COMP 3522

Object Oriented Programming in C++

Week 7 Day 1

Agenda

1. auto keyword
2. Ranged for
3. Intro to the STL
4. STL containers

COMP 3522

Roadmap – Topics

- C++ Standard Template Library
  - Containers <- Today/this week
  - Iterators
  - Algorithms
  - Functors
- Template programming
- Design patterns and idioms
- Smaller topics as well

auto KEYWORD

The auto keyword

- When used as a variable type, auto specifies that **the type of the variable will be deduced automatically from its initializer**.
- When used as a function return type, auto specifies that **the return type will be deduced from the return statements**

auto

```
double sum = 5.0;
auto a; //ERROR, auto requires initializer
auto d = 5.0;
auto i = 1 + 2;

int add(int x, int y) { return x + y; }
int main()
{
    auto sum = add(5, 6);
}
```

auto can’t be used with function parameters

```
void add_and_print(auto x, auto y)
{
    std::cout << x + y;
}
```

This won’t work because the compiler can’t infer types for function parameters x and y at compile time

auto can be used with function return types

```
auto add(int x, int y)
{
    return x + y;
}
```

I would like to discourage this:

- Using auto for variables is fine because the object is right there
- Using auto for functions means we have to dig into the function to find out what it's supposed to be returning.

Some programmers like to do this:

Instead of this:

```
int add(int x, int y);
```

They like to do this:

```
auto add(int x, int y) -> int;
```

In this case, auto does not perform type inference, it is just part of the syntax to use a **trailing return type**. But why, though?

So we can do this (so easy to read!)

```
auto add(int x, int y) -> int;
auto divide(double x, double y) -> double;
auto print_something() -> void
auto calculate_that(int x, double d) -> string
```

Additional reading: http://en.cppreference.com/w/cpp/language/auto

RANGED FOR

“The ranged for” aka for-each loop

- Identical to Java
- Some of you have already been using it
- Formally it executes a for loop over a specified range

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};
for (const int& i : v) // const reference
    std::cout << i << ‘ ‘;
```

“The ranged for” aka for-each loop

```
std::vector<int> v = {0, 1, 2, 3, 4, 5};
for (auto i : v) // access by value, i is int
    std::cout << i << ' ‘;

// the initializer may be a braced-init-list
for (int n : {0, 1, 2, 3, 4, 5})
    std::cout << n << ' ';
std::cout << '\n';
```

“The ranged for” aka for-each loop

```
// the initializer may be an array
int a[] = {0, 1, 2, 3, 4, 5};
for (int n : a)
    std::cout << n << ' ';

// the loop variable doesn’t have to be used
for (int n : a)
    std::cout << something_unrelated << ' ';
std::cout << '\n';
```

INTRO TO THE STL

Standard Template Library

- C++ **STL**
- Like the Java Collections Framework **ON STEROIDS**
- One of the most fun and interesting reasons to work with C++
- Composed of:
  1. Containers
  2. Iterators
  3. Algorithms
  4. Function objects ← So much fun.

About the STL

- Uses **value semantics** – the containers get a copy of the object we are putting in it
- This means our element class must have:
  - Copy constructor
  - Assignment operator
  - Destructor
- STL **performs almost no checking** – the programmer is responsible for meeting preconditions
- STL uses **half-open ranges** [included, not included)
  - Imagine array size 5. array[0,5)
  - When iterating through include 0^{th} index, exclude 5^{th} index

Containers

1. **Sequence** containers
    1. We specify the order
    2. array, vector, deque, list, forward_list
2. **Associative** containers
    1. Objects are automatically sorted
    2. Can be searched with O(logn) complexity
    3. set, multiset, map, multimap
3. **Unordered associative** containers
    1. Stored using hash
    2. Can be searched O(1) amortized, O(n) worst case
    3. unordered_set, unordered_multiset, unordered_map, unordered_multimap

Suppose we have a container object c…

c.insert(x); // Inserts a copy of x into an associative container
c.insert(position, x); // Inserts a copy of x into a sequence container

c.begin(); // Returns an iterator pointing to the first element in the container if it exists
c.end(); // Returns an iterator one past the end of the container

for (**auto** it = c.begin(); it != c.end(); ++it) { // process! }

About vectors (again…)

A vector is a **dynamic array** that offers random access and insertion/deletion at the end

