26.x — Chapter 26 summary and quiz

Templates allow us to write functions or classes using placeholder types, so that we can stencil out identical versions of the function or class using different types. A function or class that has been instantiated is called a function or class instance.

All template functions or classes must start with a template parameter declaration that tells the compiler that the following function or class is a template function or class. Within the template parameter declaration, the template type parameters or expression parameters are specified. Template type parameters are just placeholder types, normally named T, T1, T2, or other single letter names (e.g. S). Expression parameters are usually integral types, but can be a pointer or reference to a function, class object, or member function.

Splitting up template class definition and member function definitions doesn't work like normal classes -- you can't put your class definition in a header and member function definitions in a .cpp file. It's usually best to keep all of them in a header file, with the member function definitions underneath the class.

Template specialization can be used when we want to override the default behavior from the templated function or class for a specific type. If all types are overridden, this is called full specialization. Classes also support partial specialization, where only some of the templated parameters are specialized. Functions can not be partially specialized.

Many classes in the C++ standard library use templates, including std::array and std::vector. Templates are often used for implementing container classes, so a container can be written once and used with any appropriate type.

Quiz time

1. It's sometimes useful to define data that travels in pairs. Write a templated class named Pair1 that allows the user to define one template type that is used for both values in the pair. The following function should work:

```
int main()
{
	Pair1<int> p1 { 5, 8 };
	std::cout << "Pair: " << p1.first() << ' ' << p1.second() << '\n';

	const Pair1<double> p2 { 2.3, 4.5 };
	std::cout << "Pair: " << p2.first() << ' ' << p2.second() << '\n';

	return 0;
}
```

and print:

```
Pair: 5 8
Pair: 2.3 4.5
```

Show Solution

2. Write a Pair class that allows you to specify separate types for each of the two values in the pair.

Note: We're naming this class differently from the previous one because C++ does not currently allow you to "overload" classes that differ only in the number or type of template parameters.

The following program should work:

```
int main()
{
	Pair<int, double> p1 { 5, 6.7 };
	std::cout << "Pair: " << p1.first() << ' ' << p1.second() << '\n';

	const Pair<double, int> p2 { 2.3, 4 };
	std::cout << "Pair: " << p2.first() << ' ' << p2.second() << '\n';

	return 0;
}
```

and print:

```
Pair: 5 6.7
Pair: 2.3 4
```

Hint: To define a template using two different types, separate the two types by a comma in the template parameter declaration. See lesson 11.8 -- Function templates with multiple template types for more information.

Show Solution

3. A string-value pair is a special type of pair where the first value is always a string type, and the second value can be any type. Write a template class named StringValuePair that inherits from a partially specialized Pair class (using std::string as the first type, and allowing the user to specify the second type).

The following program should run:

```
int main()
{
	StringValuePair<int> svp { "Hello", 5 };
	std::cout << "Pair: " << svp.first() << ' ' << svp.second() << '\n';

	return 0;
}
```

and print:

```
Pair: Hello 5
```

Hint: When you call the Pair constructor from the StringValuePair constructor, don't forget to include the template parameters as part of the Pair class name.

Show Solution