If this approach doesn't work for you, you may be using a compiler that installed a non-official predraft version of templates in which templates were chosen *ahead* of ordinary functions. With this rule in effect, the compiler would use the template `Swap()` instead of the `job` version. So to get the desired effect, you’d have to use an explicit specialization that didn't quite have the modern form. That is, instead of using

\[
\text{template} <> \text{void Swap<job>(job \&j1, job \&j2); // ISO/ANSI C++}
\]

you would use the following form:

\[
\text{void Swap<job>(job \&, job \&);} \quad \text{// earlier form of specialization}
\]

Notice that it is missing the `template <>` preface. You’d make the same adjustment in the function heading. That is,

\[
\text{template <> void Swap<job>(job \&j1, job \&j2); // specialization}
\]

\[
\{
\quad ...
\}
\]

becomes

\[
\text{void Swap<job>(job \&j1, job \&j2) // old form of specialization}
\]

\[
\{
\quad \text{// code unchanged}
\}
\]

If you’re using a contemporary C++ compiler (and we hope you are), you won’t have to deal with these adjustments.

**Instantiations and Specializations**

To extend our understanding of templates, we need to investigate the terms instantiation and specialization. Keep in mind that including a function template in your code does not in itself generate a function definition. It's merely a plan for generating a function definition. When the compiler uses the template to generate a function definition for a particular type, the result is termed an *instantiation* of the template. For example, in Listing 8.11, the