```
#include <vector>
vector<int> v;
v.push_back(2);
vector<int>::iterator it;
for (it = v.begin(); it != v.end(); ++it);
     cout << *it << endl;
```

More about vectors…

```
vector<int> v2 {2, 4, 6, 8, 10};
v2.insert(v2.begin(), 6); // 6, 2, 4, 6, 8, 10
v2.insert(v2.begin() + 3, -4); // 6, 2, 4, -4, 6, 8, 10
v2.erase(v2.begin() + 1); //6, 4, -4, 6, 8, 10

int a [] = { 3, 6, 9, 12, 15 };
vector<int> v3 {a, a + 5 }; // copies in the range
[first, last)
```

Another example

```
void print(const vector<int> &v)
{
     for (vector<int>::size_type i = 0; i < v.size(); ++i)
          cout << v[i] << ‘ ‘;
     cout << endl;
}
```

(Re-)familiarize yourself with the vector

- empty()
- size()
- reserve()
- capacity()
- clear()
- insert()
- push_back() and pop_back()
- front() and back()
- assign()

http://www.cplusplus.com/reference/vector/vector/

STL CONTAINERS

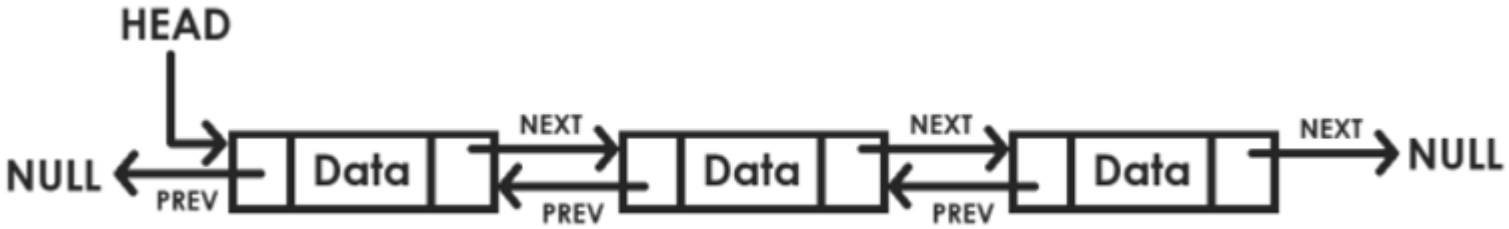
I'm so tired of vectors

- We've talked about vectors enough
- Let's visit some different containers in the STD
- Java has the Java Collections Framework
- C++ has the **Standard Template Library**
- Java has collections
- C++ has **containers**

Container classes

- **First-class containers**
  - Vector
  - List
  - Deque
  - Map and Multimap
  - Set and Multiset
- **Container adaptors (modify and restrict first class container)**
  - Stack (default implementation is the deque)
  - Queue (default implementation is also the deque)
  - Priority_queue (default implementation is the vector)

list



- <list>
- std::list
- Sequence container
- Iterator in both directions
- Implemented as a doubly-linked list
- Fast insertions, deletions, and swaps when using an iterator
- Linear travel time to a node from one of the ends

http://www.cplusplus.com/reference/list/list/

Deque – Pronounced ‘deck’

- <deque>
- std::deque
- Double-ended queue
  - Queue is French for tail
  - Also used in England instead of line-up, i.e., waiting in line
- Sequence container
- Provides linear storage, with fast inserts at both ends
- Unlike std::vector, elements are not stored contiguously
- Typically allocated with a sequence of fixed size arrays (**neat**)

deque.cpp http://www.cplusplus.com/reference/deque/deque/

Side-note: pair

- <utility>
- Object that can hold two values of different types
- Has member functions (accessors) called first and second
- See **pair.cpp** for a sample

- There is a built-in C++ function to simplify the creation of a pair: **make_pair**.

http://www.cplusplus.com/reference/utility/pair/

Map

- The value_type of a map is a Pair
- How do we add something to a map?
- Suppose we have a phonebook that maps strings to longs:

