CS 11 Data Structures and Algorithms

Lesson 19: Templates, Exceptions, and STL

Skip to Main Content

Section 19.1: Using the STL -- Vector

The C++ Standard Template Library (STL) is a set of classes available for you to use that act as "Containers" for other objects. (There's a little more to the STL than this, but this is a good place to start for now.) For example, one of the STL classes is named "vector". An STL vector is similar to an array, but easier to work with. Just like arrays, you could have a vector of anything: a vector of ints, or a vector of feetInches objects, or a vector of strings, for example. Unlike arrays, you can always add something to a vector and not have to worry about resizing the vector if its capacity is exceeded, you can at any point ask for the size of (number of items in) a vector, or even easily insert something into the middle of a vector without worrying about shifting everything down to make room for the new item.

When you declare a vector (or any other STL container class), you place the type of item that will be placed in the vector in angle brackets after the name of the STL class. For example, to declare a vector that will contain variables of type int, the declaration would be

```
vector<int> vector1;
```

The most fundamental way of adding items to a vector is with the "push_back()" function. For example, the statement

```
vector1.push_back(47);
```

will add the item 47 to the back of the vector. We can continue to add items to the back of a vector indefinitely.

The capacity of vectors change as they are being used. The capacity can be set manually by the programmer by using the "resize()" function, but if the items are added using the push_back() function, the vector will automatically change its capacity as needed. The C++ standard does not require that this be done according to a particular formula, but a simple and common technique is that the size of the vector will be doubled each time the capacity is about to be exceeded. For example, if the capacity of a vector is 8, and it already contains 8 items, then when we call push_back again to add an additional item, the capacity of the vector will be doubled to 16. The output of the sample program below will demonstrate this.

You can see a list of member functions that are available to vector objects in your text, but to get us started we will work with just a few:

- size(): Returns the number of items in the vector
- capacity(): Returns the current capacity of the vector
- operator[]: Allows access to individual elements of the vector; however, unlike arrays, it can only be used to access elements that already exist. New elements cannot be created. Also, this operator does not check to ensure that the index is within the bounds of the vector.
- at(): Like operator[], but includes bounds checking.

Here is an example that incorporates these member functions and demonstrates how size and capacity work together. You may want to try this out on your own, and experiment with uncommenting the last cout statement.

```
#include <iostream>
#include <vector>
using namespace std;

void print(vector<int>& v);

int main() {
    vector<int> v;
    int currentSize;

    for (int i = 0; i < 20; i++) {
        v.push_back(i);

        currentSize = v.size();
        cout << "size: " << currentSize << " capacity: " << v.capacity() << endl;
    }

    print(v);
}



void print(vector<int>& printMe) {
    for (int i = 0; i < printMe.size(); i++) {
        cout << printMe.at(i) << " ";
    }
    cout << endl;
```

```
    for (int i = 0; i < printMe.size(); i++) {
        cout << printMe[i] << " ";
    }
    cout << endl;

    // cout << printMe.at(150) << endl;
}
```

```
size: 1 capacity: 1
size: 2 capacity: 2
size: 3 capacity: 4
size: 4 capacity: 4
size: 5 capacity: 8
size: 6 capacity: 8
size: 7 capacity: 8
size: 8 capacity: 8
size: 9 capacity: 16
size: 10 capacity: 16
size: 11 capacity: 16
size: 12 capacity: 16
size: 13 capacity: 16
size: 14 capacity: 16
size: 15 capacity: 16
size: 16 capacity: 16
size: 17 capacity: 32
size: 18 capacity: 32
size: 19 capacity: 32
size: 20 capacity: 32
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

Section 19.2: size_type

Each STL class includes a type named "size_type". Anytime a client program declares a variable that is used to count items in an STL container class, it should be declared to be of type size_type instead of int (or some other integer type). Because the size_type is defined inside the STL container class, we need to precede it with the scope resolution operator. The syntax, using vector as an example, is "vector<int>::size_type myVariable".

The rationale, part 1: This makes it so that there won't be type mismatches between the types used by the client and the types used by the STL class. For example, if the STL class has a function that returns the number of items in a container and it returns an unsigned int, but the client assigns that value to a variable that has been declared to be an int, there is a type mismatch. But if both use the "size_type" type, there won't be a mismatch.

The rationale, part 2: On any given system, it's not guaranteed that a variable of type int (or any other integer type) will be big enough to represent the number of items in a container. The "size_type" type provided by each STL class is guaranteed to be able to represent the maximum number of items that an STL container class object can contain.

Here is the resulting code:

