We recompile once again and inspect our newly generated object files:

```
$ gcc -I. -c app.cpp lib_interval.cpp
$ nm -C app.o lib_interval.o

app.o:
                 U lib::Interval<double>::Interval(double const&, double const&)
0000000000000000 W lib::Interval<int>::Interval(int const&, int const&)
                 U bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
0000000000000000 T main

lib_interval.o:
0000000000000000 W lib::Interval<double>::Interval(double const&)
0000000000000000 W lib::Interval<double>::Interval(double const&, double const&)
0000000000000000 W lib::Interval<double>::low() const
0000000000000000 W lib::Interval<double>::high() const
0000000000000000 W lib::Interval<double>::length() const
0000000000000000 W bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
```

The application object file, `app.o`, naturally remained unchanged. What's new here is that the functions that were missing from the `app.o` file are now available in the `lib_interval.o` file, again as *weak* (`W`), as opposed to strong (`T`), symbols. Notice, however, that explicit instantiation forces the compiler to generate code for all of the member functions of the class template for a given specialization. These symbols might all be linked into the resulting executable unless we take explicit precautions to exclude those that aren't needed[3]:

```
$ gcc -o app app.o lib_interval.o -Wl,--gc-sections
$ nm -C app
00000000004005ca W lib::Interval<double>::Interval(double const&, double const&)
000000000040056e W lib::Interval<int>::Interval(int const&, int const&)
000000000040063d W bool lib::intersect<double>(lib::Interval<double> const&,
                                               lib::Interval<double> const&)
00000000004004b7 T main
```

The **`extern template`** feature is provided to enable software architects to reduce code bloat in individual object files for common instantiations of class, function, and, as of C++14, variable templates in large-scale C++ software systems. The practical benefit is in reducing the **physical** size of libraries, which *might* lead to improved link times. Explicit-instantiation declarations do *not* (1) affect the meaning of a program, (2) suppress inline template implicit instantiation, (3) impede the compiler's ability to **`inline`**, or (4) meaningfully improve

[3]To avoid including the explicitly generated definitions that are being used to resolve undefined symbols, we have instructed the linker to remove all unused code sections from the executable. The `-Wl` option passes comma-separated options to the linker. The `--gc-sections` option instructs the compiler to compile and assemble and instructs the linker to omit individual unused sections, where each section contains, for example, its own instantiation of a function template.