```
phonebook.
    insert(map<string, long>::
                        value_type("Sam", 6045551212));
```

- This fails if an element with the same key is already in the map:

```
phonebook.insert(make_pair("Sam", "2505551212"));
```

http://www.cplusplus.com/reference/map/map/

map

- <map>
- std::map
- Sorted associative container
- Provides a collection of 1-to-1 mappings, i.e. a collection of key/value pair objects
- value_type is a pair type that combines key and value
- Keys must be unique
- Keys are sorted using a comparison function (like the Java Comparator)
- Logarithmic speed for search, insertion, and removal (**fast!**)
- Map.cpp

multimap

- <map>
- std::multimap
- Associative container of key-value pairs
- Permits multiple entries with the same key (**crazy!**)
- Sorting is performed using comparison function
  - Equivalent keys sorted in order of insertion
- Logarithmic speed for search, insertion, and remove (**fast!**)

Multimap.cpp

http://www.cplusplus.com/reference/map/multimap/

set

- <set>
- std::set
- Associative container contains a sorted set of **unique** objects of type Key
- The value of an element also identifies it
- Elements cannot be edited after being added (are const)
- Sorting performed using key comparison function
- Logarithmic speed for search, insertion, and remove (fast!)
- Typically implemented as a binary search tree

Comparing (and sorting) set elements

- The C++ compare concept:
  - Type T satisfies Compare if it:
    1. Satisfies BinaryPredicate (evaluates to true/false)
    2. Induces a strict weak ordering. *
- Suppose we have a struct called myPair that stores 2 ints, x and y
- We need to write a < operator to use it in a set

```
bool operator<(Const myPair& lhs, const myPair& rhs) {
    return lhs.x + lhs.y < rhs.x + rhs.y;
} // Sorts myPairs by sum of components
```

* https://en.wikipedia.org/wiki/Weak_ordering#Strict_weak_orderings

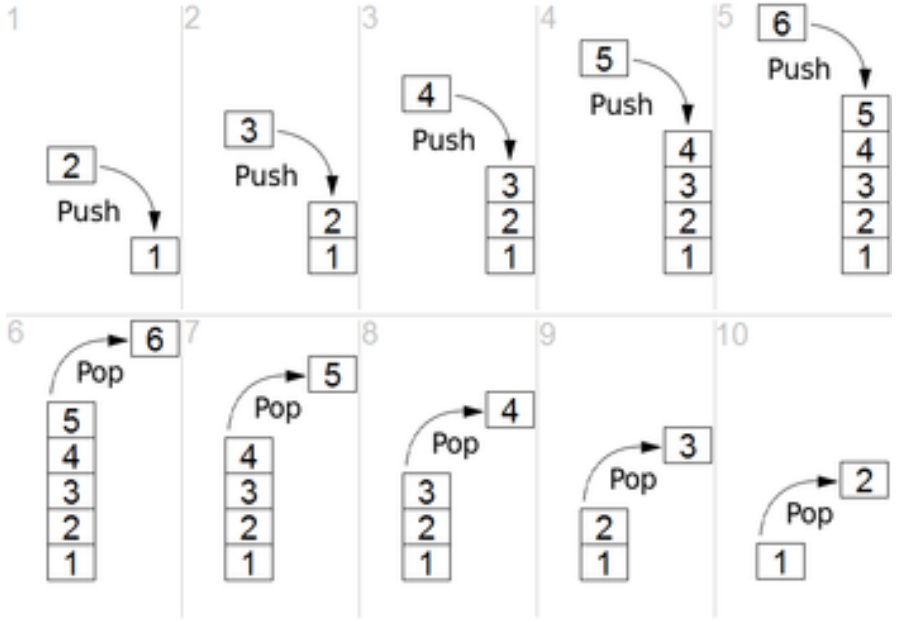
multiset

- <set>
- std::multiset
- Associative container that contains a sorted set of objects
- The objects do not have to be unique
- Elements cannot be edited after being added (are const)
- Logarithmic speed for search, insertion, and remove (fast!)

Multiset.cpp

http://www.cplusplus.com/reference/set/multiset/

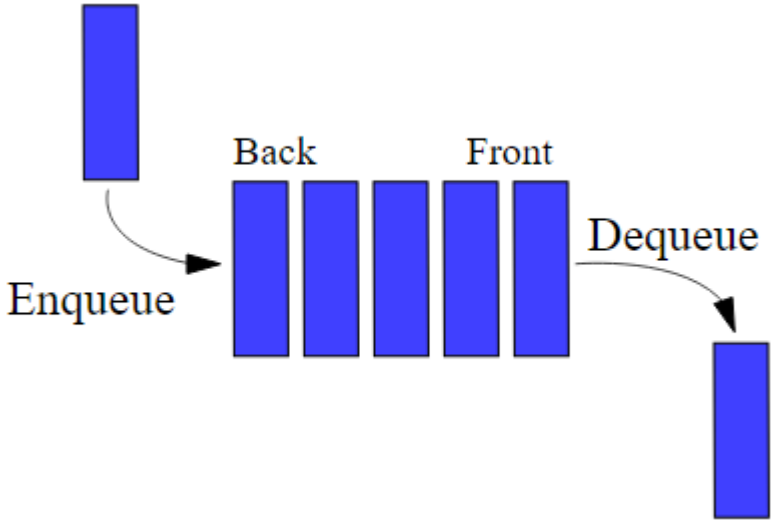
CONTAINER ADAPTOR - Stack



- A Container Adaptor
- **FILO/LIFO**
- Class template wraps the implementation
- <stack>
- std::stack
- **You're already an expert on implementing and using the stack**

http://www.cplusplus.com/reference/stack/stack/

CONTAINER ADAPTOR - Queue



- Faire la queue = French for “to wait in line”
- <queue>
- std::queue
- FIFO
- Wraps an underlying implementation, and only a specific set of functions is provided
- Push elements into the back, and retrieve from the front

http://www.cplusplus.com/reference/queue/queue/

CONTAINER ADAPTOR – Priority_queue

- <queue>
- std::priority_queue
- First element is always the "greatest/largest/first"
- Similar to a heap, in which elements can be added anytime and only the max_heap element can be retrieved
- Constant time (can't be faster!) lookup of the first element
- Logarithmic insertion and extraction

PriorityQueue.cpp http://www.cplusplus.com/reference/queue/priority_queue/

STL CONTAINERS & TYPEDEFS

Review: typedef

The typedef keywork creates an **alias** that can be used anywhere instead of a (possibly) complex type name

