




University of
Nottingham

UK | CHINA | MALAYSIA

A large, high-resolution image of the Earth as seen from space, showing the curvature of the planet and the blue oceans. The image is framed by a thin white border.

Computer Engineering and Mechatronics MMME3085

Dr Louise Brown





Software Engineering Best Practice

Part 3



Introduction

Today's lecture will cover:

- Continuation of software best practice
 - Documentation
 - Testing
- Feedback on the project planning assignment
- Going for speed
 - Ways to make your code run faster



Documentation

Why and when should
software be documented?

<https://padlet.com/louisebrown7/why-and-when-do-we-document-code-xitu4wsj69zfpg23>





Documenting

In software there are many places we undertake documentation:

- The design stage of the code
- Documentation within the code
 - Which is more than just the odd comment!
- Documentation on how to use the software once written



Documenting code

Perform documentation when and as required (e.g. %10 of total production time)

No documentation is never a good solution!

- nor is excessive documentation!

Documents, like code, need to be managed (and shared)

– consider using version controlling tools and/or web-based platforms



What to document

Documentation can include:

- code documentation
- requirement specifications
- design documents
- test documents
- user manuals etc.



Code IS documentation!

Your code is ultimately the best documentation!

What we write about the code is generally provided for those who:

- Don't know the code (someone else wrote it)
- Don't have time to read the code (it's too complex)
- Don't want to read the code (who wants to read code to understand what's going on??)
- Don't have access to the code (although they could still decompile it)

For everyone else the code is what they are working on, so make life easy for them

- And yourself if you ever need to go back to the code!



Code documentation

Code documentation is important!

Good programming style goes a long way to creating self-documenting code.

- Use meaningful variable and function names
- Use named constants instead of 'magic numbers'
- Use clear formatting
- Keep flow control and data structure simple

Then... use comments!



Commenting Code

Kinds of comments¹:

- Repeat of the code
 - Adds no value – avoid
- Explanation of the code
 - Used to explain complicated piece of code
 - May be better to improve the code!
- Marker in the code
 - `// **** TODO: Fix before release!`
 - A standard system may help to identify work to be done
 - Shouldn't be left in the code!
- Summary of the code
 - Distils a few lines of code into one or two sentences
 - Useful if trying to scan code quickly
- Description of the code's intent
 - Purpose of the code, e.g. `// get current employee information`
- Information that cannot be expressed by the code itself
 - Copyright notices, confidentiality notices, version numbers, references etc

¹McConnell, S. (2004). Code Complete, Microsoft Press.



Self-documenting code

The easiest way to document code is by producing self-documenting code (this goes back to the ideas of best practice).

Tools can be used to take comments from the code and automatically generate code documentation.

- Doxygen for C/C++ www.doxygen.org
- Sphinx for Python <https://www.sphinx-doc.org>



The best comment is perhaps

**Always code as if the person who ends up maintaining your code
is a violent psychopath who knows where you live.**



Testing

Never assume that just because your code runs that it's given you the right answers!

- Test with a small set of data where you know what the results should be. Make sure that your program doesn't crash when you give it invalid data.
- Use defensive programming
 - Test validity of variables at start of functions
 - Test validity of results returned from functions
 - Use assertions¹
- Use a unit testing library
 - Built into many IDE's

¹Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. (2014) Best Practices for Scientific Computing. PLoS Biol 12(1): e1001745. doi:10.1371/journal.pbio.1001745



Use Assertions

**Use error-handling for conditions you expect to occur;
use assertions for conditions that should *never occur*¹**

An *assertion* is simply a statement that something holds true at a particular point in a program²

- Typically used to check values of inputs in functions
- May help to identify if an error has crept into code during development

Generally disabled for release versions

- Do not put anything in the assertion which changes the state of the code
 - e.g. Don't use `assert(x = 5)`

Assertions can be removed at compile time using the preprocessor `NDEBUG`

²Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. (2014) Best Practices for Scientific Computing. PLoS Biol 12(1): e1001745. doi:10.1371/journal.pbio.1001745



Unit Testing Library

Programs are available which provide testing frameworks. Typically a unit test is written for each function in a program.

When new features are added or changes are made to code, the tests can be rerun to make sure that everything still works as it should.

Test Driven Development (TDD) specifies how each unit should work and the test is written before the actual code.

