Development

Progression

You should be aiming in your degree to move through these stages:

- *coder:* able to get programs to work
- *programmer:* understand design, precision, style, algorithms, data structures, libraries, efficiency, ...
- *developer:* able to handle larger programs, work in teams, follow industry practices, ...

Size and speed

You probably have a personal program size limit of tens or hundreds of lines

You may take much longer to write a program than you should, because debugging takes up too much of your time

You can view this chapter as providing techniques to push your limit up to thousands of lines, while reducing your debugging time to a low percentage

Golden rules

We will cover some very basic and practical development issues, relevant to programmers working on their own

They are presented here as golden rules

Take small steps

Your program should always be in a working state

Only write one or two more lines before re-compiling and re-running

Start with a hello world program and gradually make it do more

The farmer's response

A tourist lost in the countryside asks for directions from a farmer

The farmer thinks for a while and says: ***"If I were you, I wouldn't start from here"***

If you don't pay attention to the small steps rule, and you ask someone for help, this may be the only possible response they can give

If you've only written 2 lines since the program was last under your control, you know where to look for the bug

Design using tiny functions

Each function should be responsible for doing "one thing", and should be so short you can see it is correct

Aim for about 5 lines - a function should be visible all at once without scrolling

Then you can develop and test one function at a time, with the previous functions giving you a solid foundation for the next one

Cleverness

Brian Kernighan, one of the creators of C, said:

"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

Don't try to be too 'clever' and out-smart yourself

So don't try to do too much in one function

And don't write code which is too compact

Top down or bottom up?

Should you work top down (start with the whole problem and break it down into smaller pieces) or bottom up (start with low level functions and build up)?

The answer is ***both***

You must do some top-down design, to work out what functions you are going to need

You must develop the functions bottom up, otherwise you can't test them

To some extent, you can alternate

Example: Noughts and crosses

You might break the problem into two: (a) handling the 3x3 grid, (b) user interaction, then (a) breaks down into:

- a data structure for storing the grid
- a function to check if a cell is empty
- a function to put O or X into a cell
- a function to check for three in a row

You can now write and test the structure and the three functions

Then you can go back and do some design work on the user interaction

Keep your code DRY

DRY stands for ***Don't Repeat Yourself***

If ever you see similar-looking code in two or more places, it is a symptom of potential problems - some people call these symptoms smells

Example: Noughts and crosses

A function for checking for a win might be:

```
if (g[0][0] == 'X' && g[0][1] == 'X' && g[0][2] == 'X' ||
    g[1][0] == 'X' && g[1][1] == 'X' && g[1][2] == 'X' ||
    g[2][0] == 'X' && g[2][1] == 'X' && g[2][2] == 'X' ||
    ...
```

This is repetitive, not very readable, and error-prone

```
if (row(g,0,'X') || row(g,1,'X') || row(g,2,'X') || ...
```

This is better, re-using a small testable function

Do automated testing

Every developer knows that this is the only thing that works, but there seems to be a big psychological barrier that programmers have to overcome before they do it

That's why we try to get you do it from day one

Look up ***unit testing***, but beware that people rarely explain properly that it is supposed to be ***automated***, and they make it over-complicated

How much?

The rule is not about doing ***lots*** of testing

The purpose of testing is to boost ***your*** confidence

So how much testing you do, and which tests you write, is entirely up to you and your experience

But follow this rule: do however much testing you think is right for you, but make sure you ***automate*** it, not run your program over and over again looking at the results by eye

Automation

In 1945, before there were any computers to speak of, Alan Turing said:

"There need be no real danger of it [computing] ever becoming a drudge, for any processes that are quite mechanical may be turned over to the machine itself."

In other words, if you find anything boring, automate it

Testing tends to be boring, so automate it

Regression testing

Experience shows that programs ***break*** as you develop

Things that used to work stop working

So if you run your program in a certain way to test it, you will need to repeat that test many times to make sure that it continues to work

So, build all your tests into your program, i.e. automate them and never delete them

Design for testing

You can't retro-fit testing into your program

You've got to design the program from the beginning with easy testing in mind

Keep functions self-contained, so that they can be tested in isolation

Especially, make sure functions ***don't include any user interaction or other I/O*** whenever possible

Separate out the I/O, and test the rest

Refactor programs

This means re-organise, i.e. rewrite, your program, ready for the next stage

You have to ***break*** your program, but the automated tests will give you the confidence to know you have repaired it

Use a versioning system

Use `git` to keep track of versions of your program, and `gitlab` to back it up in the cloud

See the aside on version control for details