```
// declares int_t to be an alias for the type int
typedef int int_t;

// arr_t is array of 3 int
typedef int arr_t[3];

// struct Foo and Foo are the same thing
struct Foo { ... };
typedef struct Foo Foo;
```

STL typedef Restaurant Analogy

Fast Food
- LCDMenu

Fancy Restaurant
- PaperMenu

Food Truck
- BoardMenu

What's a menu?

What's a menu?

What's a menu?

Me:
Can I get a menu?

STL typedef Restaurant Analogy

Fast Food - LCDMenu	Fancy Restaurant - PaperMenu	Food Truck - BoardMenu
OK!	**What's a LCDmenu?**	**What's a LCDmenu?**

Me:
Can I get a LCDMenu?

STL typedef Restaurant Analogy

Fast Food
- LCDMenu

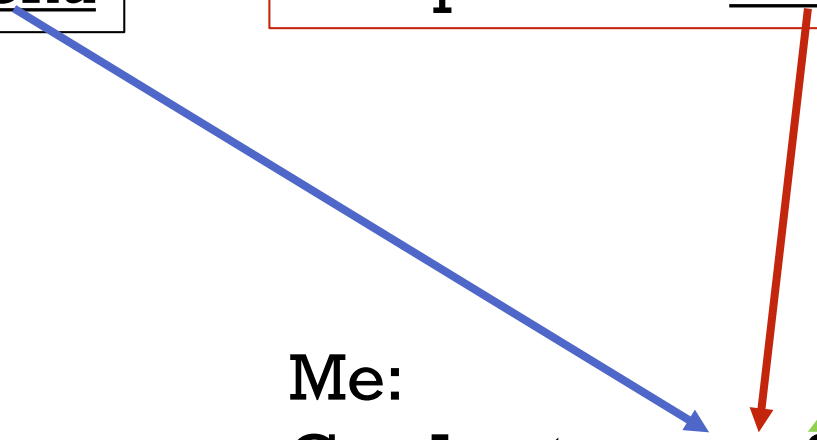
- typedef LCDMenu menu

Fancy Restaurant
- PaperMenu

- typedef PaperMenu menu

Food Truck
- BoardMenu

- typedef BoardMenu menu



Me:
Can I get a menu?

STL typedef Restaurant Analogy

*psuedocode

Vector*
- RandomAccessIterator

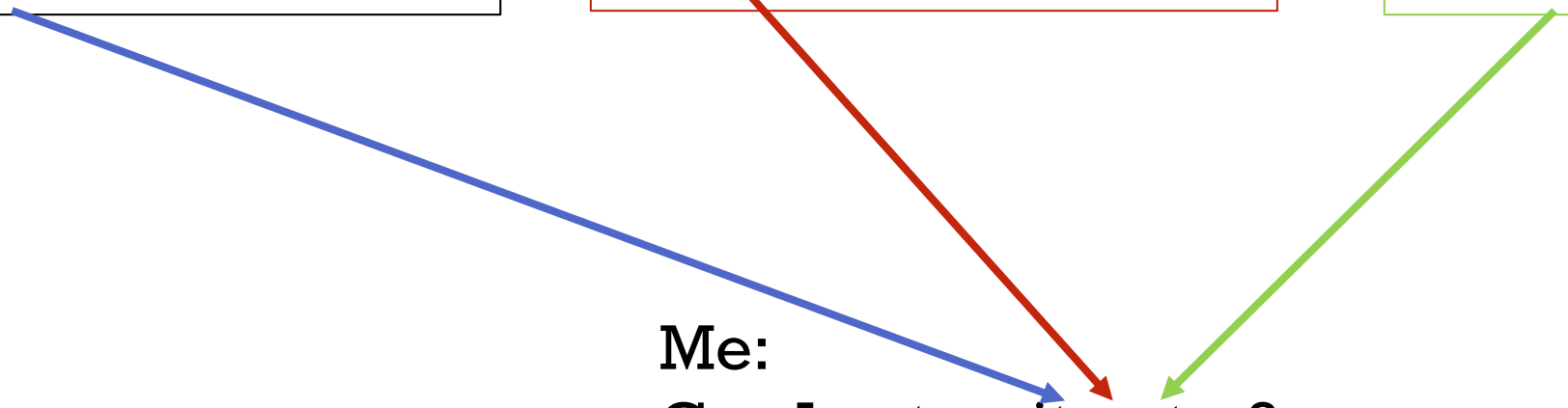
- typedef RandomAccessIterator iterator

Map*
- BiDirectionalIterator

- typedef BiDirectionalIterator iterator

Set*
- BiDirectionalIterator

- typedef BiDirectionalIterator iterator



Me:
Can I get an iterator?

STL typedef Restaurant Analogy

*psuedocode

Vector*
- RandomAccessIterator

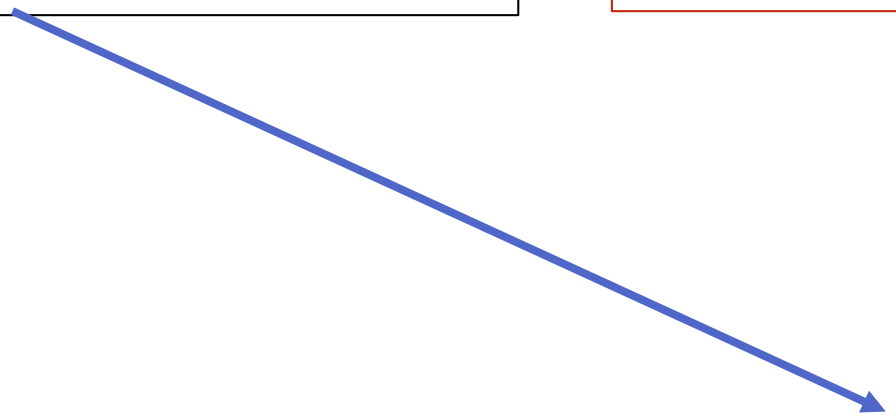
- typedef RandomAccessIterator iterator

Map*
- BiDirectionalIterator

- typedef BiDirectionalIterator iterator

Set*
- BiDirectionalIterator

- typedef BiDirectionalIterator iterator



