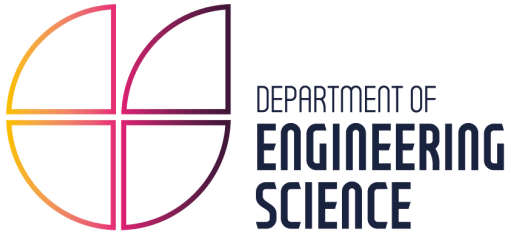


An Introduction to HPC and Scientific Computing

Lecture four: Using repositories and good coding practices.

Ania Brown
Ian Bush

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?
  - What tests would help them trust the results of the code?
- Document your code
  - You can ask for user feedback on documentation too!

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

Use tools to make life easier

- The compiler
- Debuggers
  - Work out while a program has crashed
  - While running pause the program at a specified point
  - While running enquire the value of variables
  - While running modify the value of variables
  - Run the program a single line at a time (step through the code)
  - Pause the program if a given variable is modified
  - Do all the above run time checks dependent on a condition
  - And many more
- Memory checkers
- Unit testing frameworks
- Continuous integration frameworks

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

Design – think before you start!

- How will you prove the code is correct?
- What do users need from the code?
- How will users interact with the code?
  - Inputs and how outputs will be handled
  - Small projects files are enough, but for many programs at some point a GUI becomes desirable
- Structure
  - What functions will you need
  - What inputs does it require
  - What outputs does it provide
  - What data structures will you require
- Especially for larger projects a bit of careful thought at the beginning can save a lot of thought in the end

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 ident code
- It is much more important for a given project to pick a reasonable one and then BE CONSISTENT

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
- And don't use tabs to do it!

```
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
- Particularly if you find yourself writing the same thing multiple times
- Ideally each function should be no more than “1 page long”
  - Lets you see it all in one go
  - One function does one thing
- Use a consistent naming convention for functions
  - A common one is “verb-noun”
  - E.g. `Calculate_Potential_Energy`

```
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 putting the parts of your code that do different things in different places
- Think of interchangeable ‘modules’ that can be swapped in and out
- 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, simply putting code with different purposes in different files as you can do in c is often enough

Wherever possible keep functions pure

- A Pure function is one whose result depends purely on the values of the parameters supplied to it

```
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

- Imagine the function to the right is causing problems
- You would have to work out the value of `magic`, and it is a global variable it could be set anywhere in the code
- The related lesson here is *avoid global variables*
  - If you you do use global variables
    - Keep them to a minimum
    - Best keep them constant, e.g. pi is fine as a global
    - At worst set once in a well defined place and never again modified, only read

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

Pure functions are easy to develop and test

- A function that is pure is easy to test
  - Simply write a main program that calls it with an appropriate set of parameters
- This makes it easy to develop
  - Make all your functions pure, write a testing program, test it to hell and back, move to developing the next function
- Such an approach is called *unit testing*
  - *Unit testing frameworks* exist to help with this

