




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



The tools used for code development

Having decided on the low level design we are in a position to start writing some code.

We need to think about:

- The tools that we use to make the coding process more straightforward and robust
 - What editor or development environment will be used?
 - How do we create build files (important once a project has more files)?
 - How do we find bugs?
- The practise of writing good code
 - There may be some overlap between the two (e.g. using a debug environment combined with coding techniques to identify bugs)
- How will we keep a track of changes to the code, particularly if being developed by a group
- How do we keep the code safe in the event of a system crash?
- How do we share code with our fellow developers?



The practise of writing good code

Superior coding techniques and programming practices are hallmarks of a professional programmer.

The bulk of programming consists of making a large number of small choices while attempting to solve a larger set of problems.

How wisely those choices are made depends largely upon the programmer's skill and expertise. ¹

¹ [https://msdn.microsoft.com/en-us/library/aa260844\(v=vs.60\).aspx](https://msdn.microsoft.com/en-us/library/aa260844(v=vs.60).aspx)



The Practise of Writing Good Code: Functions

Try to keep a function for one purpose only. For example don't write a function to calculate some variables and then plot them. Create two functions – you may also want to do the calculations without plotting and a general purpose plot function is more likely to be reused.

If you find yourself repeating a very similar piece of code around your program it should probably be a function

Where possible use a title which describes what both what the function **does** and an object

- PrintDocument()
- CalcPenPosition()
- CalcStartPosition()

Where possible, limit the number of parameters passed. Make sure all parameters are used.



The Practise of Writing Good Code

Even though we've selected an IDE and designed code to the level of function definitions there is another step before actually writing the code:

Pseudocode - a plain language description of how an algorithm, function, class or program will work.

- Describe specific operations using English-like statements
- Do not use syntax from the final programming language
- Write at the level of intent. Describe the meaning rather than how it will be done
 - If the pseudocode is written in the IDE as comments these will stay in your code
- Write at a low enough level that generating the code will be almost automatic. It may be an iterative process.



Example Pseudocode

```
ReadShape( fileHandle, ShapeData )
{
    read shape name from file
    read number of strokes from file

    allocate memory for number of pen strokes
    if failed to allocate memory
        return false
    endif
    for each stroke in file
        read x coord into ShapeData penStroke array
        read y coord into ShapeData penStroke array
        read pen up/down into ShapeData penStroke array
    end loop

return true
}
```



Best Practice – a guide

- As you start to develop code that will be both shared and that will ‘grow’ over time it is important that you start to adopt some best practices
- Often companies will have their own ‘house’ style
 - For example, how to align the brackets when ‘blocking’ code for an loop/condition etc.
- This best practice guide produced by Microsoft is a few years old but still provides examples of very good practice:
 - [https://msdn.microsoft.com/en-us/library/aa260844\(v=vs.60\).aspx](https://msdn.microsoft.com/en-us/library/aa260844(v=vs.60).aspx)
- Another, more general guide is given here:
 - <https://mitcommlab.mit.edu/broad/commkit/coding-and-comment-style/>
 - Links at the end of the page give style guides for specific languages.



The tools used for code development

Having decided on the low level design we are in a position to start writing some code.

We need to think about:

- The tools that we use to make the coding process more straightforward and robust
 - What editor or development environment will be used?
 - How do we create build files (important once a project has more files)?
 - **How do we find bugs?**
- The practise of writing good code
 - There may be some overlap between the two (e.g. using a debug environment combined with coding techniques to identify bugs)
- How will we keep a track of changes to the code, particularly if being developed by a group
- How do we keep the code safe in the event of a system crash?
- How do we share code with our fellow developers?



Debugging

“Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?”

- *Brian Kernighan*



“Software quality must be built in from the start. The best way to build a quality product is to develop requirements carefully, design well, and use high-quality coding practices. **Debugging is a last resort**”

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



How not to debug!