vector<int>::iterator myIter;

Gets RandomAccessIterator using iterator typedef

STL typedef Restaurant Analogy

*psuedocode

Vector*
- RandomAccessIterator

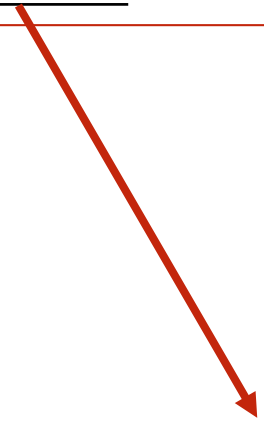
- typedef RandomAccessIterator iterator

Map*
- BiDirectionalIterator

- typedef BiDirectionalIterator iterator

Set*
- BiDirectionalIterator

- typedef BiDirectionalIterator iterator



map<string, int>::iterator myIter;

Gets BiDirectionalIterator using iterator typedef

typedefs.cpp

Containers and typedefs

- STL containers have standard typedefs
  - size_type
  - value_type
  - pointer
  - reference and const_reference
  - difference_type, etc.
- This makes it possible to write generic functions that work on containers
- If we create a container, we should implement these standard typedefs

Why? **Consistency**!

- Using the typedef mechanism means we can give the **same name to the same conceptual entity** across different container classes
- For example, consider the iterator:
  - vector<int>::iterator gives us a **random access iterator** for a vector of integers
  - list<string>::iterator gives us a **bidirectional iterator** for a list of strings

container_type

- The type of first-class container upon which a container adaptor is based

size_type

- Unsigned integer type
- Sufficiently large to hold the size of any object of that class
- Appears in all first-class containers and in the container adaptors.

size_type using deque

