

CS 103 Unit 4b – C++ Standard Template Library (Vectors and Deques) C++ References

CSCI 103L Teaching Team

Standard Template Library

TEMPLATES AND C++ STL

Templates

- We've built a linked 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
 - 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 (variable) specified by the programmer**
- Compiler will generate separate class/struct code versions for any type desired (i.e instantiated as an object)
 - `List<int> ilist` causes an `int` version of the code to be generated by the compiler
 - `List<double> dlist` causes a `double` version of the code to be generated by the compiler
 - `List<Quad> qlist` causes a `Quad` version of the code to be generated by the compiler

```
// declaring templated 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 templated code
// (instantiating templated objects)
int main()
{
    List<int> ilist;
    List<double> dlist;

    ilist.push_back(5);
    dlist.push_back(5.5125);

    double x = dlist.pop_front();
    int y = ilist.pop_front();
    return 0;
}
```

C++ STL

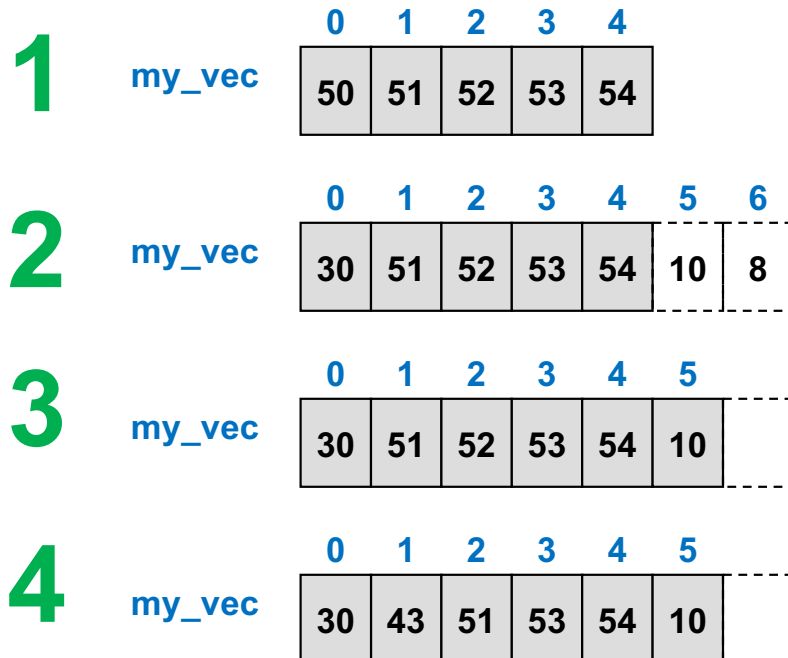
- C++ has defined a whole set of templated classes for you to use "out of the box"
- Known as the **Standard Template Library (STL)**
 - `vector<T>`
 - `list<T>`
 - `deque<T>`
 - `set<T>`
 - `priority_queue<T>`
 - `map<K,V>`
- These and their implementation are the focus of CS 104

Vector Class

- A full-featured, managed **array** implementation
 - Will **grow the array** as needed
 - Will perform **deep** copies
 - Tracks its **size**
 - Still allows array-like indexing (aka random-access) such as `data[i]`
 - And several other capabilities
- See more online [documentation](#)

Vector Class

- Container class (what it contains is up to the programmer via a template)
- Implements an array-based list that can be used like (and gives the benefits of) an array but can **resize automatically**



```

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

    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;
    }