- By guessing:
 - Scatter code with print statements
 - Randomly change things until it works
 - Don't back up original version
 - Don't keep notes
- By not spending time understanding the problem
- Fixing the problem with a workaround
 - Make a special case to deal with the error



Debugging tools

Source-code comparators

- Diff, WinDiff or git diff to see what has changed from the last working version

Compiler warning messages

- Set the compiler to the highest warning level
 - Uninitialised variables, pointers etc often cause problems
- Some compilers allow warnings to be treated as errors

Lint utility and static code analysis tools

- Check for code issues

Symbolic debugger

- Part of the IDE
- Use to step through code to see exactly what the code is doing
 - It won't solve the problem for you – it will help you to find it
- Great for understanding someone else's code
- Set breakpoints to home in on a particular part of the code



The tools used for code development

Having decided on the low level design we are in a position to start writing some code.

We need to think about:

- The tools that we use to make the coding process more straightforward and robust
 - What editor or development environment will be used?
 - How do we create build files (important once a project has more files)?
 - How do we find bugs?
- The practise of writing good code
 - There may be some overlap between the two (e.g. using a debug environment combined with coding techniques to identify bugs)
- How will we keep a track of changes to the code, particularly if being developed by a group
- How do we keep the code safe in the event of a system crash?
- How do we share code with our fellow developers?



Git Revisited

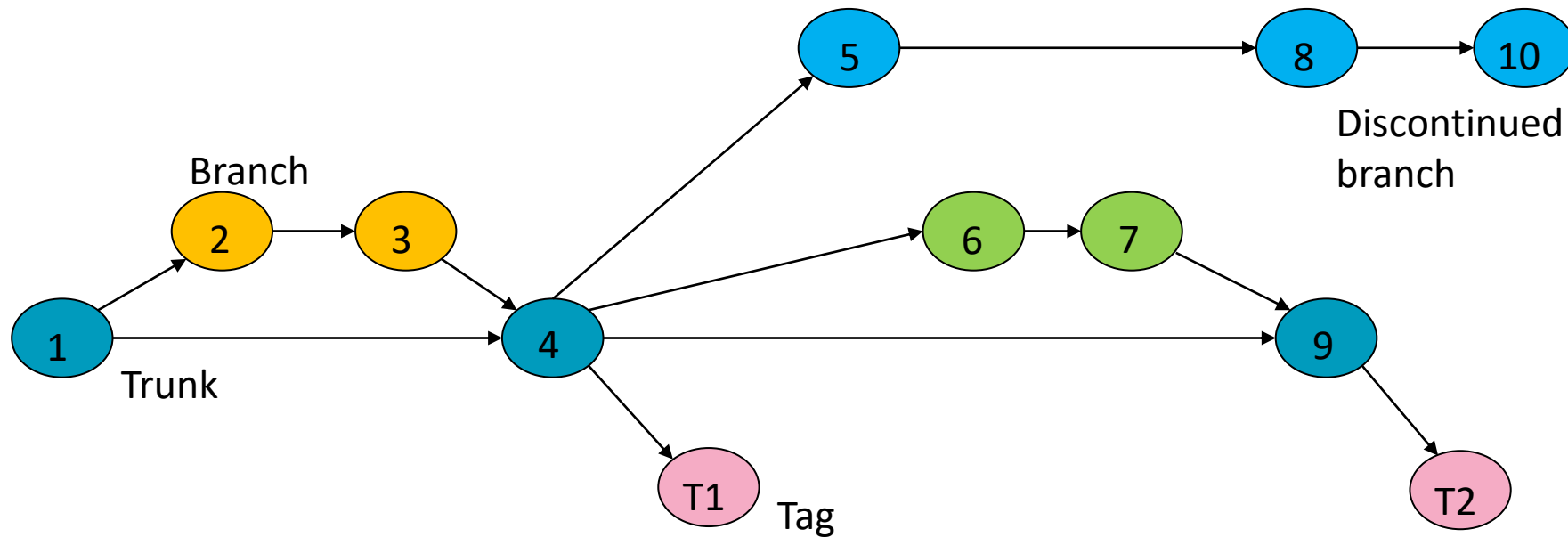
- git: To keep our code 'safe'
 - A free version control system
 - It allows us to keep version of the code so we can 'go back'
 - We can 'branch' code to try things
 - Share code with others who can then 'check in' code when they have finished with it
 - <https://git-scm.com/downloads>





