Introduction to Software Development

Chris Brady
Heather Ratcliffe

"The Angry Penguin", used under creative commons licence from Swantje Hess and Jannis Pohlmann.

Warwick RSE

24/10/2018

Aims and some Vital Rules

Aims

- Understand concepts in software design/ development
  - NOT coding
- How to plan, write and license code
- Debugging your code
- Testing research code
- Making code shareable

Software “design” and “development” are not really any different to “writing code”. They’re just usually at a more abstract level. It’s impossible to write even a slightly complicated code without some degree of design, even if this happens only as you write. The only real distinction is that the “design” encompasses much more, including how you write the code (which language, libraries, techniques), what the code should actually do (essential features, limitations etc). The “development” process also includes things like how to collaborate on code, how to debug it etc.

Setup/Disclaimer

- Software development is massive, changing field
  - Web technologies change yearly
  - Most languages add features every few year if not more
- Even “best practice” neither universal nor fixed
- You’ll never know everything:
  - Should spend lots of time saying “aha! I’m sure I read/ heard something about this. Now what was the key word to look for?”
  - Then you turn to a book index/search engine/colleague and find out what you need to know

We mostly stay away from specific languages here, only discussing interpreted vs compiled since they are written, debugged, tested, quite differently

Setup/Disclaimer

- Short intro - “hooks”
  - You’ve seen the concepts you’ll need
  - You recognise when you need this stuff
  - You’ll know what to search for when you do
- Slides/notes available at https://warwick.ac.uk/research/rtp/sc/rse/training/introdevshort after this session
  - Don’t expect you’ll remember everything without them!
- December - workshop where you can get hands-on experience/support - watch the Calendar

Idea of “hooks” crops up all over computing. Basically lets one thing happen whenever something else does. E.g. program X starts then program Y should start too. If you use Gmail, and write “attachment” in an email body, you trigger an “attachment hook” which runs code to remind you to attach a file if you didn’t already. Hopefully these notes will give you some hooks to remind you of really useful concepts. Like hooks themselves, which are closely related to “callbacks” (a function effectively saying “when this happens, call me”), which themselves depend on being able to pass around and store functions. But this is going off topic.

Disclaimer

- Once taught “sqrt(-1)” makes no sense. Later learn about complex numbers
- Atom is “small solar system”. Later learn about shells, probability density, delocalisation
- Pratchett, Stewart, Cohen *Science of Discworld*
  - Lies to Children: “… a statement that is false, but which nevertheless leads the child’s mind towards a more accurate explanation, one that the child will only be able to appreciate if it has been primed with the lie”

Discworld book is really good: real science by a mathematician and biologist done by comparison to fiction
Concept of lies to children: NOT really lies, simplifications. Essential ones to reach the next, “more true” step

Disclaimer

- Only say “must” if we mean it, everything else is “should”
- “Should”s are “lies to children” Question them. One day they may cease to be useful. Take care to be sure before breaking them - they’re not said for no reason
- Also have glossaries pdf if terms are unfamiliar. Words in **pale blue** should be found there.

When/if you can explain why something we’ve said you ”should” do doesn’t apply to whatever you’re doing, that’s great, you can break the rule. They’re not universal. They don’t work forever, or in every situation.
However, things we say you “must” do, are basically absolute. It’s very very rarely a good idea to break those rules, so don’t do it.

Ultimate Rule

As researchers, your priority is research. Your code needs to work, and you need to know it works. You need to be able to do it again in ten* years. Anything that does not enhance that goal is decoration

*replace ten with whatever your funder/supervisor/ conscience dictates

This is really a rule.
Any sufficiently large or complex piece of code will have bugs - i.e. sometimes or somehow it doesn’t quite work. The trick is distinguishing the known knowns (bugs you know, understand, and can quarantine away from your work), the unknown knowns (bits you know might not work or shouldn’t be trusted) and the really risky one, the unknown unknowns (bits you don’t even know are suspect or in error). Testing and validation is all about trying to eliminate the last one. Rewriting, refactoring etc are about making it right afterwards.

Planning and Creating Software

Warning

- If your code involves any sensitive data, you really **must** read up on relevant regulations and protocol, and get advice where necessary
- Includes:
  - Personal data
  - Credit card or other monetary data
  - Protected information (e.g. medical data)
  - Safety critical interactions

This is also an absolute rule. Data security is really really hard. For some details on University policy see https://warwick.ac.uk/services/idc/informationsecurity/handling/classifications

Note that, for instance, even our sign-up form for this course is a yellow class, since we collected contact emails. Publicising emails without making it very clear you will do so is not OK; the data listed above is even more protected.

