CIT 5950
Recitation 8

Templates, STL and Copy Constructor

Logistics

Mid-semester survey

- On Canvas
- Due March 17th @ 11:59pm

HW3 & Project details

- Will be released later this week

Exercise 1 - Copy Constructors

Copy Constructors

- Copy constructors (cctors) are invoked when we create a new object as a copy of an existing object
  - Point y(x);
  - Point z = y;
  - Void foo(Point x) {...}
  - Point foo() {
    ```
        Point y;
        Return y;
    }
    ```
- If we do not implement our own copy constructor, then the compiler will “synthesize” a default cctor, which uses “shallow copies” and can lead to problems
  - Shallow copy = copy the value of every field, without doing anything more
  - Why is a shallow copy problematic? We will find out in the following exercise!

Exercise 1: Bad Copy

```
class BadCopy {
 public:
  BadCopy()  { arr_ = new int[5]; }
  ~BadCopy() { delete [] arr_; }
 private:
  int *arr_;
};

int main(int argc, char** argv) {
  BadCopy *bc1 = new BadCopy;
  BadCopy *bc2 = new BadCopy(*bc1); // cctor
  delete bc1;
  delete bc2;
  return EXIT_SUCCESS;
}
```

Stack

Heap

Exercise 1: Bad Copy

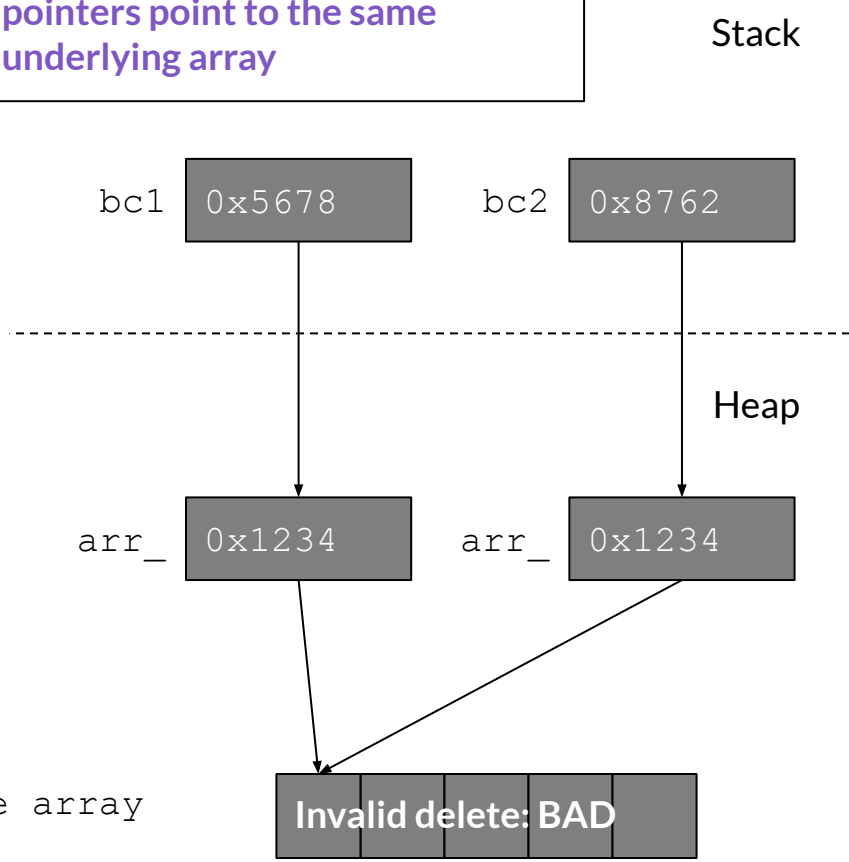
```
class BadCopy {
 public:
  BadCopy()  { arr_ = new int[5]; }
  ~BadCopy() { delete [] arr_; }
 private:
  int *arr_;
};

int main(int argc, char** argv) {
  BadCopy *bc1 = new BadCopy;
  BadCopy *bc2 = new BadCopy(*bc1);
  delete bc1;
  delete bc2;
  return EXIT_SUCCESS;
}
```

Double delete!

Bonus Q: how should we correct the code?

Synthesized cctor copies the arr_ pointer in bc1 to bc2, so the two pointers point to the same underlying array



Templates

Templates

- C++ syntax to generate code that works with *generic types*
- Generates a new implementation in assembly for every type it is used with:
  - *e.g.*, calls to `foo<int>()` and `foo<double>()` generate two implementations
  - *e.g.*, calls to `foo<int>()` and another `foo<int>()` require only one implementation
  - *e.g.*, if `foo` is never used, zero implementations are generated

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);
add3<char*>("a str");
add3<string>("a str");
```

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);              // uses add3<double>, returns 8.5
add3<char*>("a str");
add3<string>("a str");
```

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);              // uses add3<double>, returns 8.5
add3<char*>("a str");   // uses add3<char*>, return ->"tr"
add3<string>("a str");
```

Template Function

```
template<typename T>
T add3(T arg) {
    T result = arg + 3;
    return result;
}
```

Results?