CppUnit is a unit testing framework for C++ <https://en.wikipedia.org/wiki/CppUnit>

- <https://freedesktop.org/wiki/Software/cppunit/>

C Unity Test Explorer for VSCode:

<https://marketplace.visualstudio.com/items?itemName=fpopescu.vscode-unity-test-adapter>



Project Planning Feedback



Project Planning Assignment Feedback

Generally very good planning 😊

- There was generally a lack of detail about how the font file would be read in and stored
- Some had missed that the pen up/down data needs to be stored as well as the x/y coordinates
- There was nothing in the brief to say that the user input font size was an integer



Flowcharts

- Try to keep the main flow of the program moving vertically down the page
- Many flowcharts didn't include checking for success opening file
 - Some checked in a function and returned an appropriate value but then didn't do anything with this in the calling program
- Many flowcharts showed function calls but didn't include the flowchart for the functions
- Don't have crossing lines in the flowchart
- Always use a diamond box for decisions
- Make sure the outcome (yes/no, true/false) is shown on the exits from the decision boxes
- Only decision boxes should have multiple exits



Data Types

- The 'Data Type' should give the actual data type, eg int, float*, struct Shape
- There is no 'str' or 'string' data type in C – it should be a char array
- Where structures are defined, give the actual definition and the data items that are stored within it
- The rationale should include the reason why you have chosen a given data type, e.g. int for pen up/down as it can only take integer values.



Test Data

This is probably the section which completed least well.

- Be precise with the test data
 - Give the actual data that would be passed into the function with the resulting output

• eg,

Function	Test Case	Input	Expected Output
ReadFile()	Invalid file	"nonexistent.txt"	Returns 0

- Ideally, there should be a test to cover each path through the program



Appendix 1

Going for Speed



Code optimisation

Going for speed

- Just use a faster PC?

General concepts of code optimisation

- Particular examples in C
- Lookup tables (all languages)





Speeding up Code

Arrays

- Say you wanted to assign a particular character based on a value...
- Method 1

```
switch ( queue )
{
    case 0 : letter = 'W';
            break;
    case 1 : letter = 'S';
            break;
    case 2 : letter = 'U';
            break;
}
```



Arrays

- Say you wanted to assign a particular character based on a value...
- Method 2

```
if ( queue == 0 )  
    letter = 'W';  
else if ( queue == 1 )  
    letter = 'S';  
else  
    letter = 'U';
```




Arrays

- Say you wanted to assign a particular character based on a value...
- The best & quickest method

```
static char *classes = "WSU";  
letter = classes[queue];
```



Registers (1)

Use the "register" declaration whenever you can, eg.

```
register float  val;  
register double dval  
register double dval
```

This is only a hint to the compiler, and many will do this anyway

- NB: You cannot take the address of a register



Registers (2)

Note: you cannot take the address of a register variable

- So no pointers for these 😊

This will fail!

```
int main()
{
    register int i = 10;
    int *a = &i;
    printf("%d", *a);
    return 0;
}
```



Integers (1)

Use integers wherever possible

Use unsigned integers if you know the value will never go negative

- Many compilers handle unsigned integer mathematics much faster than signed

So the ideal definition would be:

```
register unsigned int    var_name;
```



Integers (2)

Remember

- integer operations are much faster as they can be done on the CPU, rather than on a floating point processor or by external libraries

If you only need 2dp accuracy,

- multiply everything by 100 and use integers, converting back to floating point at the last moment



Loop Jamming

NEVER use two loops when one will suffice

NO!

```
for(i=0; i<100; i++)  
{  
    stuff();  
}  
  
for(i=0; i<100; i++)  
{  
    morestuff();  
}
```

YES

```
for(i=0; i<100; i++)  
{  
    stuff();  
    morestuff();  
}
```



Unrolling loops can make a huge difference in speed

```
for(i=0; i<3; i++)  
{  
    something(i);  
}
```

Is much less efficient than

```
something(0);  
something(1);  
something(2);
```

As the code has to keep check the value of i



But for large loops...

It would clearly be impractical to do this for huge loops, or where the upper limit is not known at design time

We can get a speed up though by using Code Blocking



Code Blocking

We define a block size and unroll the code into blocks of this size

We then handle any leftovers in a final statement

It works as the processor can get on with processing data, rather than examining loop counters



An example (part 1)

```
#include<stdio.h>

#define BLOCKSIZE (8)

int main(void)
{
    int i = 0;
    int limit = 33;  /* could be anything */
    int blocklimit;

    /* The limit may not be divisible by BLOCKSIZE,
     * go as near as we can first, then tidy up.
     */
    blocklimit = (limit / BLOCKSIZE) * BLOCKSIZE;
```



