Today's Plan

Upcoming:

- A1 posted!
- Labs this week

Last time:

- Function Calls and the Stack
- Testing

Today's topics:

- From last time:
  - Unit Testing
- Test Drivers
- If-style Tests
- Assert-style Tests
- Table-based Tests
- Property Testing
- Using an LLM

Some ways to test code: Test Drivers

- We usually want automated testing
- But there is one kind of manual testing that can be helpful: **test drivers**
- Test drivers are small interactive programs that print the input to functions
- They let you experiment with functions and get **immediate feedback**
  - Like an interpreter in Python or Java … but you need to write the code!

```
// return a copy of s with trailing and leading spaces
// removed
string strip(const string &s)
{
    int begin = 0;
    while (begin < s.size() && s[begin] == ' ')
    {
        begin++;
    }
    int end = s.size() - 1;
    while (end >= 0 && s[end] == ' ')
    {
        end--;
    }
    return s.substr(begin, end - begin + 1);
}
```

Some ways to test code: Test Drivers

```
// return a copy of s with trailing and leading
// spaces removed
string strip(const string &s)
{
    int begin = 0;
    while (begin < s.size() && s[begin] == ' ')
    {
        begin++;
    }
    int end = s.size() - 1;
    while (end >= 0 && s[end] == ' ')
    {
        end--;
    }
    return s.substr(begin, end - begin + 1);
}
```

Function we wish to test

```
int main() {
    for (;;) {
        cout << "--> ";
        string input;
        getline(cin, input);
        string result = strip(input);
        cout << "  strip(\"" << input << "\") = \""
             << result << "\"\n";
    }
}
```

Test Driver

```
-->  cat
  strip(" cat ") = "cat"
-->    cat
  strip("   cat   ") = "cat"
-->
  strip("  ") = ""
-->
  strip("") = ""
```

Sample test driver interaction

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing
5. Using LLM to help with testing

```
// returns a copy of s in quotation marks
// e.g. quote("Hi!") returns "\"Hi!\""
string quote(const string& s)
```

We'll use **quote(s)** as a sample function for looking at different testing techniques.

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing
5. Using LLM to help with testing

```
// returns a copy of s in quotation marks
// e.g. quote("Hi!") returns "\"Hi!\""
string quote(const string& s)
```

```
void test_quote_if_style() {
    if (quote("cat") != "\"cat\"") {
        cout << "test failed";
    }
    if (quote("hot soup") != "\"hot soup\"") {
        cout << "test failed";
    }
    if (quote("") != "\"\"") {
        cout << "test failed";
    }
}
```

Pros

- Relatively simple and flexible
- Put whatever code you want in the if-statement bodies,

Cons

- A lot of typing; programmer may get tired of doing it.

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing
5. Using LLM to help with testing

```
// returns a copy of s in quotation marks
// e.g. quote("Hi!") returns "\"Hi!\""
string quote(const string& s)
```

```
#include <cassert> // assert is from here

void test_quote_assert_style() {
    assert(quote("\"cat\"") == "\"cat\"");
    assert(quote("hot soup") == "\"hot soup\"");
    assert(quote("") == "\"\"");
    cout << "all quote tests passed\n";}
}
```

Pros

- Short and simple to type
- When an assert fails, it immediately crashes the program and includes:
  - Exact text of the assertion
  - Line number of the failure

Cons

- Not very flexible: can only crash on failure

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing
5. Using LLM to help with testing

```
// returns a copy of s in quotation marks
// e.g. quote("Hi!") returns "\"Hi!\""
string quote(const string& s)
```

```
#include <cassert> // assert is from here

void test_quote_assert_style() {
    // ...
    assert(quote("cat") == "\"dog\"");
    // ...
}
```

assert(expr) fails when **expr** evaluates to **false** (it does nothing when it evaluates to true).

```
❯ ./assertexample
assertexample: assertexample.cpp:17: int main():
Assertion `quote("cat") == "\"dog\""' failed.
fish: './assertexample' terminated by signal SIGABRT (Abort)
```

A failed **assert** prints its content and its line number.

What is assert()?

- `assert(expr)` is a **macro**, *not* a function!
- In C++:
  - **Functions** evaluate their arguments first before being passed into the function
  - **Macros** pass their arguments into the function unevaluated
- Because `assert(expr)` is a macro, it can print its unevaluated input
- In general, C++ macros are complex and error-prone and you should only use them in a few specific cases (like assert)

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing

Pros
- Very easy to add tests
- Flexible: do whatever you need inside the if-statement

Cons
- More overhead to set up

```
// returns a copy of s in quotation marks
// e.g. quote("Hi!") returns "\"Hi!\""
string quote(const string& s)
```

```
struct Testcase {
    string input;
    string expected;
};

vector<Testcase> all_tests = {
    Testcase{"cat", "\"cat\""},
    Testcase{"hot soup", "\"hot soup\""},
    Testcase{"", "\"\""},
};

void test_quote_table_style() {
    for(Testcase tc : all_tests) {
        string actual = quote(tc.input);
        if (actual != tc.expected) {
            cout << "test failed\n";
}}}
```

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing
5. Using LLM to help with testing

```
// returns a copy of s in quotation marks
// e.g. quote("Hi!") returns "\"Hi!\""
string quote(const string& s)
```

Some Properties of quote(s)

- quote(s).size() == s.size() + 2
- quote(s)[0] == '"' && quote(s)[s.size()-1] == '"'
- quote("") == "\"\""

```
for(int i = 0; i < 100; i++) {
   string s = random_string();  // you write this
   if (quote(s).size() != s.size() + 2) {
      cout << "failure\m";
}}
```

Pros

- No need to write (input, output) pairs.
- Can test with random data, which is relatively easy to generate, and avoids bias.