```
add3<int>(3);           // uses add3<int>, returns 6
add3(5.5);              // uses add3<double>, returns 8.5
add3<char*>("a str");   // uses add3<char*>, return ->"tr"
add3<string>("a str"); // Compiler error! No `+` for string
                        //                  and num
```

Template Function

```
template<typename T, int N = 2> // Templatize values
T modulo(T arg) {
    T result = arg % N;
    return result;
}

modulo(5) == 1  (=5%2)
modulo<int, 5>(17) == 2  (=17%5)

// C++ template system is very powerful
// Simple type-substitution is enough for most programs
```

Template Class

- A member variable of a template class can be declared using one of the class' template types
  - Very useful for implementing data structures that support *generic types*:

```
template<typename K, typename V>
struct HTKeyValue {
  K HTKey;
  V* HTValue;
};
```

```
typedef uint64_t HTKey_t;
typedef void*    HTValue_t;
typedef struct {
  HTKey_t   key;
  HTValue_t value;
} HTKeyValue_t;
```

Exercise 2 - Templates

Exercise 2 Solution

```
_______________                 // template type definition
struct Node {
  _______________               // two-argument constructor

  ~Node() { delete value; }     // destructor cleans up the payload

  _______________               // public field value
  _______________               // public field next
};
```

Exercise 2 Solution

```
template <typename T>          // template type definition
struct Node {
  ______________              // two-argument constructor

  ~Node() { delete value; }   // destructor cleans up the payload

  ______________              // public field value
  ______________              // public field next
};
```

Exercise 2 Solution

```
template <typename T>          // template type definition
struct Node {
  ______________              // two-argument constructor

  ~Node() { delete value; }   // destructor cleans up the payload

  T* value                    // public field value
  Node<T>* next               // public field next
};
```

Exercise 2 Solution

```
template <typename T>          // template type definition
struct Node {
  Node(T* val, Node<T>* node): value(val), next(node) {}
                              // two-argument constructor

  ~Node() { delete value; }   // destructor cleans up the payload

  T* value                    // public field value
  Node<T>* next               // public field next
};
```

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`)

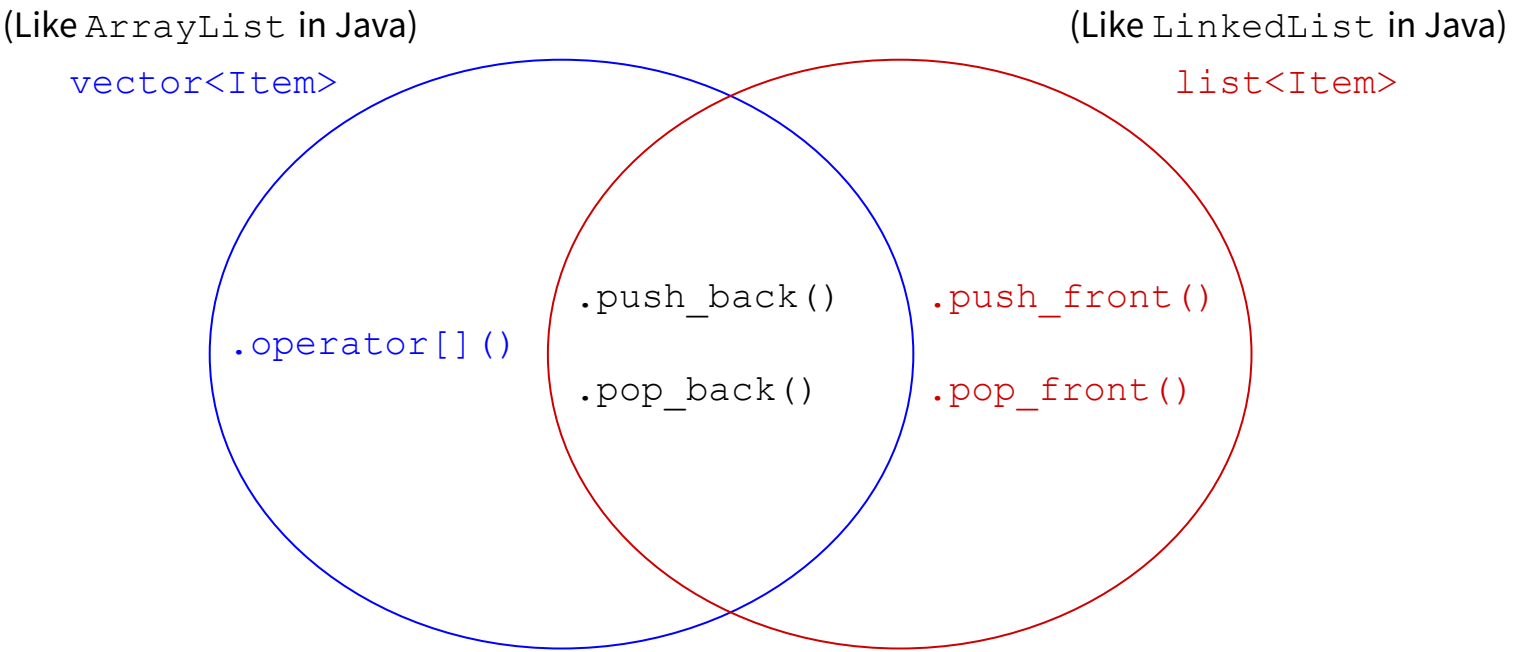
Common C++ STL Methods

	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					

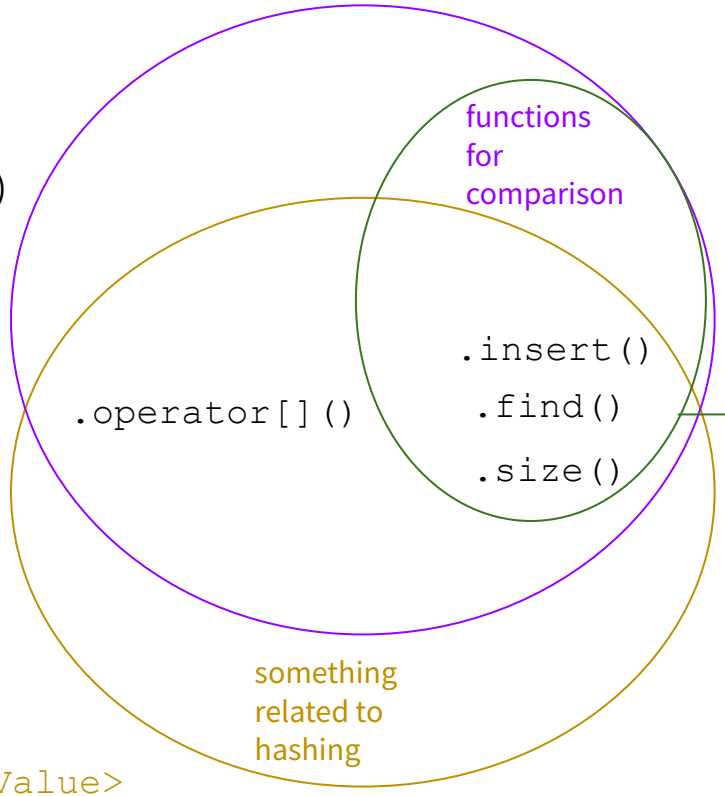
Common C++ STL Methods

	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 - STL Debugging

Here is a little program that has a small class Thing and main function
(assume that necessary #includes and using namespace std; are included).

