

# Section Notes 8

CS51—Spring, 2008

Week of April 13, 2008

## 1 Templates

In C and C++, the usefulness of your code is limited when it is written to support only a particular data type. For instance, if you were to write a linked list class in C, you might specify the type of data to store in the list (for instance, an int). But what if you later want to store another type of data in your linked list?

Templates are designed to solve this problem. The programmer can specify the type(s) used in a particular instance of a class when the object (or pointer to an object) is declared. (Rather than having the type specified explicitly when the code for the class is written.) By adding a little bit of work up front to make the class definition work for *any* data type, you can avoid having to go back and change the specification whenever a new data type must be supported. The programmer using the class can then say, "give me support for X type" when declaring an object, and the necessary type is filled into the template provided. (This type of programming is sometimes known as "generic programming" because the class is generic rather than tied to a particular data type.)

To declare a templated class, you must specify that the class is a template by using the `template` keyword, and specify a list of the *template parameters*, separated by commas, in angle brackets (< and >), followed by a standard class definition. Template parameters can be either constant values of a particular type, such as an int, or they can be a type itself. For now, we restrict our use to defining templates that accept types as arguments. To do so, the template parameter should be specified as `typename` (or `class` - the two terms are interchangeable in this context).

```
template <typename T>
class LinkedListNode { // ... };
```

Within the class, member and method declarations are mostly unchanged. The only difference between a templated class and a non-templated class is that whenever a variable or return value should be of a type given by a template parameter, you must use the name of the parameter in place of the type.

```
template <typename T>
class LinkedListNode {
public:
    T getVal();
    void setVal(T v) { val = v;}

private:
    T val; // store a value of whatever type is specified
};
```

Declaring an instance of a template is as simple as passing the desired types into the template specification: `LinkedListNode<int> node`. Note that parameters to a templated class can be passed into other templates. With this, we can finish our `LinkedListNode` interface.

```

template <typename T>
class LinkedListNode {
public:
    T getVal();
    void setVal(T v) { val = v; }

    // We'll define this outside the class declaration
    LinkedListNode(T v, LinkedListNode<T> *n);

private:
    T val; // store a value of whatever type is specified
    LinkedListNode<T> *next;
};

```

When defining member functions outside the class declaration for a templated class, you have to re-specify that you are working with a template, give the template parameters, and include the list of parameters to the template as part of the class name.

```

template <typename T>
LinkedListNode<T>::LinkedListNode(T v, LinkedListNode<T> *n)
{
    val = v;
    next = n;
}

```

Mercifully, we don't need to include the template parameter as part of the constructor name.

## 1.1 Templated functions

It is also possible to create templated functions. When templated functions are used, it is often possible for the template type to be correctly deduced from the arguments to the function. There may be no need to explicitly pass the type as a template argument (this isn't always the case, however—why not?).

```

template <typename T>
T max(T a, T b)
{
    return a > b ? a : b;
}

// The type for max is correctly deduced from the arguments
int k = max(4, 5);

// But you could be explicit if you want
int k = max<int>(4, 5);

```

## 1.2 Template issues

When declaring an instance of a templated class with a templated type as a parameter, you might think the declaration would look something like this

```
templateOne<templateTwo<int>> x; // bad code!
```

But this code is wrong. Notice the presence of two > in a row. This (>>) is the right-shift operator (which `cin` uses). Having what appears to be the right-shift operator in the middle of a variable declaration will confuse the compiler, so you must write this kind of declaration with a space between the two greater-than signs or this code will not compile. You may wish to center the entire parameter.

```
templateOne< templateTwo<int> > x;
```

Template errors are notoriously inscrutable, especially once you start to use the Standard Template Library, where you will not necessarily know what all of the parameters do. (Many parameters are specified using defaults that work for common cases.)

Another gotcha is that the **code for a templated class must appear in the header file**. You cannot split the class declaration and the class implementation across different files. There are some exceptions to this rule that are used in very large projects, but they are well beyond the scope of this class.<sup>1</sup>

Finally, for every different template parameter given to a templated class, entirely new code will be generated. The compiler should restrict itself to generating only the functions necessary, but this can nevertheless increase the resulting program size significantly.

