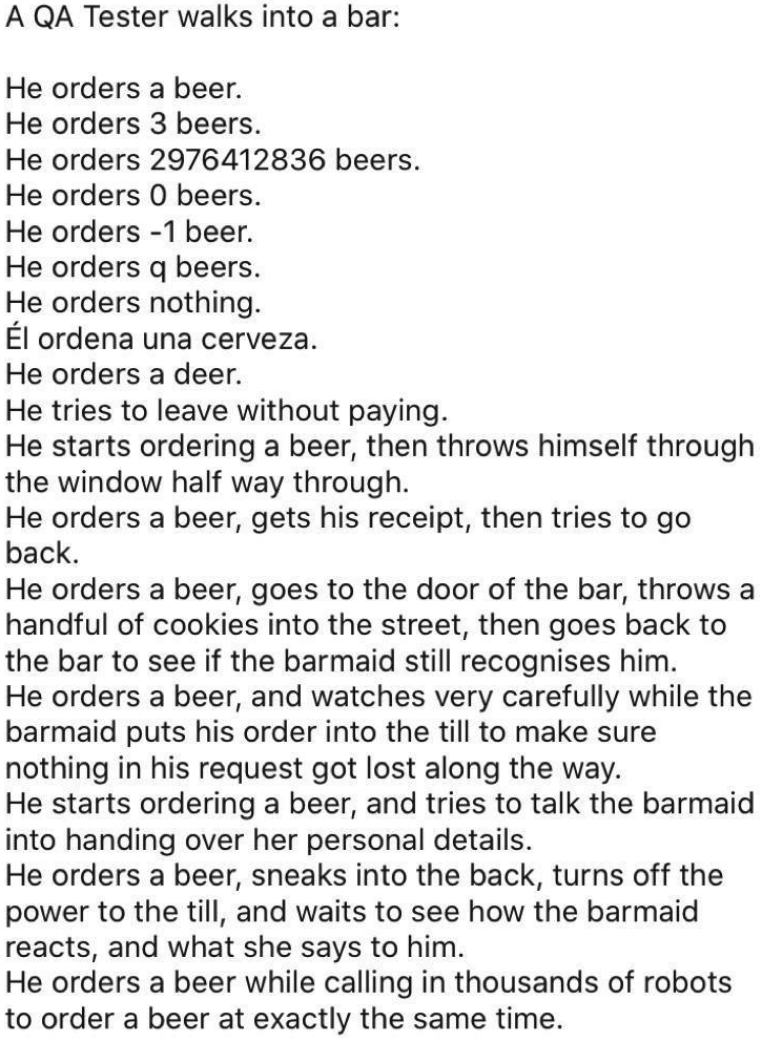
Introduction to testing



“The job of a programmer is to build things. The job of a tester is to break things.”
Learn to wear both hats.

Testing Overview:

We test our code to make sure it’s hard to break (robust), even if users do something crazy or unexpected.

Imagine you are in charge of testing an ATM machine before it gets installed. Write out a plan to make sure it works consistently. Think about both hardware and software. Consider:

- Normal functionality
- Unexpected inputs (i.e. someone requests money in pesos)
- Malicious inputs (i.e. someone tries to break into it)

Test-Driven Development:

You didn't need to have code in front of you to consider how you would test the software for the ATM. In fact, considering how you would test the software, what sort of things it would need to be able to do and what things it should prevent or be prepared for, can be helpful to consider *before* writing code. This is called test-driven development.

Write the outline of a program for the ATM machine, considering the tests you designed for it.

Writing Testable Code:

Now, as you write the code, you need to make sure it's testable. It's *best* to make sure it's automatically testable, because manually checking the output, especially for long or complex programs, is unreliable and time consuming.

Look at the function below, which tells the exchange rate from gbp to an input currency.

```
void exchange_rate(void)
{
	char cur_type;
	printf("Enter the currency you would like to receive: \n"/
		“Enter p for gbp, d for usd, or e for eur.\n”);
	cur_type = getchar();
	switch(cur_type) {
		case ‘p’:
			printf(“The exchange rate for gbp is 1.0.”);
			break;
		case ‘d’:
			printf(“The exchange rate for usd is 1.3.”);
			break;
		case ‘e’:
			printf(“The exchange rate for eur is 1.11.”);
			break;
		default:
			printf(“Error: Unrecognised input.”);
			break;
	}
}
```

Is this code very testable? Rewrite it as a more testable function (you can assume you'll be using assert testing, like you are doing in C).

Unit Testing:

Next, break up your test plan to test "units" of code (functions, in your case) as you write them.

Arrange, Act, Assert

1. Initialise the code you want to test.

2. Call the function you want to test.
3. Assert that the result is what you expect.

Once you've rewritten the above currency function, write a thorough set of unit tests for it.
Remember to base your tests on your original test plan and check edge cases!

Integration Testing:

All that's left is to check that the individual "units" work together as a whole. Follow your developed test plan to ensure that the program can handle everything you can think to throw at it.

In the C unit – checking that the program functions correctly as a whole. (Unit/component integration testing)

In real life – checking that a large suite of programs, packages, and even outside databases, internet sources, etc. work together and function as expected. (System integration testing)
You'll learn more about this in software engineering next term.

Performance Testing:

Testing of a system's speed, scalability, and stability.
This is used on client-server based systems only (i.e. systems that see a lot of traffic from users).
It becomes an issue on larger applications that will be seeing a lot of traffic (e.g. web or mobile apps) or must perform consistently (e.g. safety systems in a hospital or on a submarine or aircraft).
You'll learn more about this type of testing in software engineering.

Simple (very simple) performance checks can be done on individual programs using timing functions (some are included in time.h). For example (pseudocode warning!!):

```
time1 = time(NULL);
/* Do some stuff */
time2 = time(NULL);
printf("Your function took %d seconds.\n", time2 – time1);
```

This is rubbish if you only run through the function once. Average millions of runs and you'll have a general idea of your code's performance. (Mostly good for comparisons.)

FOR FURTHER INFORMATION:

Here are some links to further information about testing. DON'T PANIC if they're beyond you. They are way beyond the scope of your current

knowledge, and they’re just if you’re curious or looking for resources later in the year.
Unit testing/writing testable code:
https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters
Integration testing:
https://softwaretestingfundamentals.com/integration-testing/
https://www.softwaretestinghelp.com/what-is-integration-testing/
Performance testing:
https://www.guru99.com/performance-testing.html