```c
#include <stdio.h>

typedef char c;

int main(){
    struct c {c c;} c;
    c: c.c = 'c';
    printf("%c\n", c.c);
}
```

Elevator API

- The C driver:
  - Defines constants
  - Does the necessary bit-fiddling
- We want a higher-level abstraction
  - What are the necessary functions?

- Reading buttons, including stop & obstruction
- Reading the floor sensor
- Setting the motor speed
- Setting the lights

Unit tests

- A unit means
  - A single function
  - A class/object
  - A module/package
  - A routine (or several)

- Keeping the definition of “unit” small is the key to both comprehending and testing software

The basic test

- Check you get the expected output from a function
- Or the expected message from a routine

- This only works for pure functions, or stateless routines

  What if a function depends on state?

    Files, network, hardware... or other parts of the program

Other tests

- Integration tests
  - Multiple units in concert
  - Test that multiple units are glued together correctly
  - Larger surface area, so they fail more often and more mysteriously

Other tests

- End-to-end tests
  - The entire process from start to finish
  - Even more flaky
  - Expensive and time-consuming to write and maintain

  - This is one of three parts of the project evaluation

The advanced test

- Testing is easier if you separate out your dependencies
  - Separate logic from data
  - Introduce failure in the dependencies
- Constructor injection (not only OO)
  - All dependencies of a unit are passed to the constructor
    - Dependencies can then be substituted with mock objects
    - However, constructor call changes when dependencies change
      - Leaky abstraction
    - MP: Dependency is a channel/mailbox

The advanced test

- Mocks do not need to be objects
  - MP: use message generators to simulate events from a routine
    - Mostly useful for integration tests
    - Does not test that a routine generates those messages
  - Functional: Closures, pass deps. in higher order functions
- This is yet another reason to strive for low coupling
  - Maintainability and testability go hand in hand

Contracts

- Define a precondition and/or postcondition
- A test within a function/An invariant within an object
- Not (necessarily) part of the core logic

- A contract is documentation that is checked at runtime
- Defensive:

  Attempts to deny incorrect usage

  The opposite is “faith-based” programming

```
bool OrdersAbove(int floor, bool[] ordersUp,
                 bool[] ordersDown, bool[] ordersInternal)
in{
    assert( ordersUp.length == ordersDown.length );
    assert( ordersUp.length == ordersInternal.length );
    assert( floor < ordersUp.length );
    assert( floor >= 0 );
}
body{
    // via contract: assuming all order tables are of same length
    for(int i = floor+1; i < ordersUp.length; i++){
        if(ordersUp[i]  ||  ordersDown[i]  ||  ordersInternal[i]){
            return true;
        }
    }
    return false;
}
```

```
int SelectBestElevator( Containers.Button_s btn,
                        Containers.State[int] elevators,
                        int thisPeerID )
in {
    assert( btn.button == Containers.ButtonType.UP  ||
            btn.button == Containers.ButtonType.DOWN,
                "SelectBestElevator called with invalid Containers.ButtonType");
    assert(elevators.length > 0,
                "SelectBestElevator called with no listed elevators");
}
out {
    assert(__result in elevators,
                "SelectBestElevator returns ID of unlisted elevator");
}
body {
    ///
}
```

Coverage Analysis

- Shows which lines of code has been executed by a test
  - Doesn’t prove the absence of bugs
  - Still surprisingly effective at removing bugs
  - Code that is never run is usually a dead giveaway

  - Also a rudimentary performance analyzer