CIT 5950 Recitation 0

Intro to C++

Introductions

Felix Yuzhou Sun

- 2nd Year MCIT student
- TA for 5920 last semester
- Love K-pop and cooking

CV - Chandravaran Kunjeti

- 2nd Year Robotics student
- TA for 5950 last semester
- Anime

Introductions

Sean Chuang

- CIT 5910 TA last semester
- Tend to have very early OH……
- I swim and use the sauna at Pottruck every day

Ice breaker

Break up into groups of ~10

Here are some questions to help you guys get to know each other…

- What's your favorite food
- What would you do with your life if you didn't have to worry about salary?
- If you were sent to a deserted island and could only bring three movies, what would they be?

Logistics

- HW0 Due in a week **Feb 1st** @ 11:59 pm
  - Don’t forget to hand in your assignment on Gradescope
  - If you need extension, please post private post on Ed
- Pre-semester survey due January 31st @ 11:59 pm
- HW1 to be released soon afterwards

Recitation

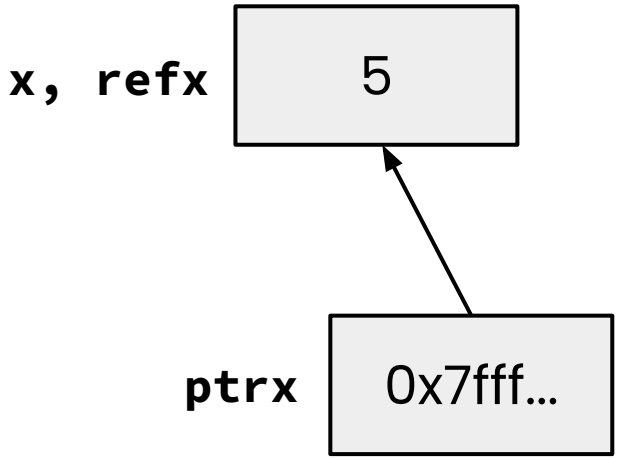
- Const & reference exercise
- STL
- optional
- Docker Setup Help

Const and References

Example

- Consider the following code:

```
int x = 5;
int &refx = x;
```



What are some tradeoffs to using pointers vs references?

Pointers Versus References

Pointers

Can move to different data via reassignment/pointer arithmetic

Can be initialized to **`NULL`**

Useful for output parameters:
`MyClass* output`

References

References the same data for its entire lifetime - *can't reassign*

No sensible "default reference," must be an alias

Useful for input parameters:
`const MyClass& input`

Pointers, References, and Parameters

- When would you prefer:
  - `void func(int &arg)` vs. `void func(int *arg)`
- Use references when you don't want to deal with pointer semantics
  - Allows real pass-by-reference
  - Can make intentions clearer in some cases
- Style wise, we want to use **references** for **input** parameters and **pointers** for **output** parameters, with the output parameters declared last
  - Note: A reference can't be NULL

References and Parameters

- When would you prefer:
  - `void func(int &arg)` vs. `void func(int arg)`
- Use void func(int arg) if the function doesn't need to modify the input or if you want to ensure that the original value remains unchanged.
- Choose reference (void func(int &arg) if the function must modify the original value.
- Style wise, we want to use **references** for **input** parameters for improved readability and consistency. It maintains a uniform coding style across functions.
  - Note: A reference can’t be NULL

Const

- Mark a variable with const to make a compile time check that a variable is never reassigned
- Does not change the underlying write-permissions for this variable

```
int x = 42;

// Read only
const int &ro_ref = x;

// Writable Reference
int& ref = x;


// Can still modify x with ref!
ref +=3;

ro_ref += 2; // does not compile
```

Exercise 1

```
int x = 5;
int &refx = x;
const int &ro_refx = x;
```

Exercise 1

```
void foo(const int &arg);
void bar(int &arg);
void baz(int arg);

int x = 5;
int &refx = x;
const int &ro_refx = x;
```

x, refx
ro_refx

5

Which result in a compiler error?

