a generic type instead of a specific type, the process is sometimes termed *generic programming*. Because types are represented by parameters, the template feature is sometimes referred to as *parameterized types*. Let's see why such a feature is useful and how it works.

Earlier we defined a function that swapped two `int` values. Suppose you want to swap two `double` values instead. One approach is to duplicate the original code but replace each `int` with `double`. If you need to swap two `char` values, you can use the same technique again. Still, it's wasteful of your valuable time to have to make these petty changes, and there's always the possibility of making an error. If you make the changes by hand, you might overlook an `int`. If you do a global search-and-replace to substitute, say, `double` for `int`, you might do something such as converting

```cpp
type x;
short interval;
```

to the following:

```cpp
double x;                // intended change of type
short doubleerval;      // unintended change of variable name
```

C++’s function template capability automates the process, saving you time and providing greater reliability.

Function templates enable you to define a function in terms of some arbitrary type. For example, you can set up a swapping template like this:

```cpp
template <class Any>
void Swap(Any &a, Any &b)
{
    Any temp;
    temp = a;
    a = b;
    b = temp;
}
```

The first line specifies that you are setting up a template and that you're naming the arbitrary type `Any`. The keywords `template` and `class` are obligatory, except that you can