Container Classes

• C++ Standard Template Library provides one or more implementations of the various ADTs
  – DynamicArrayList => C++: `std::vector<T>`
  – LinkedList => C++: `std::list<T>`
  – Deques => C++: `std::deque<T>`
  – Sets => C++: `std::set<T>`
  – Maps => C++: `std::map<K,V>`

• Question:
  – Consider the get(i) method. What is its time complexity for...
  – ArrayList => O(____)
  – LinkedList => O(____)
Container Classes

• ArrayLists, LinkedList, Deques, etc. are classes used simply for storing (or contain) other items

• C++ Standard Template Library provides implementations of all of these containers
  – DynamicArrayList => C++: std::vector<T>
  – LinkedList => C++: std::list<T>
  – Deques => C++: std::deque<T>
  – Sets => C++: std::set<T>
  – Maps => C++: std::map<K,V>

• Question:
  – Consider the get(i) method. What is its time complexity for...
  – ArrayList => O(1) // contiguous memory, so just go to location
  – LinkedList => O(i) or O(n) // must traverse the list to location i
Iteration

- Consider how you iterate over all the elements in a list
  - Use a for loop and get() or operator[]
- For an array list this is fine since each call to at() is $O(1)$
- For a linked list, calling get(i) requires taking $i$ steps through the linked list
  - 0th call = 1 step
  - 1st call = 2 steps
  - 2nd call = 3 steps
  - $1+2+...+n-2+n-1+n = O(n^2)$
- You are re-walking over the linked list a lot of the time

```cpp
ArrayList<int> mylist;
...
for(int i=0; i < mylist.size(); ++i)
{
    cout << mylist.get(i) << endl;
}
```

```cpp
LinkedList<int> mylist;
...
for(int i=0; i < mylist.size(); ++i)
{
    cout << mylist.get(i) << endl;
}
```

```
head
  0x148
  0x148 0x1c0
  0x1c0 0x3e0
  0x3e0 5 NULL
```

get(0)
get(1)
get(2)
Iteration: A Better Approach

• Solution: Don't use get()
• Use an **iterator**
  - An object containing an internal state variable (i.e. a pointer or index) that moves one step in the list at a time as you iterate, saving your position
• Iterator tracks the internal location of each successive item
• Iterators provide the semantics of a **pointer** to the values in the list
• Assume
  - mylist.begin() returns an iterator to the beginning item
  - mylist.end() returns an iterator "one-beyond" the last item
  - ++it (preferred) or it++ moves iterator on to the next value

```
// new iterator approach
LinkedList<int> mylist;
...
iterator it;
for(it = mylist.begin(); it != mylist.end(); ++it)
{
  cout << *it << endl;
}
```

```
// old index approach
int size = mylist.size();
int i;
for(i = 0; i < size; i++)
{
  cout << mylist[i];
}
```
Iterators

• List implementations may allow us to use array-like indexing (e.g. `myvec[i]`, `myvec.at(i)`, `myvec.get(i)`) that finds the correct data “behind-the-scenes” (giving the illusion that data is contiguous in memory though it may not be)

• 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 the data structure 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 store these internal pointers and 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**
  - Type is `container::iterator` (`vector<int>::iterator` is a type)
- 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 = my_vec.begin() ; it != my_vec.end(); ++it){
        cout << *it << endl;
    }
}
```

```cpp
#include <vector.h>
template<class T>
class vector
{
    class iterator {
    }
};
```
Iterators

- Iterator variable has the same semantics as a `pointer` to an item in the container
  - Use `*` to 'dereference' and get the actual item
  - Since you're storing integers in the vector below, the iterator acts and looks like an `int*`

```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 = my_vec.begin(); it != my_vec.end(); ++it){
        cout << *it << 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 but really returns an iterator to the first item.

• Think of `end()` as returning the address AFTER the last item (i.e. off the end of the collection or maybe NULL) but really returns an iterator to the one-off-the-end
  
  – So as long as your iterator is less than or not equal to the `end()` iterator, you are safe
Iterator Pro Tip 1

- **NEVER** (accidentally) compare iterators from different containers (i.e. always compare iterators obtained from the same instance of the data structure)
  - May allow iterator to go off the end of a container

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

int main()
{
    Scores s;
    ...
    for(vector<int>::iterator it = s.mtGrades().begin();
        it != s.mtGrades().end();
        ++it)
    {
        cout << *it << endl;
    }
    return 0;
}
```

### Wrong!