Rules (Must's) for Code

- Create a working code - key thing is to end up with code that works, does what was intended, and is reasonably dependable, i.e. wont break unexpectedly or in hard to detect ways.

- Follow your chosen **language standard** - standards dictate what is valid code and a valid program. E.g. in a valid Fortran program, all variable definitions must go at the top of a function. In a valid C program, all functions must be defined before use. Avoid **undefined behaviour.**

These are the absolute minimum rules. There's only four of them, and the two of them on the next slide even verge into a matter of opinion.
The first one might seem flippant, but it is important. Incomplete code is no use, nor is wonderful code that does the wrong task. Code with hidden limitations can be dangerous. Consider programming a quadratic equation solver for ax^2 + bx + c (https://en.wikipedia.org/wiki/Quadratic_formula). You have to divide by 'a'. Are you sure 'a' is not zero? How are you handling the fact that there are two roots (positive or negative)?
Following a standard seems obvious, but most languages have several versions (F77, F95, F2003; C++98, C++11, C++14, Python 2.6, 2.7, 3 series). Which one to use? As a starting point, C99, C++11, F95 are all well supported. If you don't need more modern features, pick those. In Python 2 vs 3 depends on the version of libraries etc you're using.

Rules (Must's) for Code

- Use some sort of versioning - keep some sort of record of when changes were made (and by whom). Either disciplined dated backups or VCS (see later).

- Validate your code enough - checking and testing is vital, but what "enough" means varies. For a one-off small script, you might just run an example case; for large projects you will want to consider formal testing and validation.

Version control is a topic we'll cover a bit later but key is to keep snapshots of code somehow.
Validation is discussed later too, but 'enough' is very subjective. It's rare, if not impossible, to be able to test every possible input or route through a code. Testing pieces individually helps, but sometimes pieces interact unexpectedly. Try to keep in mind that anything you didn't test for, you don't know, and don't assume testing has magically made your code correct.

Planning Code

1. Are any program sections dependent on others?
2. Where do you need user input?
3. Which sections are likely to be the crucial ones for testing and optimising?
4. Where does your code link to other code such as libraries?
5. Do any parts require research, study or unfamiliar libraries?

Planning before getting to work is a great idea. For big projects, do it on paper and in detail. For small scripts I personally like to rough out the idea in comments or function names before filling in the bodies, and I consider that a plan.
On point 5. this is the point to step back and try prototyping before using something new in a bigger code. Write a simple test script, get a basic understanding of how it works, how fast it is etc.

Libraries

- Often a **library** exists to address (part of) your problem
- Powerful, saves work and avoids errors. But are drawbacks
- Left-Pad - micro dependency hell
- **Dependencies are complexity!** Often gets forgotten
- Be selective!

We're not going to go into detail here, because entire books can and are written about when and how to use libraries. Suffice to say that a well written library is a real advantage, but finding those isn't so easy. And what about an excellent library that just hasn't been updated in years? Is it perfect, or deprecated? Would you know if it had bugs?

Left-pad, since we mention it, is an example of over-reliance on libraries. A very simple task, padding a string with spaces on the left so it had a fixed length, was delegated to a library by many web pages and apps using Node.js. A copyright battle over the name of ANOTHER library by the same author led to him taking down his work, breaking other packages and resulting in an embarrassing domino-effect. This dependency on lots (hundreds?) of tiny packages, each of which might itself depend on other packages, is sometimes called "micro-dependency hell".
DO NOT think of a library as reducing the complexity of your code. All it does is make some of it somebody else's problem, and leave you in charge of keeping up with changes to the library.
At the same time, don't be afraid to use libraries, just be a little selective and try and make sure they're saving you work and issues, not adding a whole new set of their own.

Common Red Flags

- **Magic numbers** - numeric constants in code without comment.
  - What is 86400?
  - How many digits of pi will you need? Don't want to edit every occurrence
- **Global variables** - too many, or the wrong ones.
  - What if a function goes wrong? Is the global in valid state? Was it changed? Partly written (**atomicity**) i.e. nonsense?
- **The god object/class/function** - one thing that tries to do everything.
  - Functions should do their one thing well
  - Dividing into functions too harshly complicates code, kills performance

86400 = 60 * 60 * 24. I.e. it's the number of seconds in 24 hours.

