University of Southern California

Viterbi School of Engineering

EE599 (to be EE595)
Software Design and Optimization

C++ Templates
STL Vectors and Deques

Reference: Professor Mark Redekopp's EE355 course materials, and online resources

Motivation for Templates

```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
   cout << myMax<int>(3, 7) << endl;
   cout << myMax<char>('g', 'e') << endl;
   return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

Advantages

Code is simple and easier to understand	More chances of making mistakes
Easy to debug	Difficult to handle errors as they are handled by pre-processor
Being a function call, they are less efficient than macros	More efficient as they are compiled inline
As they is compile time checking, templates are considered type safe	They do not have any type checking and therefore are type unsafe

Exercises: temp , mcr

Templates

- We've built a list to store integers
- But what if we want a list of doubles or chars or other objects
- We would have to define the same code but with different types
  - We could do better!
- 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

- **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;
}
```

Templates

- Templates for
  - Functions
  - Classes
  - Variables

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**

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	

```
#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;
}
```

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 Class (cont.)

- Vectors use a dynamically allocated array to store their elements
- This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it
- This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container
- Reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (i.e., constant amortized delay)

Exercises

- vector_eg
- vector_bad
- middle
- concat
- parity_counts
- rpn

Vector Suggestions

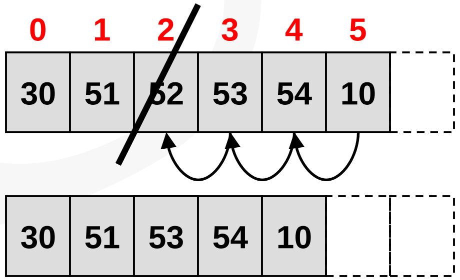
- If you don't provide an initial size to the vector, you must add items using push_back()
- When iterating over the items with a for loop, use an 'unsigned int'
- When adding an item, a copy will be made to add to the vector

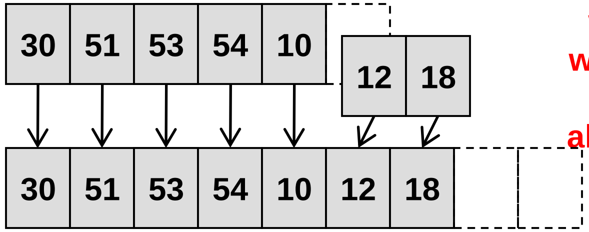
```
#include <iostream>
#include <vector>

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;

}
```

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

Deques vs Vectors

- Used for similar purposes and provide similar interfaces
- Internally, deques are more complex than vectors
  - A vector uses a single array that needs to be occasionally reallocated for growth
  - Why deques are more complex?
    - A deque's elements can be scattered in different chunks of storage, with the container keeping the necessary information internally to provide direct access to any of its elements in constant time
    - deque's Iterators provide a uniform sequential interface
    - Deques grow more efficiently for cases that reallocations become expensive, e.g., in case of long sequences

Deque Class (cont.)

- 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);
  }
  cout << "At index 2 is: " << my_deq[2] ;          // 1
  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;

}
```

Vector/Deque/String Suggestions

- When you pass a vector, deque, or even C++ string to a function a deep copy will be made which takes time
- Copies may be desirable in a situation to make sure the function alters your copy of the vector/deque/string
- But passing by const reference saves time and provide the same security

```
#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);
  }

  // can myvec be different upon return?
  do_something1(myvec);

  // can myvec be different upon return?
  do_something2(myvec);
  return 0;
}
void do_something1(vector<int> v)
{
  // process v;
}
void do_something2(const vector<int>& v)
{
  // process v;
}
```