

CSE 333

Section 6

Templates, STL and Smart Pointers

Administrivia

Mid-quarter survey

Due Tonight 11:59pm

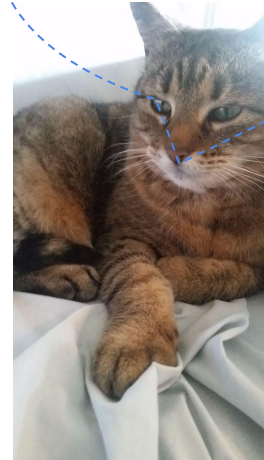
Midterm

Opens Friday 12:00am, due Friday 11:59pm

Exercise 11

Due Monday @ 10 AM

John's cat Caty
wishes everyone



GOOD LUCK ON THE MIDTERM!

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 1

Exercise 1 Solution

```
-----  
struct Node {  
    -----  
    ~Node() { delete value; } // destructor cleans up the payload  
  
    -----  
    -----  
};  
// template type definition  
// two-argument constructor  
// public field value  
// public field next
```

Exercise 1 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 1 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 1 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++ 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
<code>.size()</code> // get number of elements					
<code>.push_back()</code> // add element to back <code>.pop_back()</code> // remove back element					
<code>.push_front()</code> // add element to front <code>.pop_front()</code> // remove front element					
<code>.operator[]()</code> // random access element					
<code>.insert()</code> // insert key					
<code>.find()</code> // find key					

Common C++ STL Methods

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

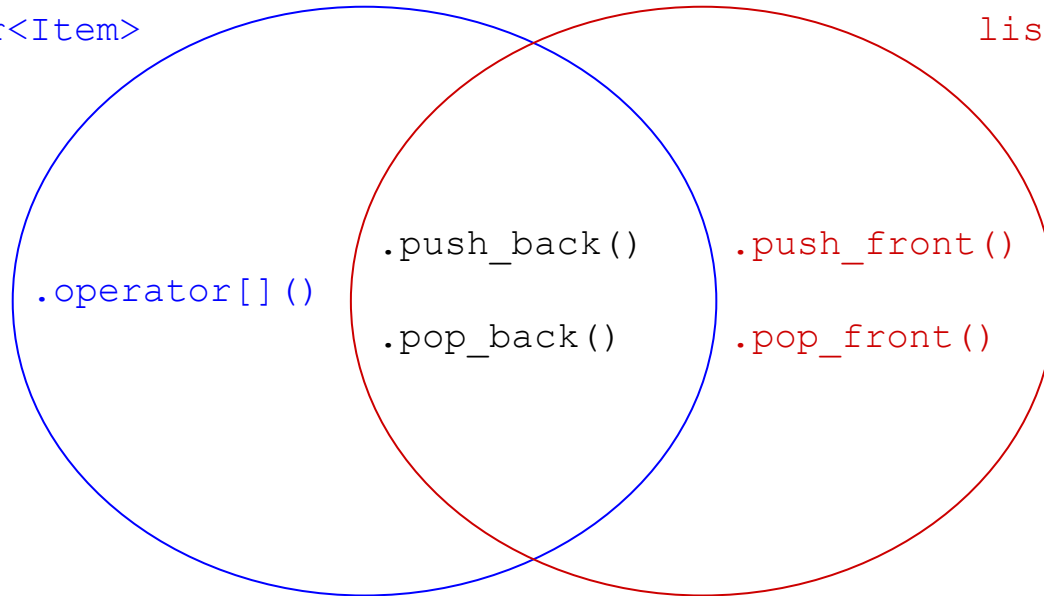
Common STL Containers (Sequence)

(Like ArrayList in Java)

