Part 1: Input/Output Testing

Programs with I/O

```
#include <iostream>
#include <string>

static void say_hello() {
  std::string name;
  std::cin >> name;
  std::cout << "Hello, " << name << "!\n";
}

int main(int argc, char **argv) {
  say_hello();
}
```

Programs with I/O

```
#include <iostream>
#include <string>

static void say_hello() {
  std::string name;
  std::cin >> name;
  std::cout << "Hello, " << name << "!\n";
}

int main(int argc, char **argv) {
  say_hello();
}
```

How can we test this?

Testing Programs with I/O

Step 1: parameterize the I/O code over streams

```
#include <iostream>
#include <string>

static void say_hello(std::istream &in, std::ostream &out) {
  std::string name;
  in >> name;
  out << "Hello, " << name << "!\n";
}

int main() {
  say_hello(std::cin, std::cout);
}
```

Testing Programs with I/O

Step 1: parameterize the I/O code over streams

```
#include <iostream>
#include <string>

static void say_hello(std::istream &in, std::ostream &out) {
  std::string name;
  in >> name;
  out << "Hello, " << name << "!\n";
}

int main() {
  say_hello(std::cin, std::cout);
}
```

`&` is like `*` without changing `.` to `->`
or `in << ...` to `(*in) << ...`

Testing Programs with I/O

Step 1: parameterize the I/O code over streams

```
#include <iostream>
#include <string>

static void say_hello(std::istream &in, std::ostream &out) {
  std::string name;
  in >> name;
  out << "Hello, " << name << "!\n";
}

int main() {
  say_hello(std::cin, std::cout);
}
```

& is like * without changing . to ->
or `in << ...` to `(*in) << ...`

effectively `&std::cin`

Testing Programs with I/O

Step 2: create string streams

```
#include <sstream>

std::stringstream in("Dolly");
std::stringstream out("");
```

Testing Programs with I/O

Step 2: create string streams

```
#include <sstream>

std::stringstream in("Dolly");
std::stringstream out("");
```

Kind of a shorthand for
`std::stringstream *out = new std::stringstream("");`
but with an automatic `delete out`

Testing Programs with I/O

Step 2: create string streams

```
#include <sstream>

std::stringstream in("Dolly");
std::stringstream out("");
```

- A `std::stringstream` works as a `std::istream` or `std::istream`
- Use `out.str()` to extract an output string

Testing Programs with I/O

Putting those together:

```
TEST_CASE( "hello" ) {
  std::stringstream in("Dolly");
  std::stringstream out("");
  say_hello(in, out);
  CHECK( out.str() == "Hello, Dolly!\n" );
}
```

Testing Programs with I/O

More tests?

```
TEST_CASE( "hello" ) {
  {
    std::stringstream in("Dolly");
    std::stringstream out("");
    say_hello(in, out);
    CHECK( out.str() == "Hello, Dolly!\n" );
  }

  {
    std::stringstream in("Kitty");
    std::stringstream out("");
    say_hello(in, out);
    CHECK( out.str() == "Hello, Kitty!\n" );
  }
}
```

Testing Programs with I/O

Sometimes it's useful to have a test helper:

```
static std::string say_hello_string(std::string s) {
  std::stringstream in(s);
  std::stringstream out("");
  say_hello(in, out);
  return out.str();
}

TEST_CASE( "hello" ) {
  CHECK( say_hello_string("Dolly") ==  "Hello, Dolly!\n" );
  CHECK( say_hello_string("Kitty") ==  "Hello, Kitty!\n" );
  CHECK( say_hello_string("world") ==  "Hello, world!\n" );
}
```

Part 2: Accumulators

Sorted Fish

Suppose we want a function on aquariums that checks whether the fish are sorted smaller to larger:

```
Aq *empty = new EmptyAq();

CHECK( new BiggerAq(5, new BiggerAq(10, empty))->is_sorted()
       == true );
CHECK( new BiggerAq(10, new BiggerAq(5, empty))->is_sorted()
       == false );
```

Sorted Fish

Suppose we want a function on aquariums that checks whether the fish are sorted smaller to larger:

```
Aq *empty = new EmptyAq();

CHECK( new BiggerAq(5, new BiggerAq(10, empty))->is_sorted()
       == true );
CHECK( new BiggerAq(10, new BiggerAq(5, empty))->is_sorted()
       == false );
CHECK( new BiggerAq(5, empty)->is_sorted()
       ==      );
```

