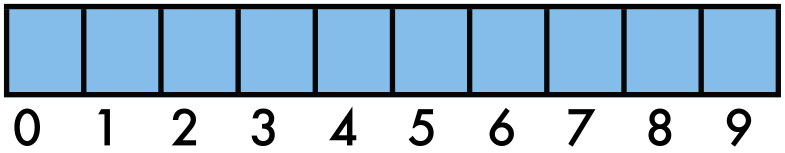
STL Vectors

taken from Chapter 8 (Budd)

What are vectors?



- Vector is an indexed collection of homogeneously typed values
  - Analogous to arrays
- Vectors provide bounds checking
- Vectors can grow and shrink as needed

Container Classes

- The type of object contained is not described
- Template classes are used to implement container classes
- The type is supplied by the user of the container class
    - Vector of int
    - Vector of string
    - Vector of critter

vector Class Operation Summary

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

Using Vectors

```
#include <iostream>
#include <vector>

// used only for convenience
using namespace std;

int main()
{
    // initialize a vector
    vector<int> numbers;
    // insert more numbers into the vector
    numbers.push_back(3);
    numbers.push_back(6);
    numbers.push_back(7);
    numbers.push_back(5);
    // the vector currently holds {3, 6, 7, 5}

    cout << numbers[2] << endl;     // outputs 7

    cout << numbers.size() << endl; // outputs 4

    if (numbers.empty())
           cout << "vector is empty" << endl;
    else
           cout << "vector is not empty" << endl;

    cout << numbers.capacity() << endl; // outputs 4
    numbers.resize(10);
    cout << numbers.capacity() << endl; // outputs 10
```

- Type this into Visual C++

Useful Generic Algorithms

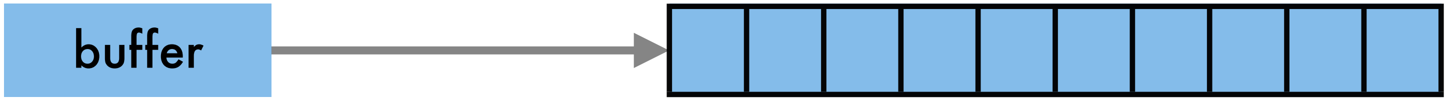
`fill (iterator start, iterator stop, value)` fill vector with a given initial value
`copy (iterator start, iterator stop, iterator destination)` copy one sequence into another
`max_element(iterator start, iterator stop)` find largest value in collection
`min_element(iterator start, iterator stop)` find 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

Using Generic Algorithms

```
...
// randomly shuffle the elements
random_shuffle( numbers.begin(), numbers.end() );

// locate the largest element, O(n)
vector<int>::const_iterator largest = max_element( numbers.begin(), numbers.end() );

cout << "The largest number is " << *largest << "\n";
cout << "It is located at index " << largest - numbers.begin() << "\n";

// sort the elements
sort( numbers.begin(), numbers.end() );

// find the position of the number 5 in the vector, O(log n)
vector<int>::const_iterator five = find( numbers.begin(), numbers.end(), 5 );

cout << "The number 5 is located at index " << five - numbers.begin() << "\n";

// print all numbers
for(vector<int>::const_iterator it = numbers.begin(); it != numbers.end(); ++it)
{
        cout << *it << " ";
}
```

- Type this into the previous program

Notes on Vector Class Implementation



- Vectors (like strings) have a dynamically allocated array buffer (growing/shrinking)
  - Buffer values can be any type
- Two sizes are maintained
  - Physical : maximum capacity
  - Logical : number of used locations
- Simple operations can be performed inline
  - Boosts performance

vector Class Declaration

```
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;
};
```

- Vector Class
    - iterator is a generic pointer
- myCapacity and mySize
- Vectors incorporate bounds checking

vector Class: Constructors

Definitions

```
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());
}
```

Usage

```
vector<string> allNames(100, "empty");
vector<int> collectedData(1000, 0);
vector<float> transactions(50);
```

```
vector<float> moreTrans(transactions);
```

- fill() an copy() are generic algorithms

vector Class Reserve()

```
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;
}
```

- Used by several methods in the vector Class
- If buffer is getting smaller then nothing needs to allocated
- If buffer getting larger then a new allocation must take place

vector Class push_back()

```
template <class T>
void vector<T>::push_back(T value)
   // add a new value to the end of the vector and
resize
   // if necessary.
{
   // grow buffer if necessary

  if (mySize >= myCapacity)
     reserve(myCapacity + 5);

  buffer[mySize] = value;
  mySize++;
}
```

- The goal is to add an item to the end
- If the buffer is full, then it must be increased in size
- What are some performance considerations of using push_back?

Generic Algorithm Implementations

```
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++;
}
```

- Generic fill() algorithm fills range of elements with a given value
- Generic copy() algorithm copies data from one container to another

Class Demonstration

- Examine the vector-based string class implementation found in this week's lessons
  - string.h: interface of the string class
  - string.cpp: implementation of the string class
  - main.cpp: the driver that compares output of your string implementation to the the std::string implementation.
- Study each method that is complete to understand how it works.
- The remainder of the methods are completed as homework.

Class Exercise

- Part I: Convert the selection sort program that uses arrays to one that uses a vector of strings
- Part II: Convert the selectionSort function to a template function
  - Use it to sort the list of names from the names.dat file
  - Create a numbers.dat file and change the program to also sort the numbers.dat file (it should sort both sets of data one after the other)