```
#include <vector>  // std::vector (general template)

template class std::vector<int>;
    // Deposit all definitions for this specialization into the .o for this
    // translation unit.
```

This explicit-instantiation directive compels the compiler to instantiate *all* functions defined by the named `std::vector` class template having the specified **`int`** template argument; any collateral object code resulting from these instantiations will be deposited in the resulting `.o` file for the current translation unit. Importantly, even functions that are never used are still instantiated, so this solution might not be the correct one for many classes; see *Potential Pitfalls — Accidentally making matters worse* on page 373.

Explicit-instantiation declaration

C++11 introduced the explicit-instantiation declaration, a complement to the explicit-instantiation definition. The newly provided syntax allows us to place **`extern template`** in front of the declaration of an ~~explicit~~ specialization of a class template, a function template, or a variable template:

```
#include <vector>  // std::vector (general template)

extern template class std::vector<int>;
    // Suppress depositing of any object code for std::vector<int> into the
    // .o file for this translation unit.
```

Using the modern **`extern template`** syntax above instructs the compiler to *refrain* from depositing any object code for the named specialization in the current translation unit and instead to rely on some other translation unit to provide any missing object-level definitions that might be needed at link time; see *Annoyances — No good place to put definitions for unrelated classes* on page 373.

Note, however, that declaring an explicit instantiation to be an **`extern template`** *in no way* affects the ability of the compiler to instantiate and to inline visible function-definition bodies for that template specialization in the translation unit:

```
// client.cpp:
#include <vector>  // std::vector (general template)

extern template class std::vector<int>;

void client(std::vector<int>& inOut)  // fully specialized instance of a vector
{
    if (inOut.size())          // This invocation of size can inline.
    {
        int value = inOut[0];  // This invocation of operator[] can be inlined.
    }
}
```