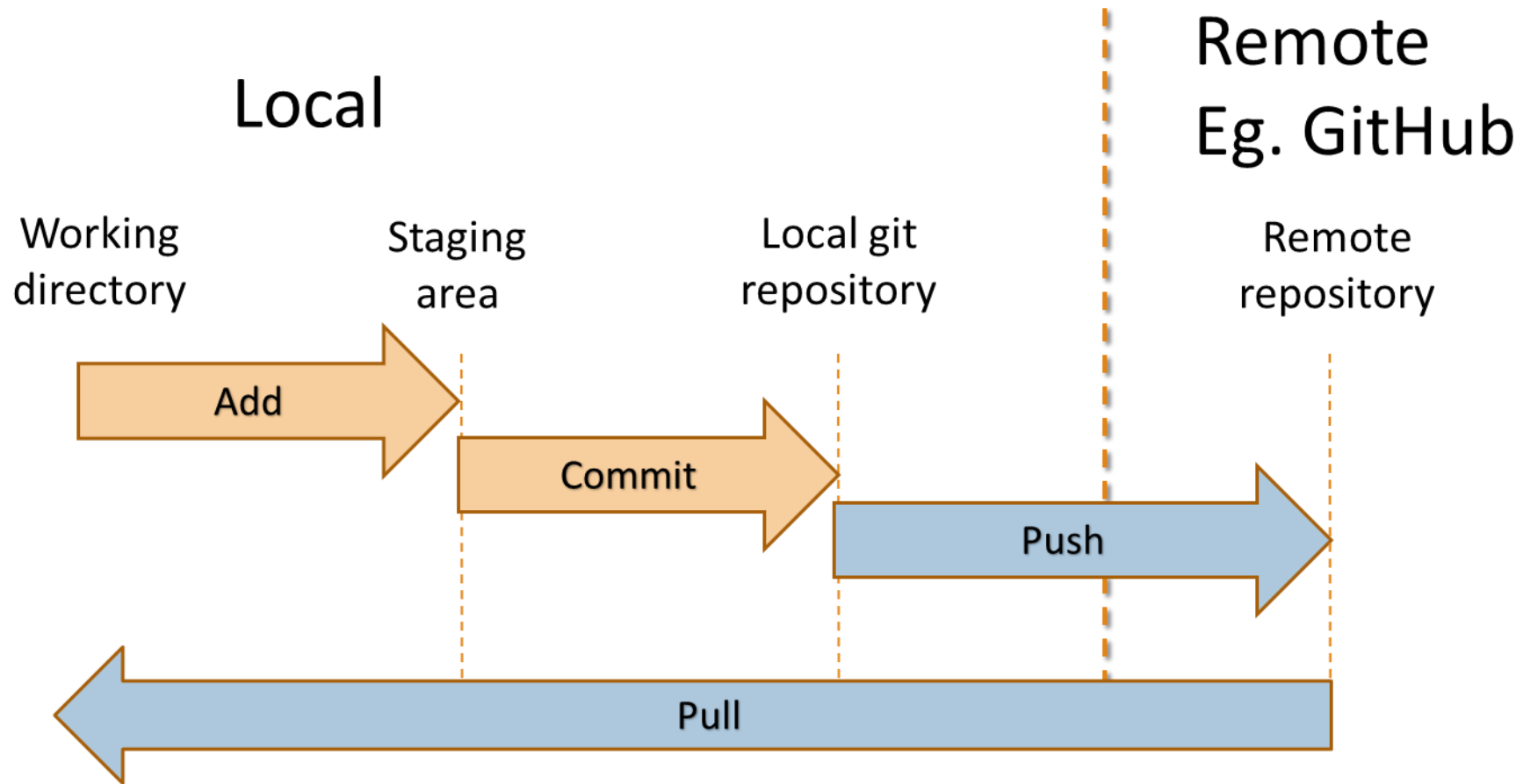
Local git workflow

- Distributed system – have own version of the repository on local computer
- Using a remote repository gives backup and easier sharing between developers
- Integrated into some IDEs eg VSCode, Visual Studio and Matlab
- Easy use of branches for experimental code development





Git workflow





Useful git blogs

- Using VSCode and Git – useful blog
- https://kbroman.org/github_tutorial/pages/init.html - introduction to starting a new repository using the command line



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



Code IS documentation!

Your code is ultimately the best documentation!

What we write about the code is generally an approximation to the code itself, 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!



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

No documentation is never a good solution!

- nor is excessive documentation!

