lecture08: Containers and Generic Programming

Sean Massung

CS 225 UIUC

20th June, 2013
Announcements

- lab_memory due tonight
- mp2.1 extra credit due Friday evening (6/21)
- lab_inheritance due Saturday evening (6/22)
- mp2 due Monday evening (6/24)
- mt1 Tuesday evening at 7pm
Our first data structure

- Let’s implement our first Abstract Data Type (ADT)
  - An ADT is a list of rules describing functionality of a data structure; it prescribes functions that must be implemented
  - Sounds like an abstract base class, huh?

- We’ll implement the Vector ADT with a resizing array
  - A sedan has a set functionality: four doors, and seats four to five passengers

- Think about this: an array is an implementation of a Vector like a Honda Civic is an implementation of a sedan
But what is a Vector?

Here’s the Vector ADT:

1. Insert a new element at the back of the Vector
2. Remove the element at the back of the Vector
3. Access any element by an index
4. Find the size of the Vector
5. Check if the Vector is empty

Not only should all these functions exist, but they should be relatively efficient. Additionally, our Vector should be able to hold any type!
template <class T>
class Vector {
    public:
        Vector();
        Vector(const Vector & other);
        Vector & operator=(const Vector & rhs);
        ~Vector();

        void push_back(const T & elem);
        T pop_back();
        T at(size_t idx) const;
        size_t size() const;
        bool empty() const;

    private:
        size_t _size;
        size_t _capacity;
        T* _data;
        void copy(const Vector & other);
        void clear();
};
Vector constructor and push_back

```cpp
template <class T>  
Vector<T>::Vector(): _size(0), _capacity(16)  
{  
    _data = new T[_capacity];  
}  

// can you write the big 3?  

template <class T>  
void Vector<T>::push_back(const T & elem)  
{  
    _data[_size] = elem;  
    ++_size;  
    if(_size == _capacity)  
    {  
        // resize and copy elements  
        // set new capacity (how?)  
    }  
}  
```
template <class T>
T Vector<T>::pop_back()
{
}

template <class T>
size_t Vector<T>::size() const
{
}

With all data structures in CS 225, we want to analyze the running times of the ADT functions. If $n$ is the number of elements in the Vector, fill in the running times with Big-$O$ notation:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Running time</th>
</tr>
</thead>
<tbody>
<tr>
<td>push_back</td>
<td></td>
</tr>
<tr>
<td>pop_back</td>
<td></td>
</tr>
<tr>
<td>at</td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
</tr>
<tr>
<td>empty</td>
<td></td>
</tr>
</tbody>
</table>
Motivation

This function doesn’t know (or care) what kind of container it’s printing, or even how it’s being printed. What a generic function!

```cpp
template <class Container, class Printer>
void print_container(const Container & things, Printer & printer)
{
    size_t idx = 0;
    while(idx < things.size())
    {
        printer(things.at(idx));
        ++idx;
    }
}
```

All it needs is that the container has a size and at function, and that the Printer class can be used like a function.
Let's see how we call `print_container`

```cpp
template <class Container, class Printer>
void print_container(const Container & things, Printer & printer)
{
    size_t idx = 0;
    while(idx < things.size())
    {
        printer(things.at(idx));  // this part's still magic...
        ++idx;
    }
}

void main()
{
    Vector<int> vec;
    // fill it up...

    SimplePrint printer;

    print_container(vec, printer);
}
```
A class that looks like a function

In order to make an object act like a function, we overload `operator()`. That’s “operator parentheses”.

```cpp
template <class T>
class SimplePrint {
    public:
        void operator()(const T & elem) const;
};

void SimplePrint<T>::operator()(const T & elem) const {
    cout << elem << " ";
}
```
Here’s a printer that only prints certain elements by saving its state (this is why functors are so powerful!).

```cpp
template <class T>
class SillyPrint {
public:
    SillyPrint(): _printed(0) { }
    void operator()(const T & elem);
private:
    size_t _printed;
};

void SillyPrint<T>::operator()(const T & elem) {
    if(++_printed % 2)
        cout << elem << " ";
}
```
Calling print\_container with SillyPrint

template <class Container, class Printer>
void print\_container(const Container & things, Printer & printer)
{
    size_t idx = 0;
    while(idx < things.size())
    {
        printer(things.at(idx));  // this part is not magic!
        ++idx;
    }
}

void main()
{
    List<int> list;
    // fill it up...

    SillyPrint printer;

    print\_container(list, silly);
}
We accomplished a lot today!

- Write a generic container data structure
- Overload operator parentheses (you should be very familiar with another class that does this...)
- Write functions (or classes) with multiple template arguments
- Write very generic functions to avoid code duplication
- Write a function that applies some operation on each element in a container (this is called “map”)

Announcements
The Vector ADT
Functors and Map
The vector class already exists in the STL

std::vector from #include <vector>

Map (and a bunch of other algorithms) also already exist in the STL

std::for_each in #include <algorithm>

Don’t reimplement them, make use of them! They are slightly more complicated than ours, but I bet you can figure them out.