```

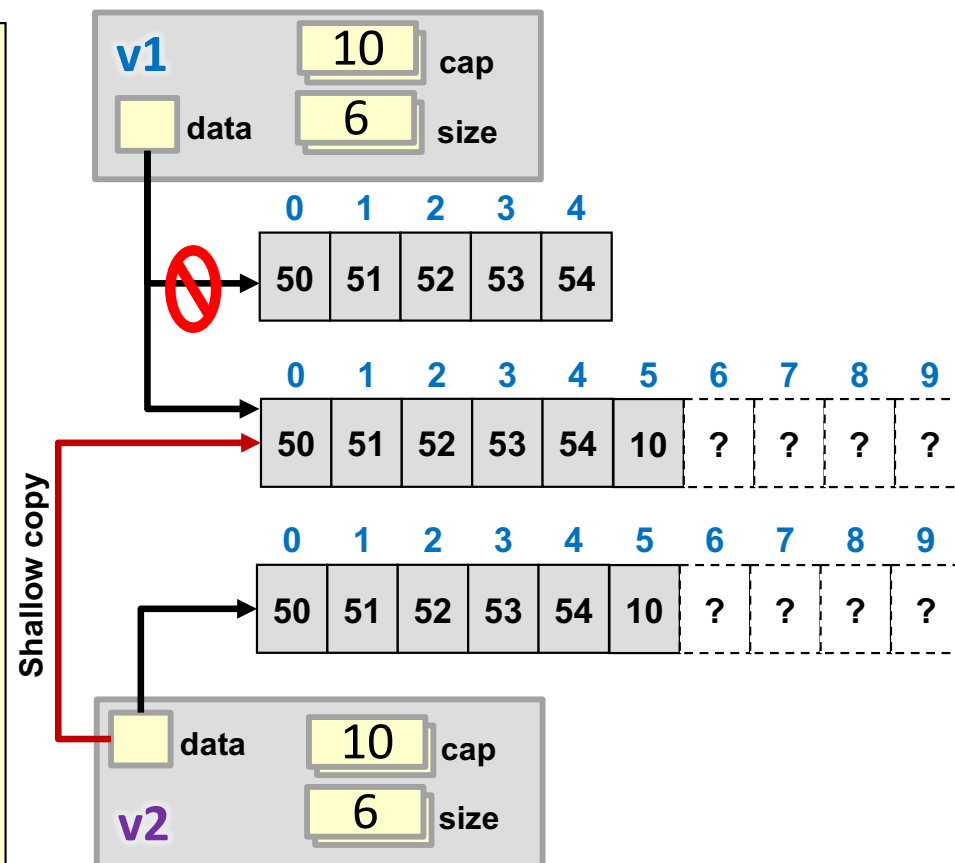
What Happens Behind the Scenes

- Vectors abstract arrays
 - Behind the scenes vectors are just creating and manipulating arrays but giving you a simplified set of operators and functions
- Vectors store a **pointer** to the array (which must be dynamically allocated since it is of variable size, the **capacity** of the array and the **size** of used/occupied elements

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v1(5);
    for(int i=0; i < 5; i++){
        v1[i] = i+50;
    }
    v1.push_back(10);
    // causes a resize behind the scenes

    vector<int> v2;
    v2 = v1; // causes a deep copy

} // Destructor called on v2 and v1.
  // What should it do?
```



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) – pop from the middle or front
 - Removes item at specified index (use begin() + index)
- insert(index, T new_val) – push to middle/front
 - Adds new_val at specified index (use begin() + index)

```
#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;
}
```

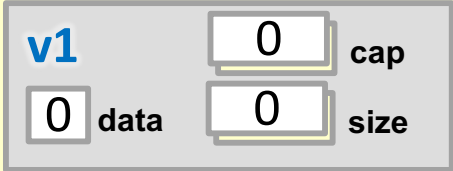
Vector Suggestions

- **Important:** 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' or 'size_t' type to match the return type of `size()`
- Makes a deep copy when passing-by-value
- When adding an item, a copy will be made to add to the vector

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

int main()
{
    vector<int> myvec;
    for(int i=0; i < 5; i++){
        // myvec[i] = i+50; // would segfault
        myvec.push_back(i+50);
    }
    for(size_t i=0; i < myvec.size(); i++)
    {   cout << myvec[i] << " "; }
    cout << endl;
    do_it(myvec); // copy of myvec passed
    string s1 = "abc";
    myvec<string> v1;
    v1.push_back(v1);
    s1 = "xyz";
    cout << v1[0] << endl; // prints "abc";
    return 0;
}

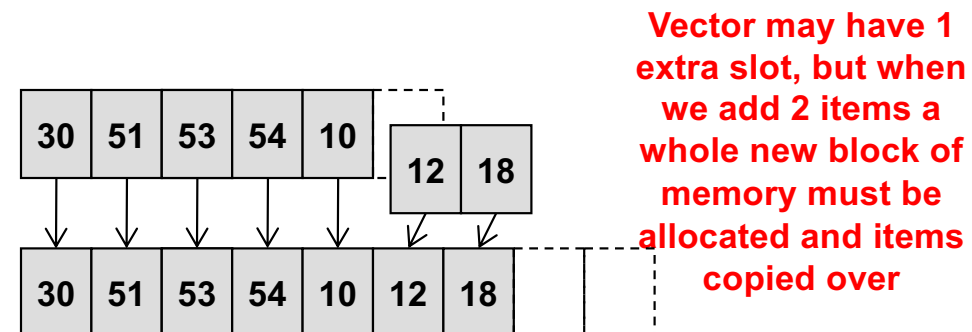
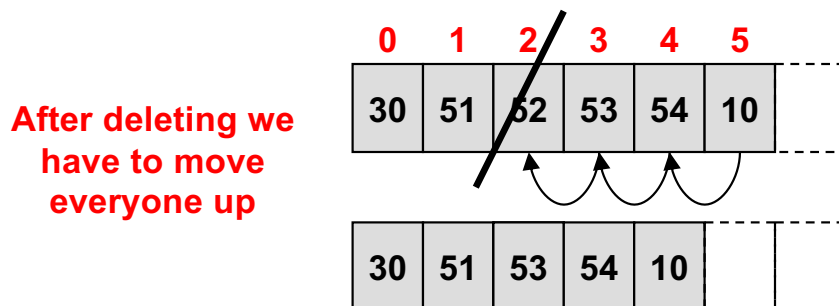
void do_it(vector<int> v)
{
    v.push_back(60); // only adds to v (the copy)
                    // not myvec
}
```



The diagram shows a vector `v1` with a capacity of 0 and a size of 0. The capacity is represented by a box labeled 'cap' with the value 0. The size is represented by a box labeled 'size' with the value 0. The data is represented by a box labeled 'data' with the value 0.

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. location i). Access to any index i is $O(1)$
 - Allows for 'fast' **addition** or **removal** of items **at the back** of the vector
- **The Bad:**
 - Inserting / removing item **at the front** or in the **middle** (it will have to shift all items up or down to add/remove the item)
 - Adding an item when the underlying array is full will trigger a $O(___)$ **resize** operation where the vector will allocate a whole new block of memory and copy over every item
 - *But if implemented wisely, it is not as bad as it sounds (more in 104)!*



Your Turn

- In-class Exercises