Documentation can include

- code documentation,
 - requirement specifications,
 - design documents,
 - test documents,
 - user manuals etc.
- Documents, like code, need to be managed (and shared) – consider using version controlling tools and/or web-based platforms



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
 - Distills 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

Code documentation is important!

The best way of doing this is 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?fpopescu.vscode-unity-test-adapter>



A few dates and times

Computer Lab: 11am-1pm Tuesday Coates C19 until 12th December.

Monday : 1-2.45pm Chemistry C15 drop-in

Robot testing: 6th, 8th and 13th December. Sign-up sheet to follow.

Project submission: 3pm Thursday 14th December



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:

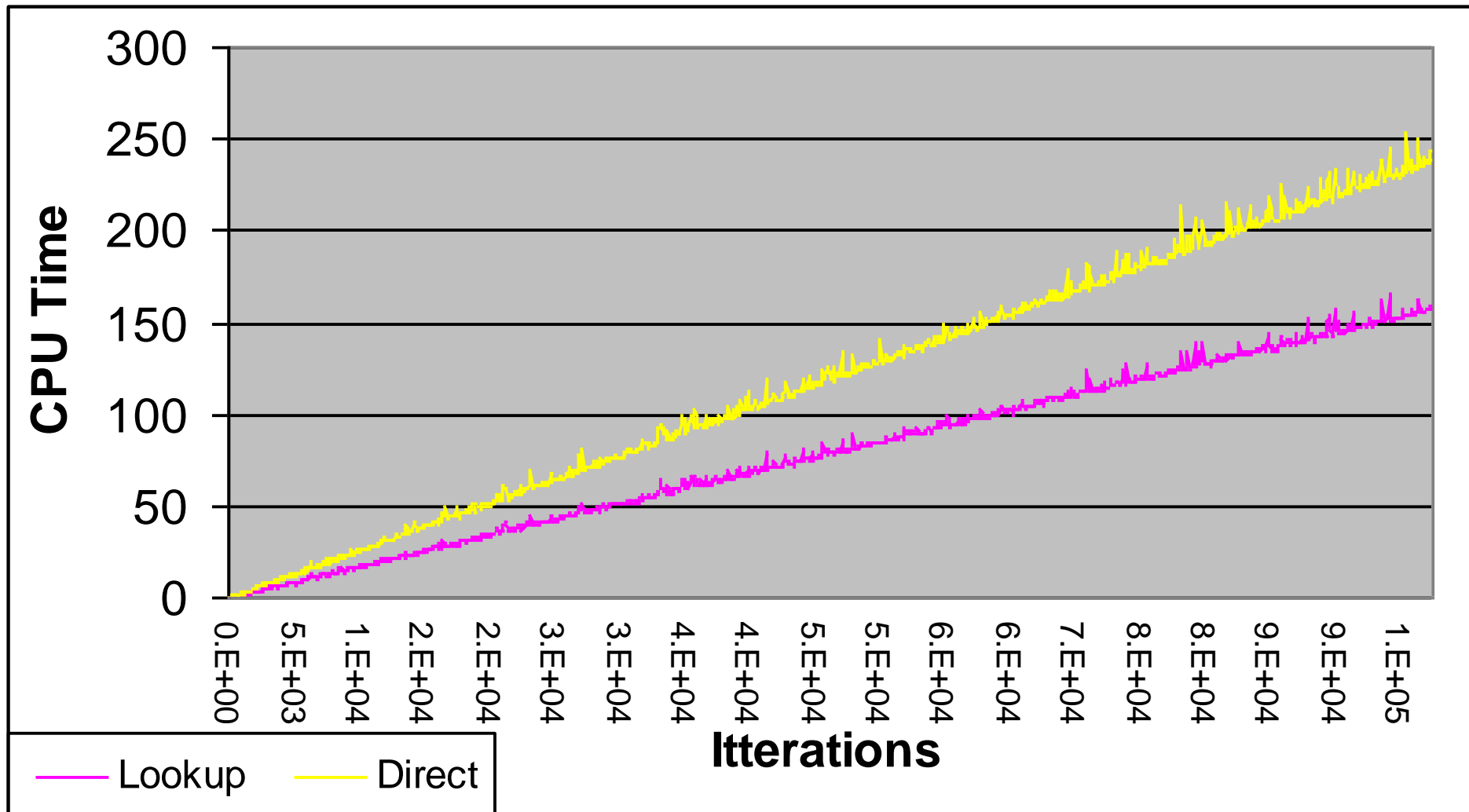
calculate sine values during calculation

