STL

Intermediate Computer Systems Programming, Fall 2025

Instructor: Travis McGaha

TAs: Theodor Bulacovschi Ash Fujiyama

Poll Everywhere

pollev.com/tqm

- How are you? Any feedback?

Administrivia

- HW02 Due tomorrow
  - I *think* it will be less word than HW01
  - Autograder posted later in the day or tomorrow
- HW03 posted on Wednesday
  - Algorithmically more complex than HW02
  - But you have the C++ standard library
  - I *think* it is less work than HW02

Lecture Outline

- **C++ Attributes**
- Memory Diagrams
- STL
  - Map
  - Set
  - Iterators
  - Algorithms
- Templates
  - Arrays

Aside: attributes

- Allows us to annotate our code in ways that may be useful to the compiler or any third-party extensions to the C++ language

- For our purposes, there are three we are most likely to use
  - `[[nodiscard]]`
    - put on the return type of a function decl to mark that the return value should not be discarded
  - `[[noreturn]]`
    - put on the return type of a function decl to mark that this function doesn't return (it always either infinite loops, throws an exception, or exits)
  - `[[maybe_unused]]`
    - put on a parameter/variable to show that the variable may not be used, and that is ok. Usually you just want to delete the variable, but that isn't always feasible.

Lecture Outline

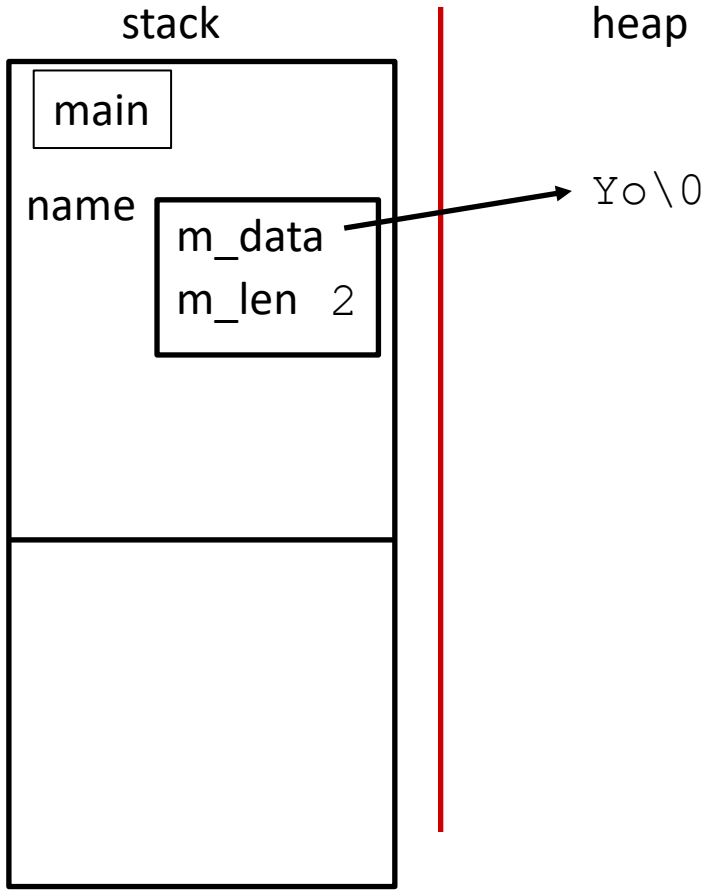
- C++ Attributes
- **Memory Diagrams**
- STL
  - Map
  - Set
  - Iterators
  - Algorithms
- Templates
  - Arrays

Memory Diagrams

- We have built up from C, but it is important to remember that all of these abstractions are built around the same concepts from C
- Everything exists in memory, either in the stack, heap or global memory
- That layout is important for understanding how our programs work, thinking about performance (more on this later) and identifying memory bugs

C++ Memory

- Draw the state of memory when the code reaches // HERE




```
void make_subs(const string& input) {
  vector<string> output{input};
  // assume initial capacity is 1
  // capacity is doubled on resize

  size_t i = 1;
  string& first = output.at(0);

  while (i < input.size()) {
    string sub = input.substr(i);
    output.push_back(sub);
    i += 1;
  }

  // HERE
  return output;
}

int main() {
  string name = "Yo";
  vector<string> subs = make_subs(name, subs);
}
```