An example (part 2)

```
/* unroll the loop in blocks of 8 */
while( i < blocklimit )
{
    printf("process(%d)\n", i);
    printf("process(%d)\n", i+1);
    printf("process(%d)\n", i+2);
    printf("process(%d)\n", i+3);
    printf("process(%d)\n", i+4);
    printf("process(%d)\n", i+5);
    printf("process(%d)\n", i+6);
    printf("process(%d)\n", i+7);

    /* update the counter */
    i += 8;
}
```



An example (part 3)

```
if( i < limit )
{
    /* Jump into the case at the place that will allow
     * us to finish off the appropriate number of items. */
    switch( limit - i )
    {
        case 7 : printf("process(%d)\n", i); i++;
        case 6 : printf("process(%d)\n", i); i++;
        case 5 : printf("process(%d)\n", i); i++;
        case 4 : printf("process(%d)\n", i); i++;
        case 3 : printf("process(%d)\n", i); i++;
        case 2 : printf("process(%d)\n", i); i++;
        case 1 : printf("process(%d)\n", i);
    }
}
```

NB: we could use a for loop, but this is yet faster!



Loops

Normally to loop, we would have (say)

```
for( i=0; i<10; i++ ) { ... }
```

Giving 0,1,2,3,4,5,6,7,8,9

If counting backwards is not a problem, do it - it is faster,

```
for ( i=9; i>= 0; i-- ) { ... }
```



Loops `for(i=10; i>0; i-- ;){}`

This works as it is faster to process `i--` as the test condition

- it says is `i==0` ?, if not decrement by 1 and loop

In the original case the compiler had to:

- Subtract `i` from 10
- Test if the result is non zero ?
- If so, increment `i` and loop



Use switch instead of if (1)

For large decisions involving

if...else if ...else..., like this:

```
if( val == 1)
    dostuff1();
else if (val == 2)
    dostuff2();
else if (val == 3)
    dostuff3();
```



Use switch instead of if (2)

It may be faster to use a switch

```
if( val == 1)
    dostuff1();
else if (val == 2)
    dostuff2();
else if (val == 3)
    dostuff3();
```

```
switch( val )
{
    case 1: dostuff1();
            break;

    case 2: dostuff2();
            break;

    case 3: dostuff3();
            break;
}
```

In the if() statement, if the last case is required, all the previous ones will be tested first. The switch lets us cut out this extra work.

If you have to use a big *if / else if / else if ..* statement, test the most likely cases first.

If you know which cases are more likely to be true put these cases first



Early loop breaking

Often it is not necessary to process for the entirety of a loop

Such a case might be where one is testing for the presence of a particular value in an array

A solution is to break out the loop as soon as you have what you need



Loop breaking (1)

Consider the case of looking to see if the number 99 is in a list of numbers

```
found = FALSE;
for(i=0;i<10000;i++)
{
    if( list[i] == 99 )
    {
        found = TRUE;
    }
}
if( found )
    printf("Yes, there is a 99. Hooray!\n");
```

Even if '99' was the first element, it would check all 10,000!



Loop breaking (2)

```
found = FALSE;
for(i=0;i<10000;i++)
{
    if( list[i] == 99 )
    {
        found = TRUE;
        break;
    }
}
if( found )
    printf("Yes, there is a 99. Hooray!\n");
```

This will break out from the loop as soon as '99' is found



Some miscellaneous ones (1)

Avoid the use of recursion.

- Recursion can be very elegant and neat, but creates many more function calls which can become a large overhead

Avoid the square root function `sqrt()` in loops

- calculating square roots is very CPU intensive

Avoid functions such as `pow()` when a simple arithmetic function can be used



Some miscellaneous ones (2)

Floating point multiplication is often faster than division

- use `val * 0.5`
- instead of `val / 2.0`

Addition is quicker than multiplication

- use `val + val + val`
- instead of `val * 3`



Some miscellaneous ones (3)

Single dimension arrays are faster than multi-dimensioned arrays

- We can make a 2D array into a 1D array (this is in effect the case with the memory anyhow).
- All we need is to be able to convert $[x][y]$ to a single $[x]$ value



It is quite easy 😊

Consider

```
int x[5][20]
```

```
int y[100]
```

We can access say `[4][7]` as

```
x[4][7]
```

or

```
y[87]            {4*20 + 7 }
```





And of course

**Remember to turn optimisation on in the
compiler settings!**



Lookup Tables – an example

This is a common optimisation in numerical processing

- We define an array and pre-populate it with values we will be making repeated use of
- Whilst the initial creation of the array will take time, the speed up is well worth it



Consider the following case

We have a calculation that needs the sine of an integer angle in degrees

Method 1:

calculate sine values during calculation

Method 2:

Create lookup table first, then perform the calculations referencing the lookup table



Comparison of Speed

