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.

element 0	element 1	element 2	element 3	element 4	element 5

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

element 0,0	element 0,1	element 0,2	element 0,3
element 1,0	element 1,1	element 1,2	element 1,3
element 2,0	element 2,1	element 2,2	element 2,3

Why Build into an ADT?

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

- Perform safety checkes (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.

```
vector<int>      a(10);
vector<double>   b(30);
vector<string>   c(15);
```

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

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

Function Templates

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

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


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

```
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
}
```

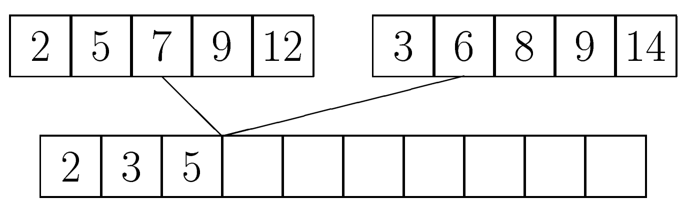
Another Example – Selection Sort

```
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
        }
}
```

Merge Sort

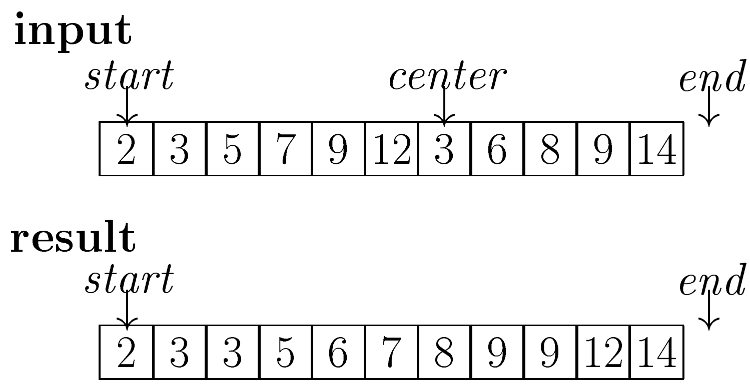
Unfortunately, Selection Sort is still $O(n^2)$ worst case.

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:

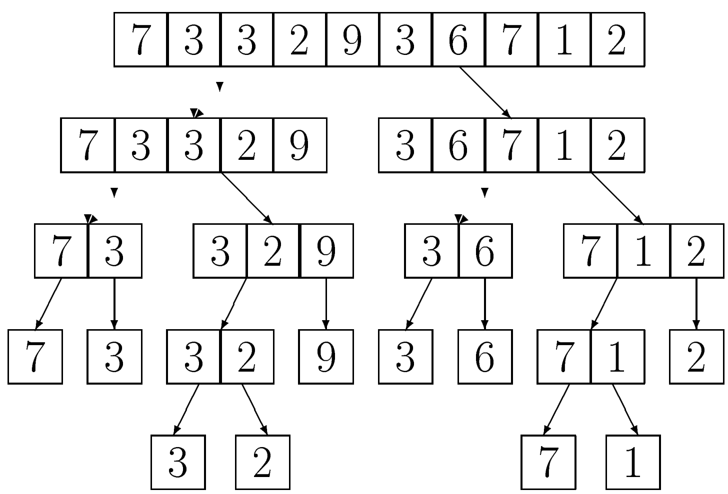


Provided by generic function

```
inplace_merge
    (iterator start, iterator center, iterator end);
```