Sorted Fish

Suppose we want a function on aquariums that checks whether the fish are sorted smaller to larger:

```
Aq *empty = new EmptyAq();

CHECK( new BiggerAq(5, new BiggerAq(10, empty))->is_sorted()
       == true );
CHECK( new BiggerAq(10, new BiggerAq(5, empty))->is_sorted()
       == false );
CHECK( new BiggerAq(5, empty)->is_sorted()
       == true );
```

Sorted Fish

Suppose we want a function on aquariums that checks whether the fish are sorted smaller to larger:

```
Aq *empty = new EmptyAq();

CHECK( new BiggerAq(5, new BiggerAq(10, empty))->is_sorted()
       == true );
CHECK( new BiggerAq(10, new BiggerAq(5, empty))->is_sorted()
       == false );
CHECK( new BiggerAq(5, empty)->is_sorted()
       == true );
CHECK( empty->is_sorted()
       ==      );
```

Sorted Fish

Suppose we want a function on aquariums that checks whether the fish are sorted smaller to larger:

```
Aq *empty = new EmptyAq();

CHECK( new BiggerAq(5, new BiggerAq(10, empty))->is_sorted()
       == true );
CHECK( new BiggerAq(10, new BiggerAq(5, empty))->is_sorted()
       == false );
CHECK( new BiggerAq(5, empty)->is_sorted()
       == true );
CHECK( empty->is_sorted()
       == true );
```

Sorted Fish

```
bool EmptyAq::is_sorted() {
    ....
}

bool BiggerAq::is_sorted() {
    .... fish .... rest->is_sorted() ....
}
```

Sorted Fish

```
bool EmptyAq::is_sorted() {
    ....
}

bool BiggerAq::is_sorted() {
    .... fish .... rest->is_sorted() ....
}
```

return true;

Sorted Fish

```
bool EmptyAq::is_sorted() {
    ....
}

bool BiggerAq::is_sorted() {
    .... fish .... rest->is_sorted() ....
}
```

Not enough information about `rest`

Sorted Fish

How about this?

```
bool BiggerAq::is_sorted() {
    return (rest->is_sorted()
            && fish <= rest->fish);
}
```

Sorted Fish

How about this?

```
bool BiggerAq::is_sorted() {
    return (rest->is_sorted()
            && fish <= rest->fish);
}
```

type is **Aq***

Sorted Fish

How about this?

```
bool BiggerAq::is_sorted() {
    return (rest->is_sorted()
            && fish <= ((BiggerAq *)rest)->fish);
}
```

Sorted Fish

How about this?

```
bool BiggerAq::is_sorted() {
    return (rest->is_sorted()
            && fish <= ((BiggerAq *)rest)->fish);
}
```

crashes on **EmptyAq**

Never try to look ahead in a field object

Sorted Fish

How about this?

```
bool BiggerAq::is_sorted() {
    return (rest->is_sorted()
            && fish <= rest->first_fish());
}
```

Sorted Fish

How about this?

```
bool BiggerAq::is_sorted() {
    return (rest->is_sorted()
            && fish <= rest->first_fish());
}
```

no good implementation for **`EmptyEq`**

Sorted Fish

How about this?

```
bool BiggerAq::is_sorted() {
    return (rest->is_sorted()
            && rest->bigger_than(first));
}
```

This works fine, but also points to a more general idea

Sorted Fish via Accumulator

Instead of pulling information "up," we can push it "down"

```
class Aq {
  bool is_sorted();
  // Checks whether the aquarium is sorted with no
  // fish smaller than max_prev_fish:
  virtual bool is_sorted_after(int max_prev_fish) = 0;
};

bool Aq::is_sorted() {
  return this->is_sorted_after(0);
}

CHECK( empty->is_sorted_after(1)
       == true);
CHECK( new BiggerAq(5, empty)->is_sorted_after(1)
       == true);
CHECK( new BiggerAq(5, empty)->is_sorted_after(10)
       == false);
```

Sorted Fish via Accumulator

```
bool EmptyAq::is_sorted_after(int max_prev_fish) {
  return true;
}

bool BiggerAq::is_sorted_after(int max_prev_fish) {
  return (fish >= max_prev_fish
          && rest->is_sorted_after(fish));
}
```

The `max_prev_fish` argument is an ***accumulator***