Method 2:

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



Comparison of Speed





Appendix 2

All About Numbers (not in the book)



Numbers

On a computer, the basic building block for a number is the BYTE

- 8 bits
- Can range from 00000000 to 11111111
- Can be signed or unsigned



Numbers – byte sizes

Each variable is made up of a number of bytes, and this defines the range of numbers possible.

We can obtain the size using the sizeof function in C

Some machines will have the same size for variables that on other machines will be different (especially short int)



Numbers - range

The number of bytes used defines (in the case of integer types) the range of numbers that can be stored:

- 2 Byte (16 bits) has the range
 - 0 to $(2^{16} - 1)$ or 0 to 65535 (unsigned)
 - or
 - (-2^{15}) to $(2^{15} - 1)$ or -32768 to 32767 (signed)



Numbers - limits

The limits of a variable are machine specific (with the possible exception of char) as they depend on the number of bytes used for storage

With each machine/compiler is shipped a file 'limits.h' which has the permissible range



Decimal numbers

What do these numbers represent?
What are their values?



1056.2458

What is this called?



Fixed point numbers (1)

A method of representing fractions using binary numbers

Fixed point representation of fractions:

- In a fixed point representation, the **binary point** is understood to always be in the same position. The bits to the **left** represent the **integer** part and the bits to the **right** represent the **fraction** part :
- The integer parts go as 2^n (1,2,4,8 etc)
- The fraction parts go as 2^{-n} (0.5, 0.25, 0.125 etc) . The overall value is formed using a sum of these.



Fixed point numbers (2)

Example:

A fixed point system uses 8-bit numbers. 4 bits for the integer part and 4 bits for the fraction:

What number is represented by **00101100**?



Fixed point numbers (3)

Example:

A fixed point system uses 8-bit numbers. 4 bits for the integer part and 4 bits for the fraction:

What number is represented by **00101100**?

8	4	2	1	0.5	0.25	0.125	0.0625
0	0	1	0	1	1	0	0



Fixed point numbers (4)

Example:

A fixed point system uses 8-bit numbers. 4 bits for the integer part and 4 bits for the fraction:

What number is represented by **00101100**?