Global variables are **not** evil **if** they're holding some sort of global state. They become a problem if they can be changed all over the place, so it becomes hard to know what value they have.

The other two are pretty self explanatory. A rule-of-thumb for programming is to do one thing, and do it well. This applies to functions, objects, scripts, everything.

However, what 'one thing' means is a matter of taste. Think about validating an online form. Checking an email address might be one thing (and it's much harder than you might think), but checking the entire form is one thing too.

Common Red Flags

- **Golden hammer** - when all you have is a hammer, everything looks like a nail
  - Don’t force favourite language/tool/technique to do everything
  - But don’t spend time learning a new tool if yours is enough
- **Premature optimisation** - optimising your code before it works, or at cost of readability/modifiability/correctness
  - Don’t optimise before code works - doing the wrong thing faster is no use
  - Don’t make choices that make your code hard to make fast enough in the end

Like the God object, the Golden hammer violates the core rule to do one thing, and do it well.

Premature optimisation is slightly tricky. To some people, everything seems premature. Certainly if your code relies on user input, there’s little point making it run any faster than the user can interact.
There’s a slightly odd series of peaks in code runtime annoyances. 30 seconds is fine. An hour is usually fine, you just have to plan slightly. But an hour is plenty of time to do something else. The worst runtimes are 10-20 minutes, and 14-24 hours. The former is too long to just wait for, but not long enough to start something else. With careful planning and a bit of scripting it can be made far less disruptive, but this is extra work and wont always mitigate the problem. The latter means to have the data tomorrow morning or midday, it’s not enough to start it when you leave today. Longer than 24 hours, you just have to accept the waiting, or plan to run things over a weekend.

Common Red Flags

- **Multiple return types** - in e.g. Python
  - Returning dictionary with varying keys “OK” with caution
  - Returning array (many results) or single value (one) generally a bad idea
  - Don’t return strings in some cases, numbers in others
- **Reflection** - allowing your code to read, modify and write itself
  - Very powerful in the right places, can also be dangerous
  - E.g. constructing a string you execute as code. Are you certain that the string is safe? **Never execute user-input!**
  - Constructing object “on-the-fly” can be useful, but don’t know what members it has

If you seem to need multiple return types, for instance, either a number or an error message, it’s usually better to use a class/struct/object containing both items. You may still make mistakes, but they’ll be less weird.

Input sanitation (making safe) is covered a bit in our full notes. Here we just note that it’s a really tricky task in general that has caught out many very good programs and websites, including those who thought they were security conscious. Be very very careful. Note we say “never” here but that is a “should” rule. Sometimes you can and have to do this, but be really careful. Look up “SQL injection attack” or see this XKCD comic https://imgs.xkcd.com/comics/exploits_of_a_mom.png

Project Scoping

- Projects grow to fit the time available for them
- Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law. – Douglas Hofstadter
- Trickiest part (?) of development, years to perfect.
- https://warwick.ac.uk/research/rtp/sc/rse/project_estimator.pdf or https://warwick.ac.uk/research/rtp/sc/rse/project_estimatorwexample. pdf
- Tries to quantify size and complexity based on inputs, outputs and libraries.

You might never need to think much about this, but it can be very tricky. The linked worksheet might help, as might the next slide.

MoSCoW

- Common problem: want everything and the kitchen sink
- Don't know how long things will take
- Have to rank features to get the most out without overrunning
- MoSCoW method Decide for each item/feature, if it:
  - **must** be included, i.e. without it the project is a failure;
  - **should** be included, i.e. it adds real benefit and is strongly desired;
  - **could** be included, i.e. it would be nice to have
  - **wont** be included, i.e. it is not required at this time

This is just one way of tagging features/capabilities, but it's common and simple and well worth knowing about. If you allocate or request enough time to do all the "should"s you have both time in case the "must" list takes longer than expected, and the work on the "could" list to occupy you if it takes less time.

Version Control

Version Control

- Record changes that you make to a file or system of files
- Allows you to keep a log of why/by whom those changes were made
- Allows you to go back through those changes to get back to old versions
- Help deal with merging incompatible changes from different sources
- Similar term “Source Code Management”

One of our sort-of rules was to use some sort of Version Control. Some insist that this means one of the dedicated systems. We are slightly more flexible. It is possible, if you’re conscientious enough, to keep a systematic, time stamped version of code and scripts that does everything you need. But its hard. On the other hand, if you take no care at all version-control wont help you. It can’t.

