Teaching with the STL

Joseph Bergin
Michael Berman
Part 1

Introduction to STL Concepts
Templates are not Classes

- These are not cookies
- You can’t eat them
- They can be used to make cookies
Templates are not Classes

- These are cookies
- They are made with a cookie cutter
- You can eat them
Templates are not Cookies

- Templates are used to create classes
- You can’t compile them
- You can instantiate them
  » This gives you a class
- The instantiations are compiled
- The instantiations are strongly typed like other classes
Templates are not Classes

```cpp
template <class E>
class stack
{
    ...
    void push(E e){...}
}

stack <int> S;  // A template class
S.push(55);      // A template class
```
Templates are not Functions

```
template <class E> E& min(E& a, E& b)
{
    if(a < b) return a;
    return b;
}

abox = min(box1, box2); <- a template function
```

<- a function template
The Standard Template Library

- **Containers**
  - array, vector, deque, list, set, map, multiset, multimap

- **Algorithms**
  - sort, search, and nearly everything else

- **Iterators**
  - generalize pointers and pointer arithmetic

- **Adaptors**
  - change the behavior of other components

- **Allocators**
  - memory management
The Standard Template Library

- **Containers**
  - array, vector, deque, list, set, map, multiset, multimap

- **Algorithms**
  - sort, search, and nearly everything else

- **Iterators**
  - generalize pointers and pointer arithmetic

- **Adaptors**
  - change the behavior of other components

- **Allocators**
  - memory management
The Major Dimensions

Independent Development of:

● Containers
  » contain values

● Algorithms
  » operate on containers

● Iterators
  » interface between containers and algorithms
The Major Dimensions

Independent Development of:

- **Containers**
  - contain values

- **Algorithms**
  - operate on containers

- **Iterators**
  - interface between containers and algorithms
STL Example

```cpp
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(5);
v.push_back(6);

vector<int>::iterator i;

for(i = v.begin(); i != v.end(); ++i) cout << *i << endl;

sort(v.begin(); v.end());

for(i = v.begin(); i != v.end(); ++i) cout << *i << endl;
```
Iterator Flavors

- Forward Iterators (operator++)
  - Input Iterators
  - Output Iterators
- Bidirectional Iterators (operator --)
- Random Access Iterators (operator +=)
Iterator Flavors

- **Forward Iterators** (operator `++`)
  - Input Iterators
  - Output Iterators

- **Bidirectional Iterators** (operator `--`)

- **Random Access Iterators** (operator `+=`)

  All Iterators have operator `*`
  All Containers produce iterators `begin()` and `end()`
  `begin` references first. `end` is “after” last
Slouching Toward Iterators

template < class T >
void selectionSort(T elements[ ], int length)
{
    for(int i = 0; i < length - 1; ++i)
    {
        int s = i;
        T small = elements[s];
        for(unsigned j = i + 1; j < length; ++j)
            if(elements[j] < small)
                { s = j;
                  small = elements[s];
                }
        elements[s] = elements[i];
        elements[i] = small;
    }
}

Pt. 1: Dependent on Arrays
Pointer Duality Law

```
int * A = new int[20];

A[i] is equivalent to *(A + i)
```

A

A + 6  (6 ints past A)
Slouching Towards Iterators

int elements[20] = ... 
selectionSort(elements, 20)

int * start = elements;
int * end = elements + 20; // or &elements[20]

selectionSort(start, end);

Pt. 2: The Goal
The Replacements

```cpp
template < class T >
void selectionSort(T elements[], int length)
{
    for(int i = 0; i < length - 1; ++i)
    {
        int s = i;
        T small = elements[s];
        for(unsigned j = i + 1; j < length; ++j)
        {
            if(elements[j] < small)
            {
                s = j;
                small = elements[s];
            }
        }
        elements[s] = elements[i];
        elements[i] = small;
    }
}
```

start = elements
end = elements + length
loc = & elements[s]
where = & elements[i]
inner = & elements[j]
Slouching Towards Iterators

template < class T >
void selectionSort(T* start, T* end)
{
    for(T* where = start ; where < end - 1 ; ++where)
    {
        T* loc = where;
        T small = *loc;
        for(T* inner = where + 1; inner < end; ++inner)
            if(*inner < *loc)
                { loc = inner;
                  small = *loc;
                }
        *loc = *where;
        *where = small;
    }
}

Pt 3: The Result (almost)
The Advantages

- This version will sort more than arrays.
  » All we need is a structure referenced by a datatype like a pointer that implements
    - operator *
    - operator++
    - operator+
    - operator-
    - operator=
    - operator<

