Data Structures
in C++

Chapter 8

Tim Budd

Oregon State University
Corvallis, Oregon
USA
Outline – Chapter 8

Vectors

• Idea of vector

• Templates
  – Class templates
  – Function templates

• Problems solved using vectors
  – Sieve of Erastosthenes
  – Selection Sort
  – Merge Sort
  – Silly Sentence Generation

• Summary of vector operations

• The implementation of the vector data type
The Idea of a Vector

Conceptually, a vector is simply a indexed collection of similarly typed values.

<table>
<thead>
<tr>
<th>element 0</th>
<th>element 1</th>
<th>element 2</th>
<th>element 3</th>
<th>element 4</th>
<th>element 5</th>
</tr>
</thead>
</table>

A matrix is a two dimensional array, again of similar type values.

<table>
<thead>
<tr>
<th>element 0,0</th>
<th>element 0,1</th>
<th>element 0,2</th>
<th>element 0,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>element 1,0</td>
<td>element 1,1</td>
<td>element 1,2</td>
<td>element 1,3</td>
</tr>
<tr>
<td>element 2,0</td>
<td>element 2,1</td>
<td>element 2,2</td>
<td>element 2,3</td>
</tr>
</tbody>
</table>
Why Build into an ADT?

The C++ language has vector and matrix values, why build something else on top of these?

- Perform safety checks (index bounds checks)
- Make values more “self describing” (and thus make programs more reliable)
- Permit the implementation of operations at a higher level of abstraction (ability to dynamically make a vector grow, for example)
Templates

One major difference between the vector ADT and the rational number or the string ADT is that the vector ADT does not, by itself, describe the type of object it holds. Can have a vector of integers, a vector of reals, a vector of strings, or any other type.

The idea of a template allows us to parameterize the type of object held by a class. You can think of a template as similar to a function parameter, only it is a data structure parameter.
The Vector Template

template<class T> class vector {

public:
    typedef T * iterator;

    // constructors
    vector (unsigned int numberElements);
    vector (unsigned int numberElements, T initialValue);
    vector (const vector & source);
    ~vector ();

    // member functions
    T back ();
    iterator begin ();
    ...

    // operators
    T & operator [ ] (unsigned int);

private: // data areas
    unsigned int mySize;
    unsigned int myCapacity
    T * data;
};
Declaring Template Types

To declare a value with a template type, a type is provided in angle brackets following the template class name.

\[
\begin{align*}
\text{vector<int>} & \quad \text{a}(10); \\
\text{vector<double>} & \quad \text{b}(30); \\
\text{vector<string>} & \quad \text{c}(15);
\end{align*}
\]
How a Template Works

A template works as if int replaced every occurrence of T and the class were renamed.

class vector<int {  
    public:  
        typedef T * iterator;

        // constructors
        vector<int (unsigned int numberElements);
        vector<int (unsigned int numberElements, T initialValue);
        vector<int (const vector & source);
        ~vector<int ();

        // member functions
        int back ();
        iterator begin ();
        ...

        // operators
        int & operator [] (unsigned int);

    private:  // data areas
        unsigned int mySize;
        unsigned int myCapacity
        int * data;
};
Naming

When the `vector<T>` template class is instantiated by `int`, its name becomes `vector<int>`.

Its constructor is named `vector<int>::vector`.

Its member functions have names like `vector<int>::size`.

To provide a *general* implementation of a member function, we use the syntax

```cpp
template <class T>
unsigned int vector<T>::size()  
{  
   return mySize;  
}
```