Why Use Version Control?

- "I didn't mean to do that!"
  - Can go back to before you made edits that haven't worked
- "What did this code look like when I wrote that?"
  - Can go back as far as you want to look at old versions that you used for papers or talks
- "How can I work on these different things without them interfering?"
  - Branches allow you to work on different bits and then merge them at the end

Important note: all of these things are true IF you use your version control system carefully and "properly". These are things the systems allow you to do: they're not in general done for you.
This is true, but you can only go back to things which are "known" to the version control, and most rely on you to tell it when to take a snapshot.

Why Use Version Control?

- "I want a secure copy of my code"
  - Most version control systems have a client-server functionality. Can easily store an offsite backup.
  - Many suitable free services, and can easily set up your own
- "How do I work with other people collaboratively?"
  - Most modern version control systems include specific tools for working with other people.
  - There are more powerful tools to make it even easier too

Version control is not a backup! BUT most systems allow you to go "back in time" so it can protect you from accidentally deleting code from a file.
Version control systems can HELP you back things up, since most systems make it easier to sync multiple copies.

For collaborative *editing* version control is awful. Few, if any, systems are designed to let two people work on one file at the same time. Actual editing systems like Googledocs or Overleaf try and do this, but even they struggle. What version control can do is help keep changes separate, and help combine them together.

Why Use Version Control?

- "My funder* wants me to"
  - More and more funding bodies want code to be managed and made available online
  - Version control is a good way of doing it

* (Or supervisor, PI, Institution etc)

What Version Control Isn't

- Not a backup
  - If you use a remote server are safe against disk failure etc
  - But other people can still wipe out your work
- Not a collaborative editing tool
  - You can merge changes from many people
  - But it is hard work, not intended to handle editing the same files
- Not magic
  - Some language awareness, has to be conservative
  - Wont fix all your problems

Git

- Git is a very popular VCS system
  - Not the only one
  - Github is NOT git and git is NOT github
- Not going to walk through every command
  - See our full Git materials at https://warwick.ac.uk/research/rtp/sc/rse/training/introgit

We're not going to go through all the details of using Git here. We're just going to introduce some of the essential words and direct you to either our materials with examples at https://warwick.ac.uk/research/rtp/sc/rse/training/introgit, the December holiday workshop we'll be running, or the many and varied online resources. There's dozens of places you can get examples and tips, but if you don't have the core ideas in place first it can be very confusing.
Github, by the way, is one remote Git server. It has its own ways of working which aren't inherent to git, and it's not the only option by far. It's just a very popular one.

Key Git Terms

- Git keeps things in a **Repository**
  - Usually keeps ‘diffs’ i.e. a record of changes
  - Slight warning - some backup systems struggle with the large number of files in the hidden `.git` folder
- **Commits** are snapshots of state with unique ID
  - Can go back to them later
  - Can use ID to be certain how things were when you produced some data

You can copy an entire folder, including the hidden `.git’ folder, and you’ll have a new copy of a valid repository. But this can go wrong, so it’s not recommended. Note that Git usually works in increments, so those hidden folders don’t contain copies of your files, they contain information for getting back to a certain state.

Key Git Terms ctd

- **Staging** area - where changes are collected prior to making a commit
  - Used in all sorts of contexts (data, military etc)
  - Make and select changes, **stage** them. Can remove some, add more, edit etc. Then commit once we're ready
  - Staging makes things known to git
- **History** and **log**
  - Git records when commits are made in the history
  - Local history is yours and you may adjust it as you wish
  - Shared history (once you "push" or put things on e.g. Github) should NOT be changed

Preserving the history is one of the key principles of git. It wont stop you messing with history, but it's not something you should ever really do.

Key Git Terms ctd

- To keep work separate, git uses **branches**
  - Separate sets of commits, given a name
  - Can make branches off branches
- Default branch is usually called ‘master’
- Many ways to manage collaborative projects using branches.
  - A popular model is called git-flow ...(https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow)
  - Master or release branch is always in a “good” (working, tested) state

https://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows gives some good examples of different git workflows

Key Git Terms ctd

- Using a remote server, such as Github, Gitlab or something else you **clone** a repository from elsewhere to get a copy of it
- You can also **checkout** branches to see or work on them, or single commits to go to a specific time etc

When you clone a remote, you end up on the master branch. If you want a different branch you have to clone the whole repository and then checkout the branch you want.
If you checkout a single commit, you can end up in the slightly scary sounding “detached HEAD” state. This basically means git doesn’t know what the history of the current state should be. Don’t panic, and see e.g. https://howtogit.net/recipes/getting-out-of-detached-head-state.html for some help.