Lecture Outline

- C++ Attributes
- Memory Diagrams
- **STL**
  - **Map**
  - **Set**
  - **Iterators**
  - **Algorithms**
- Templates
  - Arrays

STL `unordered_map`

- One of C++'s *associative* containers: a key/value table, implemented as a Chaining Hash Map
  - http://www.cplusplus.com/reference/unordered_map/
  - General form: `unordered_map<key_type, value_type> name;`
  - Keys must be *unique*
    - **`multimap`** allows duplicate keys
  - Efficient lookup (O(1)) and insertion (O(1))
    - Access `value` via **`operator[]`** (example: **`map_name[key]`**)
      - if key doesn't exist in map, it is added to the map with a "default" value
  - Elements are type `pair<key_type, value_type>`
    (key is field **`first`**, value is field **`second`**)
    - Key type must be hashable

unordered_map Example

#include <unordered_map>

```
int main(int argc, char** argv) {
  unordered_map<int,string> table{};
  unordered_map<int,string>::iterator it{};

  table.insert(pair<int,string>(2, "hello"));
  table[4] = "NGNM";
  table[6] = "mutual aid"; // inserts a value
  table[6] = "sleep";  // updates a value
  cout << "table[6]:" << table[6] << endl;
  it = table.find(7);
  if (it != table.end()) {
    it->second += "1"; // update value if it does exist
  }
  if (table.contains(4)) {
    cout << "4 exists as a key in the map" << endl;
  }

  cout << "iterating:" << endl;
  for (auto& p : table) {
    cout << "[" << p.first << "," << p.second << "]" << endl;
  }
  return 0;
}
```

Map elements

Equivalent behavior

use .find() to see if a key exists, gets an iterator to existing pair if it does exist

use .contains() to see if a key exists

Access the key and value stored in the pair

STL `unordered_set`

- One of C++'s *associative* containers: a container of unique values, implemented as a hash set
  - http://www.cplusplus.com/reference/unordered_set/
  - General form: `unordered_set<element_type> name;`
  - elements must be *unique*
    - **`multiset`** allows duplicate elements
  - Efficient lookup (O(1)) and insertion (O(1))
  - Inserting an element that already exists does nothing
  - Can use `contains(element)` to see if the element exists
  - Elements are stored in *unsorted* order
    - element type must be hashable

unordered_set Example

```
#include <unordered_set>

int main(int argc, char** argv) {
  unordered_set<string> names {};

  names.insert("bjarne");
  names.insert("ken");
  names.insert("dennis");
  names.insert("travis");
  names.insert("bjarne");

  bool exists = names.contains("bjarne");
  cout << "Is bjarne in the set?: " << exists << endl;

  numbers.erase("travis");

  for (string& name : names) {
    cout << name << endl;
  }
  return EXIT_SUCCESS;
}
```

Doesn't insert duplicate elements

prints "true"

Removes the element "travis"

Prints every name in the set

Ordered Containers (C++11)

- `map`, `set`
  - Average case for key access is O(log(n)), so generally not preferred
    - But range iterators can be more efficient than unordered `map`/`set`
    - Usually implemented as a self balancing (e.g. red/black) tree.
  - See *C++ Primer*, online references for details

map Example

#include <map>

```
int main(int argc, char** argv) {
  map<int,string> table{};
  map<int,string>::iterator it{};

  table.insert(pair<int,string>(2, "hello"));
  table[4] = "NGNM";
  table[6] = "mutual aid"; // inserts a value
  table[6] = "sleep";  // updates a value
  cout << "table[6]:" << table[6] << endl;
  it = table.find(7);
  if (it != table.end()) {
    it->second += "1"; // update value if it does exist
  }
  if (table.contains(4)) {
    cout << "4 exists as a key in the map" << endl;
  }

  cout << "iterating:" << endl;
  for (auto& p : table) {
    cout << "[" << p.first << "," << p.second << "]" << endl;
  }
  return 0;
}
```

Unordered vs Ordered Containers

- The comparison between `unordered_map` vs `map` is similar to how HashMap vs TreeMap are related in java.
  - Both use the same interface, but have different implementations
  - If you want things to be in sorted order, use `map` (TreeMap)
  - In almost all other cases, use `unordered_map` (HashMap)