8	4	2	1	0.5	0.25	0.125	0.0625
0	0	1	0	1	1	0	0

NOTE : The headings for the fraction part are divided by 2 successively to the right...

The number **00101100** represents the number **2.75**.



Fixed point numbers (5)

Example:

Converting to Fixed Point – there is an easy way !

EG. 56.78125

Stage 1 : Integer part: easy

Sign	64	32	16	8	4	2	1
0	0	1	1	1	0	0	0



Fixed Point Numbers (56.78125)

Example: - fraction part

0	.	7	8	1	2	5
					x	2
(1)	.	5	6	2	5	0
					x	2
(1)	.	1	2	5	0	0
					x	2
(0)	.	2	5	0	0	0
					x	2
(0)	.	5	0	0	0	0
					x	2
(1)	.	0	0	0	0	0

You keep multiplying, ignoring the value in brackets until you get zero or run out of bits to use

So reading down we get

11001

So final answer is

00111000 11001000



Fixed Point Numbers - Example

Your Turn :

Convert the value 25.3 to a fixed point representation,
8 bit mantissa, 8 bit exponential



Fixed Point Numbers – Solution (1)

Example: - integer part

0 0011001
s 25



Fixed Point Numbers – Solution (2)

Example: - fraction part

0	.	3	0
(0)	.	6	0
(1)	.	2	0
(0)	.	4	0
(0)	.	8	0
(1)	.	6	0
(1)	.	2	0
(0)	.	4	0
(0)	.	8	0

|



Fixed Point Numbers – Solution (3)

Example: - fraction part

0	.	3	0	
(0)	.	6	0	0.5
(1)	.	2	0	0.25
(0)	.	4	0	0.125
(0)	.	8	0	0.0625
(1)	.	6	0	0.03125
(1)	.	2	0	0.015625
(0)	.	4	0	0.0078125
(0)	.	8	0	0.00390625

$$\begin{aligned} &01001100 \\ &= 0.25 + 0.03125 + 0.015625 \\ &= 0.296575 \end{aligned}$$

Not exactly 0.3!

Answer: 0 0011001 01001100
s 25 . 3



Limitations of fixed point representation

Fixed window limits representation of both very large and very small numbers

Prone to loss of precision when two large numbers are divided.



Scientific notation

Most common solution is to use scientific notation with a base and exponent:

- $123.456 \rightarrow 1.23456 \times 10^2$ in decimal
- $789.abc \rightarrow 7.89abc \times 16^2$ in hex
- $1010.110 \rightarrow 1.010110 \times 2^3$ in binary

- Giving a sliding scale of precision to maximise precision for both very large and very small numbers



Floating Point Numbers

These are held using the IEEE 754 Floating Point Model which has 3 components:

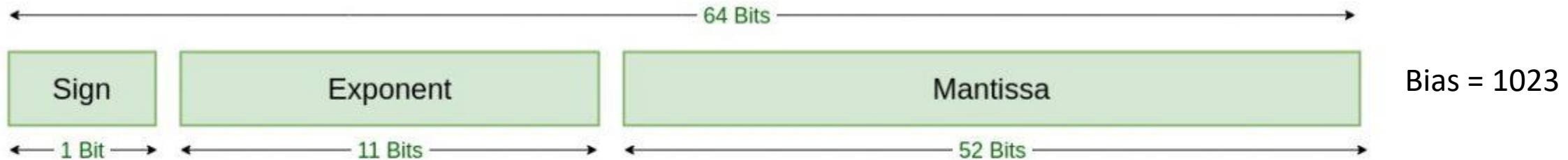
- Sign of mantissa – 0 for positive number, 1 for negative
- Biased exponent – addition of a bias enables both positive and negative exponents to be represented
- Normalised mantissa – part of number in scientific notation giving the significant digits. In binary this must be 0 or 1 so a normalised mantissa has 1 to the left of the decimal
- These may be single or double precision



