In this Chapter:

4.3 Class Templates

Sue understands how to make a template out of a function, but that does not help her make an abstract datastructure. She would like to make an array-like data-structure that works with any data type. If only there was a way to apply the template mechanism to a class...

Objectives

By the end of this chapter, you will be able to:

- Design a class with an abstract data type
- Create a class template
- Explain situations when class templates would be a useful tool

Prerequisites

Before reading this chapter, please make sure you are able to:

- Write a class definition using built-in data types
- Create a function template

What are class templates and why you should care

A class template is similar to a function template with one exception: one or more member variable (rather than a parameter) uses an abstract data type. While function templates were useful for implementing generic algorithms (like the bubble sort or the swap function), class templates are useful for implementing generic data-structures (like the stack or an array).

```
template <class T>
class Store
{
    public:
        Store(const T & t) : t(t) {}
        T get() const { return t; }
    private:
        T t;
};
int main()
{
    Store <double> storeNum(3.14149);
    cout << storeNum.get() << endl;
    string text("Text");
    Score <string> storeText(text);
    cout << storeText.get() << endl;
    return 0;
}
```

```
class Store
{
    public:
        Store(const double & t) : t(t) {}
        double get() const { return t; }
    private:
        double t;
};
class Store
{
    public:
        Store(const string & t) : t(t) {}
        string get() const { return t; }
    private:
        string t;
};
int main()
{
    Store <double> storeNum(3.14149);
    cout << storeNum.get() << endl;
    string text("Text");
    Score <string> storeText(text);
    cout << storeText.get() << endl;
    return 0;
}
```

Defining a class template

Defining a class template is exactly the same as defining any other class with the exception of the template prefix and the type parameter. There are, however, two additional quirks. First, the template prefix needs to go before every method definition. This can get tedious, but the C++ language demands it. Second, the full name of a template class includes the template designation. In other words, a template class Store is actually “Store ” because the template designation is an integral part of the class identity.

Template prefix

Just as a template prefix needed to go before the function definition of a template function, the template prefix needs to go before the class definition for a class template. Consider, for example, a class designed to store a value:

```
template <class T>
class Store
{
    public:
        Store(const T & t) : t(t) {}
        T get() const { return t; }
    private:
        T t;
};
```

The template prefix signifies that the data type T can be used anywhere in the class definition. This includes as a parameter to a member function, a local variable within a member function, and as a member variable.

Of course it is possible to define more than one data type in a template prefix:

```
template <class T1, class T2>
class StoreTwo
{
    public:
        Store(const T1 & t1, const T2 & t2) : t1(t1), t2(t2) {}
        T1 getOne() const { return t1; }
        T2 getTwo() const { return t2; }
    private:
        T1 t1;
        T2 t2;
};
```

Notice how the getOne() method returns a T1 but does not take a T1 as a parameter. This was impossible with function templates because the client cannot specify which version of the function to use. We do not have this constraint with class templates. The client specifies the version of the class at declaration time. See the section “Declaring an object from a class template” in two pages for details of how this works.

Type parameter

Once the template prefix has been defined, it can be used anywhere a standard data type can be used. In the case of class templates, this almost always means it is used to define member variables. However, it is also commonly used as parameters, return data types, and local variables.

```
template <class T>
class Store
{
    public:
        // type parameter used as a parameter
        T get() const { return t; }
    private:
        T t;
};
```

Class name

When a template is used with a class, the complete class name includes the template designation. Back to our previous example, there is no such thing as the “Store” class! Instead there is the “Store” class.

To illustrate this point, consider the copy constructor. As you may recall from Chapter 2.4, the copy constructor takes a constant parameter of the same data type as the class. Since the data type of the class is Store (not Store!), the copy-constructor must be:

```
template <class T>
class Store
{
    public:
        Store(const T & t) : t(t) {}
        Store(const Store<T> & t) // notice the data type of the parameter
        {
            this->t = t.get();
        }
        T get() const { return t; }
        void set(const T & t);
    private:
        T t;
};
```

A common mistake is to forget the template designation in the class name.

Method definitions

The final aspect of defining a class template pertains to defining a method outside the class definition. Recall that methods defined inside a class definition are inline. Inline functions and methods should be trivial functions, consisting of just a few lines. Unlike normal non-template classes, template classes must be defined entirely in the header (.h) file. This includes the template class definition, inline method definitions, and noninline definitions.

Every method definition needs to be preceded by the template prefix. Additionally, since method definitions include the class name, we need to include the template designation as well. Consider a simple `set()` method for our Store $< T >$ class:

```
/*************************************
    * STORE<T> :: SET
    *************************************/
template <class T>
void Store<T> :: set(const T & t)
{
    this->t = t;
}
```

Notice the template prefix immediately before the method definition. Also notice that the full class name of "Store" must be used.

Sam's Corner

You do not need to use T for your template parameter name. Any other designation can be used. However, we almost always use T so it is clearly recognized as a template parameter. If you teé, particularly creative, you could use T for the template parameter name in the class definition and Q in the method name. There is nothing to keep the programmer from doing this. That being said, it is generally considered to be a very bad idea! Always use T (or $T1$ and $T2$) for your template parameter names.