```
#include <iostream>
#include <vector>
using namespace std;

void print(vector<int>& v);

int main() {
    vector<int> v;
    vector<int>::size_type currentSize;

    for (int i = 0; i < 20; i++) {
        v.push_back(i);

        currentSize = v.size();
        cout << "size: " << currentSize << " capacity: " << v.capacity() << endl;
    }

    print(v);
}



void print(vector<int>& printMe) {
    for (vector<int>::size_type i = 0; i < printMe.size(); i++) {
        cout << printMe.at(i) << " ";
    }
    cout << endl;

    for (vector<int>::size_type i = 0; i < printMe.size(); i++) {
        cout << printMe[i] << " ";
    }
    cout << endl;
}
```

Section 19.3: Iterators

In our example, we iterate over all of the items in the vector in order to print them. There are plenty of other reasons we might want to iterate over all of the items in a container. For example, we might want to iterate over all of the items because we are searching for a particular item in the container, or because we need to somehow modify the value of each item. As you can see in the example from section 1, it's pretty easy to iterate over all of the items in a vector. Because vectors allow for random access just like arrays do, we can simply use a for loop with an integer counter. However, most containers do not allow random access and are not as easy to iterate over. For this reason, there is a common notation used by all STL container classes that allows a client to iterate over all of the items in the container. As you'll see, the notation adopts the most convenient features of array indexing and pointers to enable this. The best way to think about iterators is probably to think of them as pointer-like but with a very limited functionality and a couple of notational additions that seem more like array indexing.

Iterating over a vector using an iterator follows exactly the same pattern as iterating over a vector using indexing. Here is how the two techniques match up in a for loop:

```
for (int i = 0; i < printMe.size(); i++) {
    cout << printMe.at(i) << endl;
}

for (vector<int>::iterator i = printMe.begin(); i != printMe.end(); i++) {
    cout << *i << endl;
}
```

The five steps in both cases are (1) declare i, (2) initialize i to the beginning of the vector, (3) compare it to the end of vector, (4) advance i to next item in the vector, and (5) access the item at position i in the vector.

1. Declare: With iterators, instead of declaring i to be a variable of type int, we declare it to be a variable of type "iterator". However, the "iterator" type is defined inside each STL container class, so we need to precede it with the scope resolution operator, as demonstrated above.

2. Initialize: With iterators, instead of setting i to 0, we set the iterator to the beginning of the container with the begin() function, which is a member function of each STL container class. The begin() function returns an iterator to the first item in the container.

3. Compare: With iterators, instead of comparing i with the size of the container, we compare i to the iterator returned by the end() function, which is a member of each STL container class. The end() function returns an iterator to **one past** the last item in the container. This may seem odd at first, but actually, when you use indexing and you compare i to the size of a container, you are comparing it to the index of the item **one past** the last item in the container. (If the size of the container is N, the index of the last item in the container is N - 1.) Another point here is that all iterators in the STL have a not-equal-to (!=) operator, but not all iterators in the STL have a less than (<) operator. So instead of using "<" to make this comparison, we use "!=".

4. Advance: Here's where the iterator notation becomes more array-like instead of pointer-like. In every STL container class, no matter what sort of structure is used to store the items in the container, you can always advance an iterator from one item to the next item using the increment (++) operator.

5. Access: With iterators, instead of accessing the item at position i in the vector with the syntax printMe[i] or printMe.at(i), we use the "dereference" operator borrowed from pointer notation: *i.

Here's our program using iterators instead of indexing:

```
#include <iostream>
#include <vector>
using namespace std;

void print(vector<int>& v);

int main() {
    vector<int> v;
    vector<int>::size_type currentSize;

    for (int i = 0; i < 20; i++) {
        v.push_back(i);

        currentSize = v.size();
        cout << "size: " << currentSize << " capacity: " << v.capacity() << endl;
    }

    print(v);
}



void print(vector<int>& printMe) {
    for (vector<int>::iterator i = printMe.begin(); i != printMe.end(); i++) {
```

```
        cout << *i << " ";
    }
    cout << endl;
}
```

Section 19.4: Const Iterators

You've worked with passing objects as parameters for awhile now, so there's something that should look wrong to you about the code above. The printMe object should be a const reference parameter, like this:

