CSE 374 Programming Concepts and Tools

Lecture 14 - Pre- and Post-Conditions, Testing

Future

More on Variable Types and Storage

Memory Architecture

C++ Intro

Today

Testing

- Interfaces in Software
- Pre- and Post- Conditions
- Testing
- Types of tests: unit, integration, etc.
- safe_assert Testing Framework

Interfaces in Software

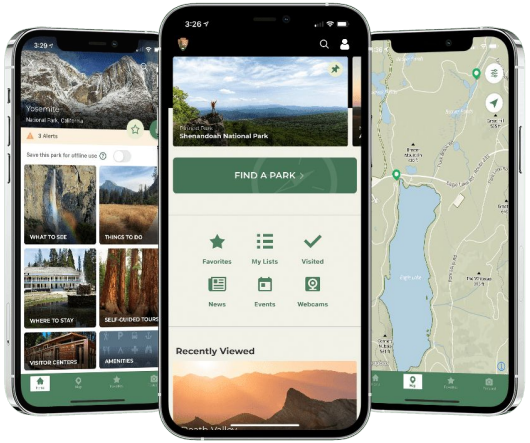
High level: What is an interface?

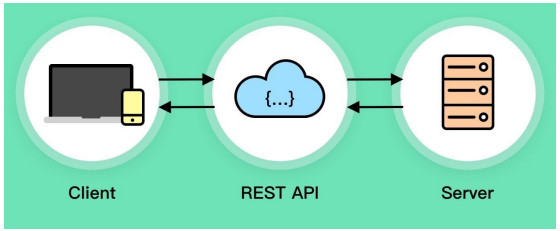
Definition: the place at which independent and often unrelated systems meet and act on or communicate with each other (Merriam-Webster)

At its most basic, an interface is how two systems can interact

- Computer-Computer
- Computer-Human
- Human-Human
- etc.

Examples of Interfaces





Interfaces in Software: Functions!

Think of your functions as a black box, where input goes in and output comes out

Interface = what kind of input and output your black box has

- Forms a contract between users and your code
- “Give us this stuff, and we will give you this result”
- Function declarations serve as a literal manifestation of this contract

Implementation = the code to make your black box work

- There can be multiple valid implementations for the same interface
- e.g. One implementation uses an array, another uses a linked list, both valid

Remember the Java keywords: `interface`, `implements`?

Interfaces in C

In C, the function declaration serves as the interface

```
char* strncpy (char* des, const char* source, size_t num);
```

What does this tell us? What doesn't it tell us?

- The header files (`.h`) serve as the interface
- The C files (`.c`) serve as the implementation
- The object files (`.o`) serve as the black box
    - We can swap out any two `.o` files that use the same interface!
        - → Why linking and compiling should be separate (Lecture 12)

Interface Design

Designing interfaces in software is *hard*.

- Once you define an interface and others start using it, it's very difficult for you to go back and change it!
- It's easy to change the implementation as long as the interface remains the same.

Many programming languages define their own tools to help define reliable interfaces

Pointer Write Permissions

Pointer types can sometimes be confusing

```
int foo(int* p);
```

- Does `foo(p)` write to p? *It's not clear from the declaration*

```
int foo(int* p) {
  return *p;
}
```

```
int foo(int* p) {
  *p = 374;
  return 0;
}
```

const Pointers

Add *const* to the pointer type to make the pointer *read-only*: i.e., cannot modify!

```
int foo(const int* p);
```

- Now `foo(p)` cannot modify `p`

```
// OK, compiles
int foo(const int* p) {
  return *p;
}
```

```
// Doesn't compile!
int foo(const int* p) {
  *p = 374;
  return 0;
}
```

Demo: Interfaces in C

Pre- and Post-Conditions

How do we know our program works?

Some indicators:

- Does it crash?
- Does it loop infinitely?
- Does it give us the correct output?
- Does it give us the correct output for every input?

