Here's the program output; you can see the process has worked:

\[ i, j = 10, 20. \]
Using compiler-generated int swapper:
\[ \text{Now } i, j = 20, 10. \]
\[ x, y = 24.5, 81.7. \]
Using compiler-generated double swapper:
\[ \text{Now } x, y = 81.7, 24.5. \]

Note that function templates don't make your executable programs any shorter. In Listing 8.9, you still wind up with two separate function definitions, just as if you had defined each function manually. And the final code doesn't contain any templates; it just contains the actual functions generated for your program. The benefits of templates are that they make generating multiple function definitions simpler and more reliable.

**Overloaded Templates**

You use templates when you need functions that apply the same algorithm to a variety of types, as in Listing 8.8. It might be, however, that not all types would use the same algorithm. To meet this possibility, you can overload template definitions, just as you overload regular function definitions. As with ordinary overloading, overloaded templates need distinct function signatures. For example, Listing 8.10 adds a new swapping template, one for swapping elements of two arrays. The original template has the signature \((\text{Any } &, \text{Any } &)\), whereas the new template has the signature \((\text{Any } [], \text{Any } [], \text{int})\). Note that the final argument in this case happens to be a specific type (int) rather than a generic type. Not all template arguments have to be template parameter types.

When, in twotemps.cpp, the compiler encounters the first use of Swap(), it notices that it has two int arguments and matches it to the original template. The second use, however, has two int arrays and an int value as arguments, and this matches the new template.

**Listing 8.10 twotemps.cpp**

```cpp
// twotemps.cpp -- using overloaded template functions
#include <iostream>
using namespace std;
```