Implementing QuantLib

Luigi Ballabio

B Code conventions

EVERY PROGRAMMER TEAM has a number of conventions to be used while writing code. Whatever the conventions (several exist, which provides a convenient *casus belli* for countless wars) it is important that they are adhered to strictly,[1] as they make it easier to understand the code; a reader familiar with their use can distinguish at a glance between a macro and a function, or between a variable and a type name.

Listing B.1 briefly illustrates the conventions used throughout the QuantLib library. Following the advice in [25], we tried to reduce their number at a minimum, enforcing only those conventions which enhance readability.

Macros are in all uppercase, with words separated by underscores. Type names start with a capital and are in the so-called camel case; words are joined together and the first letter of each word is capitalized. This applies to both type declarations such as `SomeType` and class names such as `SomeClass` and `AnotherClass`. However, an exception is made for type declarations that mimic those found in the C++ standard library; this can be seen in the declaration of the two iterator types in `SomeClass`. The same exception might be made for inner classes.

About everything else (variables, function and method names, and parameters) are in camel case and start with a lowercase character. Data members of a class follow the same convention, but are given a trailing underscore; this makes it easier to distinguish them from local variables in the body of a method (an exception is often made for public data members, especially in structs or struct-like classes.) Among methods, a further convention is used for getters and setters. Setter names are created by adding a leading `set` to the member name and removing the trailing underscore. Getter names equal the name of the returned data member without the trailing underscore; no leading `get` is added. These conventions are exemplified in `AnotherClass` and `SomeStruct`.

[1] However, the QuantLib developers are human. As such, they sometimes fail to follow the rules I am describing.

A much less strict convention is that the opening brace after a function declaration or after an `if`, `else`, `for`, `while`, or `do` keyword are on the same line as the preceding declaration or keyword; this is shown in `someFunction`. Moreover, `else` keywords are on the same line as the preceding closing brace; the same applies to the `while` ending a `do` statement. However, this is more a matter of taste than of readability; therefore, developers are free to use their own conventions if they cannot stand this one. The shown function exemplifies another convention aimed at improving readability, namely, that function and method arguments should be aligned vertically if they do not fit a single line.

Last but not least, there should be no real tabs in the code; four spaces should be used instead. If you decide to keep just one convention in this set (don't—it is just a figure of speech) please keep this one. Failing to do so will most likely ruin indentation for developers viewing the code in another editor. Whatever your editor, it can be configured (you are a programmer, right?) so that this convention is enforced without any conscious effort on your part.

Listing B.1: Illustration of QuantLib code conventions.

```
#define SOME_MACRO

typedef double SomeType;

class SomeClass {
  public:
    typedef Real* iterator;
    typedef const Real* const_iterator;
};

class AnotherClass {
  public:
    void method();
    Real anotherMethod(Real x, Real y) const;
    Real member() const;    // getter, no "get"
    void setMember(Real);   // setter
  private:
    Real member_;
    Integer anotherMember_;
};

struct SomeStruct {
    Real foo;
    Integer bar;
};

Size someFunction(Real parameter,
                  Real anotherParameter) {
    Real localVariable = 0.0;
    if (condition) {
        localVariable += 3.14159;
    } else {
        localVariable -= 2.71828;
    }
    return 42;
}
```