```cpp
vector<int> g = s.mtGrades();
for(vector<int>::iterator it = g.begin();
    it != g.end(); ++it)
    { ... }
```

### Right!

```cpp
class Scores {
    public:
        vector<int> mtGrades()
        { return mt; }
    private:
        vector<int> mt;
};
```
C++ STL Algorithms

• Many useful functions defined in <algorithm> library

• All of these functions usually accept iterator(s) to elements in a container

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

int main()
{
    vector<int> my_vec(5); // 5 = init. size
    for(int i=0; i < 5; i++){
        my_vec.push_back(rand());
    }
    sort(my_vec.begin(), my_vec.end());
    for(vector<int>::iterator it = my_vec.begin(); it != my_vec.end(); ++it){
        cout << *it << endl;
    }
    return 0;
}
```
Maps (a.k.a. Dictionaries or Ordered Hashes)

ASSOCIATIVE CONTAINERS
class Student {
public:
    Student();
    Student(string myname, int myid);
    ~Student();
    string get_name() { return name; } // get their name
    void add_grade(int score);  // add a grade to their grade list
    int get_grade(int index);   // get their i-th grade
private:
    string name;
    int id;
    vector<int> grades;
};

Note: This class is just a sample to hold some data and will be used as the 'value' in a map shortly.
Creating a List of Students

• How should I store multiple students?
  – Array, Vector, LinkedList?

• It depends on what we want to do with the student objects and **HOW we want to access them**
  – If we only iterating over all elements a list performs fine
  – If we want to access random (individual) elements where we have to search for them, lists give poor performance.
  – O(n) [linear search] or O(log n) [binary search] to find student or test membership

```cpp
#include <vector>
#include "student.h"
using namespace std;

int main()
{
    vector<Student> studs;
    ...

    unsigned int i;  // ITERATE OVER ALL ELEMENTS (LIST GIVES FINE PERFORMANCE)
    // compute average of 0-th score
    double avg = 0;
    for(i=0; i < studs.size(); i++){
        avg += studs[i].get_grade(0);
    }
    avg = avg / studs.size();

    // check "Tommy"'s score
    int tommy_score = -1;
    for(i=0; i < studs.size(); i++){
        if(studs[i].get_name() == "Tommy"){
            tommy_score = studs[i].get_grade(2);
            break;
        }
    }
    cout<< "Tommy’s score is: " <<
        tommy_score << endl;
}```
Index and Data Relationships

- Arrays and vectors are indexed with integers 0...N-1 and have no relation to the data
- Could we somehow index our data with a meaningful "keys" — studs["Tommy"].get_score(2)
- YES!!! Associative Containers

#include <vector>
#include "student.h"
using namespace std;

int main()
{
    vector<student> studs;
    ...

    unsigned int i;

    // compute average of 0-th score
    double avg = 0;
    for(i=0; i < studs.size(); i++){
        avg += studs[i].get_grade(0);
    }
    avg = avg / studs.size();

    // check "Tommy"'s score
    int tommy_score = -1;
    for(i=0; i < studs.size(); i++){
        if(studs[i].get_name() == "Tommy"{    
            tommy_score = studs[i].get_grade(2);    
            break;
        }
    }
    cout<< "Tommy’s score is: “ <<
    tommy_score << endl;
}
Maps / Dictionaries

• Stores key,value pairs
  – Example: Map student names to their GPA
• Keys must be unique (can only occur once in the structure)
• No constraints on the values
• No inherent ordering between key,value pairs
  – Can't ask for the 0th item...
• Operations:
  – Insert
  – Remove
  – Find/Lookup

Grade Inflation in the Ivy League!!
C++ Pair Struct/Class

- C++ library defines a struct ‘pair’ that is templatized to hold two values (first and second) of different types
  - Templates (more in a few weeks) allow types to be specified differently for each map that is created
- C++ map class internally stores its key/values in these pair objects
- Defined in ‘utility’ header but if you #include <map> you don't have to include utility
- Can declare a pair as seen in option 1 or call library function make_pair() to do it