✔ OK ❌ ERROR

- ✔ `bar(refx);`
- ❌ `bar(ro_refx);` ro_refx is const
- ✔ `foo(refx);`
- ✔ `int y = ro_refx;`
- ❌ `int& other_ref = ro_refx;` ro_refx is const
- ✔ `const int z = x`
- ✔ `baz(ro_refx);` ok since we pass a copy

Exercise 2

this function attempts to modify a string so that it is all capital letters.

```
void all_caps(string to_capitalize);

int main() {
  string name {"mf doom"};
  all_caps(name);
  cout << name << endl;
  // should print out "MF DOOM"
}
```

Exercise 2

to help implement this function, we use a function from the C standard library:

- `toupper()` takes in a character and returns the uppercase version. If it is not a lowercase letter, it returns the same character that was passed in

There are two issues that make this code output the wrong answer, what are they?

```
void all_caps(string to_capitalize) {
  for (auto c : to_capitalize) {
    c = toupper(c);
  }
}
```

Exercise 2

to help implement this function, we use a function from the C standard library:

- `toupper()` takes in a character and returns the uppercase version. If it is not a lowercase letter, it returns the same character that was passed in

There are two issues that make this code output the wrong answer, what are they?

```
void all_caps(string& to_capitalize) {
  for (auto& c : to_capitalize) {
    c = toupper(c);
  }
}
```

Explanation: Pass-By-Value vs Pass-By-Reference

1. The function all_caps takes its argument to_capitalize by value, meaning it works on a copy of the string passed to it, not the original string. As a result, the modifications made inside the function do not affect the original string in main. To fix this, we should pass the string by reference: void all_caps(string& to_capitalize)

2. In the loop for (auto c : to_capitalize), c is a copy of each character in the string, not a reference to it. Modifying c does not change the original string. To fix this, we should iterate over references to the characters in the string: for (auto& c : to_capitalize).

C++ STL

C++ standard lib is built around templates

- *Containers* store data using various underlying data structures
  - The specifics of the data structures define properties and operations for the container
- *Iterators* allow you to traverse container data
  - Iterators form the common interface to containers
  - Different flavors based on underlying data structure
- *Algorithms* perform common, useful operations on containers
  - Use the common interface of iterators, but different algorithms require different ‘complexities’ of iterators

Common C++ STL Containers (and Java equiv)

- *Sequence* containers can be accessed sequentially
  - **`vector<Item>`** uses a dynamically-sized contiguous array (like `ArrayList`)
  - **`list<Item>`** uses a doubly-linked list (like `LinkedList`)
- *Associative* containers use search trees and are sorted by keys
  - **`set<Key>`** only stores keys (like `TreeSet`)
  - **`map<Key,Value>`** stores key-value `pair<>`'s (like `TreeMap`)
- *Unordered associative* containers are hashed
  - **`unordered_map<Key,Value>`** (like `HashMap`)

