COMP6771 Week 2

STL Algorithms

Common mistakes

- Bazel won't sync properly
  - Is bazel version < 0.26?
  - Using the old VM (old VM is running CLion 2018, new one is 2019)?
  - Uninstall bazel, then reinstall the newest version using the custom apt repository method
  - https://docs.bazel.build/versions/master/install-ubuntu.html#install-on-ubuntu
- Debugger not working
  - Settings > build, execution > toolchains > debugger
    - switch bundled to GDB
  - chmod a+x ~/.CLion2019.*/config/plugins/clwb/gdb/gdbserver

Starting on your assignment

- git clone https://github.com/cs6771/comp6771.git
- Open up clion
  - If a project is already open, file > close project
  - File > import bazel project > select comp6771 directory
    - Default settings are fine
  - VCS > enable version control integration
  - Bazel > sync

Profiling

- **If you want your code to go faster, we've replaced std::set with std::unordered_set**
  - vcs > update project to get the changes
  - You may need to run the following if you're using the VM:
    -

- Profiling **can** be handy to work out where to optimise
- Clion has support for profiling (run > profile 'bazel run binary')
  - We don't recommend using it for this assignment except for your own learning
  - We won't show you how to use profiling here
    - You can probably use this (but we haven't tried) https://www.jetbrains.com/help/clion/cpu-profiler.html

Git

- To use git with clion, VCS > enable version control systems (if not enabled)
- VCS > update project to download updates
  - Always select rebase, not merge
  - If there are conflicts, you should get a nice UI to merge changes
- VCS > commit
  - Has a nice UI to see what you're committing
  - When you commit, you can also push
    - Only relevant if you have your own repo (eg. a fork)
- "checkout" (switch between) commits using button in bottom right
- Version control tab
  - Both local changes and log are very useful

Debugging

- See slide 2.1 for common mistakes and how to fix them
- Breakpoints to pause at certain points
  - Right click on breakpoints for fine-grained control
- Look at variables in your debugger window while paused
- Play button = Continue
- Two-step arrow = Run until next line
- Down arrow = Go inside the next function call
- Up arrow = leave the function
- Calculator icon = Type an expression in to evaluate

Principles of testing

- Test API, not implentation
- Don't make tests brittle
  - If your code changes, your tests should change minimally
- Make tests simple
  - It should be obvious what went wrong
  - Don't put if statements or loops in your tests
  - Any complex code should be put in a well-named function

Testing - Build rules

- Works this way no matter what test framework you use
- You can't test anything in a file with a main function
  - Why not?

```
cc_library(
    name = "factorial",
    srcs = ["factorial.cpp"],
    hdrs = ["factorial.h"],
)

cc_test(
    name = "factorial_test",
    srcs = ["factorial_test.cpp"],
    deps = [
        ":factorial",
        "//:catch",
    ],
)
```

Testing in general

- Testing almost always has a form that looks something similar to this
  1. Do some setup (eg. initialise variables)
  2. Run some code that should be tested (call your function to be tested)
  3. Check that things are as expected
- Sometimes these things are hard to distinguish, but usually not

Or in other words

```
SCENARIO("scenario") {
  GIVEN("Some starting condition") {
    // Initialise the variables
    WHEN("My function is called") {
      // Call my function
      THEN("something should have happened") {
        // Check that the thing happened as expected
      }
    }
  }
}
```

Catch2 testing

```
1  // Feel free to remove the string if there's
2  // only one GIVEN in this scenario.
3  SCENARIO("vectors can be sized and resized") {
4    GIVEN("A vector with some items") {
5      std::vector<int> v(5);
6
7      REQUIRE(v.size() == 5);
8      REQUIRE(v.capacity() >= 5);
9
10     WHEN("the size is increased") {
11       v.resize(10);
12
13       THEN("the size and capacity change") {
14         REQUIRE(v.size() == 10);
15         REQUIRE(v.capacity() >= 10);
16       }
17     }
18     WHEN("the size is reduced") {
19       v.resize(0);
20
21       THEN("the size changes but not capacity") {
22         REQUIRE(v.size() == 0);
23         REQUIRE(v.capacity() >= 5);
24       }
25     }
26   }
27 }
```