```
void print(const deque<int>& d)
{
    for (deque<int>::size_type i = 0;
         i < d.size(); ++i) {
        cout << d[i] << ‘ ‘;
    }
    cout << endl;
}
```

reference

- A reference to a component of a container object (i.e., T&, where T is the component type of the container object)
- typedef T& reference;

```
int num = 5;
vector<int>::reference vecRef = num;
```

const_reference

- A const reference to a component of a container object (i.e., const T&, where T is the component type of the container object)

iterator

- An iterator of the default type for a (first-class) container type

```
vector<int> myVec = {1,2,3};
vector<int>::iterator vecIter = myVec.begin();
```

const_iterator

- A const iterator of the default type for a (first-class) container type

reverse_iterator

- A reverse iterator of the default type for a (first-class) container type

```
vector<int>::reverse_iterator = myVec.rbegin();
```

const_reverse_iterator

- A const iterator of the default type for a (first-class) container type

pointer

- A pointer to a component of a (first-class) container object (i.e., same as T*, where T is the component type of the container object)
- typedef T* pointer;

```
int *otherPtr = new int{5};
vector<int>::pointer vecPtr = otherPtr;
```

const_pointer

- A const pointer to a component of a (first-class) container object (i.e., same as const T*, where T is the component type of the container object)

value_type

- The same type as the type of the values stored in a container object
- the same as T for sequential containers and container adaptors

```
vector<int>::value_type num; //num is type int
```

- Usually pair<const K, V> for associative map containers.

```
map<string, int>::value_type myMap; //myMap is a pair
type of <string, int>
```

key_type

- The same type as K in <K, V>
- Only used by map and multimap

```
map<string, int>::key_type myKey = “string”;
```

mapped_type

- The same type as V in <K, V>

```
map<string, int>::mapped_type myMapped = 5;
```

- Not used by set, vector, deque, or list

ACTIVITY

1. Check out this website http://cs.stmarys.ca/~porter/csc/ref/stl/containers_typedefs.html which has a GREAT CHART of the different STL container class typedefs with some good, short definitions for each one.
2. For each YES in the chart, i.e., the YES for size_type, check out the corresponding type in the containers, i.e., vector at http://www.cplusplus.com/reference/vector/vector/ and at https://en.cppreference.com/w/cpp/header/vector
3. Summarize your findings in a one-page PDF