EE 355 Unit 10

C++ STL - Vectors and Deques

Mark Redekopp
Templates

- We’ve built a list to store integers
- But what if we want a list of double’s or char’s or other objects
- We would have to define the same code but with different types
  - What a waste!
- Enter C++ Templates
  - Allows the one set of code to work for any type the programmer wants

```cpp
struct IntItem {
    int val;
    IntItem *next;
};

class ListInt{
public:
    ListInt(); // Constructor
    ~ListInt(); // Destructor
    void push_back(int newval); ...  
private:
    IntItem *head;
};

struct DoubleItem {
    double val;
    DoubleItem *next;
};

class ListDouble{
public:
    ListDouble(); // Constructor
    ~ListDouble(); // Destructor
    void push_back(double newval); ...  
private:
    DoubleItem *head;
};
```
Templates

- Enter C++ Templates
- Allows the type of variable to be a parameter specified by the programmer
- Compiler will generate separate class/struct code versions for any type desired (i.e instantiating as an object)
  - List<int> my_int_list causes an ‘int’ version of the code to be generated by the compiler
  - List<double> my_dbl_list causes a ‘double’ version of the code to be generated by the compiler

```cpp
// declaring templatized code
template <typename T>
struct Item {
    T val;
    Item<T> *next;
};

template <typename T>
class List{
    public:
        List();  // Constructor
        ~List();  // Destructor
        void push_back(T newval); ...
    private:
        Item<T> *head;
};

// Using templatized code
// (instantiating templatized objects)
int main()
{
    List<int> my_int_list();
    List<double> my_dbl_list();
    my_int_list.push_back(5);
    my_dbl_list.push_back(5.5125);
    double x = my_dbl_list.pop_front();
    int y = my_int_list.pop_front();
    return 0;
}
C++ STL

• C++ has defined a whole set of templatized classes for you to use “out of the box”
• Known as the Standard Template Library (STL)
Vector Class

- Container class (what it contains is up to you via a template)
- Mimics an array where we have an indexed set of homogenous objects
- Resizes automatically

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> my_vec(5); // init. size of 5
    for(unsigned int i=0; i < 5; i++)
    {
        my_vec[i] = i+50;
    }
    my_vec.push_back(10);
    my_vec.push_back(8);
    my_vec[0] = 30;
    unsigned int i;
    for(i=0; i < my_vec.size(); i++)
    {
        cout << my_vec[i] << " ";
    }
    cout << endl;
    int x = my_vec.back(); // gets back val.
    x += my_vec.front(); // gets front val.
    // x is now 38;
    cout << "x is " << x << endl;
    my_vec.pop_back();

    my_vec.erase(my_vec.begin() + 2);
    my_vec.insert(my_vec.begin() + 1, 43);
    return 0;
}
```
Vector Class

- **constructor**
  - Can pass an initial number of items or leave blank

- **operator[ ]**
  - Allows array style indexed access (e.g. `myvec[i]`)

- **push_back(T new_val)**
  - Adds a *copy* of `new_val` to the end of the array allocating more memory if necessary

- **size(), empty()**
  - Size returns the current number of items stored as an unsigned int
  - Empty returns True if no items in the vector

- **pop_back()**
  - Removes the item at the back of the vector (does not return it)

- **front(), back()**
  - Return item at front or back

- **erase(index)**
  - Removes item at specified index (use `begin() + index`)

- **insert(index, T new_val)**
  - Adds `new_val` at specified index (use `begin() + index`)

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> my_vec(5); // 5= init. size
    for(unsigned int i=0; i < 5; i++){
        my_vec[i] = i+50;
    }
    my_vec.push_back(10); my_vec.push_back(8);
    my_vec[0] = 30;
    for(int i=0; i < my_vec.size(); i++){
        cout << my_vec[i] << " ";
    }
    cout << endl;
    int x = my_vec.back(); // gets back val.
    x += my_vec.front(); // gets front val.
    // x is now 38;
    cout << "x is " << x << endl;
    my_vec.pop_back();

    my_vec.erase(my_vec.begin() + 2);
    my_vec.insert(my_vec.begin() + 1, 43);
    return 0;
}
```
Exercises

• [http://bits.usc.edu/websheets/?group=vectors](http://bits.usc.edu/websheets/?group=vectors)
  – vector_eg
  – middle
  – concat
  – parity_counts
  – rpn
  – vector_test
Vector Suggestions

- If you don’t provide an initial size to the vector, you must add items using `push_back()`
- When iterating over the items with a for loop, use an 'unsigned int'
- When adding an item, a copy will be made to add to the vector

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> my_vec;
    for(int i=0; i < 5; i++)
        my_vec.push_back(i+50);

    for(unsigned int i=0; i < my_vec.size(); i++)
        cout << my_vec[i] << " "
    cout << endl;
    do_something(myvec); // copy of myvec passed
    return 0;
}

void do_something(vector<int> v)
{
    // process v;
}
```
Understanding Performance