```cpp
template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;
};

#include <iostream>
#include <utility>
#include <string>
using namespace std;

void func_with_pair_arg(pair<char,double> p)
{
    cout << p.first << " " << p.second << endl;
}

int main()
{
    string mystr = "Bill";
    pair<string, int> p1(mystr, 1);
    cout << p1.first << " " << p1.second << endl;

    // Option 1: Anonymous pair constructed and passed
    func_with_pair_arg( pair<char, double>('c', 2.3));

    // Option 2: Same thing as above but w/ less typing
    func_with_pair_arg( make_pair('c', 2.3));
}
```

Bill 1
c 2.3
c 2.3
Associative Containers

- C++ STL ‘map’ class can be used for this purpose
- Maps store (key,value) pairs where:
  - key = index/label to access the associated value
  - Stored value is a copy of actual data
- Other languages refer to these as ‘hashes’ or ‘dictionaries’
- Keys must be unique
  - Just as indexes were unique in an array or list
- Value type should have a default constructor [i.e. Student()]
- Key type must have less-than (<) operator defined for it
  - Use C++ string rather than char array
- Efficient at finding specified key/value and testing membership (O(\log_2 n))

#include <map>
#include "student.h"
using namespace std;

int main()
{
    map<string,Student> stumap;
    Student s1("Tommy",86328);
    Student s2("Tina",54982);
    ...
    // Option 1: this will insert the pair:
    //   {Tommy,Copy of s1}
    stumap[ "Tommy" ] = s1;
    // Option 2: using insert()
    stumap.insert( pair<string,Student>("Tina", s2));
    // or stumap.insert( make_pair("Tina", s2));
    ...
    int tommy_score = stumap["Tina"].get_grade(1);
    // Returns 'Copy of s2' and then you can call Student member functions
    stumap.erase( "Tommy" );
    cout << "Tommy dropped the course..Erased!";  
    cout << endl;
}

stumap is a map that associates C++ strings (keys) with Student objects (values)
Maps & Iterators

- If you still want to process each element in a map you can still iterate over all elements in the map using an iterator object for that map type
- Recall: Iterator is a "pointer'/iterator to a pair struct
  - it->first is the key
  - it->second is the value

```cpp
#include <map>
#include "student.h"
using namespace std;

int main()
{
  map<string,student> stumap;
  Student s1("Tommy",86328);
  Student s2("Jill",54982);
  ...
  stumap["Tommy"] = s1;
  stumap[s1.get_name()].add_grade(85);
  ...
  stumap["Jill"] = s2
  stumap["Jill"].add_grade(93);
  ...
  map<string,student>::iterator it;
  for(it = stumap.begin(); it != stumap.end(); ++it){
    cout << "Name/key is " << it->first;
    cout << " and their 0th score is ";
    cout << it->second.get_grade(0);
  }
}
```

Name/key is Tommy and their 0th score is 85
Name/key is Jill and their 0th score is 93
Map Membership [Find()]

- Check/search whether key is in the map object using `find()` function
- Pass a key as an argument
- Find returns an iterator
- If key is IN the map
  - Returns an iterator/pointer to that (key,value) pair
- If key is NOT IN the map
  - Returns an iterator equal to `end()`’s return value
- Runs in log(n) time
  - Do not loop/iterate to find something in a map (or set)

```cpp
#include <map>
#include "student.h"
using namespace std;

int main()
{
    map<string,student> stumap;
    Student s1("Tommy",86328), s2("Jill",14259);
    string name;

    stumap["Tommy"] = s1;    // Insert an item
    stumap["Tommy"].add_grade(85);  // Access it

