CS 103 Unit 12 Slides

Standard Template Library
Vectors & 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

```
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 instantiated 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

```
// 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)

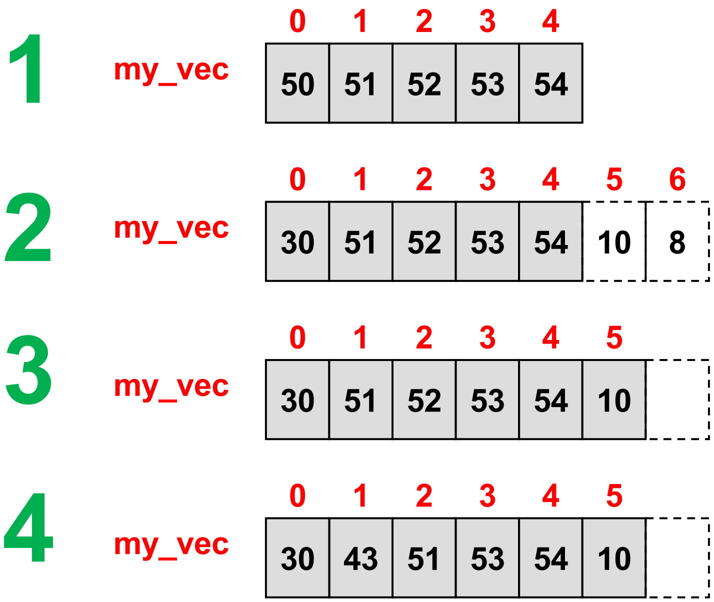
Vector Class

- 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

```
  #include <iostream>
  #include <vector>
  using namespace std;
  int main()
  {
    vector<int> my_vec(5); // init. size of 5
1   for(unsigned int i=0; i < 5; i++){
      my_vec[i] = i+50;
    }
    my_vec.push_back(10); my_vec.push_back(8);
2   my_vec[0] = 30;
    unsigned int i;
    for(i=0; i < my_vec.size(); i++){
      cout << my_vec[i] << " ";
    }
    cout << endl;

3   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();

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



1 my_vec

0	1	2	3	4
50	51	52	53	54

2 my_vec

0	1	2	3	4	5	6
30	51	52	53	54	10	8

3 my_vec

0	1	2	3	4	5	
30	51	52	53	54	10	

4 my_vec

0	1	2	3	4	5	
30	43	51	53	54	10	

Inserting or erasing an element in the end needs only one step. However, the worse case of erasing and inserting an element in the middle of or at the beginning of the vector is O(n)

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)

```
#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;
}
```

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

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

using namespace std;

int main()
{
  vector<int> my_vec;
  for(int i=0; i < 5; i++){
    // my_vec[i] = i+50; // doesn’t work
    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;

}
```

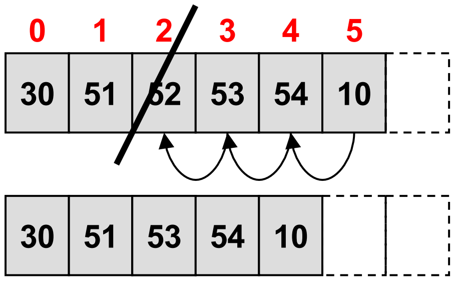
Your Turn

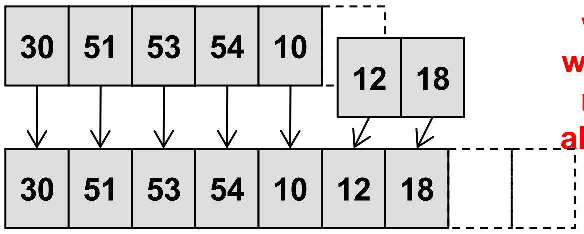
- In-class Exercises
  - vector_eg
  - middle
  - concat
  - parity_counts
  - rpn

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

After deleting we have to move everyone up





Vector may have 1 extra slot, but when we add 2 items a whole new block of memory must be allocated and items copied over

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

1 my_deq

0	1	2	3	4
50	51	52	53	54

2 my_deq

0	1	2	3	4
51	52	53	54	60

after 1st iteration

3 my_deq

0	1	2	3	4
60	61	62	63	64

after all iterations

4 my_deq

```
#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);
  }                                                    1
  cout << “At index 2 is: “ << my_deq[2] ;
  cout << endl;

  for(int i=0; i < 5; i++){
    int x = my_deq.front();                            2
    my_deq.push_back(x+10);
    my_deq.pop_front();                                3
  }
  while( ! my_deq.empty()){
    cout << my_deq.front() << “ “;
    my_deq.pop_front();
  }                                                    4
  cout << endl;

}
```