V ectors  Chapter 8
Function Templates

Functions can also be parameterized using templates, as in the following:

```cpp
template <class T> T max(T a, T b)
    // return the maximum of a and b
{
    if (a < b)
        return b;
    return a;
}
```

```cpp
template <class T> void swap (T & a, T & b)
    // swap the values held by a and b
{
    T temp = a;
    a = b;
    b = temp;
}
```
Example Program – Sieve of Erastosthenes

```cpp
void sieve(vector<int> &values)
    // leave vector holding only prime numbers
{
    unsigned int max = values.size();

    // first initialize all cells
    for (int i = 0; i < max; i++)
        values[i] = i;

    // now search for non-zero cells
    for (i = 2; i*i < max; i++) {
        if (values[i] != 0) {
            // inv: i has no factors
            for (int j = i + i; j < max; j += i)
                values[j] = 0;
            // inv: all multiples of i have been cleared
        }
        // all nonzero values smaller than i are prime
    }
    // inv: all nonzero values are prime
}
```
Another Example —
Selection Sort

```cpp
template<class T>
void selectionSort(vector<T> & data)
    // sort, in place, the vector argument
    // into ascending order
{
    unsigned int top;
    for (top = data.size() - 1; top > 0; top = top - 1) {
        // find the position of the largest element
        unsigned int largeposition = 0;
        for (int j = 1; j <= top; j++) {
            // inv: data[largeposition] is largest element
            // in 0..j-1
            if (data[largeposition] < data[j])
                largeposition = j;
            // inv: data[largeposition] is
            // largest element in 0 .. j
        }
        if (top != largeposition)
            swap(data, top, largeposition);
            // inv: data[top .. n] is ordered
    }
}
```

Vectors

Chapter 8
Merge Sort

Unfortunately, Selection Sort is still $O(n^2)$ worst case.

Better algorithm can be built using the idea that two vectors can be merged in linear time.
In-Place Merge

An in place merge can be performed for adjacent vector ranges:

Provided by generic function

\[
\text{inplace}_\text{merge} \quad (\text{iterator start, iterator center, iterator end});
\]
How to build a Sorting Algorithm

First, break things apart, until you reach a single element

```
7 3 3 2 9 3 6 7 1 2
```

```
7 3 3 2 9
3 6 7 1 2
```

```
7 3
3 2 9
3 6
7 1 2
```

```
7 3
3 2
9 3 6
7 1 2
```

```
3 2
7 1
```
Then Put Together

Then merge adjacent ranges as you come back out of the sequence of recursive calls.

```
3 2
7 3 2 3 9 3 6 1 7 2
3 7 2 3 9 3 6 1 2 7
2 3 3 7 9 1 2 3 6 7
1 2 2 3 3 6 7 7 9
```
The Merge Sort Algorithm

template <class ltr>
void m_sort(ltr start, unsigned low, unsigned high)
{
    if (low + 1 < high) {
        unsigned int center = (high + low) / 2;
        m_sort (start, low, center);
        m_sort (start, center, high);
        inplace_merge
            (start + low, start + center,
             start + high);
    }
}

template <class T>
void mergeSort(vector<T> & s)
{
    m_sort(s.begin(), 0, s.size());
}
What is the Asymptotic Complexity?

- Complexity is work at each level times number of levels of call.
- Work at each level is linear
- Number of recursive calls in \( \log n \)
- Total amount of work is \( O(n \log n) \! \)
- Much better than bubble sort or insertion sort
Picture of Complexity

\[
\begin{array}{c c c c}
3 & 2 & 7 & 1 \\
7 & 3 & 2 & 3 & 9 & 3 & 6 & 1 & 7 & 2 \\
3 & 7 & 2 & 3 & 9 & 3 & 6 & 1 & 2 & 7 \\
2 & 3 & 3 & 7 & 9 & 1 & 2 & 3 & 6 & 7 \\
1 & 2 & 2 & 3 & 3 & 6 & 7 & 7 & 9 \\
\end{array}
\]
Example Problem – Silly Sentence Generation

Generate a sequence of silly sentences.

Each sentence has form subject - verb - object.

First, allocate three vectors, with initially empty size.

```cpp
vector<string> subject, verb, object;
```
Dynamically Extending the Size of Vectors

Next, push values on to the end of the vectors.

Vectors are automatically resized as necessary.

