Dr. Ole Klein, Stefan Meggendorfer
IWR, Heidelberg University

Exercise Sheet 9

Exercise 1: Lambda Expressions for Integration (10 points)

Introduced in C++11, lambda expressions are a very convenient way to define function objects (functors). We will have a second look at numerical integration, so that you can decide for yourself whether using lambdas makes the resulting code more readable or not.

In a previous exercise you had to implement several functors to write a program for numerical integration (quadrature):

- scalar real functions (polynomials and trigonometric functions)
- quadrature rules (trapezoidal rule and Simpson's rule)
- composite rules (only the equidistant one in our case)
- ...

Each of these functors was derived from some abstract base class that defined its interface.

Your task is the modification of the program you already have, so that it uses lambda expressions instead of the classes you created (or to write the program from scratch if you skipped that exercise). Proceed as follows:

(a) Remove all abstract base classes. Lambda expressions introduce anonymous auto-generated classes that are not derived from anything[1], so these base classes become irrelevant. The usual replacement for these abstract base classes would be templates, to model the polymorphism at compile-time. However, you need to pass lambdas as arguments to lambdas, i.e., these lambda expressions themselves would need to be templates. This isn't possible in C++11, since a lambda expression defines a functor class, not a functor class template, and the type of function arguments is therefore fixed. C++14 introduces generic lambdas, as we will see, but for now another solution has to be used.

(b) C++11 provides a class template called `std::function` (in header `<functional>`). This template is an instance of *type erasure*: you can assign any type of function or function object to it, and it will forward the function calls to it, while its type is independent of the exact type of its contents. This means you can capture lambdas in `std::function` objects and ignore the fact that each lambda defines its own type. Inform yourself how this class is used and what its drawbacks are[2].

(c) Replace each functor object with an equivalent lambda expression, using `std::function` wherever you have to pass a lambda to another lambda. Note how the `std::function` argument serves as documentation of the expected function interface. Wherever appropriate, capture local variables instead of handing them over as arguments, and make sure that the right capture type is used (by-value vs. by-reference).

(d) Run the test problems as described in the previous exercise, and make sure that your new version produces the same results. Use the `time` command to compare the compile time for the two versions, as well as the runtime for a large number of subintervals. Which version is faster, if any, and which is easier to read resp. implement? Discuss.

[1] https://en.cppreference.com/w/cpp/language/lambda
[2] https://en.cppreference.com/w/cpp/utility/functional/function

Exercise 2: Variadic Templates: Type-Safe Printing (10 points)

The concept of variadic functions is nothing new: C has them as well, e.g., the `printf` function that creates formatted output. However, these C-style variadic functions[3] have two drawbacks:

- They need special macros, like `va_start`, `va_arg`, `va_end`, etc., which are not very intuitive to use and incur a certain runtime cost.
- They are not type-safe, since there is no way to know what data types are going to be passed as arguments.

Variadic functions based on the variadic templates introduced in C++11, however, have full access to the type information of their arguments. Use this to write a flexible `print()` function:

(a) Start by writing several variants of this function for a single argument, using a combination of, e.g., SFINAE and simple function overloads:

- Any class that has a `print()` method should be printed by simply calling this method. Create a simple numerical vector class with such a method that prints its components as comma-separated values enclosed by brackets, e.g., `[2.72,3.12,6.59]`, and test your function with it. There is no need to implement any of the usual methods of vector classes, just an appropriate constructor and the `print()` method.
- Strings (`std::string`) and string literals (`const char*`) should be placed between quotation marks (you can ignore the fact that internal quotation marks in the strings would have to be escaped in practice).
- Floating-point numbers should be printed with six-digit precision in scientific notation. Use the appropriate I/O manipulator[4] and the `precision()` method of the stream object, and don't forget to restore the original precision after printing.
- Any other single argument should simply be printed by passing it to `std::cout`.

(b) Write a variadic `print()` function that is able to print an arbitrary number of arguments using the rules above. Base your implementation on the single-argument versions you have already produced, and note that the order of function definitions is important. Test your function with some sequences of mixed arguments. There is no need to specify any types, like you would for `printf`: the compiler can deduce those types from the function argument. Note that this also means that there is no way to accidentally specify the wrong type, of course.

[3] https://en.cppreference.com/w/cpp/utility/variadic
[4] https://en.cppreference.com/w/cpp/io/manip