Key Git Terms ctd

- A **merge** means combining two sets of changes
  - **Fast forward** is the nicest sort of merge, where the destination hasn't changed after the branch that you are merging in, so the changes are just popped into place
- You may also encounter the **rebase**
  - Take some changes and "replay" them onto a new original
- Merge vs rebase is a vexed question. Use whichever you prefer UNLESS you're on somebody else's project, then whichever THEY prefer

Seriously, the merge versus rebase debate is a hot one and those on both sides are vehement that their way is better. We think both have their uses, and the only thing to absolutely NOT do is mix and match styles as you tangle up history too much.

A Brief Intermission

Musical Interlude

- **Grab these slides, our list of bugs (we'll get to this later), our glossaries etc at**

https://warwick.ac.uk/research/rtp/sc/rse/training/introdevshort

Building and Packaging

Simple Build Tools

- For compiled code (C, Fortran etc) first thing you'll meet is build scripts
  - Save typing complicated compile line over and over
- Next step up is a build `system` like Make where you can specify "dependencies" (which file depends on what)
  - Allows recompiling only changed files
  - Can be really nice speedup
  - Allows parallel building
  - Can use for simple install jobs (download and build this, then this ...)

Build Tools

- Also have more complex systems
  - Allow “proper” install
  - Check compilers, libraries etc
  - Require or exclude libraries etc by version
    - Note this can be red-flag - shouldn’t usually need particular version
- Cmake, GNU autotools, Qmake

We wont go into these at all. When you need them, you’ll know, and there are ample guides and tutorials available online.

Installers

- Also have tools to install or share an entire program and its toolchain
- https://easybuilders.github.io/easybuild/
  - SCRTP uses this internally for installs
  - Works smoothly with “module” utility
- For really complicated things might go as far as a Docker or VM image
  - “Container” holding all code, dependencies etc

Probably not going to need this stuff, but it’s useful to have heard of.

Make

- Originated in 1976 after someone wasted a morning debugging a program that he just hadn’t compiled correctly
- Basic Makefile contains list of rules
  - Target - what is to be made
  - Prerequisites - what needs to be made before this target can be. Can be files or other targets
  - Recipe - The command to build this target

You’d be amazed how many indispensable bits of software originated like this in the early days. Sometimes these lead to odd little legacies in style or syntax.

Make

- Rules look like

```
Target : prerequisites
→|       recipe
```

- Note the TAB (→|) character. This is essential. Spaces wont do!
- The first rule is special and is the default rule. It is built when you just type "`make`"
- Can make any rule using "`make {target}`"

Remember the "{}" here means "substitute the correct word in here. You don't need to type them.

Make

- Much more to Make than this - see our slides at https://warwick.ac.uk/research/rtp/sc/rse/training/make for details
- Note that can use Make for other ordering tasks
  - E.g. if data file changed, rerun analysis script
  - Wasn't designed for this - hard to get right
  - We DO NOT recommend
  - Use a tool designed for this if wanted

Python Package Manager

- “Packages” are just code with some specified set of abilities and functions
- Basic instructions on how to turn Python code into a package: https://packaging.python.org/tutorials/packaging-projects/
- Mostly probably going to use other peoples packages

Python Package Manager

- Probably know `pip`
  - Nice easy way to install Python packages
  - Install script and web-repository of package code
  - Can install libraries only for one user (`pip -u`)
- Main alternative `easy_install`
- https://packaging.python.org/discussions/pip-vs-easy-install/
- On SCRTP systems you can install your own packages in user space

On SCRTP desktop, info is at https://wiki.csc.warwick.ac.uk/twiki/bin/view/Desktop2018/PythonUserGuide

User space installs are mostly easy. For instance, with pip supply “--user” Not everything can work this way, but most things just do.

Debugging Code

Bugs

- Definition: error, flaw, failure or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways.” [Wikipedia (Software bug)]
- Not every example of a program “misbehaving” is a bug
- Sometimes there are several valid ways of doing something, or the programmer may have chosen a compromise.
- Compromises should be covered in code documentation. Make sure to consider this in your own code, as you may not remember your reasons in a few months time.