```
void print(const vector<int>& printMe) {
```

However, if I modify the parameter list like this and try to compile the program, I get an ugly error message. Here's what the error message looks like on my system:

```
c.cpp: In function 'void print(const std::vector >&)':
c.cpp:24: error: conversion from '__gnu_cxx::__normal_iterator > >' to non-scalar type '__gnu_cxx::__normal_iterator > >' requested
```

Don't worry if that doesn't make a lot of sense. The problem is that when we have a const vector instead of a vector, the begin() member function doesn't return a normal iterator. It returns something called a const_iterator. And we can't assign that const_iterator to the normal iterator "i".

The solution is to declare the iterator "i" to be of type const_iterator instead of just plain iterator. Here's the updated version of our print() function:

```
void print(const vector<int>& printMe) {
    for (vector<int>::const_iterator i = printMe.begin(); i != printMe.end(); i++) {
        cout << *i << " ";
    }
    cout << endl;
}
```

Section 19.5: Other Iterator Details

There are different categories of iterators in C++, which vary according to what operations are defined for them. My recommendation at this point is to just use the operations that you know are defined for ALL iterators (or at least all iterators that you are likely to encounter). That's just dereference (*), pre-increment and post-increment (++), equal-to (==), not-equal-to (!=), and dereference-and-select (->). If at some point in the future you need another operation (such as other relational operators or decrement (--)), you can do some research online. Start with **http://www.cplusplus.com/reference/iterator**.

Section 19.6 Template Functions

You can create a function that can accept any type of parameter. For example, we can write a sort function that will sort an array of any type of element. First, here is a program that includes a normal (not-templated) sort, along with a sample output. It randomly generates 25 integers, sorts them, then prints them.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

const int ARRAY_SIZE = 25;

int indexOfSmallest(const int list[], int startingIndex, int numItems);
void sort(int list[], int numItems);

int main()
{
    srand(static_cast(time(0)));

    int list[ARRAY_SIZE];
    int numItems;

    for (int i = 0; i < ARRAY_SIZE; i++) {
        list[i] = rand() % 1000;
    }

    sort(list, ARRAY_SIZE);

    for (int i = 0; i < ARRAY_SIZE; i++) {
        cout << list[i] << " ";
    }
    cout << endl;
}



void sort(int list[], int numItems)
{
    for (int count = 0; count < numItems - 1; count++){
        swap(list[indexOfSmallest(list, count, numItems)],
             list[count]);
    }
```

```
}


int indexOfSmallest(const int list[], int startingIndex, int numItems)
{
    int targetIndex = startingIndex;

    for (int count = startingIndex + 1; count < numItems; count++){
        if (list[count] < list[targetIndex]){
            targetIndex = count;
        }
    }

    return targetIndex;
}
```

```
42 73 157 165 169 249 272 303 327 440 492 503 544 560 612 658 709 709 729 807 840 878 923 930 987
```

In order to write a function that can work with any type of parameter, we just add something called a "template prefix" on the line before the function header. The template prefix looks like this:

```
template <class T>
```

Then, inside the function itself, everywhere we see the type that we want to replace, we use "T" instead of the type. "T" is called the "type parameter".

(The "T" is just an identifier. You can use something else if you want. I've seen "object" used instead of "T", for example. However, there is a strong convention to use "T".)

Here is our program again, using a templated version of the sort() function, so that we can call it with an array of ints first, and then an array of strings.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <string>
using namespace std;

const int ARRAY_SIZE = 25;

template <class T>
void sort(T list[], int numItems);

template <class T>
int indexOfSmallest(const T list[], int startingIndex, int numItems);


int main()
{
    srand(static_cast<unsigned> (time(NULL)));

    int list[ARRAY_SIZE];

    for (int i = 0; i < ARRAY_SIZE; i++) {
        list[i] = rand() % 1000;
    }

    sort(list, ARRAY_SIZE);

    for (int i = 0; i < ARRAY_SIZE; i++) {
        cout << list[i] << " ";
    }
    cout << endl;

    string stringList[ARRAY_SIZE];

    for (int i = 0; i < ARRAY_SIZE; i++) {
        stringList[i] = "";
        int stringlength = rand() % 6 + 1;
        for (int j = 0; j < stringlength; j++) {
            stringList[i] = stringList[i] + char(rand() % 26 + 'a');
        }
    }

    sort(stringList, ARRAY_SIZE);

    for (int i = 0; i < ARRAY_SIZE; i++) {
        cout << stringList[i] << " ";
    }
    cout << endl;
}


template <class T>
void sort(T list[], int numItems)
{
    for (int count = 0; count < numItems - 1; count++){
        swap(list[indexOfSmallest(list, count, numItems)],
             list[count]);
```