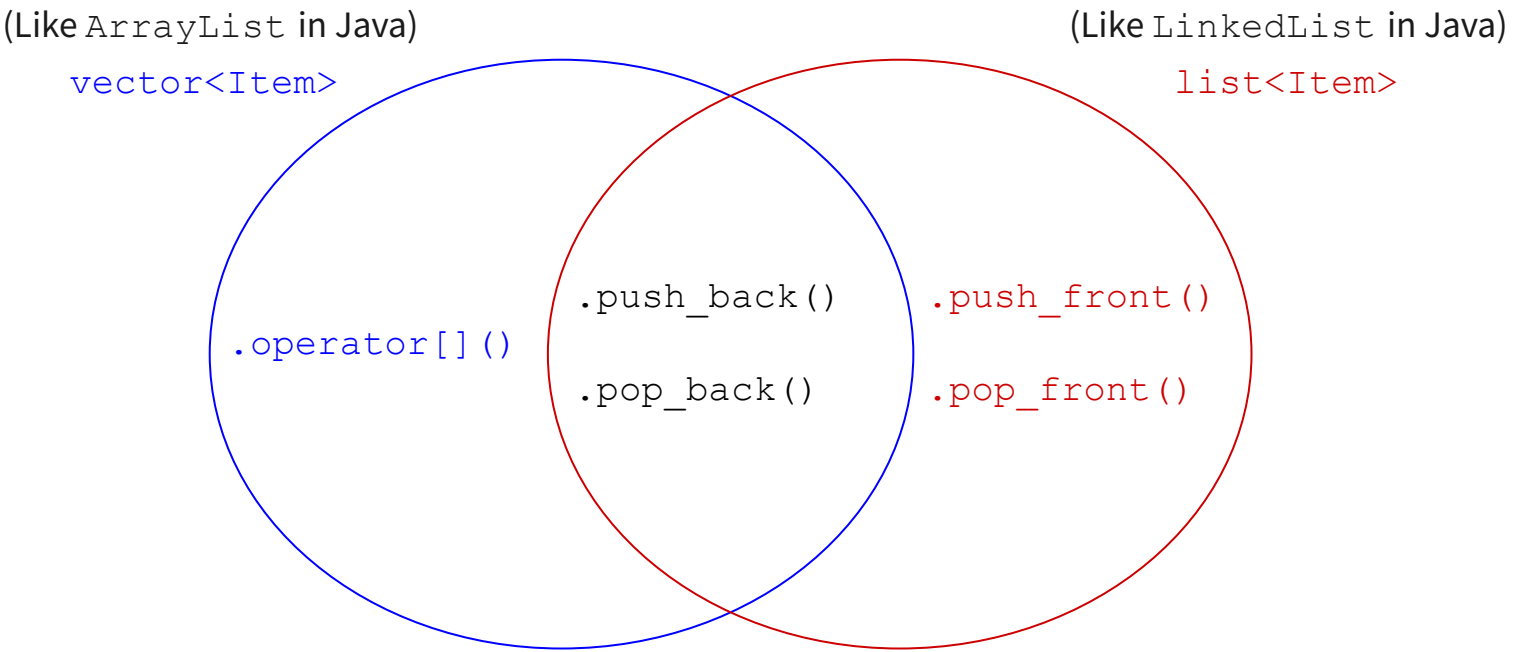
Common C++ STL Methods

	vector	list	set	map	unordered_map
`.size()` // get number of elements					
`.push_back()` // add element to back `.pop_back()` // remove back element					
`.push_front()` // add element to front `.pop_front()` // remove front element					
`.operator[]()` // random access element					
`.insert()` // insert key					
`.find()` // find key					

Common C++ STL Methods

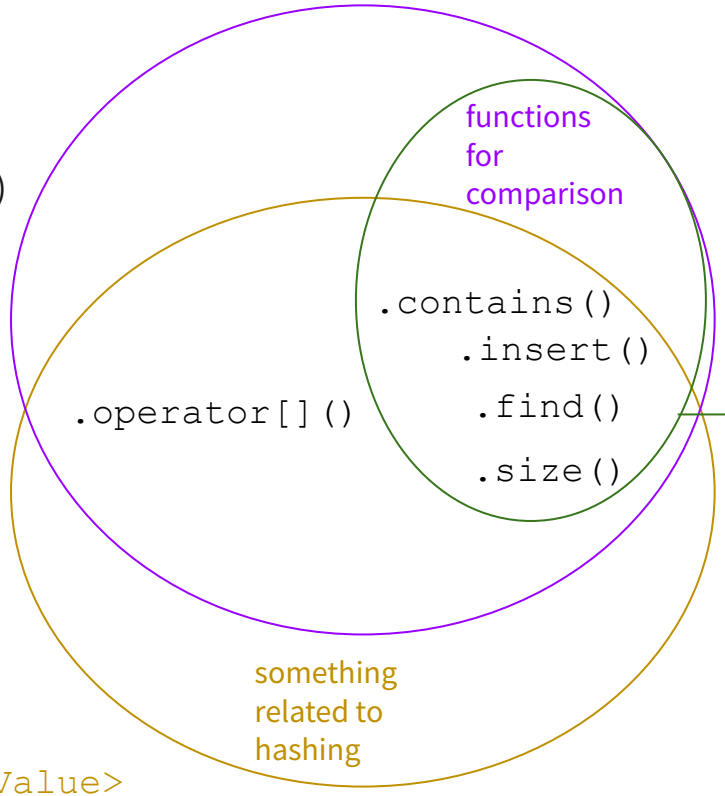
	vector	list	set	map	unordered_map
`.size()` // get number of elements	✔	✔	✔	✔	✔
`.push_back()` // add element to back `.pop_back()` // remove back element	✔	✔			
`.push_front()` // add element to front `.pop_front()` // remove front element		✔			
`.operator[]()` // random access element	✔			✔	✔
`.insert()` // insert key			✔	✔	✔
`.find()` // find key			✔	✔	✔

Common STL Containers (Sequence)



Common STL Containers (Associative)

`map<Key, Value>`
(Like `TreeMap` in Java)



`set<Item>`
(Like `TreeSet` in Java)

(Like `HashMap` in Java)
`unordered_map<Key, Value>`

Common STL Containers

Many more containers and methods!

