Today's Plan

Upcoming:

- A1 posted!
- Labs this week

Last time:

- Function Calls
- Stacks

Today's topics:

- From last time:
  - How Function Calls Work with the Stack
- Verifying Code
- Inspection vs. Testing
- Unit & System Testing

How do civil engineers ensure their constructions "work"?

They do many things!

- Hire experts
- Follow safety standards
- Use good materials and techniques
- Inspect and test as they go
- …

How do you know if a program is correct?

Two basic techniques:

1. **Inspection**: read the source code to make sure there are no errors, and that all cases are handled
   - Many software companies (e.g. Google, Microsoft) require at least one other person to read any code you submit
2. **Testing**: run the program on some sample inputs and make sure it does the right thing
   - **Continuous** and **automated** testing is the general standard that most good software companies follow, or at least aim for

Occasionally, it may be possible to mathematically **prove** a program is correct. A mathematical proof can be thought of as a detailed and systematic inspection. In practice, it is rarely used since the proofs are usually more complex than the code.

How do you know if a program is correct?

Is this function correct?

```
void f(int& n)
{
    n++;
}
```

How do you know if a program is correct?

Are these functions correct?

```
// adds 1 to n
void f(int& n)
{
   n++;
}
```

```
// adds 2 to n
void g(int& n)
{
   n++;
}
```

Important!
A function is correct (error-free) if it does what we want it to do. So, to test it, we must **specify** what the function does.

How do you know if a program is correct?

In C++, **specifications** are usually put in comments. Or if it's obvious, in the function name.

```
// adds 1 to n
void f(int& n)
{
   n++;
}
```

```
// adds 2 to n
void g(int& n)
{
   n++;
}
```

Important!
A function is correct (error-free) if it does what we want it to do. So, to test it, we must **specify** what the function does.

How do you know if a program is correct?

Suppose I give you a function with this header:

void **f(const string& v);**

I claim `f(s)` returns `s` duplicated, e.g.

`f("cat")` returns `"catcat"`
`f("bird")` returns `"birdbird"`
`f("a house")` returns `"a housea house"`
etc.

How can I prove to you that my claim about **f** is true?

Bad answers:

- It's true because I say so.
- It's so simple that it couldn't be wrong.
- I got it from the web.
- I got it from CoPilot/ChatGPT.
- Someone told me how I did it is correct.

How do you know if a program is correct?

Suppose I give you a function with this header:

void **f(const string& v);**

I claim `f(s)` returns `s` duplicated, e.g.

`f("cat")` returns `"catcat"`
`f("bird")` returns `"birdbird"`
`f("a house")` returns `"a housea house"`
etc.

How can I prove to you that my claim about **f** is true?

Better Answers:

- Ask to see how **f** is implemented
- Call it on a few different strings and check what it returns

How do you know if a program is correct?

Inspection

- Done manually by the programmer
- Works well for small/simple programs
- Especially useful when done by someone other than the original programmer
- Can be tedious/complicated for bigger programs

Testing

- Can be done automatically or manually
- Usually requires making (input, output) pairs
- Comes in many varieties …

Varieties of Testing

- **Unit testing**: test one part ("unit") of a program, e.g. a single function
  - **Whitebox testing**: make test cases based on the implementation
  - **Blackbox testing**: make test cases based on the specification only, without seeing an implementation
- **System testing**
  - Testing the entire system

Can start as soon as you write one function. **Test-driven development** (**TDD**) is when you write unit tests at the same time as you write your code.

Needs the system to be in a working state. **Waiting until a system is mostly working is a very bad idea!**

System testing on a bridge: load testing

Load testing on the Pelješac Bridge in 2022

- 40 tonnes of trucks drove across the bridge
- Engineers measured to ensure the bridge deformed as expected
- Obviously, the bridge had to be **nearly finished** to do this test
- In software, we often do load testing for websites or servers to make sure they can handle a lot of requests

Varieties of Testing

- **Unit testing**: test one part ("unit") of a program, e.g. a single function
  - **Whitebox testing**: make test cases based on the implementation
  - **Blackbox testing**: make test cases based on the specification only, without seeing an implementation
- **System testing**
  - Testing the entire system

Advantages
- Can check that every line of code is run by a test.
- Also allows you to do some inspection.

Disadvantages
- Test cases need to added/removed if implementation changes

Advantages
- Can write test cases before you write the code.
- Same test cases can work if implementation changes but header stays the same.

Disadvantages
- Can't test implementation-specific details.

Testing: Challenge 1

Suppose I give you a function with this header:

void **f(const string& s);**

I claim `f(s)` returns `s` duplicated, e.g.

`f("cat")` returns `"catcat"`
`f("bird")` returns `"birdbird"`
`f("a house")` returns `"a housea house"`
etc.

How can I prove to you that my claim about **f** is true?

Challenge 1
Write a version of function **f** that is not completely correct, but does work correctly for the three example cases.

Testing: Challenge 1

Suppose I give you a function with this header:

void **f(const string& s);**

I claim `f(s)` returns `s` duplicated, e.g.

`f("cat")` returns `"catcat"`
`f("bird")` returns `"birdbird"`
`f("a house")` returns `"a housea house"`
etc.

How can I prove to you that my claim about **f** is true?

Testing: Challenge 2

Suppose I give you a function with this header:

void **f(const string& s);**

I claim `f(s)` returns `s` duplicated, e.g.

`f("cat")` returns `"catcat"`
`f("bird")` returns `"birdbird"`
`f("a house")` returns `"a housea house"`
etc.

How can I prove to you that my claim about **f** is true?

Challenge 2
Write a version of function **f** that is not completely correct, but fails on **exactly one input**.

Testing: Challenge 2

Suppose I give you a function with this header:

void **f(const string& s);**

I claim `f(s)` returns `s` duplicated, e.g.

`f("cat")` returns `"catcat"`
`f("bird")` returns `"birdbird"`
`f("a house")` returns `"a housea house"`
etc.

How can I prove to you that my claim about **f** is true?

Testing: Challenge 3

Suppose I give you a function with this header:

void **f(const string& s);**

I claim `f(s)` returns `s` duplicated, e.g.

`f("cat")` returns `"catcat"`
`f("bird")` returns `"birdbird"`
`f("a house")` returns `"a housea house"`
etc.

How can I prove to you that my claim about **f** is true?

Challenge 3
Write a version of function **f** that for any given input string may, or may not, return the correct answer.

Testing: Challenge 3

Suppose I give you a function with this header:

void **f(const string& s);**

I claim `f(s)` returns `s` duplicated, e.g.

`f("cat")` returns `"catcat"`
`f("bird")` returns `"birdbird"`
`f("a house")` returns `"a housea house"`
etc.

How can I prove to you that my claim about **f** is true?

rand() returns a random int from 0 to about 2 billion (for 32-bit ints)