- A scenario is a named group of tests
- GIVEN, WHEN, and THEN work the exact same way
  - GIVEN should be labelled with the initialisation performed
  - WHEN should be labelled with the code that you ran
  - THEN should be labelled with the expectation you have of the result
  - They just give us really nice errors
- REQUIRE is the thing that actually runs your tests

More advanced Catch2 testing

```
1  const auto hasAbc = Catch::Matchers::Contains(
2      "aBC", Catch::CaseSensitive::No);
3
4  SCENARIO("Do that thing with the thing", "[Tags]") {
5    GIVEN("This stuff exists") {
6      // make stuff exist
7      AND_GIVEN("And some assumption") {
8        // Validate assumption
9        WHEN("I do this") {
10         // do this
11         THEN("it should do this") {
12           REQUIRE(itDoesThis());
13           AND_THEN("do that") {
14             REQUIRE(itDoesThat());
15             REQUIRE_THAT(
16                 getResultOfThat(), hasAbc);
17           }
18         }
19       }
20     }
21   }
22 }
```

- You can chain together GIVE/WHEN/THEN
  - Do it like you would english
  - You can write these before writing code tests
- To run actual tests, use CHECK, CHECK_THAT, REQUIRE, or REQUIRE_THAT
  - Require kills the test if it fails, check keeps on going
  - REQUIRE and CHECK take in a boolean
  - REQUIRE_THAT and CHECK_THAT take a value, and a matcher (https://github.com/catchorg/Catch2/blob/master/docs/matchers.md)

Common algorithms

- What was the writer of this code trying to do?
- Does it do what it should?
- How long does it take you to work that out?
- How easy is it to read?

```
1 std::vector<int> nums;
2
3 int sum = 0;
4 for (int i = 0; i <= nums.size(); ++i) {
5   sum += i;
6 }
```

What about this?

```
1 std::vector<int> nums;
2
3 int sum = 0;
4 for (auto it = nums.begin(); i != nums.end(); ++i) {
5   sum += *i;
6 }
```

C++ range-for loops

```
1 std::vector<int> nums;
2
3 int sum = 0;
4
5 // Internally, this uses begin and end,
6 // but it abstracts it away.
7 for (const auto& i : nums) {
8   sum += i;
9 }
```

Algorithms

- Surely we can write a function that looks like this?
  - We can (but it doesn't quite look like this)
  - But we don't need to (the STL has us covered)

```
1 // What type of iterator is required here?
2 template <typename T, typename Container>
3 T sum(iterator_t<Container> first, iterator_t<Container> las
4   T total;
5   for (; first != last; ++first) {
6     total += *first;
7   }
8   return total
9 }
```

```
1 template <typename T>
2 T sum(iterable<T> cont) {
3   T total;
4   for (auto it = std::begin(cont); std::end(cont); ++
5     total += *it;
6   }
7   return total
8 }
```

Standard Algorithms

- Surely we can write a function that looks like this?
  - Turns out we can (but it doesn't quite look like this)
  - But we don't need to (the STL has us covered)

```
1 std::vector<int> v{1, 2, 3};
2 int sum = std::accumulate(v.begin(), v.end(), 0);
```

Very powerful

What if we want the product instead of the sum?

```
1 // What is the type of std::multiplies<int>()
2 int product = std::accumulate(v.begin(), v.end(), 1, std::multiplies<int>());
```

What if we want to only sum up the first half of the numbers?

```
1 auto midpoint = v.begin() + (v.size() / 2);
2 // This looks a lot harder to read. Why might it be better?
3 auto midpoint = std::next(v.begin(), std::distance(v.begin(), v.end()) / 2);
4
5 int sum = std::accumulate(v.begin(), midpoint, 0);
```