- Sorted map requires `bool operator<(const& T, const& T)` for the key so that it can sort values. (set requires this too)
- Unordered requires `operator==` on keys and definition of a struct:

```
template <>
struct std::hash<Thing> {
  size_t operator()(const Thing& to_hash) {
     // TODO: calculate hash
  };
};
```

C++ Iterators

- C++ Containers all support Iterators to navigate the container.
- To get an iterator to the front, you use `.begin()`
  - Or `.cbegin()` if you want a constant iterator
- To get an iterator to "one past the end" you use `.end()` (or `.cend` for const)
- Use ++ to move an iterator forward
- Use * (dereference to get the value iterator is currently at)
- Supports == and != to compare iterators

```
int main() {
  unordered_set<string> names = {"ash", "theodor", "travis"};
  for(auto it = names.begin(); it != names.end(); ++it) {
    cout << *it << endl;
  }
}
```

C++ Iterators

- Different contains also have other operations that they support.
  - List and vector can use `operator--` to move backwards
  - Vector iterators can use other operators (e.g. `<`) to compare iterators
  - Etc.
  - Stick to only the operations you need if you want to try and make your function “generic”

- Iterators are how std::algorithms interface with containers “generically”

Iterator/Reference Invalidation

- Iterators can become invalidated when the container it refers to changes. Usually because the things the iterator pointed to are now in a different location in memory.
  - Same is true for references or pointers that may be referring to an element in the container

- For vector, what operations can invalidate an interator?
  - `push_back()`
  - `reserve()`
  - Anything that may cause a resize

- What about for std::list? (doubly linked list)
  - An iterator is only invalidated when corresponding element is removed/deleted.

STL Algorithms

- A set of functions to be used on ranges of elements
  - Range: any sequence that can be accessed through *iterators* or *pointers*, like arrays or some of the containers
  - General form: `algorithm(begin, end, ...);`

Rest depends on the algo

Takes a range of a sequence to operate on

- Algorithms operate directly on range *elements* rather than the containers they live in
  - Make use of elements' copy ctor, =, ==, !=, <
  - Some do not modify elements
    - *e.g.* `find`, `count`, `for_each`, `min_element`, `binary_search`
  - Some do modify elements
    - *e.g.* `sort`, `transform`, `copy`, `swap`

Appropriate operator(s) must be defined for the element to use an STL algorithm

Algorithms Example

```
#include <vector>
#include <algorithm>
#include "Tracer.h"
using namespace std;

void PrintOut(const string& p) {
  cout << " printout: " << p << endl;
}

int main(int argc, char** argv) {
  vector<string> vec;

  vec.push_back("theo");
  vec.push_back("ash");
  vec.push_back("trav");
  cout << "sort:" << endl;
  sort(vec.begin(), vec.end());
  cout << "done sort!" << endl;
  for_each(vec.begin(), vec.end(), &PrintOut);
  auto it = find(vec.begin(), vec.end(), "ash");
  return 0;
}
```

Not in order ☹

Sort elements from [vec.begin(), vec.end())

Runs function on each element.
In this case, prints out each element

Returns iterator to target if found.
end() if not found

Higher Order Functions

- Common operations have their C++ counterparts
  - Filter -> `copy_if()`
  - map -> `transform()`
  - reduce -> `reduce()`

Aside: Lambdas

- In C++ you can define lambda's or "anonymous functions". commonly used for std::algorithm

```
auto lambda = [](const string& a) { cout << a << endl; };
// to store lambda in a variable use auto
for_each(vec.begin(), vec.end(), lambda);


// can put the lambda inline
// can use -> to declare a lambda with a return value
transform(nums.begin(), nums.end(), nums.begin(),
          [](int num) -> int { return num * 2; });

// can put captures in the square brackets
// a capture is an outside variable accesible from within the lamnda
// use & to capture by reference. Ommitt to capture by value.
int factor = 3;
transform(nums.begin(), nums.end(), nums.begin(),
          [&factor](int num) -> int { return num * factor; });
```

ranges::

- Algorithms that are better named and less clunky to use were added in C++23
  - You will still see the old algorithms used a lot

- Part of <algorithm> header

```
auto it = find(nums.begin(), nums.end(), 5);
```

- becomes

```
auto it = ranges::find(nums, 5);
```

Ed programming

- See Ed ☺

- Note: Ed uses C++17 ☹
  - No ranges::
  - No contains() function on map or sets
    - Use .count() instead
      (given a key, returns the number of occurrences of that key. Should be 0 or 1)