On the other hand, the fact that this code is generated at compile time means that using templates should not have a run-time cost. (This is why all template parameters must be constant—if they weren't, it would be impossible to generate code for templates at compile time.)

## 2 A taste of the STL—using vector

The C++ standard library provides numerous container classes that give you a variety of ways to store data. (Why might you need more than one way of storing data?) Arrays are one example of a structured way of storing and accessing data, but arrays have some problems—they aren't resizable and they are a common source of memory errors.

The *Standard Template Library (STL)* provides a replacement for arrays—the **vector** template class. Note that vectors require you to include `using namespace std;` and the `<vector>` header.

Vectors can be specified to have a particular initial size with the constructor:

```
using namespace std;
vector<int> intVector(10);
```

Vectors will automatically resize if necessary when new elements are added using the `push_back` function, which adds an element to the end of a vector. (What data structure does this remind you of?)

Vectors also allow numerically indexed access using the same syntax as an array—note that vectors, like arrays, are not bounds-checked, so it's perfectly possible to walk off the end of a vector:

```
intVector[0] = 1;
intVector[2] = 1;
intVector[10] = 1; // Oops!
```

---

<sup>1</sup>For a brief discussion of how templates are broken across multiple files including “tpp” files, Google for a paper by Daniel Spiegel called “Teaching template classes with all the advantages”.

### 3 Iterators

It would be convenient to have a single interface for accessing data stored in different types of STL containers (such as `vector`—if we decide to use a different data type, we won't have to change how we access the data.

An “iterator” provides sequential access to all elements stored in a container. An iterator stores a current location in the container whose data is being accessed. Iterators are a common concept in program design when working with abstract data types.

The syntax for declaring an iterator is a bit complicated: the iterator is defined within the templated class, and must be accessed using the scope operator. For example, to declare an iterator over a `vector` containing `ints`:

```
vector<int>::iterator vectorIterator;
```

Placing the iterator class inside the container class makes it clear that different iterators have different ways of accessing the elements of the container. (Why might this be?)

Once you have an iterator, you will need to initialize it using the container you are accessing. All STL containers have member functions `begin` and `end`, which return iterators placed either at the beginning or immediately past the end of the collection. (That is, the iterator returned by `end` does not refer to an element in the collection.)

```
vectorIterator = aVector.begin();
```

To move an iterator onto the next element in a collection, you can use the `++` operator, and to move backward (if supported by the particular collection), you can use `--`.

```
vectorIterator++;
```

To access the element of the collection at which the iterator is located, you can use the `*` operator. Notice that this is very much like dereferencing a pointer. This makes a lot of sense if you think of an iterator as storing a location, just as a pointer stores a location in memory.

What does this code do?

```
using namespace std;

vector<int> intVector;
vector<int>::iterator vectorIterator;

for(int i = 0; i < 10; i++) {
    intVector.push_back(i);
}

// print everything in the vector
for(vectorIterator = intVector.begin();
    vectorIterator != intVector.end();
    vectorIterator++) {
    cout << *vectorIterator << endl;
}
```

## 4 const

We've already said that it's more efficient to pass references to objects rather than whole objects. However, the downside of this is that a function with a reference might modify the passed object unpredictably. You can pass a `const` reference to prevent this:

```
void Screen::drawShape(const Shape &s);
```

This indicates that `drawShape` is allowed to examine the referenced `Shape` in order to draw it, but it may not modify the shape.

### 4.1 const methods

So far so good, but how does `drawShape` know which operations on the `Shape` are fair game in order not to modify it? The answer is that you must declare which `Shape` methods modify shapes and which do not:

```
class Shape
{
    ...
    void translate(double dx, double dy);
    void getX() const;
    void getY() const;
    ...
}
```

### 4.2 const pointers vs. pointers to const

The syntax for `const` can be confusing when declaring a reference or pointer:

```
int const *pInt;
const int *pInt;
```

What, exactly, is constant in these declarations? It turns out that in both cases, `pInt` will be an ordinary pointer to a constant integer. `pInt` may change, but `*pInt` may not. On the other hand, with

```
int * const pInt;
```

`pInt` may not change, but `*pInt` may. Can you guess how you would declare a pointer to an integer where both the integer and the address will not be changed?