Bugs

- Bugs which you can reproduce are troublesome - bugs which you can't are awful. Not only are you working in the dark to diagnose it, you can never be sure it has gone!
- If you've ever had one fire alarm with a dying battery you can imagine this
  - Beep
  - ..... beep
  - .....
  - beep

Every time you think you hear it, it's gone, and you can't work out where it came from. Bugs like this are a nightmare!

Bug Catalogue

- We put together a rough “catalogue” of sorts of bug
- Copy at https://warwick.ac.uk/research/rtp/sc/rse/training/bugcatalogue.pdf
- Not exhaustive, but worth a read. It also covers a bit about numerics
- Bug spotting comes with experience. As you get to know the symptoms you can often “see” what the problem must be and then you just have to find it

Debuggers

- "**Symbolic" debuggers** understand your code. They know what symbols (variables, functions etc) it contains
- They let you step through the code one line at a time
- Can examine variables as you do
- Can set "**breakpoints**" - run until this point and then stop
- Learn to use them, they are invaluable

GDB is classic example for C/Fortran/C++ code. Python has pdb.
Important note: bad enough errors can crash the debugger, especially pdb. So sometimes you have to turn to other tools, or the method on the next slide.

If you are working with C/C++ or Fortran code called from within python (usually from a library) then you might need to use python extensions for gdb (see https://wiki.python.org/moin/DebuggingWithGdb) so that you can track where a crash in the compiled code was called from your python code.

Print or Caveman Debugging

- Sometimes, especially when working with parallel code, or when remotely debugging something it's not practical to use "proper" debugger
- Old method but it works
- Insert prints. Slowly restrict down to troublesome area and find bug
- Advantage - forces you to think about what is happening and what you expect to
  - Can work better when error is a logical one

Two really important things to remember.
First is practical - code output can be buffered, so if problem is a crash, you might have to take care to flush output to see your prints. See https://blogs.warwick.ac.uk/researchsoftware/entry/pretty_please_print/
Second is a real timesaver - **bisection**. Possibly the most important method you'll ever call on. When trying to find name in phone book, you don't start from the beginning. Flip to somewhere around the middle. Is the target before or after this? Now flip to middle of the correct chunk and repeat. Each time the number of pages drops by half. This is the core idea behind binary searches, several sorts of sorting algorithm and lots of more complicated stuff in data structures. For N items to search through, this way takes log_e(N) flips, rather than N/2 (on average you find your name after flipping through half of the phone book)

Symbolic Debugging

- Using symbolic debugger not much changes except how you work
- Set breakpoint on the failing line
  - Examine state of variables
    - Trace back anything which is wrong
      - Either bisection
      - Or careful stepping
  - Adjust (if possible) bad value
    - Step on and make sure things work

Handy Things

- Features of gdb and family to keep in mind
- Delayed breakpoints - break only on the *n*th time we run this line
- Conditional breakpoints - break only if *x* is true (e.g. i > 0)
- gdb can be attached to a running process - useful with parallel jobs (gdb -p ***PID***)
- Can edit a variable and continue running

Other Tools you Must Know Exist

Memory Errors

- Python, Java have **garbage collector**
  - Once you no longer need objects they get cleaned up
- C, Fortran and C++ are all languages where you manage memory
  - No garbage collector overhead
  - The dreaded **seg faults**
  - Memory leaks

Garbage collectors run periodically and clean up any stuff you no longer have references to. You can't (generally) control when they run, or if they run. They have many potential problems, but those only arise when doing fairly complicated or time critical things.
Languages like C which are used for all kinds of microcontrollers, relying on precise timings and real-time responses cannot be garbage collected like this. More control of memory is needed.
Segfaults, or segmentation faults are a type of memory error. Usually a result of trying to read/write the wrong memory, which usually arises due to an uninitialised reference/pointer. This is not fixed by a garbage collector, but is harder to cause in non memory managing languages, usually because they don't give you the tools to let you access memory so directly.
Memory leaks are when you delete your last reference to some memory, so no longer have any way to access it, or clean it up. Garbage collectors do fix this, but add their own issue - they only clean up when they think it's needed. So you can end up using more memory than needed.