    // add subjects
    subject.push_back("alice and fred");
    subject.push_back("cats");
    subject.push_back("people");
    subject.push_back("teachers");

    // add verbs
    verb.push_back("love");
    verb.push_back("hate");
    verb.push_back("eat");
    verb.push_back("hassle");

    // add objects
    object.push_back("dogs");
    object.push_back("cats");
    object.push_back("people");
    object.push_back("donuts");
Generating Sentences

Use \texttt{size} to compute size, \texttt{randomInteger} to get a random subscript.

\begin{verbatim}
for (int i = 0; i < 10; i++)
    cout << subject[randomInteger(subject.size())] << " "
    << verb[randomInteger(verb.size())] << " "
    << object[randomInteger(object.size())] << "\n"
\end{verbatim}

Example Output

\begin{itemize}
    \item alice and fred hate dogs
    \item teachers hassle cats
    \item alice and fred love cats
    \item people hassle donuts
    \item people hate dogs
\end{itemize}
Matrices

Can even build vectors whose elements are themselves vectors – this is a reasonable approximation to a matrix.

```cpp
vector<vector<int>> mat(5);
```

Initially each row has zero elements. Must be resized to correct limit.

```cpp
for (int i = 0; i < 5; i++)
    mat[i].resize(6);
```
Vector Operations
## Constructors

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector&lt;T&gt; v;</code></td>
<td>default constructor</td>
</tr>
<tr>
<td><code>vector&lt;T&gt; v (int);</code></td>
<td>initialized with explicit size</td>
</tr>
<tr>
<td><code>vector&lt;T&gt; v (int, T);</code></td>
<td>size and initial value</td>
</tr>
<tr>
<td><code>vector&lt;T&gt; v (aVector);</code></td>
<td>copy constructor</td>
</tr>
</tbody>
</table>

## Element Access

<table>
<thead>
<tr>
<th>Access</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v[i]</code></td>
<td>subscript access</td>
</tr>
<tr>
<td><code>v.front ()</code></td>
<td>first value in collection</td>
</tr>
<tr>
<td><code>v.back ()</code></td>
<td>last value in collection</td>
</tr>
</tbody>
</table>

## Insertion

<table>
<thead>
<tr>
<th>Insertion</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v.push_back (T)</code></td>
<td>push element on to back of vector</td>
</tr>
<tr>
<td><code>v.insert(iterator, T)</code></td>
<td>insert new element after iterator</td>
</tr>
<tr>
<td><code>v.swap(vector&lt;T&gt;)</code></td>
<td>swap values with another vector</td>
</tr>
</tbody>
</table>

## Removal

<table>
<thead>
<tr>
<th>Removal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v.pop_back ()</code></td>
<td>pop element from back of vector</td>
</tr>
<tr>
<td><code>v.erase(iterator)</code></td>
<td>remove single element</td>
</tr>
<tr>
<td><code>v.erase(iterator, iterator)</code></td>
<td>remove range of values</td>
</tr>
</tbody>
</table>

## Size

<table>
<thead>
<tr>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>v.capacity ()</code></td>
<td>number of elements buffer can hold</td>
</tr>
<tr>
<td><code>v.size ()</code></td>
<td>number of elements currently held</td>
</tr>
<tr>
<td><code>v.resize (unsigned, T)</code></td>
<td>change to size, padding with value</td>
</tr>
<tr>
<td><code>v.reserve (unsigned)</code></td>
<td>set physical buffer size</td>
</tr>
<tr>
<td><code>v.empty ()</code></td>
<td>true if vector is empty</td>
</tr>
</tbody>
</table>

## Iterators

<table>
<thead>
<tr>
<th>Iterators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vector&lt;T&gt;::iterator itr</code></td>
<td>declare a new iterator</td>
</tr>
<tr>
<td><code>v.begin ()</code></td>
<td>starting iterator</td>
</tr>
<tr>
<td><code>v.end ()</code></td>
<td>ending iterator</td>
</tr>
</tbody>
</table>

---

**Vectors**  Chapter 8
Sizes of Vector

Vectors will maintain an internal buffer. Like the string, the physical size of the buffer need not be the same as the logical size.

The two sizes can be accessed or set using member functions.

As with the string, a new buffer is allocated when the physical size is exceeded.
## Useful Generic Algorithms