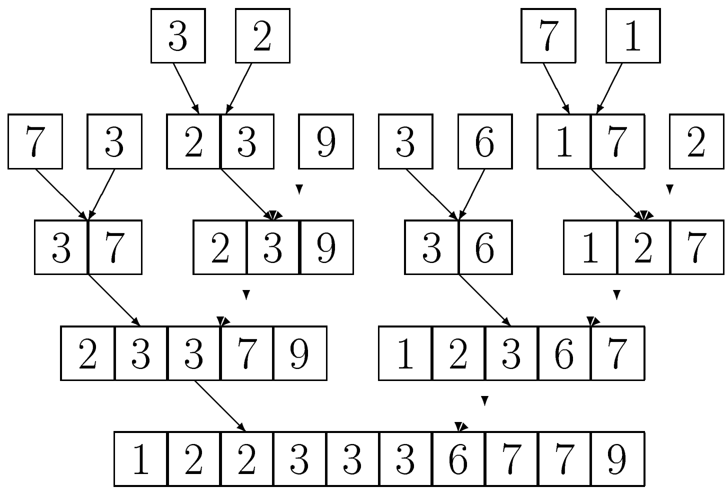
How to build a Sorting Algorithm

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



Then Put Together

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



The Merge Sort Algorithm

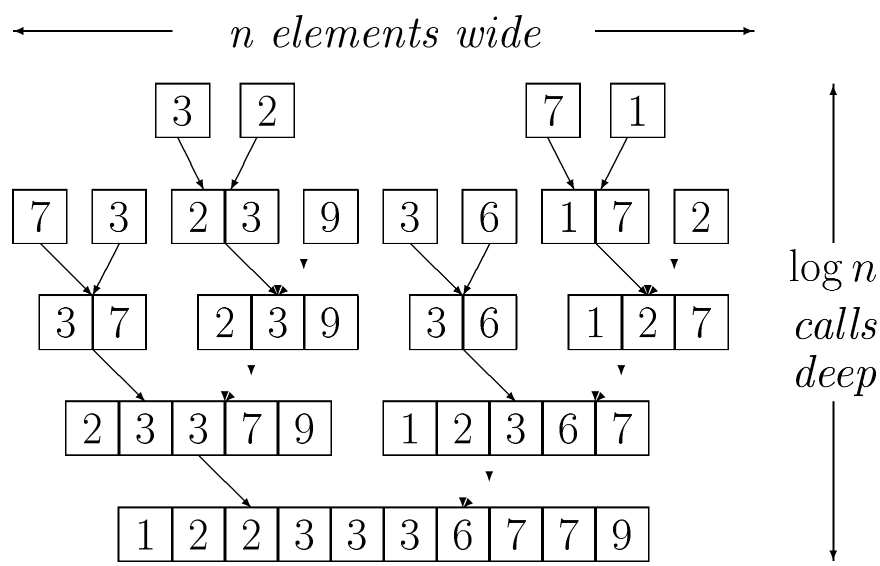
```
template <class Itr>
void m_sort(Itr start, unsigned low, unsigned high)
{
    if (low + 1 < high) {
        unsigned int center = (high + low) / 2;
        m_sort (start, low, center);
        m_sort (start, center, high);
        inplace_merge
            (start + low, start + center,
        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

template <class T>
void mergeSort(vector<T> & s)
{
    m_sort(s.begin(), 0, s.size());
}
```

What is the Asympototic 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



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.

```
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 `size` to compute size, `randomInteger` to get a random subscript.

```
for (int i = 0; i < 10; i++)
    cout << subject[randomInteger(subject.size())]
        << " "
        << verb[randomInteger(verb.size())]
        << " "
        << object[randomInteger(object.size())]
        << "\n"
```

Example Output

alice and fred hate dogs

teachers hassle cats

alice and fred love cats

people hassle donuts

people hate dogs

Matrices

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

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

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

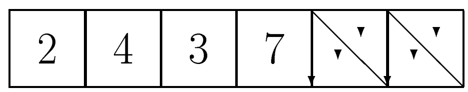
```
for (int i = 0; i < 5; i++)
    mat[i].resize(6);
```

Vector Operations

Constructors	
vector<T> v;	default constructor
vector<T> v (int);	initialized with explicit size
vector<T> v (int, T);	size and initial value
vector<T> v (aVector);	copy constructor
Element Access	
v[i]	subscript access
v.front ()	first value in collection
v.back ()	last value in collection
Insertion	
v.push_back (T)	push element on to back of vector
v.insert(iterator, T)	insert new element after iterator
v.swap(vector<T>)	swap values with another vector
Removal	
v.pop_back ()	pop element from back of vector
v.erase(iterator)	remove single element
v.erase(iterator, iterator)	remove range of values
Size	
v.capacity ()	number of elements buffer can hold
v.size ()	number of elements currently held
v.resize (unsigned, T)	change to size, padding with value
v.reserve (unsigned)	set physical buffer size
v.empty ()	true if vector is empty
Iterators	
vector<T>::iterator itr	declare a new iterator
v.begin ()	starting iterator
v.end ()	ending iterator

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

`fill (iterator start, iterator stop, value)` fill vector with a given initial value
`copy (iterator start, iterator stop, iterator destination)` copy one sequence into another
`max_element(iterator start, iterator stop)` find largest value in collection
`min_element(iterator start, iterator stop)` find smallest value in collection
`reverse (iterator start, iterator stop)` reverse elements in the collection
`count (iterator start, iterator stop, target value, counter)` count elements that match target value, incrementing counter
`count_if (iterator start, iterator stop, unary fun, counter)` count elements that satisfy function, incrementing counter
`transform (iterator start, iterator stop, iterator destination, unary)` transform elements using unary function from source, placing into destination
`find (iterator start, iterator stop, value)` find value in collection, returning iterator for location
`find_if (iterator start, iterator stop, unary function)` find value for which function is true, returning iterator for location
`replace (iterator start, iterator stop, target value, replacement value)` replace target element with replacement value
`replace_if (iterator start, iterator stop, unary fun, replacement value)` replace lements for which fun is true with replacement value
`sort (iterator start, iterator stop)` places elements into ascending order
`for_each (iterator start, iterator stop, function)` execute function on each element of vector
`iter_swap (iterator, iterator)` swap the values specified by two iterators

Example, counting elements

```
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 properitary)

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:

```
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;
        myCapaicty = 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.

```
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++;
}
```