Memory Errors

- Can’t always avoid manually managing memory
  - Using variable length arrays in C (before C99)
  - Fortran since 90/95 has allocatable arrays which help you a lot, but not completely
  - C++ proper use of constructors/destructors and new helps
- Sooner or later need to check for leaks and out of range access

Memory Errors

- Out of range access
  - in Fortran don't forget array-bounds checking
  - in C++ check for or write debug mode with extra checking for classes you rely on
- **Valgrind** memory checker is invaluable
  - Run your compiled program inside it
  - Identifies memory leaks, access errors etc

Valgrind works by replacing all the normal system calls to allocate and free memory with it's own monitored versions. This is great - it can identify leaks, tell you about access out of range etc. Two major caveats and one minor one. First, valgrind is intended to check correctness - so it won't warn you until something has a side-effect. The check-origins flag is very useful for tracking back to the source of the problem. Secondly, since you run inside valgrind, your program can run much slower - you may need to run a cut-down version and this may not show up errors if they only arise in certain setups. Thirdly and more minorly, it takes a bit of skill to understand what it's telling you, so it can be frustrating at first.

Profiling

- We mentioned “premature optimisation” as a red flag
- When it is time to optimise, how do you make sure you’re
    1. targeting the crucial (rate determining) parts
    2. making things faster/more efficient
    3. not breaking other things
- First two are job of a profiler, third is solved by testing and verification

The whole point of profiling is to try and work out which parts of your code are doing all of the work (quite often these are called “hot spots”) and then make those faster. Thinking about it a bit it’s quite obvious why you have to do that. Imagine a program that has two functions, one that takes 9 seconds and one that takes 1 second. Even if you made the 1 second function so much faster that it was literally instantaneous then the code would still take 9 seconds to run. Improving the 9 second function so that it takes 8 seconds would give the same reduction in total runtime and would probably be easier because you only have to speed it up by 12% rather than make it infinitely faster.

Note that even a function that starts being the rate limiting step may cease to be so once you’ve optimised. Imagine a function which accounts for half your total runtime. Make it 2x as fast and your runtime is 75% what it was. Make it 10x as fast and you hit 55%. But now, even if you made it a million times faster you’ll only reach 50.0%. This function is no longer rate limiting. Waste of time to go further. May make code less readable/maintainable/ etc

Profiling

- Profiler sits inside and around your code and monitors all or some of:
  - How many times functions are called
  - Where they're called from
  - How long they take
- Shows where to focus optimisation effort
  - Target functions which make up much of runtime
  - Make them "fast enough"
    - Optimisation means trade-offs in clarity, simplicity etc

Note: just because a function is called a lot doesn't mean you'll gain from optimising it. Might be worth looking at **inlining** - just the function call can be substantial time sink

Parallel Profiling

- For MPI codes there is a free tool called MpiP
  - Replaces MPI calls with its own monitored ones, so can tell you call stats
- For trickier problems there are commercial tools
  - Arm/Allinea Map/DDT/Forge
  - Intel Advisor

The Arm (formerly Allinea) tools are available on some supercomputers and are very handy.
Intel Advisor tells you things about optimising your code like how well it vectorises

Documentation

Self-Documenting Code

- Recall 86400: store in variable seconds_per_day
  - Variable documents itself!
- Function “solve_quadratic(a, b, c) “ pretty obvious
- Function parameter “input_file” pretty obvious
- “seconds”, “quadratic” and “file” insufficient
- logical variable called “flag” unhelpful and redundant
- This is NOT a substitue for “actual” docs

Note first that the quadratic is obvious to me (a physicist) and to many people, because the ax^2 + bx + c quadratic is “almost standard”. “input_file” is pretty clear IF the action of the function is clear (read_file, get_configuration etc) However these may (usually do) have other restrictions. Can ‘a’ be zero in the quadratic? How many roots are returned? What format should “input_file” be? All of these are a task for the “real” docs. What self-documenting code is about is making it easier for you/others to read your code, without having to try and remember what a function does. A parameter called “seconds” may as well be called “bob” in most contexts. If you think this “self-documenting” is a very fancy name for the “giving things useful names” taught in many beginner programming courses, you’re not wrong, this is just the equivalent for larger programs, where you might have classes, namespaces and many other ways to be helpful.