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fill</td>
<td>Fill vector with a given initial value</td>
</tr>
<tr>
<td>copy</td>
<td>Copy one sequence into another</td>
</tr>
<tr>
<td>max_element</td>
<td>Find largest value in collection</td>
</tr>
<tr>
<td>min_element</td>
<td>Find smallest value in collection</td>
</tr>
<tr>
<td>reverse</td>
<td>Reverse elements in the collection</td>
</tr>
<tr>
<td>count</td>
<td>Count elements that match target value, incrementing counter</td>
</tr>
<tr>
<td>count_if</td>
<td>Count elements that satisfy function, incrementing counter</td>
</tr>
<tr>
<td>transform</td>
<td>Transform elements using unary function from source, placing into destination</td>
</tr>
<tr>
<td>find</td>
<td>Find value in collection, returning iterator for location</td>
</tr>
<tr>
<td>find_if</td>
<td>Find value for which function is true, returning iterator for location</td>
</tr>
<tr>
<td>replace</td>
<td>Replace target element with replacement value</td>
</tr>
<tr>
<td>replace_if</td>
<td>Replace elements for which function is true with replacement value</td>
</tr>
<tr>
<td>sort</td>
<td>Places elements into ascending order</td>
</tr>
<tr>
<td>for_each</td>
<td>Execute function on each element of vector</td>
</tr>
<tr>
<td>iter_swap</td>
<td>Swap the values specified by two iterators</td>
</tr>
</tbody>
</table>

Vectors Chapter 8
Example, counting elements

```cpp
vector<int>::iterator start = aVec.begin();
vector<int>::iterator stop = aVec.end();

if (find(start, stop, 17) != stop)
    ...  // element has been found

int counter = 0;
count (start, stop, 17, counter);
if (counter != 0)
    ...  // element is in collection
```
Vector Implementation

- Like string, the vector holds a buffer that can dynamically grow if needed
- Maintains two sizes, physical and logical size
- Most operations have simple implementations, can be performed inline
- (Note that this implementation is simpler than the actual commercial implementations, which are proprietary)
Inline Definitions

template <class T> class vector {
public:
  typedef T * iterator;

  // constructors
  vector () { buffer = 0; resize(0); }
  vector (unsigned int size) { buffer = 0; resize(size); }
  vector (unsigned int size, T initial);
  vector (vector & v);
  ~vector () { delete buffer; }

  // member functions
  T back () { assert(!empty()); return buffer[mySize - 1];}
  iterator begin () { return buffer; }
  int capacity () { return myCapacity; }
  bool empty () { return mySize == 0; }
  iterator end () { return begin() + mySize; }
  T front () { assert(!empty()); return buffer[0]; }
  void pop_back () { assert(!empty()); mySize--; }
  void push_back (T value);
  void reserve (unsigned int newCapacity);
  void resize (unsigned int newSize)
    { reserve(newSize); mySize = newSize; }
  int size () { return mySize; }

  // operators
  T & operator [] (unsigned int index)
    { assert(index < mySize); return buffer[index]; }

private:
  unsigned int mySize;
  unsigned int myCapacity;
  T * buffer;
};
Constructors

The constructors use generic algorithms to fill initial values:

```cpp
template <class T>
vector<T>::vector (unsigned int size, T initial)
    // create vector with given size,
    // initialize each element with value
{
    buffer = 0;
    resize(size);
    // use fill algorithm to initialize each
    fill (begin(), end(), initial);
}

template <class T>
vector<T>::vector (vector & v)
    // create vector with given size,
    // initialize elements by copying
{
    buffer = 0;
    resize(size);
    // use copy algorithm to initialize
    copy (v.begin(), v.end(), begin());
}
```
Reserve – the workhorse method

template <class T>
void vector<T>::reserve (unsigned int newCapacity)
   // reserve capacity at least as large as argument
{
   if (buffer == 0) {
      mySize = 0;
      myCapacity = 0;
   }
   // don’t do anything if already large enough
   if (newCapacity <= myCapacity)
      return;
   // allocate new buffer, make sure successful
   T * newBuffer = new T [newCapacity];
   assert (newBuffer);
   // copy values into buffer
   copy (buffer, buffer + mySize, newBuffer);
   // reset data field
   myCapacity = newCapacity;
   // change buffer pointer
   delete buffer;
   buffer = newBuffer;
}
Implementing Generic Algorithms

Templates are also the key to the implementation of generic algorithms.

```cpp
template (class ItrType, class T)
void fill (ItrType start, ItrType stop, T value)
{
    while (start != stop)
        *start++ = value;
}

template (class SourceItrType, class DestItrType)
void copy (SourceItrType start,
            SourceItrType stop, DestItrType dest)
{
    while (start != stop)
        *dest++ = *start++;
}
```