In the previous example, the two tiny member functions of `vector`, namely, `size` and **`operator[]`**, will typically be inlined — in precisely the same way they would have been had the **`extern template`** declaration been omitted. The *only* purpose of an **`extern template`** declaration is to suppress object-code generation for this particular template instantiation for the current translation unit.

Finally, note that the use of explicit-instantiation directives has absolutely no effect on the logical meaning of a well-formed program; in particular, when applied to specializations of function templates, they have no effect on overload resolution:

```
template <typename T> bool f(T v) {/*...*/}  // general template definition

extern template bool f(char c);  // specialization of f for char
extern template bool f(int v);   // specialization of f for int

bool bc = f((char)    0);  // exact match: Object code is suppressed locally.
bool bs = f((short)   0);  // not exact match: Object code is generated locally.
bool bi = f((int)     0);  // exact match: Object code is suppressed locally.
bool bu = f((unsigned)0);  // not exact match: Object code is generated locally.
```

As the example above illustrates, overload resolution and template argument deduction occur independently of any explicit-instantiation declarations. Only *after* the template to be instantiated is determined does the **`extern template`** syntax take effect; see also *Potential Pitfalls — Corresponding explicit-instantiation declarations and definitions* on page 371.

A more complete illustrative example

So far, we have seen the use of explicit-instantiation declarations and explicit-instantiation definitions applied to only a standard *class* template, `std::vector`. The same syntax shown in the previous code snippet applies also to full specializations of individual function templates and variable templates.

As a more comprehensive, albeit largely pedagogical, example, consider the overly simplistic `my::Vector` class template along with other related templates defined within a header file, `my_vector.h`:

```
// my_vector.h:
#ifndef INCLUDED_MY_VECTOR  // internal include guard
#define INCLUDED_MY_VECTOR

#include <cstddef>  // std::size_t
#include <utility>  // std::swap

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

template <typename T>
class Vector
```