```
    }

}


template <class T>
int indexOfSmallest(const T list[], int startingIndex, int numItems)
{
    int targetIndex = startingIndex;

    for (int count = startingIndex + 1; count < numItems; count++){
        if (list[count] < list[targetIndex]){
            targetIndex = count;
        }
    }

    return targetIndex;
}
```

```
5 25 26 36 54 74 105 142 260 337 372 423 427 472 500 534 551 630 704 821 850 854 857 892 954
b cpq eqwr esfq fih folxxd fvv hmwrec ics jjtrjr jqym lug me pgmj pp qsow r ros sbawwh ugh v v wbct wko ws
```

Section 19.7 Templated Classes

Being able to write functions that operate on any type of parameter is nice, but the real win comes when we can write container classes that can contain any type of item. As we saw with the vector class, this is what the STL container classes can do. We used a vector of int's by putting "int" in the angle brackets after the word "vector", but we could declare a vector of strings or feetInches objects or any other type.

We're going to demonstrate this by writing our own version of the vector class, which we will call "MyVector". We'll start with a MyVector class that can only contain int's. Then we'll make the necessary changes so that MyVector objects can be created to hold any type of item. First, the MyVector class that can only contain int's:

```
// file myvector.h

#include <cstdlib>

class MyVector {
    public:
        typedef std::size_t size_type;
        typedef int value_type;
        MyVector(size_type inSize = 0, const value_type& inValue = value_type());
        size_type size() const;
        size_type capacity() const;
        value_type at(size_type i) const;
        void push_back(const value_type& insertMe);
    private:
        value_type* items;
        size_type mSize;
        size_type mCapacity;
};
```

```
// file myvector.cpp

#include <cassert>
#include "myvector.h"
using namespace std;

MyVector::MyVector(size_type inSize, const value_type& inValue) {
    mSize = inSize;
    mCapacity = inSize;
    items = new value_type[inSize];
    for (size_type i = 0; i < mSize; i++) {
        items[i] = inValue;
    }
}



MyVector::size_type MyVector::size() const {
    return mSize;
}



void MyVector::push_back(const value_type& inValue){
    if (mSize < mCapacity) {
        items[mSize] = inValue;
        mSize++;
    } else {
        if (mCapacity == 0) {
            mCapacity = 1;
        } else {
            mCapacity *= 2;
```

```
        }
        value_type* temp = new value_type[mCapacity];
        for (size_type i = 0; i < mSize; i++) {
            temp[i] = items[i];
        }
        temp[mSize] = inValue;
        mSize++;
        delete [] items;
        items = temp;
    }
}



MyVector::size_type MyVector::capacity() const {
    return mCapacity;
}



MyVector::value_type MyVector::at(size_type i) const {
    assert(i >= 0 && i < size());
    return items[i];
}
```

```
// file myvector-client.cpp
// notice, very similar to the STL vector client we wrote in section 19.1

#include <iostream>
#include "myvector.h"
using namespace std;

void print(MyVector& printMe);

int main() {
    MyVector v;
    MyVector::size_type currentSize;

    for (int i = 0; i < 20; i++) {
        v.push_back(i);

        currentSize = v.size();
        cout << "size: " << currentSize << " capacity: " << v.capacity() << endl;
    }

