Exercises for the Lecture Series

"Object-Oriented Programming for Scientific Computing"

Dr. Ole Klein
ole.klein@iwr.uni-heidelberg.de

To be handed in on 27. 06. 2017 before the lecture

EXERCISE 1 STATIC VS. DYNAMIC POLYMORPHISM

We want to investigate runtime performance differences of programs when using static or dynamic polymorphism. As an example we again consider numerical integration (quadrature).

Instead of using dynamic polymorphism, we may also select the quadrature rule and/or the function to be integrated using static polymorphism (templates).

As a reminder, we use the following interface, which you have implemented in this or a very similar form in a previous exercise:

- `Functor`s take a single `double` argument and return a function value (also `double`).
- `Rule`s take a `Functor` and two interval boundaries (as `double`) and return an approximate integral of the `Functor` on that interval.
- A `CompositeRule` takes a `Functor` and a `Rule` and integrates the `Functor` on an interval by applying the `Rule` on a sequence of subintervals and summing up the results.

1. The following choices can be made, among others, when implementing this program:
   (a) Dynamic or static selection of the quadrature rule.
   (b) Dynamic or static selection of the function.
   (c) Inline definition of operators or definition outside of the class declaration.

   Implement several versions of this program, with the following choices:

   (a) Dynamic selection for both quadrature rule and function.
   (b) Dynamic selection for the function, but static for the rule.
   (c) Static selection for both quadrature rule and function.

   Provide these variants both with inline definition of operators and definition outside of the class declaration, so six versions in total.

   Due to the large number of very similar classes, this might be a good time to become acquainted with namespaces and separate header files. This also reduces, depending on your coding style and discipline, the work required for implementation, since it becomes easier to copy and reuse code. Make sure that it is always clear through the name of the namespace which version it is. Check that all methods are declared `const`, and that classes for static polymorphism don't contain any virtual methods.

2. Measure the time required for integration by each of the different versions in your main function. Use the `<chrono>` header for this, it provides a namespace `std::chrono` with timer functionality. Try to familiarize yourself with this header, here is an example on how to use it:

```
const auto start = std::chrono::high_resolution_clock::now();
// compute integral here
const auto end = std::chrono::high_resolution_clock::now();
const auto ms = std::chrono::duration_cast<std::chrono::nanoseconds>
(end-start).count() / 1000.;
// ms contains elapsed time in milliseconds
// print it using std::cout, for example
```

3. Measure the times for all versions using different optimization levels by specifying the optimization flags "-O0", "-O1" and "-O2". Use the integration of $\int_{-3}^{13} 2t^2 + 5$ with 100.000 nodes as a test case. We restrict ourselves to the midpoint rule. This means it is enough to implement different versions of these two classes, one specific function and one specific quadrature rule per variant (plus abstract base classes for the cases with dynamic polymorphism). Fill a table with the results.

4. Are there time differences between the different versions? If yes, do they depend on the optimization level? What effect does the optimization level have on each of the versions? Interpret and justify the results! Is it worth it to think about the form of polymorphism in this situation? If so, why? If not, why not?

20 Points