Lecture Outline

- C++ Attributes
- Memory Diagrams
- STL
  - Map
  - Set
  - Iterators
  - Algorithms
- **Templates**
  - **Arrays**

Macros

- In C we had macros
  - Can define constants and “functions”

```
#define ADD(x, y) (x + y)


#define MAGIC 7


int main() {
  int n = MAGIC;
  int m = 8;

  int res = ADD(n, m);
}
```

Steps to compilation

- Compilation goes through multiple steps
  - First the C-Pre-Processor (also called CPP) which resolves any macro definitions
  - AFTERWARDS the compiler gets what CPP outputs
  - Code -> intermediate representation -> assembly -> machine code
- Our #defines are just text replacement that is done before the compiler is run

Macros

- In C we had macros
  - Can define constants and “functions”
  - We see

```c
#define ADD(x, y) (x + y)


#define MAGIC 7


int main() {
  int n = MAGIC;
  int m = 8;

  int res = ADD(n, m);
}
```

- Notably: there is no “ADD” function anywhere
  - Compiler gets:

```c
int main() {
  int n = 7;
  int m = 8;


  int res = (n + m);
}
```

Never use C style macros in C++

- C style macros are really easy to mess up
- What does this print?

```
#define MAX(x, y) (x < y ? y : x)

int main() {
  int n = 3;
  int m = 5;


  int res = MAX(n, ++m);


  // what is res?
}
```

Templates

- Like macros but harder to fuck up
- Still fuck-up-able!!!

- Commonly used to make some object or function type "generic"

```
struct Pair {
  int x;
  int y;
};


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

Can become ->

```
template<typename T>
struct Pair {
  T x;
  T y;
};


template<typename T>
T add(const T& x, const T& y) {
  return x + y;
}
```

Templates in Compilation

- Like macros, C++ templates are “not real code” but instead “template code” templates that the compiler will use to generate code.

- If I were to compile this C++ code, how much assembly is generated?

```hpp
// example.hpp
template<typename T>
struct Pair {
  T x;
  T y;
};


template<typename T>
T add(const T& x, const T& y);
```

```cpp
// example.cpp
#include "./example.hpp"


template<typename T>
T add(const T& x, const T& y) {
  return x + y;
}
```

- No assembly. We get an empty object file

Templates in Compilation

- Like macros, C++ templates are “not real code” but instead “template code” templates that the compiler will use to generate code.
- Put all template code in a .hpp file

```hpp
// example.hpp
template<typename T>
struct Pair {
  T x;
  T y;
};

template<typename T>
T add(const T& x, const T& y) {
  return x + y;
}
```

```cpp
// main.cpp
#include "./example.hpp"


int main() {
  string x = "hi"
  string y = "bye";

  cout << add(x, y) << endl;
  cout << add(5, 12) << endl;
}
```

- In this program, there are two instance of add generated, zero instance of Pair add<string> and add<int>

Templates uh-oh

- What does this print?

```cpp
// example.hpp
template<typename T>
struct Pair {
  T x;
  T y;
};

template<typename T>
T add(const T& x, const T& y) {
  return x + y;
}
```

```cpp
// main.cpp
#include "./example.hpp"


int main() {
  cout << add("hi", "bye") << endl;
  cout << add(5, 12) << endl;
}
```

- Add resolves to type char* not string. What is a char* + char* ?
- Can fix by being explicit `cout << add<string>("hi", "bye") << endl;`

Arrays in C++

- Not just a type T, but size as well!

```
array<string, 3> my_array {"ash", "theodor", "travis"};
```

- If size is a template parameter, then……
- Size must be known at compile time

- Has .at() to index into (while checked)
- Has a .size() function to get the length
- Has .data() to get a pointer to the elements

- The array elements are stored wherever the array is stored. Vector always puts things on the heap.

Later Lecture: Compile Time Computation

- We can force/trick the compiler to do computation at compile time
- Computation done at compile time does not have to be done at run time
- It already does this for you in some cases:
  - int x = 3 + 1074 + 22 * 5
- Can do more explicit & complex things with:
  - Template meta-programming
  - constexpr
  - This will be talked about much later ☺

That's all for now!

- Hopefully you are doing well ☺