What can we do to help us know the program behaves the way it's supposed to?

Input

What type?

- e.g. `void foo(int, int*);`

What needs to be true about our input?

- e.g. `void foo(int len, int* arr);`
- "`arr`" must be at least "`len`" long, "`len`" is non-negative

This is called a "pre-condition"

- A fact that must be true about the inputs to a piece of code
- If input doesn't satisfy the pre-condition, all bets are off

Polling Time!

Please answer in the LP14 assignment on Gradescope!

Consider the function `free(void* p)`;

If you were explaining how to use `free`, what would the *pre*condition be?

- `p` is a pointer to any type
- `p` is a pointer to the heap
- `p` is a pointer to an array on the heap
- `p` is a pointer to the heap, which hasn't been `free`'d

man 3 free

`free` has multiple manual pages, one is a command line tool

Use `man 3 free` to look up "`free`" as a C function

The **free**() function frees the memory space pointed to by *ptr*, which must have been returned by a previous call to **malloc**() or related functions. Otherwise, or if *ptr* has already been freed, undefined behavior occurs. If *ptr* is NULL, no operation is performed.

Output

What's type?

- e.g. `int fact(int n);`

What needs to be true about our output?

- e.g. `int res = fact(n);`
- "`res`" must be equal to "`n`" factorial

This is called a "post-condition"

- Something that must be true about your output

Contracts

A function's pre- and post-conditions form a contract

The user and the function must uphold their end of the deal

- If the input provided by the user doesn't satisfy the pre-condition:
  - "the deal is off", no need to satisfy the post-condition
- If the output provided by the function doesn't satisfy the post-condition:
  - The function has broken the contract, meaning it is incorrect (a bug!)

Undefined behavior: calling a function with input which doesn't satisfy the pre-condition

- Don't rely on undefined/undocumented behavior - it might change!

Hyrum's Law

With a sufficient number of users of an ***API***,

it does not matter what you promise in the contract:

all observable behaviors of your system

will be depended on by somebody.

It's *vital* that we design our interfaces, pre-conditions, and post-conditions very carefully! The more programmatic guards we can use (like `const`), the better.

Summary So Far

Valid **input** = input satisfying the **pre**-condition

Correct **output** = output satisfying the **post**-condition

Correct functions generate correct output for every valid input

We don't care what happens on invalid input

- Your function is free to do anything, even segfault!

Ideally, we want to fail fast if we're following defensive programming (for Hyrum!)

Specification = list of pre- and post-conditions for every function in the program

Example: `pow()`

```
int pow(int base, int exp);
```

Pre: `exp >= 0`

Post: returned value is equal to `base` to the power of `exp`

- The user of `pow()` is the person who calls `pow()`
- The designer of `pow()` is the person who wrote the code for `pow()`

Polling Time!

Please answer in the LP14 assignment on Gradescope!

Part 1:

If `pow(-10, 3) != -1000 = (-10)`^{3},
whose fault is it?

- User
- Designer

Part 2:

If `pow(1, -2) != 1 = 1`$^{-2}$,
whose fault is it?

- User
- Designer

```
int pow(int base, int exp);
```

Pre: `exp >= 0`

Post: returned value is equal to `base` to the power of `exp`

Testing

Testing

We can write tests to check if our functions generate **correct** output for **valid** input

Writing a test:

- Start with input which satisfies the pre-condition
- Run the function on that input
- Check whether the post-condition is satisfied

* There used to be a very clear separation between testers & developers. Today (and probably for the past decade) it's very common for developers to be testers too!

Testing Isn't Perfect

Program testing can be used to show the presence of bugs, but never to show their absence!

– Edsger Dijkstra, 1970 (1972 Turing Award Winner)

In other words,

“Absence of evidence is *not* evidence of the absence.”

“Just because you did not detect a bug, doesn't mean you don't have one.”

“Even when you don't detect any bugs through testing, you can still have one.”