Cons

- Need to write functions like random_string().
- Can be hard to know which, and how many, properties to test.

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing
5. Using LLM to help with testing

```
// returns a sorted copy of v
// e.g. sort({4,1,7}) returns {1,4,7}
void sort(const vector<int>& v)
```

Some Properties of sort(v)

1. sort({}) == {}
2. sort({x}) == {x}
3. sort(v).size() == v.size() (length preservation)

Challenge

Write some properties of sort(v) that might be useful for testing.

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing
5. Using LLM to help with testing

```
// returns a sorted copy of v
// e.g. sort({4,1,7}) returns {1,4,7}
void sort(const vector<int>& v)
```

Some Properties of sort(v)

1. sort({}) == {}
2. sort({x}) == {x}
3. sort(v).size() == v.size() (length preservation)

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing
5. Using LLM to help with testing

```
//
// Returns a copy of s with all leading and
// trailing spaces removed.
// E.g. strip("  apple  ") returns "apple".
//
string strip(const string& s)
{
    int i = 0;
    while (i < s.size() && s[i] == ' ')
    {
        i++;
    }
    int j = s.size() - 1;
    while (j >= 0 && s[j] == ' ')
    {
        j--;
    }
    return s.substr(i, j - i + 1);
}
```

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing
5. Using LLM to help w

This text is “boilerplate” that could be saved and re-used for any testing you want to do. Tweak it to get the results you want.

```
Write test code for this C++ function:

//
// Returns a copy of s with all leading and trailing spaces removed.
// E.g. strip(" apple  ") returns "apple".
//
string strip(const string& s)
{
    int i = 0;
    while (i < s.size() && s[i] == ' ')
    {
        i++;
    }
    int j = s.size() - 1;
    while (j >= 0 && s[j] == ' ')
    {
        j--;
    }
    return s.substr(i, j - i + 1);
}

Use table-style testing where each test starts with a string description
followed by an input and an expected output. Include a variety of test cases:
small inputs, big input, extreme inputs, random inputs, etc. Use code to check
that all the inputs are correct. If a test fails, print a neatly indented error
message showing the expected and actual results in a way that's easy for the
programmer to understand. Also number the test cases, and at the end print a
summary of how many tests passed and how many failed.
```

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing
5. Using LLM to help with testing

It output this entire program …

```
// ... some code removed ...

// Struct to hold test cases
struct TestCase {
    string description;
    string input;
    string expectedOutput;
};

// Test function
void testStripFunction()
{
    // Table of test cases
    vector<TestCase> testCases = {
        {"Leading spaces", "   Hello", "Hello"},
        {"Trailing spaces", "Hello   ", "Hello"},
        {"Leading and trailing spaces", "   Hello   ", "Hello"},
        {"No spaces", "Hello", "Hello"},
        {"Only spaces", "     ", ""},
        {"Empty string", "", ""},
        {"Mixed spaces", "  Hello World  ", "Hello World"},
        {"Long string with spaces",
         "  This is a test string with spaces at both ends  ",
         "This is a test string with spaces at both ends"},
        {"String with no leading or trailing spaces",
         "No extra spaces", "No extra spaces"},
        {"String with tabs and spaces", "\t   Hello\t ", "\t   Hello\t"}};

// ... continued on next slide ...
```

Testing Techniques

1. If-style tests
2. Assert-style tests
3. Table-based tests
4. Property testing
5. Using LLM to help w

Careful!
ChatGPT and other large language models **sometimes make mistakes** and don't usually give the same results for the same input.

```
// ... Continued from previous slide ...
    int passedTests = 0, failedTests = 0;
    for (size_t i = 0; i < testCases.size(); ++i)
    {
        const auto &testCase = testCases[i];
        string result = strip(testCase.input);
        if (result != testCase.expectedOutput) {
            cout << "\nTest " << i + 1 << " failed: "
                 << testCase.description << "\n";
            cout << "    Expected: \"" << testCase.expectedOutput
                 << "\"\n";
            cout << "    Actual:   \"" << result << "\"\n";
            failedTests++;
        }
        else {
            cout << "Test " << i + 1 << " passed: "
                 << testCase.description << endl;
            passedTests++;
        }
    }

    cout << "\nTest Summary: " << passedTests << " passed, "
         << failedTests << " failed.\n";
}

int main() {
    testStripFunction();
    return 0;
}
```

Final Thoughts on Testing

- You must do it! How else do you know your program works correctly?
- Always try to do it automatically
- Test as you go
- "Eat your own dog food"
  - Use the programs you write in your own work … shows you trust it
- Tools like LLMs seem promising, make it less work to do testing
- Other kinds of testing:
  - **Usability** testing, e.g. how easy is it to use your system?
  - **Security** testing, e.g. does your program protect private information?
  - **Performance** testing, e.g. is your program fast enough?
  - **Stress** testing, e.g. can your program handle lots of input?