26.1 — Template classes

In a previous chapter, we covered function templates (11.6 -- Function templates), which allow us to generalize functions to work with many different data types. While this is a great start down the road to generalized programming, it doesn't solve all of our problems. Let's take a look at an example of one such problem, and see what templates can further do for us.

Templates and container classes

In the lesson on 23.6 -- Container classes, you learned how to use composition to implement classes that contained multiple instances of other classes. As one example of such a container, we took a look at the IntArray class. Here is a simplified example of that class:

```
#ifndef INTARRAY_H
#define INTARRAY_H

#include <cassert>

class IntArray
{
private:
    int m_length{};
    int* m_data{};

public:

    IntArray(int length)
    {
        assert(length > 0);
        m_data = new int[length]{};
        m_length = length;
    }

    // We don't want to allow copies of IntArray to be created.
    IntArray(const IntArray&) = delete;
    IntArray& operator=(const IntArray&) = delete;

    ~IntArray()
    {
        delete[] m_data;
    }

    void erase()
    {
        delete[] m_data;
        // We need to make sure we set m_data to 0 here, otherwise it will
        // be left pointing at deallocated memory!
        m_data = nullptr;
        m_length = 0;
    }

    int& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() const { return m_length; }
};

#endif
```

While this class provides an easy way to create arrays of integers, what if we want to create an array of doubles? Using traditional programming methods, we'd have to create an entirely new class! Here's an example of DoubleArray, an array class used to hold doubles.

```
#ifndef DOUBLEARRAY_H
#define DOUBLEARRAY_H

#include <cassert>

class DoubleArray
{
private:
    int m_length{};
    double* m_data{};

public:

    DoubleArray(int length)
    {
        assert(length > 0);
        m_data = new double[length]{};
        m_length = length;
    }

    DoubleArray(const DoubleArray&) = delete;
    DoubleArray& operator=(const DoubleArray&) = delete;

    ~DoubleArray()
    {
        delete[] m_data;
    }

    void erase()
    {
        delete[] m_data;
        // We need to make sure we set m_data to 0 here, otherwise it will
        // be left pointing at deallocated memory!
        m_data = nullptr;
        m_length = 0;
    }

    double& operator[](int index)
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() const { return m_length; }
};

#endif
```

Although the code listings are lengthy, you'll note the two classes are almost identical! In fact, the only substantive difference is the contained data type (int vs double). As you likely have guessed, this is another area where templates can be put to good use, to free us from

having to create classes that are bound to one specific data type.

Creating template classes works pretty much identically to creating template functions, so we'll proceed by example. Here's our array class, templated version:

Array.h:

```
#ifndef ARRAY_H
#define ARRAY_H

#include <cassert>

template <typename T> // added
class Array
{
private:
    int m_length{};
    T* m_data{}; // changed type to T

public:

    Array(int length)
    {
        assert(length > 0);
        m_data = new T[length]{}; // allocated an array of objects of type T
        m_length = length;
    }

    Array(const Array&) = delete;
    Array& operator=(const Array&) = delete;

    ~Array()
    {
        delete[] m_data;
    }

    void erase()
    {
        delete[] m_data;
        // We need to make sure we set m_data to 0 here, otherwise it will
        // be left pointing at deallocated memory!
        m_data = nullptr;
        m_length = 0;
    }

    // templated operator[] function defined below
    T& operator[](int index); // now returns a T&

    int getLength() const { return m_length; }
};

// member functions defined outside the class need their own template declaration
template <typename T>
T& Array<T>::operator[](int index) // now returns a T&
{
    assert(index >= 0 && index < m_length);
    return m_data[index];
}
```

```
#endif
```

As you can see, this version is almost identical to the IntArray version, except we've added the template declaration, and changed the contained data type from int to T.

Note that we've also defined the `operator[]` function outside of the class declaration. This isn't necessary, but new programmers typically stumble when trying to do this for the first time due to the syntax, so an example is instructive. Each templated member function defined outside the class declaration needs its own template declaration. Also, note that the name of the templated array class is Array<T>, not Array -- Array would refer to a non-templated version of a class named Array, unless Array is used inside of the class. For example, the copy constructor and copy-assignment operator used Array rather than Array<T>. When the class name is used without template arguments inside of the class, the arguments are the same as the ones of the current instantiation.

Here's a short example using the above templated array class:

```
#include <iostream>
#include "Array.h"

int main()
{
	const int length { 12 };
	Array<int> intArray { length };
	Array<double> doubleArray { length };

	for (int count{ 0 }; count < length; ++count)
	{
		intArray[count] = count;
		doubleArray[count] = count + 0.5;
	}

	for (int count{ length - 1 }; count >= 0; --count)
		std::cout << intArray[count] << '\t' << doubleArray[count] << '\n';

	return 0;
}
```

This example prints the following:

```
11     11.5
10     10.5
9       9.5
8       8.5
7       7.5
6       6.5
5       5.5
4       4.5
3       3.5
2       2.5
1       1.5
0       0.5
```

Template classes are instantiated in the same way template functions are -- the compiler stencils out a copy upon demand, with the template parameter replaced by the actual data type the user needs, and then compiles the copy. If you don't ever use a template class, the compiler won't even compile it.

Template classes are ideal for implementing container classes, because it is highly desirable to have containers work across a wide variety of data types, and templates allow you to do so without duplicating code. Although the syntax is ugly, and the error messages can be cryptic, template classes are truly one of C++'s best and most useful features.

Template classes in the standard library

Now that we've covered template classes, you should understand what std::vector<int> means now -- std::vector is actually a template class, and int is the type parameter to the template! The standard library is full of predefined template classes available for your use. We'll cover these in later chapters.