    print(v);
}



void print(MyVector& printMe) {
    for (int i = 0; i < printMe.size(); i++) {
        cout << printMe.at(i) << " ";
    }
    cout << endl;
}
```

I would stop what you are doing and sit and study this example for at least 30 minutes or until you have it practically memorized. You need to understand what is going on at every point, and we're going to be working on this example for quite some time, so you'll want to be familiar with it.

Here are three observations about the example.

1. We are defining a new type called "value_type" to represent "int". Then, throughout the rest of the class, we use value_type instead of int whenever we refer to the type of the items in the vector. This will make it easier to change the type to something other than int if we want to. Also, we might as well do this since every STL container has a public member "value_type" that represents the type of the items in the container.

2. The push_back() function is the only member function that is non-trivial. In that function we have to check to see if the vector capacity has been reached. If it has not, we place the item at the back of the MyVector. If on the other hand the capacity HAS been reached, we must double the capacity (or make it 1 if it was originally 0), create a temporary dynamic array with the new capacity, copy all of the items to the temporary array, and finally point our "items" data member to the new array we created.

3. We are defining a new type called "size_type". The reasons for providing this typedef are covered in section 2 of this lesson. The new part is the pre-defined type "std::size_t". This is a type defined by C++ that is guaranteed to store the maximum size of a theoretically possible object of any type. So, it's a good choice for our "size_type" type. We could have just used "typedef int sizetype;" but it wouldn't have been as safe.

Now let's take our MyVector class and turn it into a templated class, so our client can have MyVectors that contain different types of data. The process of converting a non-templated class into a templated class is a 6 step process.

Step 1: Place the template prefix before the class declaration and before each member function definition that occurs outside the class declaration.

Step 2: Place a <T> after each occurrence of the class name that occurs outside the class declaration.

For example, in the case of MyVector, that means that almost every time you see the word "MyVector" outside of the class declaration, you'll put a <T> after it. The only exception is when the word is not used as the name of a class, but rather as the name of a function, as is the case with the constructor.

Step 3: Replace each occurrence of the original type with "T"

Because we used "value_type" throughout our code instead of using "int" everywhere, this step will be easy. We just replace the "int" in the typedef to "T". If we hadn't used the value_type typedef, we would have to search through our entire class looking for int's that should be replaced with "T".

Let's take a look at the code after the first three steps.

```
// file myvector.h
// only two changes in this file

#include <cstdlib>

template <class T>        // added this line
class MyVector {
    public:
        class NegativeSize{};

        typedef std::size_t size_type;
        typedef T value_type;        // changed int to T
        MyVector(size_type inSize = 0, const value_type& inValue = value_type());
        size_type size() const;
        size_type capacity() const;
        value_type at(size_type i) const;
        void push_back(const value_type& insertMe);
    private:
        value_type* items;
        size_type mSize;
        size_type mCapacity;
};
```

```
// file myvector.cpp

#include <cassert>
#include "myvector.h"
using namespace std;

template <class T>
MyVector<T>::MyVector(size_type inSize, const value_type& inValue) {
    mSize = inSize;
    mCapacity = inSize;
    items = new value_type[inSize];
    for (size_type i = 0; i < mSize; i++) {
        items[i] = inValue;
    }
}



template <class T>
MyVector<T>::size_type MyVector<T>::size() const {
    return mSize;
}



template <class T>
void MyVector<T>::push_back(const value_type& inValue){
    if (mSize < mCapacity) {
        items[mSize] = inValue;
        mSize++;
    } else {
        if (mCapacity == 0) {
            mCapacity = 1;
        } else {
            mCapacity *= 2;
        }
        value_type* temp = new value_type[mCapacity];
        for (size_type i = 0; i < mSize; i++) {
            temp[i] = items[i];
        }
        temp[mSize] = inValue;
        mSize++;
        delete [] items;
        items = temp;
    }
```

```
}


template <class T>
MyVector<T>::size_type MyVector<T>::capacity() const {
    return mCapacity;
}




template <class T>
MyVector<T>::value_type MyVector<T>::at(size_type i) const {
    assert(i >= 0 && i < size());
    return items[i];
}
```

```
// file myvector-client.cpp

#include <iostream>
#include "myvector.h"
using namespace std;

void print(MyVector<int>& printMe);

int main() {
    MyVector<int> v;
    MyVector<int>::size_type currentSize;

    for (int i = 0; i < 20; i++) {
        v.push_back(i);

        currentSize = v.size();
        cout << "size: " << currentSize << " capacity: " << v.capacity() << endl;
    }