```
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;
}
```

Documentation

- Again, there are many forms of documentation
  - Inline documentation of variables, functions, files etc
  - Comments
  - High level developer guides
  - Installation guides

- If you use best practices, you get some documentation for free
  - Use sensible variable names
  - Avoid magic numbers
  - Avoid hard coded paths
  - Version control as documentation — commits, pull requests, issues

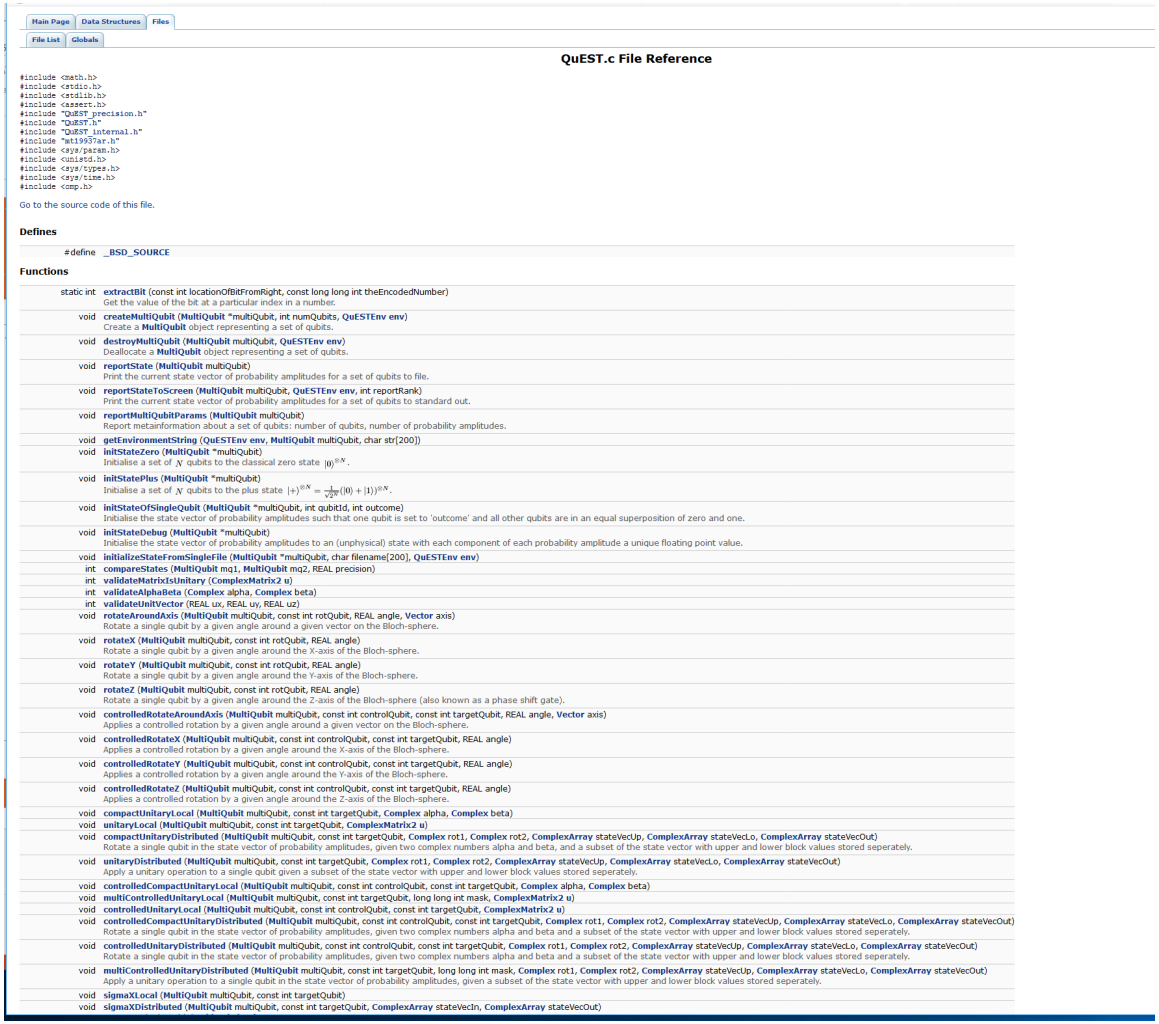
If stuck for time, comment the parts of the code that gave you problems to write

Comments and self documenting code

- Comments should be useful
  - Good comment:
    ```
    // Update the charge density
    ```
  - Bad comment
    ```
    i++; // Increment i
    ```
- Code should as far as is possible be self-documenting
  - Use meaningful variable and function names
    - Call variables by *what* they are, not *how* they are to be used
  - Use white space to break the code into logical blocks

```
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;
}
```

Doxygen



Main Page | Data Structures | Files

File List | Globals

QuEST.c File Reference

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "QuEST_precision.h"
#include "QuEST.h"
#include "QuEST_internal.h"
#include "mt19937ar.h"
#include <sys/param.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>
#include <omp.h>
```

Go to the source code of this file.

Defines

#define _BSD_SOURCE

Functions

static int **extractBit** (const int locationOfBitFromRight, const long long int theEncodedNumber)
Get the value of the bit at a particular index in a number.

void **createMultiQubit** (**MultiQubit** *multiQubit, int numQubits, **QuESTEnv env**)
Create a **MultiQubit** object representing a set of qubits.

void **destroyMultiQubit** (**MultiQubit** multiQubit, **QuESTEnv env**)
Deallocate a **MultiQubit** object representing a set of qubits.

void **reportState** (**MultiQubit** multiQubit)
Print the current state vector of probability amplitudes for a set of qubits to file.

void **reportStateToScreen** (**MultiQubit** multiQubit, **QuESTEnv env**, int reportRank)
Print the current state vector of probability amplitudes for a set of qubits to standard out.

void **reportMultiQubitParams** (**MultiQubit** multiQubit)
Report metainformation about a set of qubits: number of qubits, number of probability amplitudes.

void **getEnvironmentString** (**QuESTEnv env**, **MultiQubit** multiQubit, char str[200])

void **initStateZero** (**MultiQubit** *multiQubit)
Initialise a set of N qubits to the classical zero state $|0\rangle^{\otimes N}$.

