C++ templates use the same source code for different types. This can improve code flexibility and make programs easier to maintain because code can be written and tested once, yet used with different types. Templates can also be used in generic programming that treats different types with the same semantics. This chapter introduces templates and static polymorphism, the STL, template metaprogramming and some generic programming methods, and shows how these can be used effectively for microcontrollers.

### 5.1 Template Functions

Consider the simple template function below.

```cpp
template<typename T>
T add(const T& a, const T& b)
{
    return a + b;
}
```

The template function `add()` returns the sum of `(a + b)`, where `a`, `b` and the return value of `add()` all have the same type as the template parameter `T`. A template parameter can be considered a placeholder for a not-yet-specified type. Template parameters can be class types, built-in types, constant integral or pointer values, but not floating-point values. See Sect. 4.3 in Vandevooorde and Josuttis [3].

When a template is used in code, the compiler instantiates it for a known type by filling in the template code corresponding to its template parameters at compiler time. This is the same vocabulary that is used for instances of a class types in object-oriented programming. The context should be considered when discerning the two.
The code below, for example, uses add() twice, one time to add two integer variables and a second time to add two variables of type std::string, the standard library’s string class.

```cpp
int n = add(1, 2); // 3
std::string s = add(std::string("abc"),
    std::string("xyz")); // "abcxyz".
```

In the calls to add() above, the template parameter is not explicitly given. The compiler can automatically deduce the types of template parameters if it has sufficient information to do so from the context of usage. Even if the template parameters could be deduced, though, they can still be optionally provided. For example,

```cpp
const int c = add<int>(a, b);
```

The template parameters must be compatible with the functionality of the templated code. In order to be used with add(), for example, a given template parameter must support the binary addition operator (in other words operator+).

The compiler does not automatically perform type conversion for templates. So if the function’s parameters do not exactly match those of the template, then the template parameters must be explicitly provided. For example,

```cpp
double d1 = add(1.1, 2.2); // OK, 3.3, double
double d2 = add(1.1, 2);   // Not OK, ambiguous
double d3 = add<double>(1.1, 2); // OK, 3.1, double
```

If multiple types are needed, template functions can have more than one template parameter.1

For instance,

```cpp
template<typename T1,
    typename T2>
T1 add(const T1& a, const T2& b)
{
    return a + T1(b);
}
double d = add(1.1, 2); // OK, 3.1, double
```

---

1 Multiple template parameters are provided in a comma-separated template parameter list in angled brackets.