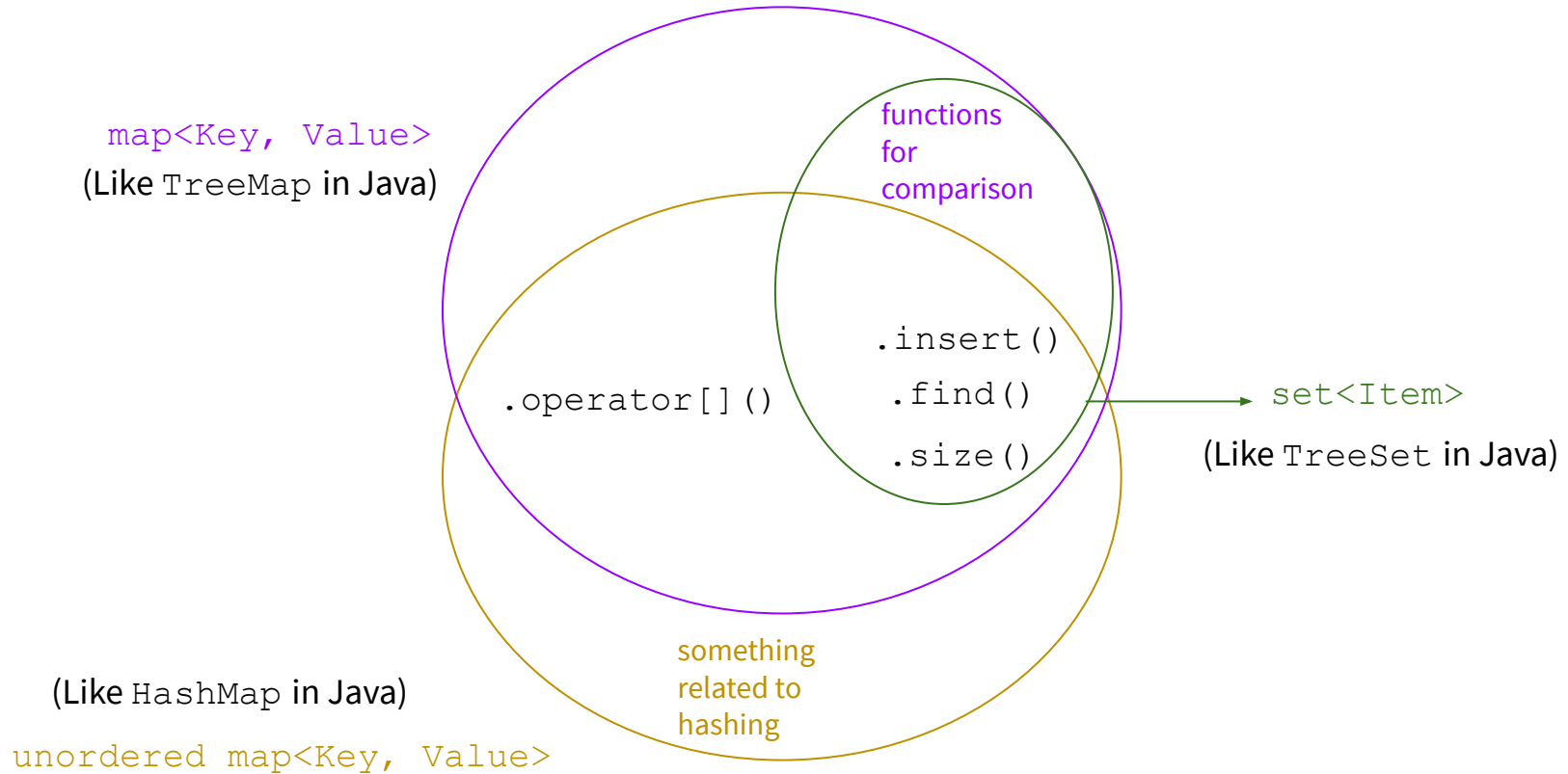
`vector<Item>`

(Like LinkedList in Java)

`list<Item>`



Common STL Containers (Associative)



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 2

Exercise 2 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;
}
```

Smart Pointers!

Smart Pointers

- `std::unique_ptr`
 - Unique owner of the managed raw pointer – disabled cctor and op=
 - Used when you want to declare unique ownership of a pointer
- `std::shared_ptr`
 - Similar to `unique_ptr` but can be copied (via cctor or op=), uses reference counting to decide when to call delete on managed raw pointer
 - **Most commonly used type of smart pointer in practice**
- `std::weak_ptr`
 - Similar to `shared_ptr` but does not contribute to reference count
 - Almost always used with `shared_ptr`

Smart Pointer Usage

- Main/typical usage:
 - Call ctor with `new` keyword or existing smart pointer
(e.g., `unique_ptr<int[]> uptr(new int[3])`)
 - Treat like a normal pointer (i.e., use `*`, `->`, `[]`)
- Other methods that may be useful in *some* cases:
 - `unique_ptr` - `.get()`, `.release()`, `.reset()`
 - `shared_ptr` - `.get()`, `.use_count()`, `.unique()`
 - `weak_ptr` - `.lock()`, `.use_count()`,
`.expired()`

Smart Pointers Operations

- `std::unique_ptr` - `.get()`, `.release()`, `.reset()`
- `std::shared_ptr` - `.get()`, `.use_count()`, `.unique()`
- `std::weak_ptr` - `.lock()`, `.use_count()`, `.expired()`

Takeaway:

- These operations exist and can be useful in *some* cases. But most of the time in practice, you would use smart pointers just as if they were normal pointers!

Exercise 4

Exercise 4 Solution

```
#include <memory>
using std::shared_ptr;

template <typename T>
struct Node {
    Node(T* val, Node<T>* node): value(val), next(node) {}

    ~Node() { delete value; }

    T* value;
    Node<T>* next;
};
```

Exercise 4 Solution

```
#include <memory>
using std::shared_ptr;

template <typename T>
struct Node {
    Node(T* val, Node<T>* node): value(val), next(node) {}

    ~Node() { delete value; }

    shared_ptr<T> value;
    shared_ptr<Node<T>> next;
};
```

Exercise 4 Solution

```
#include <memory>
using std::shared_ptr;

template <typename T>
struct Node {
    Node(T* val, Node<T>* node): value(shared_ptr<T>(val)),
                                next(shared_ptr<Node<T>>(node)) {}

    ~Node() { delete value; }

    shared_ptr<T> value;
    shared_ptr<Node<T>> next;
};
```

Exercise 4 Solution

```
#include <memory>
using std::shared_ptr;

template <typename T>
struct Node {
    Node(T* val, Node<T>* node): value(shared_ptr<T>(val)),
                                next(shared_ptr<Node<T>>(node)) {}

    ~Node() { delete value; }

    shared_ptr<T> value;
    shared_ptr<Node<T>> next;
};
```

Exercise 4 Solution

```
#include <iostream>
```

```
using std::cout;
```

```
using std::endl;
```

```
int main() {
```

```
    shared_ptr<Node<int>> head =
```

```
        shared_ptr<Node<int>>(new Node<int>(new int(351), nullptr));
```

```
    head->next = shared_ptr<Node<int>>(new Node<int>(new int(333), nullptr));
```

```
    shared_ptr<Node<int>> iter = head;
```

```
    while (iter != nullptr) {
```

```
        cout << *(iter->value) << endl;
```

```
        iter = iter->next;
```

```
    }
```

```
}
```