void **initStatePlus** (**MultiQubit** *multiQubit)
Initialise a set of N qubits to the plus state $|+\rangle^{\otimes N} = \frac{1}{\sqrt{2}^N}(|0\rangle + |1\rangle)^{\otimes N}$.

void **initStateOfSingleQubit** (**MultiQubit** *multiQubit, int qubitId, int outcome)
Initialise the state vector of probability amplitudes such that one qubit is set to 'outcome' and all other qubits are in an equal superposition of zero and one.

void **initStateDebug** (**MultiQubit** *multiQubit)
Initialise the state vector of probability amplitudes to an (unphysical) state with each component of each probability amplitude a unique floating point value.

void **initialiseStateFromSingleFile** (**MultiQubit** *multiQubit, char filename[200], **QuESTEnv env**)

int **compareStates** (**MultiQubit** mq1, **MultiQubit** mq2, REAL precision)

int **validateMatrixIsUnitary** (**ComplexMatrix2 u**)

int **validateAlphaBeta** (**Complex** alpha, **Complex** beta)

int **validateUnitVector** (REAL ux, REAL uy, REAL uz)

void **rotateAroundAxis** (**MultiQubit** multiQubit, const int rotQubit, REAL angle, **Vector** axis)
Rotate a single qubit by a given angle around a given vector on the Bloch-sphere.

void **rotateX** (**MultiQubit** multiQubit, const int rotQubit, REAL angle)
Rotate a single qubit by a given angle around the X-axis of the Bloch-sphere.

void **rotateY** (**MultiQubit** multiQubit, const int rotQubit, REAL angle)
Rotate a single qubit by a given angle around the Y-axis of the Bloch-sphere.

void **rotateZ** (**MultiQubit** multiQubit, const int rotQubit, REAL angle)
Rotate a single qubit by a given angle around the Z-axis of the Bloch-sphere (also known as a phase shift gate).

void **controlledRotateAroundAxis** (**MultiQubit** multiQubit, const int controlQubit, const int targetQubit, REAL angle, **Vector** axis)
Applies a controlled rotation by a given angle around a given vector on the Bloch-sphere.

void **controlledRotateX** (**MultiQubit** multiQubit, const int controlQubit, const int targetQubit, REAL angle)
Applies a controlled rotation by a given angle around the X-axis of the Bloch-sphere.

void **controlledRotateY** (**MultiQubit** multiQubit, const int controlQubit, const int targetQubit, REAL angle)
Applies a controlled rotation by a given angle around the Y-axis of the Bloch-sphere.

void **controlledRotateZ** (**MultiQubit** multiQubit, const int controlQubit, const int targetQubit, REAL angle)
Applies a controlled rotation by a given angle around the Z-axis of the Bloch-sphere.

void **compactUnitaryLocal** (**MultiQubit** multiQubit, const int targetQubit, **Complex** alpha, **Complex** beta)

void **unitaryLocal** (**MultiQubit** multiQubit, const int targetQubit, **ComplexMatrix2 u**)

void **compactUnitaryDistributed** (**MultiQubit** multiQubit, const int targetQubit, **Complex** rot1, **Complex** rot2, **ComplexArray** stateVecUp, **ComplexArray** stateVecLo, **ComplexArray** stateVecOut)
Rotate a single qubit in the state vector of probability amplitudes, given two complex numbers alpha and beta, and a subset of the state vector with upper and lower block values stored seperately.

void **unitaryDistributed** (**MultiQubit** multiQubit, const int targetQubit, **Complex** rot1, **Complex** rot2, **ComplexArray** stateVecUp, **ComplexArray** stateVecLo, **ComplexArray** stateVecOut)
Apply a unitary operation to a single qubit given a subset of the state vector with upper and lower block values stored seperately.

void **controlledCompactUnitaryLocal** (**MultiQubit** multiQubit, const int controlQubit, const int targetQubit, **Complex** alpha, **Complex** beta)

void **multiControlledUnitaryLocal** (**MultiQubit** multiQubit, const int targetQubit, long long int mask, **ComplexMatrix2 u**)

void **controlledUnitaryLocal** (**MultiQubit** multiQubit, const int controlQubit, const int targetQubit, **ComplexMatrix2 u**)

void **controlledCompactUnitaryDistributed** (**MultiQubit** multiQubit, const int controlQubit, const int targetQubit, **Complex** rot1, **Complex** rot2, **ComplexArray** stateVecUp, **ComplexArray** stateVecLo, **ComplexArray** stateVecOut)
Rotate a single qubit in the state vector of probability amplitudes, given two complex numbers alpha and beta and a subset of the state vector with upper and lower block values stored seperately.