Documenting Code

- For libraries and helper functions, interface docs vital
  - What function/subroutine does
  - What each parameter means
    - In Python and similar, this is the perfect place to mention if parameters should have a particular type/class
  - What is returned
    - Again, in Python type languages, what type it is
- An example call, ideally with context

This is the stuff you find in library docs. E.g. https://docs.scipy.org/doc/numpy/reference/generated/numpy.core.defchararray.capitalize.html#numpy.core.defchararray.capitalize

Documenting Code

- Often nice to have interface docs both in the code, and separated out
  - Avoids drift - if you change the code, docs right there are easy to change at the same time
  - Easier to skim and check parameter names etc are correct
- Many tools will extract your docs and make HTML/ Tex etc from them
- Some also check signatures in docs against code

Drift between code and comments is always a bad thing, and is one reason why good code comments are about the “general what” and the “why” and not the details of “how”. The idea of self-documenting code is all about letting the code itself show what is happening, and making that easier to understand. That’s also the reason why we suggest leaving things like equations in a form matching a paper, even when this has minor inefficiencies, rather than optimising but reducing clarity.

Generally, “what” a block of code is doing should never change - but how it does it regularly can. “what” a function does should never change, but which of several equivalent methods it uses also can. This is also why “interface” docs are quite brief, and should usually omit any details other than limitations on parameters etc. More detailed docs, sometimes called “implementation” docs, are also really useful. Opinion varies whether these must be separate. I favour putting them in, but with a tag/ highlight/something to distinguish them.

Documentation Packages

- Sphinx widely used for python, JS
- JSDoc, Javadoc etc
- Doxygen great for C++, good for Fortran
  - Outputs HTML or LaTeX among other formats
- Many many more exist - see https://en.wikipedia.org/wiki/Comparison_of_documentation_generators
- Also several online services to build/host. E.g. readthedocs.org

(Most) options can tell you what things aren't documented (including function params), which is really useful. Can also generate class-hierarchy diagrams, and tell you some other things about interconnections. This relies on them parsing the actual source code though, so the ones that work this way can only handle a restricted set of languages. Others read only specially formatted comments - these work with any language but don't have those nice features.

Code Licensing and Sharing

Licenses

- Sharing code with fellow researchers:
  - Put your name/date at the start of your files
- If you're using other people's libraries:
  - Check if they impose restrictions
  - "Download *this* from *here* to use"
- If distributing your code online, for example using Github, consider choosing a proper license.

The simplest way to use libraries from other people is just to specify "Download this from here to use" rather than including their code. This doesn't get around their license, but would mostly be considered a fair use, as long as you attribute to them. You're not taking credit for their code, and crucially, you're not accidentally distributing some copy which may not be up-to-date or canonical.

Licenses

- https://choosealicense.com/
  - Describes most of the options
  - Helps you choose based on restrictions
- Primary concern is whatever your funder/institute demands you do
  - They may wish to retain rights
  - May require fully-open source
- Otherwise simplest option - protect yourself from unintended errors
  - https://choosealicense.com/licenses/mit/

Mostly you shouldn’t need to worry about licensing much. Either your funder has restrictions which you should know about, or they don’t. In the latter case, all you need to consider is whether you want to use something like the MIT license, or text to the effect of “By using this code you agree that anything it does, unintended or otherwise, is your own responsibility”. This protects you in the unlikely event that your code messes up somebody else’s work or computer.

Sharing Considerations

Meme created by Imgflip. Image sequence from the film https://en.wikipedia.org/wiki/Despicable_Me_(franchise)

Sharing Considerations

- Tempting to “release” all your code
  - Are you actually happy with somebody using it?
  - Does it work?
    - Is it free of silent failure cases?
    - Does it handle errors?
  - Is it documented?
    - Especially any assumptions!
  - Are you going to fix any issues you find?
    - Or at very least add them to documentation.

Be very careful if the answers to any of these are “no”. You don’t want unfinished or broken code getting away from you.

NOTE: this doesn’t apply to just popping code on somewhere like Github, only when you start presenting it as something for others to use. E.g. giving it a name, putting it on a Python repo, giving usage instructions

Closing Notes

Things to Take Away

- Probably very obvious that we've only barely touched things
- Lots of links for further reading to come at https://warwick.ac.uk/research/rtp/sc/rse/training/introdevshort
- Pick the things you're interested by or need
- Let us know if there's anything missing
  - rse@warwick.ac.uk

Reverse Favour

- This is an experiment for us
- We're hoping sometime in the next months/years you'll have some of those "aha" moments
  - If you do, maybe let us know so we know which bits worked
  - Don't hesitate to tell everybody where you heard about it!
- Going to leave the feedback for a while - we may email in a few months with a link