    print(v);
}




void print(MyVector<int>& printMe) {
    for (int i = 0; i < printMe.size(); i++) {
        cout << printMe.at(i) << " ";
    }
    cout << endl;
}
```

Unfortunately, this code won't compile until we complete steps 4 and 5.

Step 4: Outside of the scope of the class, add the keyword "typename" before any use of one of the types defined in the class.

The types defined in the class are size_type and value_type. Recall that "inside the scope of the class" includes inside the class declaration itself as well as inside a definition of a class member function. Parameter lists are considered inside the scope of the class, because the scope resolution operator has already occurred in the code when the compiler gets to the parameter list. So, typically the only place these will occur outside the scope of the class will be when they are the return type of a class member function. In our example we have three occurrences.

Step 5: Don't compile the implementation file separately

The problem here is that compilers have a difficult time when templated class implementation files are compiled separately. The reason is not important and requires an understanding of code compilation that is too advanced for us. So, instead of compiling the implementation file separately, we will be including the implementation file from the bottom of the header file. This makes it so that to the compiler it looks like the class declaration and implementation are in one file.

Here is step 5 in more detail:

a. Delete the #include of the header file from the top of the implementation file.
b. Add a #include of the implementation file at the bottom of the header file after the close of the namespace block.
c. Remove the implementation file from your project. This is important and easy to forget!
d. Eliminate any using directives in the implementation file. The implementation file is now part of the header file, and we don't want using directives in the header file.

Here is a video tutorial on step 5c using Xcode: **https://youtu.be/JyfyK-eVoR8**. Here is a video tutorial on step 5c using Visual C++: **https://youtu.be/yc8GzCWZ4SI**.

Step 6: Some compilers require default parameters to be in both the prototype and the function implentation.

I wouldn't worry about step 6 unless your compiler starts complaining.

Here is the updated code:

```
// file myvector.h

#include <cstdlib>

template <class T>
class MyVector {
    public:
        class NegativeSize{};

        typedef std::size_t size_type;
        typedef T value_type;
        MyVector(size_type inSize = 0, const value_type& inValue = value_type());
        size_type size() const;
        size_type capacity() const;
        value_type at(size_type i) const;
        void push_back(const value_type& insertMe);
    private:
        value_type* items;
        size_type mSize;
        size_type mCapacity;
};

#include "myvector.cpp"
```

```
// file myvector.cpp

#include <cassert>

template <class T>
MyVector<T>::MyVector(size_type inSize, const value_type& inValue) {
    mSize = inSize;
    mCapacity = inSize;
    items = new value_type[inSize];
    for (size_type i = 0; i < mSize; i++) {
        items[i] = inValue;
    }
}



template <class T>
typename MyVector<T>::size_type MyVector<T>::size() const {
    return mSize;
}



template <class T>
void MyVector<T>::push_back(const value_type& inValue){
    if (mSize < mCapacity) {
        items[mSize] = inValue;
        mSize++;
    } else {
        if (mCapacity == 0) {
            mCapacity = 1;
        } else {
            mCapacity *= 2;
        }
        value_type* temp = new value_type[mCapacity];
        for (size_type i = 0; i < mSize; i++) {
            temp[i] = items[i];
        }
        temp[mSize] = inValue;
        mSize++;
        delete [] items;
        items = temp;
    }
}


template <class T>
typename MyVector<T>::size_type MyVector<T>::capacity() const {
    return mCapacity;
}



template <class T>
typename MyVector<T>::value_type MyVector<T>::at(size_type i) const {
    std::assert(i >= 0 && i < size());
    return items[i];
}
```

```
// file myvector-client.cpp

#include <iostream>
#include "myvector.h"
using namespace std;
```

```
void print(MyVector<int>& printMe);

int main() {
    MyVector<int> v;
    MyVector<int>::size_type currentSize;

    for (int i = 0; i < 20; i++) {
        v.push_back(i);

        currentSize = v.size();
        cout << "size: " << currentSize << " capacity: " << v.capacity() << endl;
    }

    print(v);
}



void print(MyVector<int>& printMe) {
    for (int i = 0; i < printMe.size(); i++) {
        cout << printMe.at(i) << " ";
    }
    cout << endl;
}
```

Section 19.8: Defining iterator and const_iterator

Let's add iterators and const_iterators to our MyVector class so that MyVectors can be iterated over using them instead of using indexing. At this point you may want to go back and review sections 19.3 and 19.4 to refresh your memory about how the client program will use iterators and const_iterators. We will implement iterators and const_iterators by defining new classes INSIDE of the MyVector class. So we will have two classes nested inside the MyVector class.

Syntactically this starts to get very complex. Because of this, the standard approach is to simply include the entire iterator and const_iterator classes inside the MyVector class, including all of the definitions of the member functions, instead of trying to keep the member functions separate from the class declaration as we usually do. Also, the functions begin() and end() will be defined inside of the MyVector class declaration instead of being defined outside.

Another thing you will notice is that we have to switch back and forth between public and private sections of the class. We have to do things in this order because of the dependencies between the different identifiers being declared. My suggestion is to just follow the pattern you see here when you write your own iterator classes.

I'm going to provide the code here and let you study it and let me know if you have questions about it. I think that the iterator class will not be too difficult to understand. If there is anything that is confusing, it might be the variety of places where "const" and "const_iterator" are used. If you have any questions feel free to ask on the forums of course, but for the most part it is not critical that you understand every detail of every const in the const_iterator class.