void **controlledUnitaryDistributed** (**MultiQubit** multiQubit, const int controlQubit, const int targetQubit, **Complex** rot1, **Complex** rot2, **ComplexArray** stateVecUp, **ComplexArray** stateVecLo, **ComplexArray** stateVecOut)
Rotate a single qubit in the state vector of probability amplitudes, given two complex numbers alpha and beta and a subset of the state vector with upper and lower block values stored seperately.

void **multiControlledUnitaryDistributed** (**MultiQubit** multiQubit, const int targetQubit, long long int mask, **Complex** rot1, **Complex** rot2, **ComplexArray** stateVecUp, **ComplexArray** stateVecLo, **ComplexArray** stateVecOut)
Apply a unitary operation to a single qubit in the state vector of probability amplitudes, given a subset of the state vector with upper and lower block values stored seperately.

void **sigmaXLocal** (**MultiQubit** multiQubit, const int targetQubit)

void **sigmaXDistributed** (**MultiQubit** multiQubit, const int targetQubit, **ComplexArray** stateVecIn, **ComplexArray** stateVecOut)

UNIVERSITY OF OXFORD

Oxford e-Research Centre

A good way to learn maintainability

- Look at open source codes and see what works!
  - Example in the practical
- Let other people see your code.
  - Everyone is scared to do it. It's still worth it!

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
- The compiler is a great tool to help you with standards compliance
  - Note by default few compilers are standard compliant
    - They have non-portable extensions
- In fact the compiler is a very useful tool in general through programming, learn to use it!

Efficiency

- We don't really have time to address efficiency here
- But note I said "fast enough"
  - If over night 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, too many people forget this

A summary of development tools

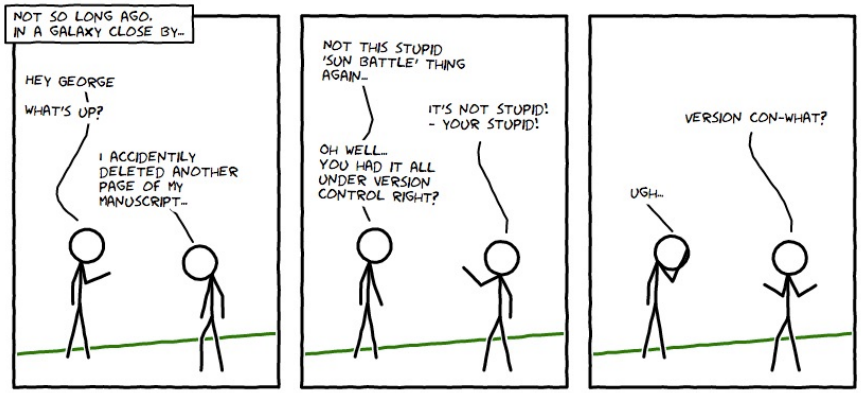
- Editors e.g. emacs, nano
  - Useful for consistent code layout, and can help find some bugs via e.g. syntax highlighting
- Compilers e.g. gcc, icc
  - Standard conformance, bug detection at both compile and run time
- Unit testing frameworks, many free ones for C e.g. Check
  - Check correctness during development cycle
- Debuggers e.g. gdb, idb, Totalview, ddt
  - Find bugs either *post mortem* or at run time
- Profilers e.g. gprof, Scalasca, Paraver
  - Diagnose efficiency issues
- Documentation e.g. Doxygen
  - Automatically generate documentation from the comments in your code

What Else?

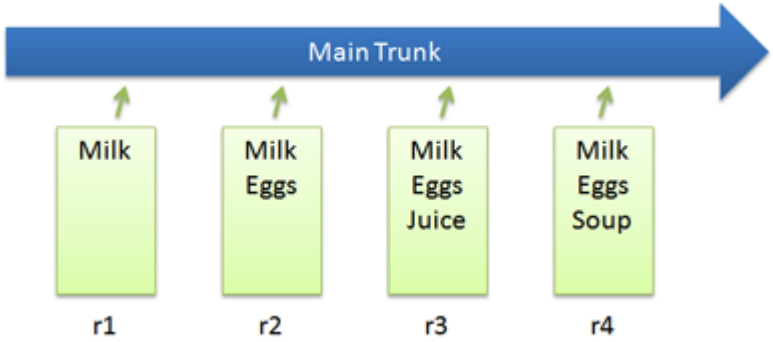
- IDE's (integrated development environments) bring a lot of these together e.g. Eclipse
  - But less popular in the Linux world than Windows – philosophical reasons I suspect
- Revision Control Systems – we'll have a quick look at this
  - a.k.a. Version Control
- Continuous Integration (CI) – a quick word once we have done Revision Control

