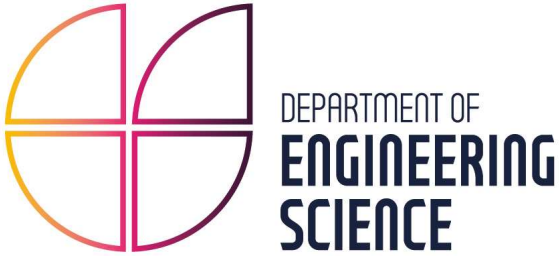


An Introduction to HPC and Scientific Computing

Lecture four: Using repositories and good coding practices.

Karel Adamek and Yishun Lu

Oxford e-Research Centre,
Department of Engineering Science

Overview

In this lecture we will learn about:

- The elements of good practice in writing code
- Some of the tools that will help us write good code
- The very basics of Revision Control

Fundamentals

Good practice is context dependent. No matter what approach you take,

- Be consistent
- Communicate what you've done

What do we want out of good software?

- Usability
  - Easy to learn and use
  - Has the features users want
- Correctness and reliability
  - Gives the right answer
  - Behaves as expected
  - Few bugs
- Maintainability
  - Easy to fix
  - Easy to extend
- Portability
  - Works everywhere, for an appropriate value of everywhere
- Efficiency
  - As fast as it needs to be

Usability – writing code that people want to use

- The purpose of a code is to address a real need that users have
- That might include
  - Having novel features
  - Running faster than other codes
  - Being easier to use or access

Don’t assume – talk to your users before you start writing code!

- “But I don’t have any users!”
  - Considering an outside perspective will still help you write better code
  - You never know when your code might escape into the wild
- Don’t reinvent the wheel

Usability – writing code that people want to use

- If your program doesn't solve the problem it is supposed to solve it's not much use
- Similarly if it does but you can't work out to use it, again it is not much use!

Talk to your users at every stage of development!

- Ask
  - What need do they have that they can't currently solve?
  - Are they struggling to learn or use any part of the code?

Usability – writing code that people want to use

- The instructions for how to use your code are part of a working code
- There are many types of documentation for users
  - User guides
  - Installation guides
  - Quick start guides
  - Tutorials

You can ask for user feedback on your documentation too!

Correctness and reliability

- Code that gives the wrong answer is not useful code
- Code that may or may not give the right answer is not useful code

Assume code with no tests is incorrect code

- Spend some time at the start of your project thinking about how you could verify your code
  - Unit tests – test the outputs of individual functions on a fine grained level
  - Integration tests/science tests – big picture tests of final outputs
  - Sanity checks – test that certain properties hold true
  - Regression tests – test that nothing has changed since a known working version. Often used to compare parallel and serial versions
- Ask your users what would convince them the code is correct

Using the compiler to catch bugs early – warnings

- If it generates a warning fix it
- Use flags on the compiler to generate additional useful warnings

```c
int main(void){

    int a = 3;
    int b;

    printf("a = %d, b = %d\n", a, b);

}
```

```
[oerc0113@login11(arcus-b) ~]$ gcc warnings.c
warnings.c: In function 'main':
warnings.c:6: warning: incompatible implicit declaration of built-in function 'printf'
[oerc0113@login11(arcus-b) ~]$ gcc -std=c99 -Wall -Wextra -pedantic warnings.c
warnings.c: In function 'main':
warnings.c:6: warning: implicit declaration of function 'printf'
warnings.c:6: warning: incompatible implicit declaration of built-in function 'printf'
warnings.c:6: warning: 'b' is used uninitialized in this function
```

Using the compiler to catch bugs early – run time checks

- Can you spot the bug in the following code?

```c
#include <stdio.h>

void zero_array(int [], int);

int main(void){

    int a[10];
    int i;

    zero_array(a, 10);

    printf("%d\n", a[9]);
}

void zero_array(int a[], int n){
    int i;

    for (i=0; i<=n; i++){
        a[i] = 0;
    }
}
```

Catching bugs at compile time – warnings

- Using the Intel Compiler (2017 or later)

```
[oerc0113@login11(arcus-b) ~]$ module load intel-compilers/2017
[oerc0113@login11(arcus-b) ~]$ icc -check-pointers=rw -debug bounds.c
[oerc0113@login11(arcus-b) ~]$ ./a.out
CHKP: Bounds check error ptr=0x7fff67abb5f8 sz=4 lb=0x7fff67abb5d0 ub=0x7fff67abb5f7 l
oc=0x400949
Traceback:
    at address 0x400949 in function zero_array
    in file /home/oerc0113/bounds.c line 19
    at address 0x400814 in function main
    in file /home/oerc0113/bounds.c line 10
    at address 0x3d9e21ed5d in function __libc_start_main
    in file unknown line 0
    at address 0x4006f9 in function _start
    in file unknown line 0
0
CHKP Total number of bounds violations: 1
```