• Vectors are good at some things and worse at others in terms of performance
  • The Good:
    – Fast access for random access (i.e. indexed access such as myvec[6])
    – Allows for ‘fast’ addition or removal of items at the back of the vector
  • The Bad:
    – Erasing / removing item at the front or in the middle (it will have to copy all items behind the removed item to the previous slot)
    – Adding too many items (vector allocates more memory that needed to be used for additional push_back()’s...but when you exceed that size it will be forced to allocate a whole new block of memory and copy over every item
Deque Class

• Double-ended queues (like their name sounds) allow for additions and removals from either ‘end' of the list/queue

• Performance:
  – Slightly slower at random access (i.e. array style indexing access such as: `data[3]`) than vector
  – Fast at adding or removing items at front or back
Deque Class

- Similar to vector but allows for `push_front()` and `pop_front()` options
- Useful when we want to put things in one end of the list and take them out of the other

```cpp
#include <iostream>
#include <deque>
using namespace std;

int main()
{
    deque<int> my_deq;
    for(int i=0; i < 5; i++){
        my_deq.push_back(i+50);
    }
    cout << "At index 2 is: " << my_deq[2] ;
    cout << endl;
    for(int i=0; i < 5; i++){
        int x = my_deq.front();
        my_deq.push_back(x+10);
        my_deq.pop_front();
    }
    while( !my_deq.empty() ){
        cout << my_deq.front() << " ";
        my_deq.pop_front();
    }
    cout << endl;
}
```

```
my_deq
0  1  2  3  4
50 51 52 53 54

my_deq
0  1  2  3  4
51 52 53 54 60
after 1st iteration

my_deq
0  1  2  3  4
60 61 62 63 64
after all iterations
```
ITERATORS
Iterators

- Vectors and deques (and other STL container classes to be discussed later) give the programmer the appearance that items are located contiguously in memory (like an array) though in implementation they may not
- Vectors and deques allow us to use array-like indexing (‘myvec[i]’) and finds the correct data “behind-the-scenes”
- To iterate over the whole set of items we could use a counter variable and the array indexing (‘myvec[i]’), but it can be more efficient (based on how STL is actually implemented) to keep an ‘internal’ pointer to the next item and update it appropriately
- C++ STL containers define ‘helper’ classes called iterators that help iterate over each item or find an item in the container
Iterators

- Iterators are a new class type defined in the scope of each container
- Initialize them with `objname.begin()`, check whether they are finished by comparing with `objname.end()`, and move to the next item with ++ operator

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> my_vec(5); // 5 = init. size
    for(int i=0; i < 5; i++){
        my_vec.push_back(i+50);
    }
    vector<int>::iterator it
    for(it = myvec.begin() ; it < my_vec.end(); it++){
        ...
    }
}
```
Iterators

- Iterator variable has same semantics as a pointer to an item in the container
  - Use * to 'dereference' and get the actual item

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> my_vec(5); // 5 = init. size
    for(int i=0; i < 5; i++){
        my_vec.push_back(i+50);
    }

    for(vector<int>::iterator it = myvec.begin(); it != my_vec.end(); ++it){
        cout << (*it)
    }
    cout << endl;
    return 0;
}
```
Iterator Tips

• Think of an iterator variable as a ‘pointer’...when you declare it, it points at nothing
• Think of begin() as returning the address of the first item and assigning that to the iterator
• Think of end() as returning the address AFTER the last item (i.e. off the end of the collection) so that as long as the iterator is less than that, you are safe
CONDITIONAL COMPILATION
Multiple Inclusion

- Often multiple file compilation will require #include's of the same file
- This may cause compiling errors when a duplicate declaration is encountered
  - See example
- Would like a way to include only once and if another attempt to include is encountered, ignore it
Conditional Compiler Directives

- Compiler directives start with '('#'
  - define XXX
    - Sets a flag named XXX in the compiler
  - #ifdef, #ifndef XXX ... #endif
    - Continue compiling code below until #endif, if XXX is (is not) defined
- Encapsulate header declarations inside a
  - #ifndef XX
    - #define XX
    - ...
    - #endif

```
#include "Image.h"
class Character {
  public:
    Image *img;
};
```

```
#include "Image.h"
#include "Character.h"
```

```
class Image { // inc. from Image.h
};
class Character{ // inc. from Character.h
...
```

```
main.cpp after preprocessing
```
Conditional Compilation

• Often used to compile additional DEBUG code
  – Place code that is only needed for debugging and that you would not want to execute in a release version

• Place code in a #ifdef XX...#endif bracket

• Compiler will only compile if a #define XX is found

• Can specify #define in:
  – source code
  – At compiler command line with (-Dxx) flag
    • g++ -o stuff -DDEBUG stuff.cpp

```c
int main()
{
    int x, sum=0, data[10];
    ... 
    for(int i=0; i < 10; i++) {
        sum += data[i];
        #ifdef DEBUG
        cout << "Current sum is ";
        cout << sum << endl;
        #endif
    }
    cout << "Total sum is ";
    cout << sum << endl;
}
```

stuff.cpp

> g++ -o stuff -DDEBUG stuff.cpp