Revision Control Systems (1)

- Imagine you are working on a piece of software
- It's working and then you make some changes
- It's now not working
  - E.g. you are doing this after the pub. Number one tip: software development and beer don't mix!
- Wouldn't it be nice if you could go back to the version that worked and start again?
  - Ideally tomorrow morning
- Revision control systems allow you to do this by keeping a complete history of all versions of the software project



Basic Checkins



https://betterexplained.com/articles/a-visual-guide-to-version-control/

Revision Control Systems (2)

- Imagine the project has got so complex that it is not just you working on it
- It would be nice to have a tool that kept track of the most up to date version which is the union of all the teams changes
  - You don't want to be emailing versions of files around – people will lose track and everybody will end up with a subtly different version of the code
- It would also be nice to have a tool that helps you merge the various changes together
  - And identify where the work of one member of the team conflicts with the work of another
- Revision control systems help you do that

Revision Control systems (3)

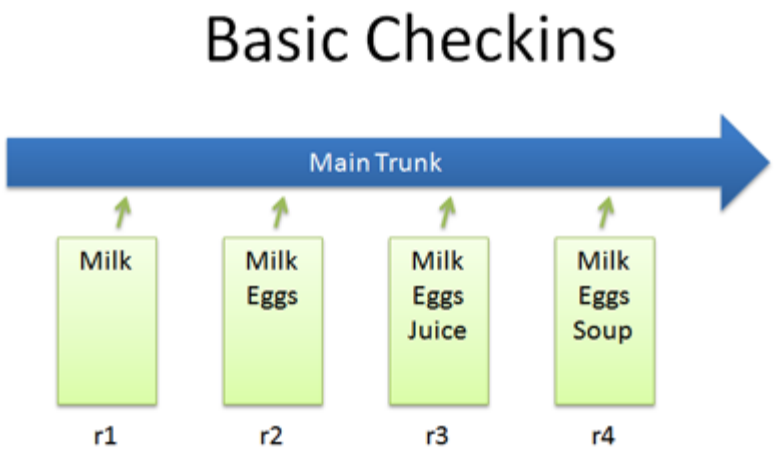
- You now want to release the software to the public
- How are people to get the code from you
- If somebody reports a bug how are you to know what version of the code it relates to?
- Revision control can help with this as well!
  - Release tags

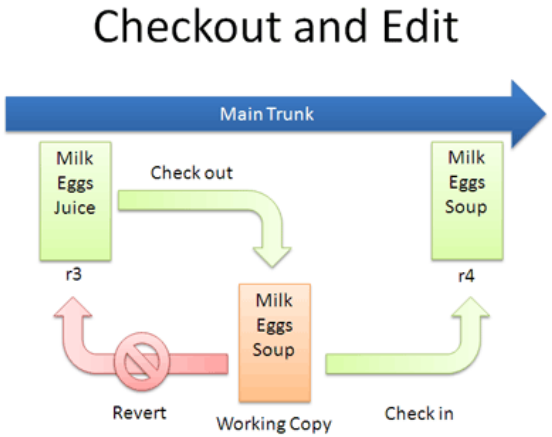
Revision Control Systems (4)

- What if somebody in the project wants to do something very experimental
- Or there are multiple developments which need to go on, and the best way to work would be to each development to be worked upon as independently as possible from any other one
- Or you want to be able to merge bug fixes into a given release, but have a separate version of the code which is being developed for the next release?
- Revision control systems can help with this!
  - Branches

Revision Control Systems

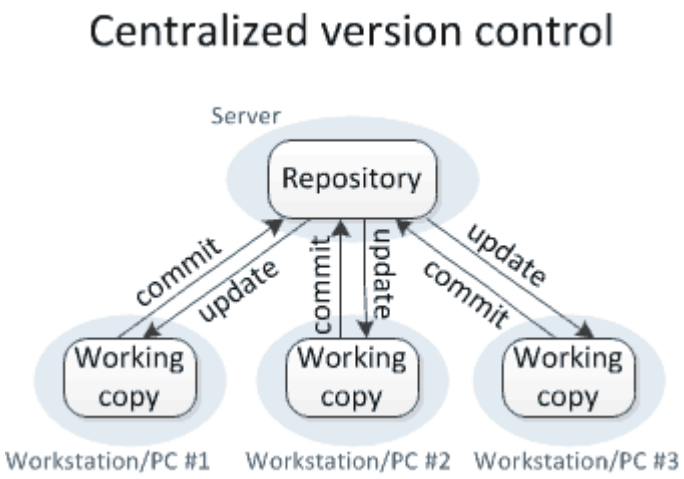
- The basic idea
  - The current version and history of the project is held in a repository
  - When you want to work on the code you *check out* a copy of the code
  - You do your work on the *working copy*
  - Once you are happy with your work you check it back into the repository