```
// file myvector.h

#include <cassert>
#include <cstdlib>

template <class T>
class MyVector {
    public:
        typedef std::size_t size_type;
        typedef T value_type;
        MyVector(size_type inSize = 0, const value_type& inValue = value_type());
        size_type size() const;
        size_type capacity() const;
        value_type at(size_type i) const;
        void push_back(const value_type& insertMe);
    private:
        value_type* items;
        size_type mSize;
        size_type mCapacity;
    public:
        class iterator {
            public:
                iterator(value_type* initial = NULL) {
                    current = initial;
                }

                value_type& operator*() const {
                    return *current;
                }

                iterator& operator++() {
                    current++;
                    return *this;
                }

                iterator operator++(int) {
                    iterator original(current);
```

```
                    current++;
                    return original;
                }

                bool operator==(iterator other) const {
                    return current == other.current;
                }

                bool operator !=(iterator other) const {
                    return current != other.current;
                }
            private:
                value_type* current;
        };

        class const_iterator  {
            public:
                const_iterator(const value_type* initial = NULL) {
                    current = initial;
                }

                const value_type& operator*() const {
                    return *current;
                }

                const_iterator& operator++() {
                    current++;
                    return *this;
                }

                const_iterator operator++(int) {
                    const_iterator original(current);
                    current++;
                    return original;
                }

                bool operator==(const const_iterator other) const {
                    return current == other.current;
                }

                bool operator !=(const const_iterator other) const {
                    return current != other.current;
                }
            private:
                const value_type* current;
        };

        iterator begin() {
            return iterator(items);
        }

        iterator end() {
            return iterator(&items[mSize - 1] + 1);
        }

        const_iterator begin() const {
            return const_iterator(items);
        }

        const_iterator end() const {
            return const_iterator(&items[mSize - 1] + 1);
        }


};

#include "myvector.cpp"
```

```
// file myvector.cpp

template <class T>
MyVector<T>::MyVector(size_type inSize, const value_type& inValue) {
    mSize = inSize;
    mCapacity = inSize;
    items = new value_type[inSize];
    for (size_type i = 0; i < mSize; i++) {
        items[i] = inValue;
    }
}



template <class T>
typename MyVector<T>::size_type MyVector<T>::size() const {
    return mSize;
}



template <class T>
void MyVector<T>::push_back(const value_type& inValue){
    if (mSize < mCapacity) {
        items[mSize] = inValue;
        mSize++;
    } else {
```

```
        if (mCapacity == 0) {
            mCapacity = 1;
        } else {
            mCapacity *= 2;
        }
        value_type* temp = new value_type[mCapacity];
        for (size_type i = 0; i < mSize; i++) {
            temp[i] = items[i];
        }
        temp[mSize] = inValue;
        mSize++;
        delete [] items;
        items = temp;
    }
}


template <class T>
typename MyVector<T>::size_type MyVector<T>::capacity() const {
    return mCapacity;
}



template <class T>
typename MyVector<T>::value_type MyVector<T>::at(size_type i) const {
    assert(i >= 0 && i < size());
    return items[i];
}
```

```
#include <iostream>
#include "myvector.h"
using namespace std;

void print(const MyVector<int>& v);

int main() {
    MyVector<int> v;
    MyVector<int>::size_type currentSize;

    for (int i = 0; i < 20; i++) {
        v.push_back(i);

        currentSize = v.size();
        cout << "size: " << currentSize << " capacity: " << v.capacity() << endl;
    }

    print(v);
}



void print(const MyVector<int>& printMe) {
    for (MyVector<int>::const_iterator i = printMe.begin(); i != printMe.end(); i++) {
        cout << *i << " ";
    }
    cout << endl;
}
```

Section 19.9: Exception Handling

Exception handling is a way to handle errors or other special cases that is more sophisticated than what we have used up to this point. In our MyVector class, if the client tries to access an MyVector index that is out of bounds, we simply abort the program. With exception handling we can send information back to the client, and allow the client to decide how to best handle the error. This action by the class is called "throwing" an exception. The client then must "catch" the exception.

When an exception is thrown, what is actually thrown is an object. The simplest form of exception handling is when the object thrown is some simple type, such as int or string. (This is rarely used in practice, but it's a good starting point to learn about exception handling.) Here's how we would rewrite our "at()" function in our MyVector class to utilize exception handling in this manner:

```
template
typename MyVector<T>::value_type MyVector<T>::at(size_type i) const {
    if (i < 0 || i >= size()) {
        string str = "out-of-range-error";
        throw str;
    }
    return items[i];
}
```

The client, then, needs to catch the exception. To do this, the code that might possibly cause an exception to be thrown must be placed inside a "try" block, and then the code that is executed if an exception is thrown is placed inside a "catch" block. Here's what

our client's print() function would look like:

```
void print(const MyVector<int>& printMe) {
        try {
                for (int i = 0; i < printMe.size(); i++) {
                        cout << printMe.at(i) << " ";
                }
                cout << endl;
        } catch (string e) {
                cout << "ERROR: MyVector index out of range." << endl;
        }
}
```

This example isn't very interesting, because the catch block isn't invoked. To see the exception handling in action, we'll need the client to call the at() function with an index that is out of bounds. Here's how that would look:

```
void print(const MyVector<int>& printMe) {
        try {
                cout << printMe.at(30) << endl;
        } catch (string e) {
                cout << "ERROR: MyVector index out of range." << endl;
        }
}
```

Now if we run our program, we'll get this output (omitting a few lines of output to save space):

```
size: 1 capacity: 1
size: 2 capacity: 2
.
.
.
size: 20 capacity: 32
ERROR: MyVector index out of range.
```

Notice that what goes in the parentheses after the word "catch" looks a lot like a parameter, and, like a parameter, is a declaration. When an exception is thrown from within a try block, C++ looks for a catch block that has the matching type of exception declared in the parentheses. You can also have multiple catch blocks associated with a single try block, in which case C++ will go to the first catch block whose exception type matches the exception that was thrown.

Another way that the client might choose to handle this exception would be to actually use the value of the object that was thrown. So far, we have only been concerned about the type of the object being thrown, not its value. Here's an alternative print() function that uses the value of the object that was thrown:

```
void print(const MyVector<int>& printMe) {
        try {
                cout << printMe.at(30) << endl;
        } catch (string e) {
                cout << e << endl;
        }
}
```

This version of the print() function produces the following output:

```
size: 1 capacity: 1
size: 2 capacity: 2
.
.
.
size: 20 capacity: 32
out-of-range-error
```

Section 19.10 Programmer Defined Exception Classes

More typically when a class programmer wants to throw an exception, they will create their own exception class. This makes it easier for the client to manage exceptions if there are many different kinds of exceptions. The class programmer simply creates an empty class whose only purpose in life is to serve as a type of object that can be thrown. A separate class is defined for each different kind of exception. Below I have repeated the MyVector example using this new exception handling technique, with some parts of the code omitted to save space. Make sure you find the location of the class declaration (near the top), the location of the throw statement, and the location of the try/catch block.

```
// file myvector.h

#include <cassert>
#include <cstdlib>

template <class T>
class MyVector {
    public:
        class outOfRangeError {};
        typedef std::size_t size_type;
        typedef T value_type;
        MyVector(size_type inSize = 0, const value_type& inValue = value_type());
.
.
.

};

#include "myvector.cpp"
```

```
// file myvector.cpp

template <class T>
MyVector<T>::MyVector(size_type inSize, const value_type& inValue) {
    mSize = inSize;
    mCapacity = inSize;
    items = new value_type[inSize];
    for (size_type i = 0; i < mSize; i++) {
        items[i] = inValue;
    }
}

.
.
.
.
.
template <class T>
typename MyVector<T>::value_type MyVector<T>::at(size_type i) const {
    throw outOfRangeError();    // default constructor is called to create the object
    return items[i];
}
```

```
#include <iostream>
#include "myvector.h"
using namespace std;

void print(const MyVector<int>& v);

int main() {
    MyVector<int> v;
    MyVector<int>::size_type currentSize;

    for (int i = 0; i < 20; i++) {
        v.push_back(i);

        currentSize = v.size();
        cout << "size: " << currentSize << " capacity: " << v.capacity() << endl;
    }

    print(v);
}



void print(const MyVector<int>& printMe) {
        try {
                cout << printMe.at(30) << endl;
        } catch (MyVector<int>::outOfRangeError e) {
                cout << "ERROR: MyVector index out of range." << endl;
        }
}
```