Tests are (usually) Good Enough

Tests can never 100% prove that your program is correct

- Even if you test every single input (if even possible), on every single system!

However, tests are *usually* good enough to give you confidence

Ideally, your tests should cover all the different "ways" your program can be used

- Maximize code coverage (more on this later)

Together, these tests form a **test suite**

- Similar concepts exist in many other languages (e.g. JUnit, GoogleTest).

Assert

In many programming languages, we can assert whether a condition is true

- If the condition is true, do nothing (it is working as expected)
- If the condition is false, alert the user and stop the program

In C, `assert()` is defined in `<assert.h>`

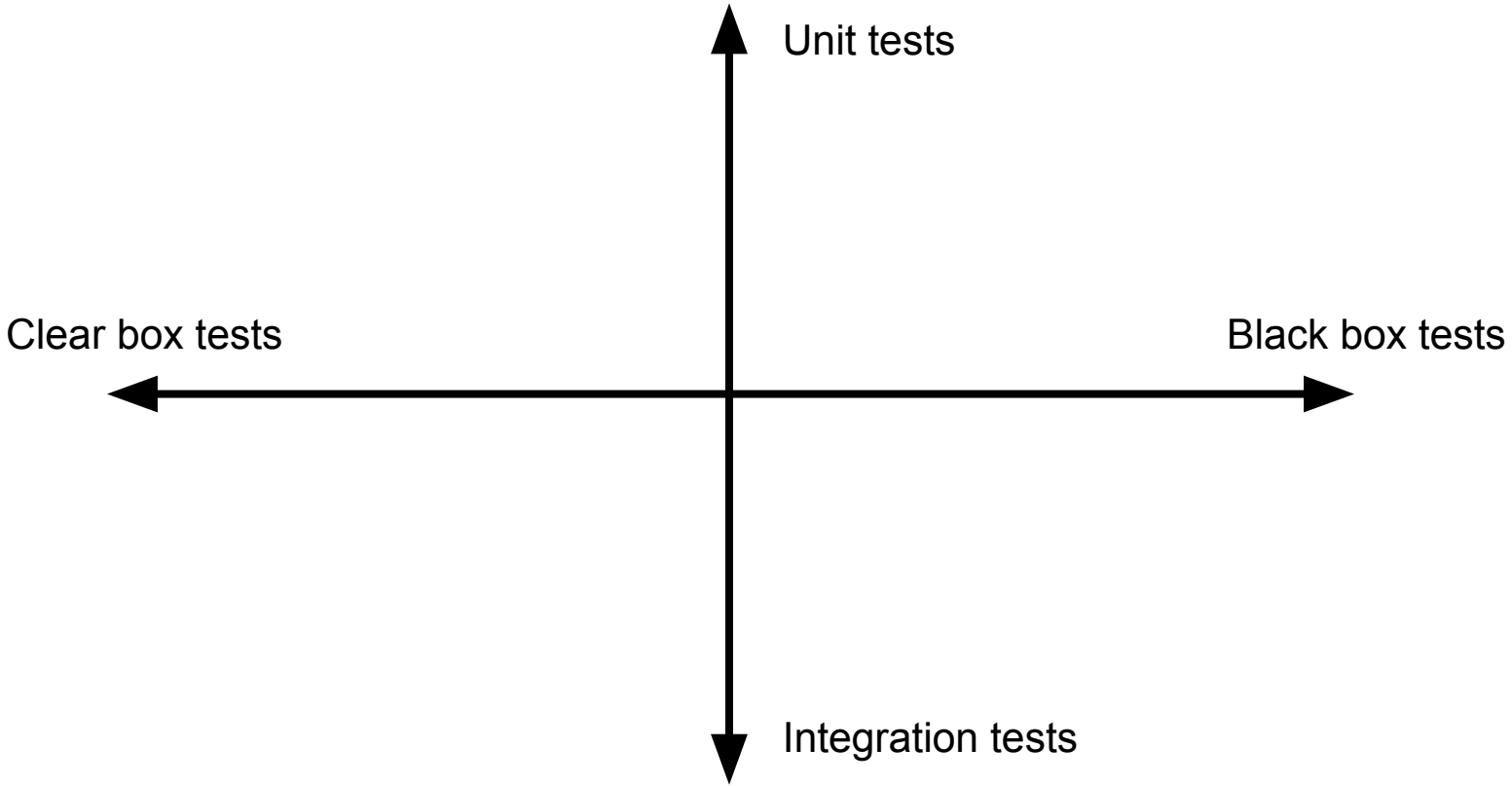
It can be used like a function, but it is technically a macro