- Getting array and pointer indexing wrong is an incredibly common error in C
- When developing use many compilers, they have different diagnostic capabilities

Tips to avoid bugs

- Validate inputs

```c
double convertInchesToCm(double inches){
    if (inches < 0){
        printf("Error, length must be nonnegative\n");
        exit(1);
    }
    return inches * 2.54;
}
```

- Check return values of functions

```c
fp = fopen("filename.txt", "r");
if (fp == NULL) {
    printf("Error, could not open file\n");
    exit(1);
}
```

- Isolate parts of your program that do different things from each other – more on this later

Tools for ensuring code correctness

- Compilers (eg gcc, clang, icc)
  - Warnings
  - Bounds checking
- Debuggers (eg gdb, Arm DDT, TotalView)
  - Pause the program at a specified point or when the program throws a runtime error
  - Run the program a single line at a time
  - Inspect the values of variables
- Memory checkers (eg Valgrind)
  - Bounds checking
  - Memory leaks
- Unit testing frameworks (eg GoogleTest, Check)
  - Simplify the bookkeeping of creating unit tests
- Continuous integration frameworks (eg Jenkins, Travis CI, GitLab CI)
  - Automate regularly running unit tests on different platforms

Maintainability

- You WILL want to modify your code
- You WON’T remember what it did 6 months after you wrote it
- Time spent in design and documentation will pay dividends

Be consistent

- There is no one true variable naming convention
- There is no one true project directory structure
- There is no one true way to indent code
- It is much more important for a given project to pick a reasonable one and then BE CONSISTENT

Some questions to ask before you start

- What do users need from the code?
- How will you prove the code is correct?
- How will users interact with the code?
  - Inputs and outputs
  - Do you need a Graphical User Interface?
- Style
- Structure
  - What functions will you need?
  - What data structures will you require?

A bit of thought at the beginning can save a lot of work in the end

Have a basic naming convention

- Choose one of underscores or camel case for variables and function names
- Use similar names for variables that do similar things
- Common conventions:
  - `i, j, k` for loop variables
  - Start with `n` for variables that refer to the number of something
  - Start with `is` for binary decisions

```
vars_should_all_look_like_this

orLikeThis

ObjectsStartWithACapitalLetter
```

```
if ( is_valid ){
 for( i=0; i<nEls; i++ ){
   for( j=0; j<nEls; j++ ){
      a[ i ][ j ] = i + j;
   }
   b[ i ] = a[ i ][ 0 ]
 }
}
```

Indent your code

- Indentation helps you identify where control structures (for, if, etc.) start and end
- It also help you identify which curly brace corresponds to which control structure
- Your Editor should help you do this
- Some editors can mangle tabs!

```
for( i=0; i<10; i++ ){
    for( j=0; j<10; j++ ){
        a[ i ][ j ] = i + j;
    }
    b[ i ] = a[ i ][ 0 ]
}
```

KISS - Keep it Simple, Stupid

- A real line submitted to Stackoverflow

```
sfs=(n*vs)**2/1.49**2*((20+2*ys)/20/ys)**(4/3) v(j,i+1)=4.905*(sqrt(sqrt(ys)*s0**2*dt**2-
2*sqrt(ys)*s0*dt*(sfs*dt-0.1019368*(vs-3.13209195*sqrt(ys)))+sfs**2*sqrt(ys)*dt**2-0.2038736*sfs*(vs-
3.132092*sqrt(ys))*sqrt(ys)*dt+0.0065092*(q((i+1)*dt)+1.596377*(vs**26.2641839*vs*sqrt(ys)+9.81*ys)*s
qrt(ys)))+ys^(1/4)*(s0*dt-sfs*dt+0.101937*(vs-3.132092*sqrt(ys))))/ys**(1/4)
```

KISS and functions

- Break large blocks of code into multiple functions
  - Rule of thumb: 1 page per function
- When you find yourself writing the same thing over and over, turn it into a function
- Use a consistent naming convention for functions
  - A common one is “verb-noun”

```
float calc_circle_area(const float r){

   /* This function calculates
      the area of a circle of radius r
   */

   float area;
   const float pi = 3.1415927;

   area = pi * r * r;

   return area;
}
```

Make your code modular

- Modularity means isolating parts of your code that do different things from each other
- Advantages
  - Makes code easier to extend
  - Makes code easier to debug
- Object orientation (which you will see in C++, Python etc) forces you to do this
- However, you should follow this general practice in all languages
  - Group functions that do the same thing into one file
  - One function should do one thing