Performance and portability

- Consider:
  - Number of comparisons for binary search on a vector is O(log N)
  - Number of comparisons for binary search on a linked list is O(N log N)
  - The two implementations are completely different
- We can call the same function on both of them
  - It will end up calling a function have two different overloads, one for a forward iterator, and one for a random access iterator
- Trivial to read
- Trivial to change the type of a container

```
1 // Lower bound does a binary search, and returns the first value >= the argument.
2 std::vector<int> sortedVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
3 std::lower_bound(sortedVec.begin(), sortedVec.end(), 5);
4
5 std::list<int> sortedLinkedList{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
6 std::lower_bound(sortedLinkedList.begin(), sortedLinkedList.end(), 5);
```

Algorithms with output sequences

Why doesn't the second one work?

```
 1 char to_upper(char value) {
 2   return std::toupper(static_cast<unsigned char>(value));
 3 }
 4
 5 std::string s = "hello world";
 6 // std::for_each modifies each element
 7 std::for_each(s.begin(), s.end(), to_upper);
 8
 9 std::string upper;
10 // Algorithms like transform, which have output iterators,
11 // use the other iterator as an output.
12 std::transform(s.begin(), s.end(), upper.end(), to_upper);
```

Back inserter

Gives you an output iterator for a container that adds to the end of it

```
 1 char to_upper(char value) {
 2   return std::toupper(value);
 3 }
 4
 5 std::string s = "hello world";
 6 // std::for_each modifies each element
 7 std::for_each(s.begin(), s.end(), toupper);
 8
 9 std::string upper;
10 // std::transform adds to third iterator.
11 std::transform(s.begin(), s.end(), std::back_inserter(upper), to_upp
```

Lambda functions

-
- Can be used with std::function<ReturnType(Arg1, Arg2)> (or auto)
  - It can be used as a parameter or variable
  - No need to use function pointers anymore

```
1 std::string s = "hello world";
2 // std::for_each modifies each element
3 std::for_each(s.begin(), s.end(), [] (char value) { return std::toupper(value); });
```

Lambda captures

- This doesn't compile
- The lambda function can get access to the scope, but does not by default

```
1 void AddN(std::vector<int>& v, int n) {
2   std::for_each(v.begin(), v.end(), [] (int val) { return val + n; });
3 }
```

Lambda captures - By Value

- Copies the value contained when the function was created
  - Doesn't update when the original updates
  - Safe
  - Potentially slow
  - May not work for non-copyable types (eg. ostream, unique pointer)

```
1 // Works great.
2 void AddN(std::vector<int> vec, int n)
3   std::for_each(vec.begin(), vec.end(),
4       [=] (int item) { item += n; });
5 }
```

```
1 // Not so great. Fails to compile.
2 void PrintList(const std::vector<int>& nums,
3                std::ostream& os) {
4   auto printer = [=] (int value) { os << value << '\n'
5   std::for_each(nums.begin(), nums.end(), printer);
6 }
```

```
1 // Even worse. This compiles successfully.
2 std::map<std::string, int> m;
3 auto emplace = [=] (const auto& key, const auto& value) { m.emplace(key, value
4 emplace("hello", 5);
```

Lambda captures - By reference

- Creates a reference to the original object
  - Remains up to date with the value of the object
  - Potentially very dangerous
    - Undefined behavior if you attempt to access it after the original goes out of scope
    - Especially prone to bugs when you do multithreading (out of scope of this course)
  - Fast
  - Works with non-copyable types

```
1 std::map<std::string, int> m;
2 auto emplace = [&] (
3     const auto& key,
4     const auto& value) {
5   m.emplace(key, value);
6 };
7 // What happens here?
8 emplace("hello", 5);
```

```
1 auto GetGenerator() {
2   int upto = 0;
3   return [&] () { return upto++; }
4 }
5
6 // What happens here?
7 auto fn = GetGenerator();
8 std::cout << fn() << fn() << '\n';
```

Lambda captures - Generic

- Can use any expression
- Most frequently used for move captures, however

```
1  std::vector<int> vec{1, 2, 3};
2  int n = 10;
3  auto fn = [vec{std::move(vec)}, y=n + 1] () {
4    std::cout << vec.size() << '\n' << y;
5  };
6
7  // Should be 0
8  std::cout << vec.size() << '\n';
9
10 fn();
```