```
class Thing {
 public:
  Thing(int n): n_(n) { }
  int getThing() const { return n_; }
  void setThing(int n) { n_ = n; }
 private:
  int n_;
};
```

```
int main() {
  Thing t(17);
  vector<Thing> v;
  v.push_back(t);
}
```

This code compiled and worked as expected, but then we added the following two lines of code (plus the appropriate #include <set>):

```
set<Thing> s;
s.insert(t);
```

The second line (s.insert(t)) failed to compile and produced dozens of spectacular compiler error messages, all of which looked more-or-less like this (edited to save space):

In file included from string:48:0, from bits/locale_classes.h:40, from bits/ios_base.h:41,from ios:42,from ostream:38, from /iostream:39,from thing.cc:3: bits/stl_function.h: In instantiation of 'bool std::less<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = Thing]': <<*many similar lines omitted*> > thing.cc:37:13: required from here bits/stl_function.h: 387:20: error: no match for 'operator<' (operand types are 'const Thing' and 'const Thing') { return __x < __y; }

What on earth is wrong? Somehow class Thing doesn't work with set<Thing> even though insert is the correct function to use here. (a) What is the most likely reason, and (b) what would be needed to fix the problem? (Be brief but precise – you don't need to write code in your answer, but you can if that helps make your explanation clear.)

Here is a little program that has a small class Thing and main function
(assume that necessary #includes and using namespace std; are included).

```
class Thing {
 public:
  Thing(int n): n_(n) { }
  int getThing() const { return n_; }
  void setThing(int n) { n_ = n; }
 private:
  int n_;
};
```

```
int main() {
  Thing t(17);
  vector<Thing> v;
  v.push_back(t);
}
```

The second line (s.insert(t)) failed to compile and produced dozens of spectacular compiler error messages, all of which looked more-or-less like this (edited to save space):

In file included from string:48:0, from bits/locale_classes.h:40, from bits/ios_base.h:41,from ios:42,from ostream:38, from /iostream:39,from thing.cc:3: bits/stl_function.h: In instantiation of 'bool std::less<_Tp>::operator()(const _Tp&, const _Tp&) const [with _Tp = Thing]': *<<many similar lines omitted>* > thing.cc:37:13: required from here bits/stl_function.h:
387:20: error: no match for 'operator<' (operand types are 'const Thing' and 'const Thing') { return __x < __y; }

What on earth is wrong? Somehow class Thing doesn't work with set<Thing> even though insert is the correct function to use here. (a) What is the most likely reason, and (b) what would be needed to fix the problem? (Be brief but precise – you don't need to write code in your answer, but you can if that helps make your explanation clear.)

STL has to compare them using operator<. Add an appropriate operator< as either a member function in Thing, or as a free-standing function that compares two Thing& parameters.