Wherever possible keep functions pure

- A Pure function is one whose result depends purely on the values of the parameters supplied to it
- Pure functions are easy to ‘unit test’
  - write a main program that calls it with an appropriate set of parameters

```
float calc_circle_area(const float r){

   /* This function calculates
      the area of a circle of radius r
   */

   float area;
   const float pi = 3.1415927;

   area = pi * r * r;

   return area;
}
```

Impure functions are difficult to debug; avoid global variables

- How to debug if `result` is wrong?
- Need to work out the value of `magic`, and it is a global variable it could be set anywhere in the code
- If you do use global variables
  - Keep them to a minimum
  - Best keep them constant, e.g. pi is fine as a global

```
float calc_something( float s ){
    float result;
    if( magic == 0 ) {
        result = 3;
    }
    else {
        result = another_function();
    }
    return result;
}
```

Documentation

- As for users, there are many forms of documentation aimed at developers
  - Comments
  - High level developer guides
  - Style guides and other contribution guides
- Comments should be useful
  - Good comment:

```
// Update the charge density
```

  - Bad comment

```
i++; // Increment i
```

If stuck for time, comment the parts of the code that gave you problems

If you use best practices, you get some documentation for free

- Use sensible variable names
- Avoid magic numbers
- There are automated tools (eg Doxygen) for turning comments into documentation
  - Requires comprehensive comments describing variables, functions, files, etc
- The process of version control can serve as documentation – more on this later

A good way to learn maintainability

- Look at open source codes and see what works!
  - Example in the practical
- Look at style guides for inspiration
  - eg Python PEP 8 (https://www.python.org/dev/peps/pep-0008/),
  - Google C++ style guide (https://google.github.io/styleguide/cppguide.html)
- Let other people see your code.
  - Everyone is scared to do it. It's still worth it!

Tools to help with maintainability

- Documentation e.g. Doxygen
  - Automatically generate documentation from the comments in your code
- Editors e.g. vim, emacs, nano
  - Useful for consistent code layout, and can help find some bugs via e.g. syntax highlighting
- IDE's (integrated development environments) e.g. Eclipse
  - Combine editors, debuggers, project management and more
- Revision Control Systems – more on this later

Portability

- Standards are one of the main ways to help ensure your code will work in as many places as possible
- C standards: C89, C99, C2011
- Note by default few compilers are standard compliant
  - They have non-portable extensions
  - Use flags to enforce standards checking:

```
gcc -std=c99 example.c
```

- Note C and C++ are not completely compatible
  - Gotchas include treatment of complex number types
- Note CUDA (which we will see in the GPU section) is similar but not identical to C++

Efficiency

- There is often a maintainability and portability cost to efficiency
- Make your code ‘fast enough’ but no faster
  - If overnight is good enough 4 hours or 8 hours makes no difference
  - But the weather forecast has to be there by tomorrow!
- You can use a *profiler* to look at efficiency problems
  - gprof is a simple free one, and you will look at nvvp in the CUDA exercises
- But remember if you get the wrong answer it doesn’t matter how fast it runs
  - Get it right and then, and only then, get it fast
  - Correctness trumps efficiency every time

Revision control systems (version control)

- A revision control system (eg git) keeps snapshots of your code throughout the history of a project
- It is often combined with a hosting platform (eg github) for storing your code
- Uses that everyone can take advantage of
  - Back up your code
  - Revert to a previous working version when something breaks
  - Make your code public
- For larger projects with teams
  - Give everyone access to an agreed ‘master copy’ of the code
  - Merge in multiple changes to the code from people working on it at the same time
  - Isolate experimental changes to the code from the master copy, and merge them in when required

Git terminology

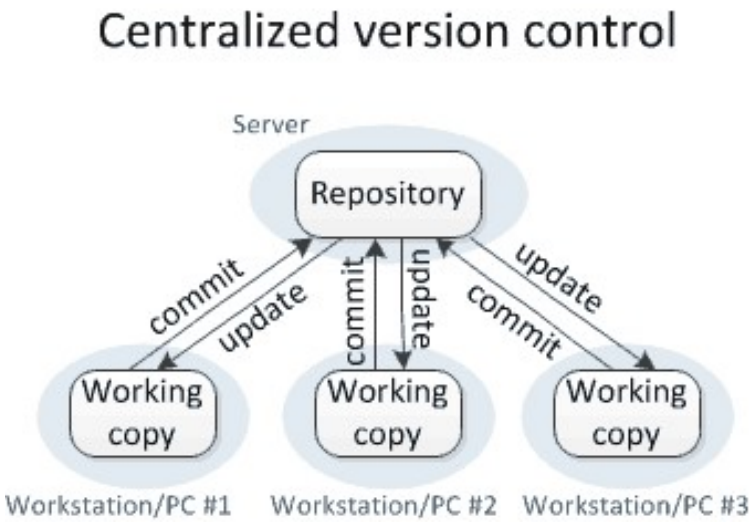
- The folder containing code to be versioned is a ‘repository’
- Snapshots in time are ‘commits’
- Different version of the code that exist at the same time are ‘branches’
- The agreed master copy of the code is called the ‘master branch’