See full documentation here:
http://www.cplusplus.com/reference/stl

Common STL Data Structures

- vector<Item> (Resizable array, like ArrayList in Java)
  - `.operator[]()` (Gets an element from the vector at a specific index)
  - `.push_back()` (Adds a new element at the end of the vector)
  - `.pop_back()` (Removes the last element in the vector)
- set<Item> (An unindexed collection of items, like Set in Java)
  - `.find()` (Searches the container for an element, returns an iterator)
  - `.insert()` (Inserts a new item into the set)
  - `.size()` (Returns the size of the set)

Common STL Data Structures

- map<Key, Value> (Store key value pairs, like TreeMap in Java)
  - `.operator[]()` (Gets a value associated with a given key. Can also be used to insert a key value pair if the given key does not exist in the map)
  - `.find()` (Searches the map for an element with the key, returns an iterator)
  - `.insert()` (Inserts a new key value pair into the map)
- unordered_map<Key, Value> (Store key value pairs, like HashMap in Java)
  - Supports mostly same operations as map does, usually faster than map

And a lot more! See full documentation here:

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

Now what's that 'std::less'? *// Out of scope*

```
std::less<T>(const T& lhs, const T& rhs) {
    return lhs < rhs;
}
```

- Much like in Java, some structures require ordering elements
  - E.g. set is implemented as a binary tree
- Want to let users store custom types.
  - Java uses Comparable, C++ uses operator< (in std::less)
- However, maybe you want to use a different ordering
  - Ordering is templated function so you can substitute
  - E.g. set<int, std::greater<int>> or set<int, myIntCompare>

Exercise 3 - STL Methods

Exercise 3: STL Methods

Complete the function `ChangeWords` that:

- Takes in a vector of strings, and a map of `<string, string>` key-value pairs
- Returns a new `vector<string>`, where every string in the original vector is replaced by its corresponding value in the map
- Example: if vector `words` is `{"the", "secret", "number", "is", "xlii"}` and map `subs` is `{{"secret", "magic"}, {"xlii", "42"}}`, then `ChangeWords(words, subs)` should return a new vector `{"the", "magic", "number", "is", "42"}`.

```
using namespace std;

vector<string> ChangeWords(const
vector<string> &words,
map<string,string> &subs) {

   #TODO: fill in the method



}
```

Exercise 3 Solution

```
using namespace std;
vector<string> ChangeWords(const vector<string> &words,
                           map<string, string> &subs) {
  vector<string> result;
  for (auto &word : words) {
    if (subs.find(word) != subs.end()) {
      result.push_back(subs[word]);
    } else {
      result.push_back(word);
    }
  }
  return result;
}
```

Exercise 4: optional

std::optional

❖ optional<T> is a struct that can either:

▪ Have some value

T(optional<string> {"Hello!"})

▪ Have nothing

(nullopt)

Exercise 4

We usually use the [] syntax to access a value in a map. However, this does not work elegantly to handle the case when the specified key is not in the map. We instead want to write a helper function to help get a value and distinguish the case when the key is not in the map.

For example: if we have the map ***`values`*** `{ 3: “hello”, 4: “bye”};`,
Then `get(values, 3)` returns “hello” and `get(values, 6)` returns `nullopt`.

```
optional<string> get(map<int, string>& table, int key) {
    // TODO: implement me

}
```

Exercise 4

```
optional<string> get(map<int, string>& table, int key) {
  if (!table.contains(key)) {
    return nullopt;
  }
  return table[key];
}
```

Does anyone need help with their docker container setup?

Bonus Exercise 5

Bonus Exercise - 5

Complete the following function ‘word_positions()’ that takes in a vector of strings and then returns an unordered_map. The map is used to keep track of where each string in the ‘words’ shows up in the vector
For instance:
Words = {“hello”, “hello”, “no”, “monte”, “sano”, “hello”, “sano”};
Would return:
{
“hello”: [0, 1, 5],
“no”: [2],
“monte”: [3],
“sano”: [4, 6]
}

unordered_map<string, vector<size_t>> word_positions(const vector<string>& words) {}

Bonus Exercise - 5

```
unordered_map<string, vector<size_t>> word_positions(const vector<string>& words) {
 unordered_map<string, vector<size_t>> result{};
 for (size_t i {0U}; i < words.length(); i++){
    result[words.at(i)].push_back(i);
  }
   return result;
}
```

Bonus Exercise 6

Bonus Exercise - 6

This implementation is broken, why?

```
unordered_map<string, vector<size_t>> word_positions(const vector<string>& words) {
 unordered_map<string, vector<size_t>> result{};

  for (size_t i {0U}; i < words.length(); i++){

    vector<size_t> current_positions = result[words.at(i)];

    current_positions.push_back(i);

  }

  return result;

}
```

Bonus Exercise - 6

Fixed

```
unordered_map<string, vector<size_t>> word_positions(const vector<string>& words) {
 unordered_map<string, vector<size_t>> result{};

  for (size_t i {0U}; i < words.length(); i++){

    vector<size_t>& current_positions = result[words.at(i)];

    current_positions.push_back(i);

  }

  return result;

}
```