condition through specialization. If you already feel comfortable with the concepts used here you can jump to the next exercise, where the same techniques are needed.

- Create a class template `factorial`, taking `n` as the first template parameter and a temporary value `temp` as the second one.

```
factorial<uint_type n, uint_type temp>
```

- The class has one static member function `eval()` which evaluates the factorial. As long as the ending condition is not reached it calls recursively `factorial` for $n-1$. The product of the previous values is stored in `temp`.
- Make a specialization of `factorial` to end the recursion appropiately. The factorial of n can then be evaluated by calling `factorial<n,1>::eval()`.

Problem 12.1 Template Meta-Programming 2, Binomial Coefficient

Evaluate the binomial coefficient[1]

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \tag{1}$$

at compile time through template meta-programming.

Problem 12.2 Template Meta-Programming 3, Expression Templates

Expression templates are based on the fact that the tree associated with any expression can be represented with types at compile-time; the compiler can then perform powerful optimizations on the expression or, under specific circumstances, evaluate the whole or parts of the expression at compile-time. Here is an example of such a tree:

```
                              +
                             / \
                            *   4
3 * (x + 1) + 4    <-->    / \
                          3   +
                             / \
                            x   1
```

Note that it consists of vertices representing some operation and leafs representing either a constant or the independent variable. For a given tree, one can easily calculate the derivative of the expression using the chain rule and the product rule.

[1] `http://en.wikipedia.org/wiki/Binomial_coefficient`

We want to combine this into a simple code that is able to calculate expressions and derivatives thereof, where we allow only addition and multiplication operations and only one independent variable, which may however occur several times. The basic ingredient is the class `Expression` that represents vertices of the tree. It has three template parameters:

- type of the left-hand vertex,
- type of the right-hand vertex,
- operation

In addition, we have two classes for the leafs, `Constant` and `Variable`. Each class must implement two functions, `T operator()(T y)` and `T derivative(T y)`, which calculate the value of the part of the expression that follows below that vertex at $x = y$ or its derivative at $x = y$. All this is being represented by types, i.e. the type of the above expression would be:

```
Expression< Expression<Constant<int>, Expression<Variable ,
Constant<int>, Add>, Multiply >, Constant<int>, Add >
```

Complete the missing parts in the skeleton code provided. With the compiler flag (`-O3`), everything should be evaluated at compile-time and the assembly should contain the final numbers $f(8) = 974$ and $f'(8) = 157$.[2]

Bonus: Enable the evaluation of higher order derivatives by making the `derivative()` to return an expression (`Constant`, `Variable` or `Expression`) instead of an evaluated expression. The evaluation would then take place only at call of the `operator()(T x)`. The following code shows the evaluation of $\left.\frac{\mathrm{d}^2(4x^2)}{\mathrm{d}x^2}\right|_{x=8}$.

```
    Variable<int> x;
    return (constant(4)*x*x).derivative().derivative()(8);
```

Problem 12.3 Parallelization

Use `OpenMP`[3] to parallelize your simpson integration routine (you may take the templated version of exercise 5 as a starting point). Parallelization is easiest done outside of the simpson routine. Each thread calculates the integral on a subinterval with a reduced number of bins. When you sum up the results at the end take care of race conditions! You can use the `reduction` shortcut and let `OpenMP` do the work for you.

Note: The `OpenMP` standard is included in current versions of the GNU compilers and Microsoft Visual Studio. For OS X you can either use the GNU compilers instead of Clang or (remotely) use the D-PHYS machines (e.g. `plumpy.ethz.ch`).

[2] Use compiler flag `-save-temps` to write the assembler output.

[3] `http://openmp.org/wp/`