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



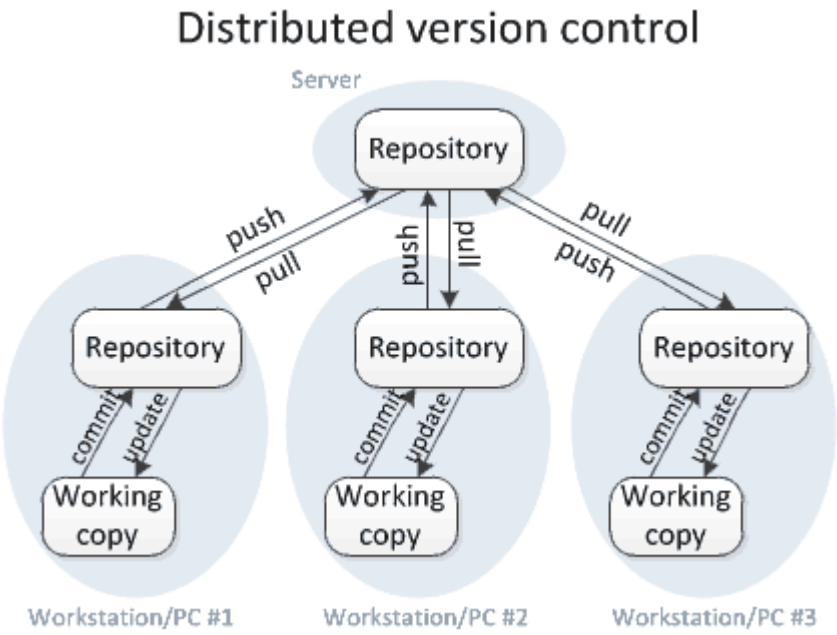
https://blog.inf.ed.ac.uk/sapm/2014/02/14/if-you-are-not-using-a-version-control-system-start-doing-it-now/

Git

- Many Revision Control systems
  - Mercurial, subversion …
  - I think I have used 7 different ones over the years
  - Admittedly all more or less the same for the level of use needed here
- Git is probably the most commonly used nowadays
  - Even if myself I'm not that fond of it …

Git Model

- Git has a slightly more complicated model
  - It is a distributed revision control system
- Definite advantage in having 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



Git – a few basic commands

- `git init` – initialise a local repository
- `git add` – add a file to the *staging area*
- `git commit` – check all changes held in the staging area into the repository
- `git status` – what is the current status of the directory, staging area and repository?

```
[oerc0085@login11(arcus-b) git_eg]$ git init
Initialized empty Git repository in /panfs/pan01/vol007/home/oerc-rse/oerc0085/git_eg/.git/
[oerc0085@login11(arcus-b) git_eg]$ git config --global user.name "Ian.Bush"
[oerc0085@login11(arcus-b) git_eg]$ git config --global user.email Ian.Bush@oerc.ox.ac.uk
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
#
# Initial commit
#
nothing to commit (create/copy files and use "git add" to track)
[oerc0085@login11(arcus-b) git_eg]$ touch milk
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       milk
nothing added to commit but untracked files present (use "git add" to track)
[oerc0085@login11(arcus-b) git_eg]$ git add milk
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   milk
#
[oerc0085@login11(arcus-b) git_eg]$ git commit -m "Added milk to shopping list"
[master (root-commit) 408bc7c] Added milk to shopping list
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 milk
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
nothing to commit (working directory clean)
[oerc0085@login11(arcus-b) git_eg]$ git log
commit 408bc7c5d22fa4f0cf254440c752c0e90b9a0ca8
Author: Ian.Bush <Ian.Bush@oerc.ox.ac.uk>
Date:   Thu May 17 09:23:45 2018 +0100

    Added milk to shopping list
[oerc0085@login11(arcus-b) git_eg]$
```