- Being a macro allows it to do special things like tell you which line failed

```c
#include <assert.h>
int main() {
    void* pointer = 0;
    assert(pointer);
    return 0;
}
```

Demo: Assert

Kinds of Tests



Unit vs. Integration Testing

Unit testing = testing one module/function by itself

- i.e. Testing only one thing at a time
- If a test fails, we can pinpoint *exactly* what input the function fails on
- Hard to catch bugs created by a cascade of smaller errors
- Often done immediately after (or hopefully before!) implementing

Integration testing = testing many modules/functions together

- i.e. Testing how functions will interact when called in sequence
- Often more realistic simulation of how code will be used

Black vs. Clear Box Testing

Black box testing = tests designed using only information from the specification

- These can be written *even before implementing the spec*
- This is what you're doing in HW5!

Clear box testing = tests influenced by the implementation

- Author may know about potential weaknesses that need to be tested
- What are the edge cases?
- These tests should still pass, even if tested on a different implementation

How Much Testing?

Writing tests is expensive in time & money

How much should we test before we're satisfied?

Many heuristics exist to answer this question

- One common one: code coverage

Code coverage = was each line of code executed?

- Having code coverage isn't enough, but **not** having code coverage is a problem
- Code coverage is part of clear box testing!

Polling Time!

Please answer in the LP14 assignment on Gradescope!

I have the *header* files (`.h`) and *object* files (`.o`) of some program and I want to test my functions *working together*. What kind of tests should I use in this situation?

- **Black**-box **unit** tests
- **Clear**-box **unit** tests
- **Black**-box **integration** tests
- **Clear**-box **integration** tests

Regression Tests

As you develop larger projects, you may edit code that you wrote earlier

You want tests to make sure that any fixes/edits don't introduce further bugs (i.e. a regression)

Regression testing is where you test against old output to make sure that the behavior has not changed

- Also used in the context of adding regression tests to detect when old bugs reoccur

Good practice is to create at least one regression test each time you fix a bug, to make sure it doesn't come back

Testing Frameworks

Various programming languages have various frameworks that make testing easier

For CSE 374, we will give you a testing framework, `safe_assert.h`

- You must `#include "safe_assert.h"` in your code
- This should be the first line in the testing file

Testing framework has its own syntax, which is different from C's syntax

Using `safe_assert.h`

In test files, instead of a `main()` function, there is a `suite()` block

- The suite block takes a string literal which is the name of the suite

Inside of the `suite()` are similar `test()` blocks

```
suite("My test suite") {
  test("Test case 1") {
    int res = pow(10, 2);
    safe_assert(res == 100);
  }
}
```

The `safe_assert()` macro

`safe_assert()` is very similar to `assert()`, except that if you segfault inside of an assert, it will tell you!

```
Assert failed (test.c:14): `*null_ptr'
```

The test suite process will survive the segfault

- Then, it will tell you that there was an error and run the other test cases
- You don't have to know how this works (it's magic ✨)
    - Again, interface vs. implementation

Demo: Testing Framework

Assignment Reminders

Mid-quarter survey due tonight!

- Opportunity for anonymous feedback!

HW3 late deadline is tonight, Monday at 11:59 PM

EX12 released today, due before class on Wednesday