Types of tests

Unit tests: test a single *unit* of your code in isolation

- Test as little code as possible at once
- A failure in one unit shouldn't break the tests for another unit
- Useful to
  - Pinpoint failures to a specific piece of code
  - Check edge cases that may not yet be exercised by other units

Integration tests: test how multiple (or perhaps all!) units work together

- Test units as they'll be configured in the real world
- A bug in a single unit may cause many integration tests to fail
- Useful to
  - Ensure the larger system works as expected

DISCUSSION QUESTION

Is a test that calls a single function always a unit test?

Type your answer, but wait for our cue to send it.

~~absolute unit~~

~~Types of tests~~ The continuum of tests

Few real tests perfectly fit into one category:

- Units often depend on other units
- Your language or architecture can prevent isolated testing of such units
- Many unit tests, therefore, end up "seeing" other units indirectly

But your *intent* still matters:

- Unit and integration tests serve different purposes
- Don't let technical constraints blind you to that fact
- Write both unit and integration tests, and keep them separate
    - Your unit tests may end up testing multiple units, and that's okay
    - Your integration tests might not cover certain units, and that's okay too

Which one should you choose?

Both.

Writing unit tests

What to test

- Your code
- Code you rely on that is not so well tested
- Code you *really* rely on

Unit testing maxims (*not* Max-isms)

- Test small units of code as directly as possible
- Avoid "round trips" through layers of software
  - Dependency injection/mocking
  - Function composition
- Avoid stateful computation
- It's not a test unless you watch it fail

- Eat food. Not too much. Mostly plants.

Hands on with UTest: Seer PeopleSoft

```
// person-test.cpp
#include "person.h"
#include "utest.h"

UTEST(PersonTests, ConstructorSetsAge) {
  Person p;
  EXPECT_EQ(p.age(), 0);
}

// test-main.cpp
#include "utest.h"

UTEST_MAIN();
```

Maxim: test small units of code

How would you test `void setAgeAndHeight(int age, int height)`?

Would it be easier if it were `setAge` and `setHeight`?

Why might a compound setter (or constructor) exist?

demo time!

Tightly-coupled units

- Most software projects consist of multiple layers of code
    - e.g. frontend, backend, database
    - Each layer can be thought of as its own unit
- Units often directly call other units
- We say two such units are *tightly-coupled*
    - You can't use the higher one without the lower one
    - A bug in the lower one can cause the higher one to fail

DISCUSSION QUESTION

How would you isolate a unit that calls other units?

Type your answer, but wait for our cue to send it.

Strategies for isolating units

1. Get rid of layer dependencies
   - Call each layer in sequence, passing one's output as the next one's input
   - Prevents any layer from tightly coupling to another
   - Top layer that chains together other layers has no logic and so needs no unit tests
   - *Very hard or impossible in many languages*
   - *Easier in functional languages and ones with evented I/O*
2. Inject your dependencies
   - Make each layer conform to an interface
   - Pass instances of lower layers as parameters to higher ones
   - When testing, can pass *mocks* instead
   - *Hard to do in non-object-oriented languages*