- `git ls-files` – list the files in my repository
- `git log` – history of repository, or a given file

```
[oerc0085@login11(arcus-b) git_eg]$ touch juice
[oerc0085@login11(arcus-b) git_eg]$ git add juice
[oerc0085@login11(arcus-b) git_eg]$ touch soup
[oerc0085@login11(arcus-b) git_eg]$ git add soup
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   juice
#       new file:   soup
#
[oerc0085@login11(arcus-b) git_eg]$ git commit -m "I need soup and juice as well"
[master 8a076b3] I need soup and juice as well
 0 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 juice
 create mode 100644 soup
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
nothing to commit (working directory clean)
[oerc0085@login11(arcus-b) git_eg]$ git log
commit 8a076b3f5d36f1d2b0fe6517d54ca426a6e755ed
Author: Ian.Bush <Ian.Bush@oerc.ox.ac.uk>
Date:   Thu May 17 09:50:30 2018 +0100

    I need soup and juice as well

commit 57291921a4b3dbf3a7906e875a94f9f80e980bb3
Author: Ian.Bush <Ian.Bush@oerc.ox.ac.uk>
Date:   Thu May 17 09:49:44 2018 +0100

    I need eggs for lunch

commit 0b22bbe22af1b53d4c6f1f456a6b22dac692e204
Author: Ian.Bush <Ian.Bush@oerc.ox.ac.uk>
Date:   Thu May 17 09:49:44 2018 +0100

    Added milk to shopping list
[oerc0085@login11(arcus-b) git_eg]$ git ls-files
eggs
juice
milk
soup
[oerc0085@login11(arcus-b) git_eg]$ git log milk
commit 0b22bbe22af1b53d4c6f1f456a6b22dac692e204
Author: Ian.Bush <Ian.Bush@oerc.ox.ac.uk>
Date:   Thu May 17 09:49:44 2018 +0100

    Added milk to shopping list
[oerc0085@login11(arcus-b) git_eg]$
```

UNIVERSITY OF OXFORD

Oxford e-Research Centre

- Updated files:

```
[oerc0085@login11(arcus-b) git_eg]$ cat > milk
Semi-skimed
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   milk
#
no changes added to commit (use "git add" and/or "git commit -a")
[oerc0085@login11(arcus-b) git_eg]$ git add milk
[oerc0085@login11(arcus-b) git_eg]$ git commit -m "The milk should be semi-skimmed"
[master e40e909] The milk should be semi-skimmed
 1 files changed, 1 insertions(+), 0 deletions(-)
[oerc0085@login11(arcus-b) git_eg]$ git log milk
commit e40e9099805527a377989063ad5cdffe5146fb5a
Author: Ian.Bush <Ian.Bush@oerc.ox.ac.uk>
Date:   Thu May 17 11:20:41 2018 +0100

    The milk should be semi-skimmed

commit 0b22bbe22af1b53d4c6f1f456a6b22dac692e204
Author: Ian.Bush <Ian.Bush@oerc.ox.ac.uk>
Date:   Thu May 17 09:49:44 2018 +0100

    Added milk to shopping list
[oerc0085@login11(arcus-b) git_eg]$
```

Whoops!

- If you realise you have made a mistake – `git checkout`

```
[oerc0085@login11(arcus-b) git_eg]$ cat > soup
Carrot and Coriander
[oerc0085@login11(arcus-b) git_eg]$ cat soup
Carrot and Coriander
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   soup
#
no changes added to commit (use "git add" and/or "git commit -a")
[oerc0085@login11(arcus-b) git_eg]$ git checkout -- soup
[oerc0085@login11(arcus-b) git_eg]$ cat soup
[oerc0085@login11(arcus-b) git_eg]$
```

Whoops And I've Added It!

- To remove it from the staging area `git reset HEAD`

```
[oerc0085@login11(arcus-b) git_eg]$ cat > soup
Carrot and Coriander
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   soup
#
no changes added to commit (use "git add" and/or "git commit -a")
[oerc0085@login11(arcus-b) git_eg]$ git add soup
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   soup
#
[oerc0085@login11(arcus-b) git_eg]$ git reset HEAD soup
Unstaged changes after reset:
M       soup
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   soup
#
no changes added to commit (use "git add" and/or "git commit -a")
[oerc0085@login11(arcus-b) git_eg]$ git checkout -- soup
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
nothing to commit (working directory clean)
[oerc0085@login11(arcus-b) git_eg]$ type git
```

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Whoops and I've Comitted It!