Declaring an object from a class template

The final aspect of using a class template is that, at object declaration time, the flavor of the class template needs to be specified. With a function template, the parameters passed to the function tell the compiler which version of the function is needed. With a class template, we need to be explicit. For example, if the user wishes to instantiate a double version of the Store class, the following declaration is used:

```
int main()
{
    Store<double> s(3.14159); // use the "double" version of Store<T>
    cout << s.get() << endl;
    return 0;
}
```

This syntax should be a bit familiar. Recall the syntax for declaring a vector of integers:

```
#include <vector>
int main()
{
    vector<int> numbers; // use the "int" version of vector<T>
    return 0;
}
```

If more than one type parameter is used in a class template, we include all the data types in the <> :

```
int main()
{
    StoreTwo<double,int> s(3.14159, 42); // use the "double, int" version of
    cout << s.getOne() << endl; // StoreTwo<T1, T2>
    return 0;
}
```

Designing with class templates

The most common use for class templates is to define a custom data-structure. A data-structure is a mechanism for storing or organizing data in a program. Thus far we have discussed data-structures only sparingly. An array is a data-structure, as is a vector. We will learn about a new data-structure called a linked list next chapter. In each of these cases, the way that the data-structure works with the individual entities should be completely independent of the nature of the entity. In other words, arrays all work the same, be they float arrays or string arrays. This is why data-structures are commonly defined with class templates.

Sue's Tips

Data-structures represent a rich topic of discussion, far too rich for this text. Fortunately we have an entire class devoted to studying data-structures: CS 235.

Example 4.3 - Store

This example will demonstrate the use of a class template with the simplest data-structure imaginable: one to store a single value for the user.

Write a class to store a single value for the user. The value can be any data type specified by the user, as long as the data type supports the copy constructor and the assignment operator. An example of the use includes:

```
{
    Store<double> s(3.14159); // use the "double" version of Store<T>
    cout << s << endl;
}
```

The class definition of the Store $< T >$ class is mostly inline because the class is trivial:

```
template <class T>
class Store
{
    public:
        Store(const T & t) : t(t) { }
        Store(const Store<T> & store) { *this = store; }
        T get() const { return t; }
        void set(const T & t);
        Store<T> & operator = (const Store<T> & store)
        {
            this->t = store.get();
            return *this;
        }
        friend ostream & operator << (ostream & out, const Store<T> & store)
        {
            out << store.get();
            return out;
        }
    private:
        T t;
};
```

Additionally one method is not defined inline (though it should!), Store::set():

```
template <class T>
void Store<T> :: set(const T & t)
{
    this->t = t;
}
```

As a challenge, can you overload the extraction operator (>>), the assignment operator with a T as the right-hand-side (as opposed to a Store $\langle T \rangle$), and the Store $\langle T \rangle$: :set() method taking a Store $\langle T \rangle$ on the right-hand-side?

The complete solution is available at 4-3-Store.html or: /home/cs165/examples/4-3-Store.cpp

Example 4.3 - Container

This example will demonstrate a more complex and worthwhile use of a class template: an array data type that can store any type of data.

Write a class to emulate the behavior of the vector $\langle T \rangle$ class in the standard template library. This class will support the copy constructor to copy one array onto another, the get method for read/write access, insert method to push an item onto the array, and a variety of other convenient methods and operators.

The most important part of the class definition includes:

```
template <class T>
template <class T>
class Container
{
public:
    // constructors
    Container() : numItems(0), capacity(0), data(0x00000000) {}
    Container(const Container & rhs) throw (const char *);
    Container(int capacity) throw (const char *);
    // destructor : free everything
    ~Container() { if (capacity) delete [] data; }
    // is the container currently empty
    bool empty() const { return numItems == 0; }
    // remove all the items from the container
    void clear() { numItems = 0; }
    // how many items are currently in the container?
    int size() const { return numItems; }
    // add an item to the container
    void insert(const T & t) throw (const char *);
    T & get(int index) throw (const char *);
private:
    T * data; // dynamically allocated array of T
    int numItems; // how many items are currently in the Container?
    int capacity; // how many items can I put on the Container before full?
};
```

As a challenge, can you modify the insert method so the container grows to accommodate any amount of items? See example “4.1 Expanding Arrays” from “Procedural Programming in C++” for details of how to make the buffer grow.

The complete solution is available at 4-3-container.html or: /home/cs165/examples/4-3-container.cpp

Problem 1 - 3

Consider a class representing a pair of values. Examples could be a pair of integers (x, y for example), a pair of names (first and last for example) or a Date paired with a string.

1. Write a class definition for a pair of values. Include three constructors, a getter, a setter, and two member variables.

2. Write the function definitions for the copy constructor, the getter, and the setter.

3. Given our new class, declare two objects that are: an integer paried with a float, a string paired with a string. Declare the objects and set them to a value.