4.1. Namespace `std`

If you use different modules and/or libraries, you always have the potential for name clashes. This is because modules and libraries might use the same identifier for different things. This problem was solved by the introduction of *namespaces* to C++. A namespace is a certain scope for identifiers. Unlike a class, a namespace is open for extensions that might occur at any source. Thus, you could use a namespace to define components that are distributed over several physical modules. A typical example of such a component is the C++ standard library, so it follows that it uses a namespace.

In fact, all identifiers of the C++ standard library are defined in a namespace called `std`. With C++11, this also applies to identifiers that were introduced with TR1 and had namespace `std::tr1` there (see Section 2.1, page 7). In addition, namespace `posix` is reserved now, although it is not used by the C++ standard library.

Note that the following namespaces nested within `std` are used inside the C++ standard library:

- `std::rel_ops` (see Section 5.5.3, page 138)
- `std::chrono` (see Section 5.7.1, page 144)
- `std::placeholders` (see Section 6.10.3, page 243)
- `std::regex_constants` (see Section 14.6, page 732)
- `std::this_thread` (see Section 18.3.7, page 981)

According to the concept of namespaces, you have three options when using an identifier of the C++ standard library:

1. You can qualify the identifier directly. For example, you can write `std::ostream` instead of `ostream`. A complete statement might look like this:

```
std::cout << std::hex << 3.4 << std::endl;
```

2. You can use a *using declaration*. For example, the following code fragment introduces the local ability to skip `std::` for `cout` and `endl`:

```
using std::cout;
using std::endl;
```

Thus, the example in option 1 could be written like this:

```
cout << std::hex << 3.4 << endl;
```

3. You can use a *using directive*. This is the easiest option. By using a using directive for namespace `std`, all identifiers of the namespace `std` are available as if they had been declared globally. Thus, the statement

```
using namespace std;
```

allows you to write

```
cout << hex << 3.4 << endl;
```

Note that in complex code, this might lead to accidental name clashes or, worse, to different behavior due to some obscure overloading rules. You should never use a using directive when the context is not clear, such as in header files.

The examples in this book are quite small, so for my own convenience, I usually use using directives throughout this book in complete example programs.