- `git checkout` can also be used to get back old versions

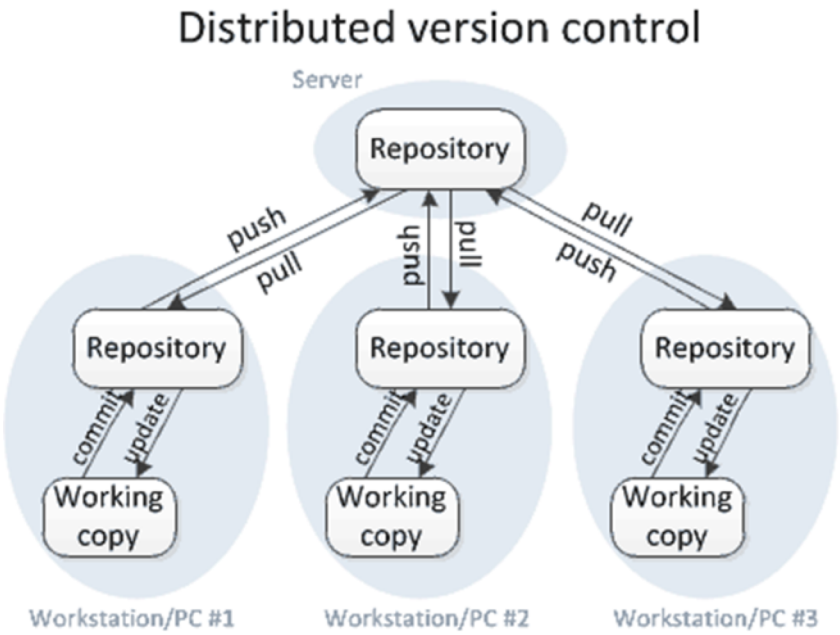
```
[oerc0085@login11(arcus-b) git_eg]$ cat > soup
Chicken
[oerc0085@login11(arcus-b) git_eg]$ git add soup
[oerc0085@login11(arcus-b) git_eg]$ git commit soup -m "It's chicken soup for lunch"
[master 2502b07] It's chicken soup for lunch
 1 files changed, 1 insertions(+), 0 deletions(-)
[oerc0085@login11(arcus-b) git_eg]$ cat > soup
Carrot and Coriander
D

[oerc0085@login11(arcus-b) git_eg]$ cat > soup
Carrot and Coriander
[oerc0085@login11(arcus-b) git_eg]$ git add soup
[oerc0085@login11(arcus-b) git_eg]$ git commit soup -m "No, it's carrot and coriander"
[master 2db7b5f] No, it's carrot and coriander
 1 files changed, 1 insertions(+), 1 deletions(-)
[oerc0085@login11(arcus-b) git_eg]$ git log soup
commit 2db7b5fbbdd06aefb82469b13836bfda4d1eabf3
Author: Ian.Bush <Ian.Bush@oerc.ox.ac.uk>
Date:   Thu May 17 11:39:13 2018 +0100

    No, it's carrot and coriander

commit 2502b07b6f9177244e650cedb39493f2a5708faf
Author: Ian.Bush <Ian.Bush@oerc.ox.ac.uk>
Date:   Thu May 17 11:37:58 2018 +0100

    It's chicken soup for lunch

commit 8a076b3f5d36f1d2b0fe6517d54ca426a6e755ed
Author: Ian.Bush <Ian.Bush@oerc.ox.ac.uk>
Date:   Thu May 17 09:50:30 2018 +0100

    I need soup and juice as well
[oerc0085@login11(arcus-b) git_eg]$ cat soup
Carrot and Coriander
[oerc0085@login11(arcus-b) git_eg]$ git checkout 2502b07b6f9177244e650cedb39493f2a5708faf soup
[oerc0085@login11(arcus-b) git_eg]$ cat soup
Chicken
[oerc0085@login11(arcus-b) git_eg]$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   soup
#
[oerc0085@login11(arcus-b) git_eg]$ git add soup
[oerc0085@login11(arcus-b) git_eg]$ git commit -m "Chnaged my ind again, back to chicken"
[master c65f895] Chnaged my ind again, back to chicken
 1 files changed, 1 insertions(+), 1 deletions(-)
```

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Remote repository

- We've looked at `add`, `commit` and `checkout` to deal with the local repository
  - Where you store your changes
- In git there is also the global repository
  - Where the team as a whole stores their changes
- `push`, `pull` and `clone` are used to access those
  - Will look at briefly during the practicals



Continuous Integration (CI) tools

- How do we make sure the code in the repository is reliable (and maybe efficient)?
- Continuous integration tools automate testing of the software by, at regular intervals, checking out the current version of the repository, running tests as specified by the development team, and reporting and problems encountered
- It's possible to run these tests on different platforms to check portatbility
- It's possible to include efficiency tests in the suite
- Serious software projects really should use this!
  - Jenkins is probably the most widely used CI tool

A Final Word About your Software: Stick A Licence On it!

- If you are going to give out your software to somebody else I strongly recommend you put a licence on it to protect yourself and make sure you get recognition
- 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

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
- The existence of software licences

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 - we'll look at this in the practicals
  - 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

In the next lecture…

We shall look further into C!