Get the most out of your revision control system

- Make commits often
- As with functions, one commit should do one thing
- Add descriptive commit messages
- Add documentation to your repository
  - Common practice to include an introductory README.md in the root directory
- Use the tools for project management that come with Github
  - Particularly ‘issues’, which are todos such as bug fixes and new features that can be logged by users and developers
  - Project boards
- Include a licence on your code
  - What is allowed may depend on departmental policy – check!
  - Common simple, open source ones:
    - BSD - https://en.wikipedia.org/wiki/BSD_licenses
    - MIT - https://en.wikipedia.org/wiki/MIT_License

Centralised Revision Control Systems

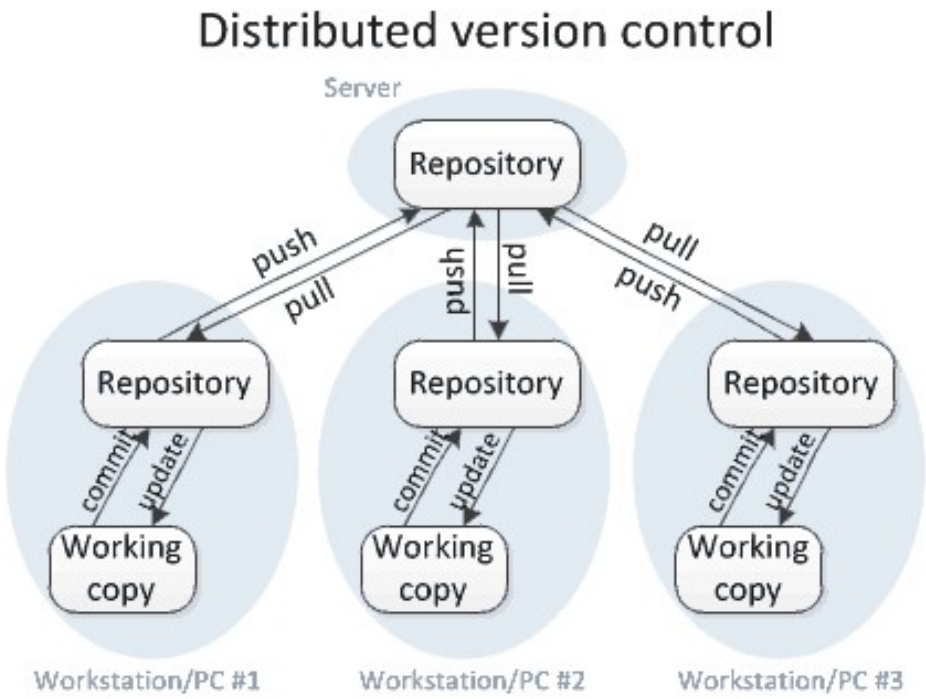
- Note the repository could be accessible by just you, or members of a team
- The simplest model is a centralised version control system



https://blog.inf.ed.ac.uk/sapm/2014/02/14/if-you-are-not-using-a-version-control-system-start-doing-it-now/

Git Model

- Git has a slightly more complicated model
  - It is a distributed revision control system
- Developers get their own local repository to work with, and then a shared external repository
- One of the reasons git is popular is there are many places to store the shared repository, e.g. github, which also makes it easy to distribute code
  - See the practical



Further reading

You only learn this stuff by doing it!
But some suggestions:

- ARC and Archer provide a number of courses, some of which cover some of the material covered here
- Consider attending one of the *Software Carpentry* courses
  - https://software-carpentry.org/
  - https://www.software.ac.uk/software-carpentry
- A few introductions to revision control and git:
  - https://blog.inf.ed.ac.uk/sapm/2014/02/14/if-you-are-not-using-a-version-control-system-start-doing-it-now/
  - https://betterexplained.com/articles/a-visual-guide-to-version-control/
  - https://try.github.io/levels/1/challenges/1
  - https://git-scm.com/book/en/v2 - an on line book on git
  - http://www-cs-students.stanford.edu/%7Eblynn/gitmagic/ - another book on git

What have we learnt?

We have learnt about

- The very basics of writing good quality code
- The names, and in some cases basic use, of some of the tools that software developers use
- A bit about revision control

In the next lecture…

We shall look further into C!