Suppose, for now, we decide we would like to suppress the generation of object code for templates related to just **`double`** type with the intent of later putting them all in one place, i.e., the currently empty `lib_interval.o`. Achieving this objective is precisely what the **`extern template`** syntax is designed to accomplish.

Returning to our `lib_interval.h` file, we need not change one line of code; we need only to *add* two explicit-instantiation declarations — one for the *class* template, `Interval<double>`, and one for the *function* template, `intersect<double>(const double&, const double&)` — to the header file anywhere *after* their respective corresponding general template declaration and definition:

```
// lib_interval.h:  // No change to existing code.
#ifndef INCLUDED_LIB_INTERVAL  // internal include guard
#define INCLUDED_LIB_INTERVAL

namespace lib  // namespace for all entities defined within this component
{

template <typename T>
class Interval
{
    // ...  (same as before)
};

template <typename T>
bool intersect(const Interval<T>& i1, const Interval<T>& i2)
{
    // ...  (same as before)
}

extern template class Interval<double>;  // explicit-instantiation declaration

extern template                          // explicit-instantiation declaration
bool intersect(const Interval<double>&, const Interval<double>&);

}  // close lib namespace

#endif  // INCLUDED_LIB_INTERVAL
```

Let's again compile the two `.cpp` files and inspect the corresponding `.o` 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&)
```