Splitting up template classes

A template is not a class or a function -- it is a stencil used to create classes or functions. As such, it does not work in quite the same way as normal functions or classes. In most cases, this isn't much of a issue. However, there is one area that commonly causes problems for developers.

With non-template classes, the common procedure is to put the class definition in a header file, and the member function definitions in a similarly named code file. In this way, the member function definitions are compiled as a separate project file. However, with templates, this does not work. Consider the following:

Array.h:

```
#ifndef ARRAY_H
#define ARRAY_H

#include <cassert>

template <typename T> // added
class Array
{
private:
    int m_length{};
    T* m_data{}; // changed type to T

public:

    Array(int length)
    {
        assert(length > 0);
        m_data = new T[length]{}; // allocated an array of objects of type T
        m_length = length;
    }

    Array(const Array&) = delete;
    Array& operator=(const Array&) = delete;

    ~Array()
    {
        delete[] m_data;
    }

    void erase()
    {
        delete[] m_data;
        // We need to make sure we set m_data to 0 here, otherwise it will
        // be left pointing at deallocated memory!
        m_data = nullptr;
        m_length = 0;
    }

    // templated operator[] function defined below
    T& operator[](int index); // now returns a T&

    int getLength() const { return m_length; }
};

// Definition of Array<T>::operator[] moved into Array.cpp below

#endif
```

Array.cpp:

```
#include "Array.h"

// member functions defined outside the class need their own template declaration
template <typename T>
T& Array<T>::operator[](int index) // now returns a T&
{
    assert(index >= 0 && index < m_length);
    return m_data[index];
}
```

main.cpp:

```
#include <iostream>
#include "Array.h"

int main()
{
        const int length { 12 };
        Array<int> intArray { length };
        Array<double> doubleArray { length };

        for (int count{ 0 }; count < length; ++count)
        {
                intArray[count] = count;
                doubleArray[count] = count + 0.5;
        }

        for (int count{ length - 1 }; count >= 0; --count)
                std::cout << intArray[count] << '\t' << doubleArray[count] << '\n';

        return 0;
}
```

The above program will compile, but cause a linker error:

```
undefined reference to `Array<int>::operator[](int)'
```

Just like with function templates, the compiler will only instantiate a class template if the class template is used (e.g. as the type of an object like intArray) in a translation unit. In order to perform the instantiation, the compiler must see both the full class template definition (not just a declaration) and the specific template type(s) needed.

Also remember that C++ compiles files individually. When main.cpp is compiled, the contents of the Array.h header (including the template class definition) are copied into main.cpp. When the compiler sees that we need two template instances, Array<int>, and Array<double>, it will instantiate these, and compile them as part of the main.cpp translation unit. Because the `operator[]` member function has a declaration, the compiler will accept a call to it, assuming it will be defined elsewhere.

When Array.cpp is compiled separately, the contents of the Array.h header are copied into Array.cpp, but the compiler won't find any code in Array.cpp that requires the Array class template or `Array<int>::operator[]` function template to be instantiated -- so it won't instantiate anything.

Thus, when the program is linked, we'll get a linker error, because main.cpp made a call to `Array<int>::operator[]` but that template function was never instantiated!

There are quite a few ways to work around this.

The easiest way is to simply put all of your template class code in the header file (in this case, put the contents of Array.cpp into Array.h, below the class). In this way, when you #include the header, all of the template code will be in one place. The upside of this solution is that it is simple. The downside here is that if the template class is used in many files, you will end up with many local instances of the template class, which can increase your compile and link times (your linker should remove the duplicate definitions, so it shouldn't bloat your executable). This is our preferred solution unless the compile or link times start to become a problem.

If you feel that putting the Array.cpp code into the Array.h header makes the header too long/messy, an alternative is to move the contents of Array.cpp to a new file named Array.inl (.inl stands for inline), and then include Array.inl at the bottom of the Array.h header (inside the header guard). That yields the same result as putting all the code in the header, but helps keep things a little more organized.

Tip

If you use the .inl method and then get a compiler error about duplicate definitions, your compiler is most likely compiling the .inl file as part of the project as if it were a code file. This results in the contents of the .inl getting compiled twice: once when your compiler compiles the .inl, and once when the .cpp file that includes the .inl gets compiled. If the .inl file contains any non-inline functions (or variables), then we will run afoul of the one definition rule. If this happens, you'll need to exclude the .inl file from being compiled as part of the build.

Excluding the .inl from the build can usually be done by right clicking on the .inl file in the project view, and then choosing properties. The setting will be somewhere in there. In Visual Studio, set "Exclude From Build" to "Yes". In Code::Blocks, uncheck "Compile file" and "Link file".

Other solutions involve #including .cpp files, but we don't recommend these because of the non-standard usage of #include.

Another alternative is to use a three-file approach. The template class definition goes in the header. The template class member functions goes in the code file. Then you add a third file, which contains *all* of the instantiated classes you need:

templates.cpp:

```
// Ensure the full Array template definition can be seen
#include "Array.h"
#include "Array.cpp" // we're breaking best practices here, but only in this one
place

// #include other .h and .cpp template definitions you need here

template class Array<int>; // Explicitly instantiate template Array<int>
template class Array<double>; // Explicitly instantiate template Array<double>

// instantiate other templates here
```

The “template class” command causes the compiler to explicitly instantiate the template class. In the above case, the compiler will stencil out definitions for Array<int> and Array<double> inside of templates.cpp. Other code files that want to use these types can include Array.h (to satisfy the compiler), and the linker will link in these explicit type definitions from template.cpp.

This method may be more efficient (depending on how your compiler and linker handle templates and duplicate definitions), but requires maintaining the templates.cpp file for each program.