

Lesson 7: Writing an Arrays



Resizable arrays, iterators

Push Back

- Let's introduce a new command that allows us to add to the end of an array
- If the capacity is not big enough we need to increase the capacity

```
template <typename T>
T Array<T>::push_back(T value) {
    if (length == capacity) {
        capacity *= 2;
        T* new_data = new T[capacity];
        for (int i=0; i<length; ++i) {
            new_data[i] = data[i];
        }
        delete [] data;
        data = new_data;
    }
    data[length] = value;
    ++length;
    return value;
}
```

Resizable Arrays

- We are finally in a position where we can make our array resizable
- We need to have both a length (which the user know about) and a capacity which is invisible to the user, but allows us to add elements while not having to create too many new arrays
- We need to change the constructor

```
template <typename T>
Array<T>::Array(unsigned n=0) {
    length = n;
    if (n==0) {
        n = 8;
    }
    data = new T[n];
    capacity = n;
}
```
- If we don't give the constructor an argument it will be set to 0

Refactoring

- It is possible to define functions inside the class definition
- This has the advantage of making the code much more compact
- It has the disadvantage of confusing the interface (a list of function that can be called) with the implementation
- It is unfortunate that when you use templates you tend to put the implementation details in the header file
- What you should do depend on whether you are after a quick solution or are writing code that many people might use

New Code

```
template <typename T>
class Array {
private:
    T *data;
    unsigned length;
    unsigned capacity;
public:
    Array(unsigned n=0) {
        length = n;
        if (n==0) {
            n = 8;
        }
        data = new T[n];
        capacity = n;
    }
    Array(const Array& other);
    ~Array() {delete[] data;}
    T& operator[](unsigned index) {return data[index];}
    const T& operator[](unsigned index) const {return data[index];}
    unsigned size() const {return length;}
    T push_back(T value);
};
```

vector

- C++ comes with a powerful library known as the **standard template library** (STL)
- This includes containers (resizable arrays, linked lists, double ended queues, sets, maps, etc.)
- The resizable array is called `vector<T>`
- We can just replace "array.h" with `<vector>` and `Array` with `vector` in our main function and everything will work the same
- Of course the STL vector is a bit more powerful and efficient than the `Array` class we wrote, although our class isn't bad

Iterators

- Iterators are designed to follow the same semantics as pointers
- Because we are just wrapping an array where we can use pointers to navigate the array we can use pointers as iterators
- All we need to do is add two new methods

```
T* begin() {return data;}
T* end() {return data+length;}
```
- We can use iterators to iterate over our array

Iterators in Other Containers

- Iterators exist for many containers
- Because of iterators it is possible to write algorithms that work for any container that supports iterators
- The STL has a bunch of algorithms that work for different containers
- You can also use a pretty for loop

```
for(T entry: container) {
    ...
}
```
- This makes code easier to read

Writing Iterators

- For most classes we don't iterator by advancing a pointer
- Thus most iterators are more complex and we have to write a class (or structure—`struct`) which keeps the information we need to iterate
- We then have to define methods to make iteration look like iterating though an array
- We will see an example in the code for linked lists