    if(stumap.find("Jill") != stumap.end() ){
        cout << "Jill exists!" << endl;
    }
    else {
        cout << "Jill does not exist" << endl;
        stumap["Jill"] = s2;  // So now add him
    }
    cin >> name;
    map<string,student>::iterator it = stumap.find(name);
    if( it != stumap.end() ){
        cout << it->first << " got score=" << it->second.get_grade(0) << endl;
    }
}
```
Another User of Maps: Sparse Arrays

- Sparse Array: One where there is a large range of possible indices but only small fraction will be used (e.g. are non-zero, etc.)

- Example 1: Using student ID’s to represent students in a course (large 10-digit range, but only 30-40 used)

- Example 2: Count occurrences of zip codes in a user database
  - Option 1: Declare an array of 100,000 elements (00000-99999)
    - Wasteful!!
  - Option 2: Use a map
    - Key = zipcode, Value = occurrences
Set Class

- C++ STL "set" class is like a list but each value **can appear just once**
- Think of it as a map that stores just keys (no associated value)
- **Keys are unique**
- **insert()** to add a key to the set
- **erase()** to remove a key from the set
- Very efficient at testing membership \( O(\log_2 n) \)
  - Is a specific key in the set or not!
- Key type must have a less-than (<) operator defined for it
  - Use C++ string rather than char array
- Iterators to iterate over all elements in the set
- **find()** to test membership

```cpp
#include <set>
#include <string>
using namespace std;

int main()
{
    set<string> people;

    people.insert("Tommy");
    people.insert("Johnny");
    string myname = "Jill";
    people.insert(myname);

    for(set<string>::iterator it=people.begin();
        it != people.end();
        ++it){
        cout << "Person: " << *it << endl;
    }

    myname = "Tommy";
    if(people.find(myname) != people.end()){
        cout<< "Tommy is a CS-related major!" << endl;
    }
    else {
        cout<< "Tommy wants to change his major!" << endl;
    }
    people.erase("Johnny"); // erase Johnny
    // more code
    return 0;
}```
Iterator Pro Tip 2a

• You should **NOT MODIFY** a container (vector, map, set) as you iterate through it
  – May allow iterator to go off the end of a container

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

int main()
{
    vector<int> s;
    ...
    for(vector<int>::iterator it = s.begin(); it != s.end(); ++it)
    {
        if(*it == 0) s.erase(it);  // WRONG! Do NOT modify the container as you iterate through it
    }
    return 0;
}
```
• If you must modify the container as you iterate through take time to understand how the iterator works and research correct methods
  – For vectors, use the `std::remove` and `std::erase` idiom ([Wikipedia link](https://en.wikipedia.org/wiki/Remove_if))
  – Though not efficient, one could iterate through a copy while erasing from the original

```cpp
vector<int>::iterator it = s.begin();
while(it != s.end()){
    if(*it == 0) s.erase(it);
    else ++it;
}
```

```cpp
vector<int>::iterator it;
it = std::remove(s.begin(), s.end(), 0);
s.erase(it, s.end());
```
A Deeper Look: Binary Tree

- Data structure where each node has at most 2 children (no loops/cycles in the graph) and at most one parent
- Tree nodes w/o children are called "leaf" nodes
- Depth of binary tree storing N elements? ______________
A Deeper Look: Binary Tree

- Data structure where each node has at most 2 children (no loops/cycles in the graph) and at most one parent
- Tree nodes w/o children are called "leaf" nodes
- Depth of binary tree storing N elements? $\log_2 n$
Binary Search Tree

- Tree where all nodes meet the property that:
  - All descendants on the left are less than the parent’s value
  - All descendants on the right are greater than the parent’s value
- Can find value (or determine it doesn’t exit) in log2n time by doing binary search

Q: is 19 in the tree?
- 19 < 25..left
- 19 > 18..right

Q: is 34 in the tree?
- 34 > 25..right
- 34 < 47..left
- 34 > 32..right
- 34 != 45 and we’re at a leaf node...34 does not exist
Trees & Maps/Sets

- Maps and sets use binary trees internally to store the keys.
- This allows logarithmic find/membership test time.
- This is why the less-than (<) operator needs to be defined for the data type of the key.

```
Map::find("Greg")
Returns iterator to corresponding pair<string, Student>
```

```
Map::find("Mark")
Returns iterator to end() [i.e. NULL]
```

```
"Frank" Student object
"Jordan" Student object
"Percy" Student object
"Anne" Student object
"Greg" Student object
"Tommy" Student object
```