## 5 Code reading: exceptions

```
#include <exception>
#include <map>
#include <vector>
#include <iostream>

class DivideByZero : public std::exception {
public:
    const char* what() const throw () {
        return "Attempt to divide by zero";
    }
};
```

```

class NullArgumentException : public std::exception {
public:
    const char* what() const throw () {
        return "Unexpected NULL pointer passed to function";
    }
};

template <class T>
class Stats
{
public:
    T mean;
    T median;
    std::vector<T> mode;
};

double safe_divide(double numerator, double denominator) throw (DivideByZero)
{
    if (denominator == 0) {
        throw DivideByZero();
    }
    return numerator / denominator;
}

double compute_average(double *inputs, int size)
{
    if (inputs == NULL) {
        throw NullArgumentException();
    }
    double sum = 0;
    for (int i = 0; i < size; ++i) {
        sum += inputs[i];
    }
    return safe_divide(sum, size);
}

double compute_median(double *inputs, int size)
{
    if (inputs == NULL) {
        throw NullArgumentException();
    }
    std::sort(inputs, inputs + size);
    return (size % 2 == 0) ? (inputs[size/2 - 1] + inputs[size/2]) / 2 :
        inputs[size/2];
}

std::vector<double>& compute_mode(double *inputs, int size)
{
    std::map<double, int> counts;
    for(int i = 0; i < size; ++i) {
        ++counts[inputs[i]];
    }
}

```

```

    }
    int max_count = 0;
    std::vector<double> max_vals;

    for (std::map<double, int>::iterator itr = counts.begin(),
         end = counts.end();
         itr != end; ++itr) {
        if ((*itr).second > max_count) {
            max_count = (*itr).second;
            max_vals.clear();
        }
        if ((*itr).second == max_count) {
            max_vals.push_back((*itr).first);
        }
    }

    return max_vals;
}

Stats<double> *compute_stats(double *inputs, int size)
{
    if (inputs == NULL) {
        throw NullArgumentException();
    }
    Stats<double> *stats = new Stats<double>;

    try {
        stats->mean = compute_average(inputs, size);
        stats->median = compute_median(inputs, size);
        stats->mode = compute_mode(inputs, size);
    }
    catch(DivideByZero&) {
        delete stats;
        throw;
    }

    return stats;
}

void print_stats(const Stats<double>& stats)
{
    std::cout << "Mean: " << stats.mean << std::endl;
    std::cout << "Mode" << ((stats.mode.size() > 1) ? "s" : "") << ": ";

    for (std::vector<double>::const_iterator itr =
         stats.mode.begin(), end = stats.mode.end();
         itr != end; ++itr) {
        if (itr != stats.mode.begin()) {
            std::cout << ", ";
        }
    }
}

```

```

        std::cout << *itr;
    }

    std::cout << std::endl;
    std::cout << "Median: " << stats.median << std::endl;
}

int main()
{
    int size;
    Stats<double> *stats = NULL;
    double *inputs = NULL;

    do {
        std::cin >> size;
    } while (size < 0);

    try {
        inputs = new double[size];
        for (int i = 0; i < size; ++i) {
            std::cin >> inputs[i];
        }
        stats = compute_stats(inputs, sizeof(inputs));
    }
    catch(std::bad_alloc& out_of_memory) {
        std::cerr << "Out of memory: " << out_of_memory.what() << std::endl;
        delete [] inputs;
        return 1;
    }
    catch(DivideByZero&) {
        std::cerr << "Division by zero ";
        delete [] inputs;
        return 1;
    }
    catch(...) {
        std::cerr << "Unexpected exception caught" << std::endl;
        delete [] inputs; // delete stats;
        return 1;
    }

    print_stats(*stats);
    delete stats;
    delete [] inputs;
    return 0;
}

```

- Is the stats class an example of good design?
- Why is the catch inside compute\_stats necessary? How could it be avoided?
- What is the purpose of catch(...)?

- Why doesn't the `catch(...)` clause delete stats?
- How could the code be rewritten to avoid the multiple calls to delete inputs in main?
- `std::sort` expects an iterator. Why are pointers acceptable instead?
- What do you think of `compute_mode` returning a reference to avoid copying the return value?
- What other inconsistencies do you notice?