    With care we could reduce this list

    Such datatypes are called iterators
The Lesson

- Implement containers separate from algorithms
- Use pointer like structures as an interfacing mechanism
The Lesson

- Implement containers separate from algorithms
- Use pointer like structures as an interfacing mechanism
Advantages

● Generality
● A framework for thinking about containers and algorithms
● Smaller written code
● Smaller compiled code
Advantages

- Generality
- A framework for thinking about containers and algorithms
- Smaller written code
- Smaller compiled code

But...
Disadvantages

- Students must become thoroughly familiar with all aspects of pointers including
  - The pointer duality law
  - Pointer arithmetic
  - Pointer “gotchas”
Part 2

STL Containers
STL Containers

- Ordinary Arrays
- Vectors -- expandable array
- Deques -- expandable at both ends
- Lists -- doubly linked circular with header
- Sets and Multisets -- red-black tree
- Maps and Multimaps -- dictionary like

Note: Implementation is not specified but efficiency is specified.
All Containers Provide

- A Storage Service
  - insert and erase...
- An Associated Iterator type
  - The type of iterator determines what can be done with the container.
- begin() and end() iterators - - - [b, e)
- A collection of types: vector::value_type...
- constructors, assignment, cast, equality...
All Iterators Provide

- `operator*`
  - may be readonly or read/write
- `copy constructor`
- `operator++` and `operator++(int)`
- `operator==` and `operator!=`
- Most provide `operator=`
Specialized Iterators

- **Forward**
  - provide operator=

- **Bidirectional (extend forward)**
  - provide operator-- and operator--(int)

- **Random Access (extend bidirectional)**
  - provide operator<..., operator+=..., operator-
Algorithms

- Defined in terms of a specific iterator type
  - e.g. sort requires random access iterators
- Work with all containers that provide that iterator type -- including user written.
- Combine good generality with good efficiency
- Do not appear within container classes
  - This is important to generality & efficiency
Function Objects 1

- Predicates
  - A function of one argument returning bool
- Comparisons
  - A function of two arguments returning bool
- Unary Operator, Binary Operator
  - A function of one or two arguments returning a value
Function Objects 2

- Can be functions or template functions
- Can be objects implementing an appropriate operator()
- Many are built in
  - less..., plus..., and..., ...
- Function adaptors too
  - not1, not2, bind1st, bind2nd,...
Function Object Example

class stringLess
{
    bool operator()(char* s1, char* s2)
    {
        return strcmp(s1, s2) < 0;
    }
    . . .
} // Defines a function object.

vector< char* > stringVec;
. . .
sort (stringVec.begin(), stringVec.end(), stringLess());
// Note the constructor call in the last argument ^^^^^
vector

- Expandable array -- operator[]
- push_back, pop_back
- Average O(1) insert at end.
- O(n) insert in middle
- Random Access Iterators
- Fastest (average) container for most purposes.
deque

- Expandable “array” at both ends
- `push_front, pop_front`
- Average O(1) insert at both ends
- Linear insert in middle
- Random Access Iterators
- Good choice for queues & such.
list

- Doubly linked list
- $O(1)$ inserts everywhere, but slower on average than vector and deque
- Bidirectional iterators
- Some specialized algorithms (sort).
set and multiset

- Sorted set (multiset) of values
- $O(\lg n)$ inserts and deletions
  - Balanced binary search tree
- Sorted with respect to operator $<$ or any user defined comparison operator
- Bidirectional iterators
- Good choice if elements must stay in order.
map and multimap

- Ordered set (multiset) of key-value pairs
- Kept in key order
- $O(\log n)$ inserts and deletions
- Bidirectional iterators
- Good choice for dictionaries, property lists, & finite functions as long as keys have comparison operation
Extending the STL

- Not standardized but available
  - hash_set
  - hash_map
  - hash_multiset
  - hash_multimap

- Like set... but have a (self reorganizing) hashed implementation

- Constant average time for insert/erase
STL in Java

- ObjectSpace has developed an equivalent library for Java
- (JGL) Java Generic Library
- Public domain, available on internet.
- Depends on run-time typing instead of compile time typing, but is otherwise equivalent.
Resources

- http://csis.pace.edu/~bergin
- http://www.objectspace.com
- ftp.cs.rpi.edu/pub/stl
**Books**

- Data Structures Programming with the STL, Bergin, Springer-Verlag (to appear)
- The STL <primer>, Glass and Schuchert, Prentice-Hall, 1996