

An Introduction to HPC and Scientific Computing

Lecture four: Using repositories and good coding practices.

Karel Adámek and Yishun Lu

Oxford e-Research Centre,
Department of Engineering Science

Thanks to Ania Brown and Ian Bush for the materials

Overview

In this lecture we will learn about:

- Fundamentals of good practice in writing code and software
- Some of the tools that will help us write good code
- The basics of Revision Control

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 consistently
 - Behaves as expected
 - Few bugs
- Maintainability
 - Easy to fix
 - Easy to extend
- Portability
 - Works everywhere where it needs to work
- 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 (tidy data format)

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

- “But I don't have any users!”
 - You are a user as well.
 - Considering an outside perspective will still help you write better code
 - You never know when your code might escape into the wild

Usability – keep it focused

- 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 how 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 – user documentation

Documentation is part of a working code

There are many types of documentation for users

- User guides – is detailed description of software capabilities
- Installation guides – how to install your software (very important)
- Quick start guides – contains minimal starting example of how to use your software
- Tutorials – describes steps “how to do X”

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

Bugs - Using the compiler to catch bugs early

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

```
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
```


Make the code resilient

- Validate inputs

```
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

```
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

- Maintainability is about how much time (money) you need to spend on fixing or extending the code
- 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

Code style – how to make code readable

There is no one true:

- variable naming convention
- project directory structure
- way to indent code

It is much more important for a given project to pick a reasonable one and then

BE CONSISTENT

Communicate what you've done

Write aesthetically pleasing code!

Have a basic naming convention

- Variable names must make sense!
- 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 helps you identify the code blocks and scope
- 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 ]  
}
```

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

Do not write ugly code

- 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)
```

Functions

- Break large blocks of code into multiple functions
 - Rule of thumb: 1 page per function
- Function helps you contain bugs
- When you find yourself writing the same thing over and over, turn it into a function
- Function should do one thing
- 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;  
}
```


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 for developers

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

- Use sensible variable and function names
 - What makes more sense? `nEventsPerSecond` or `nEPS`
- Avoid global variables
- 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

Learn from others

- 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, notepad++
 - Useful for consistent code layout, and can help find some bugs via e.g. syntax highlighting
- IDE's (integrated development environments) e.g. Eclipse, Visual Studio
 - Combine editors, debuggers, project management and more
- Revision Control Systems – more on this later
- Formatting: clang (clang format), Uncrustify (<https://github.com/uncrustify/uncrustify>)

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 little 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 Nsight Compute/Systems in the CUDA
- 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

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?
 - How does input and output data looks like?

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

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'

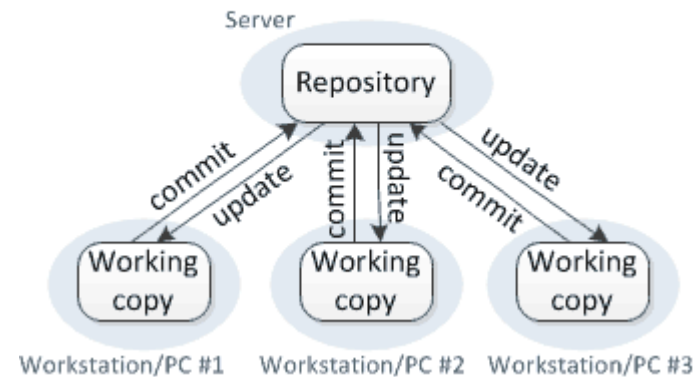
Get the most out of your revision control system

- 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:
 - Creative Commons - <https://creativecommons.org/>
 - BSD - https://en.wikipedia.org/wiki/BSD_licenses
 - MIT - https://en.wikipedia.org/wiki/MIT_License
 - GPL - <https://www.gnu.org/licenses/gpl-3.0.en.html>

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

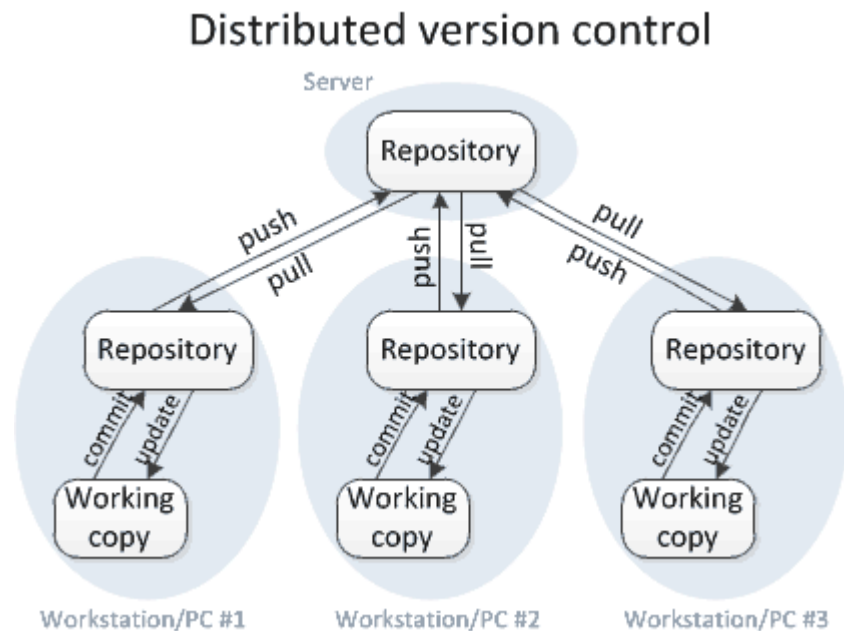
Centralized version control



<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!