Single and Double Precision Floating Point Numbers



Single Precision IEEE 754 Floating-Point Standard



Double Precision IEEE 754 Floating-Point Standard

<https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/>



In summary

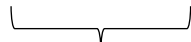
1. The sign bit is 0 for positive, 1 for negative.
2. The exponent base is two.
3. The exponent field contains 127 plus the true exponent for single-precision, or 1023 plus the true exponent for double precision.
4. The first bit of the mantissa is typically assumed to be 1, yielding a full mantissa of $1.f$, where f is the field of fraction bits.

<https://steve.hollasch.net/cgindex/coding/ieeefloat.html>



Convert 25.3 to single precision floating point binary (1)

- $25 = 11001$
- $0.3 = 010011001100110011\dots$



This section will repeat

0	.	3	0	
(0)	.	6	0	0.5
(1)	.	2	0	0.25
(0)	.	4	0	0.125
(0)	.	8	0	0.0625
(1)	.	6	0	0.03125
(1)	.	2	0	0.015625
(0)	.	4	0	0.0078125
(0)	.	8	0	0.00390625



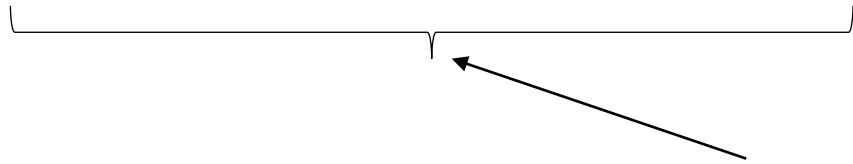
Convert 25.3 to single precision floating point binary (2)

$$25 = 11001$$

$$0.3 = 010011001100110011\dots$$

$$25.3 = 11001.010011001100110011$$

$$= 1.10010100110011001100110 \times 2^4$$



$$\text{Normalised mantissa} = 10010100110011001100110 \quad (23 \text{ bits})$$

$$\text{Sign} = 0$$

$$\text{Biased exponent} = 127 + 4 = 131$$

$$= 10000011$$

Single precision is: **0 10000011 10010100110011001100110**



It's not actually 25.3!

IEEE 754 Converter (JavaScript), V0.22

	Sign	Exponent	Mantissa
Value:	+1	2^4	1.5812499523162842
Encoded as:	0	131	4875878
Binary:	<input type="checkbox"/>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/>	<input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input checked="" type="checkbox"/> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/>
Decimal representation		<input type="text" value="25.2999992371"/>	<input type="text" value="+1"/>
Value actually stored in float:		<input type="text" value="25.299999237060546875"/>	<input type="text" value="-1"/>
Error due to conversion:		<input type="text"/>	
Binary Representation		<input type="text" value="01000001110010100110011001100110"/>	
Hexadecimal Representation		<input type="text" value="0x41ca6666"/>	

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>



Floating point numbers

This representation however has limitations:

- Even with a 23 bit mantissa some numbers can appear the same
- We can extend to double precision (51 bit mantissa, 12 bit exponent) but we can still have inaccuracies
- Adding very large to very small numbers can cause considerable problems
 - https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- Care should be taken with comparisons



Variable Selection

We aim to use the most suitable variable for the type of number we are storing

The choice of variable has consequences both for overall memory usage and speed of calculation

Integer mathematics is considerably faster than floating point calculations.



Byte ordering (1)

In an ideal world, all machines would use the same byte size for variables and would arrange the bytes in the same way
However... (as mentioned when we discussed binary files)



Byte ordering (2)

Different machines arrange the bytes used to make a numbers using one of two ordering types

- Big endian – high order byte comes first
- Little endian – low byte comes first

So for a two byte integer

- Big Endian - High byte, low byte
- Little Endian - Low byte, high byte
- For more information: <https://developer.ibm.com/articles/au-endianc/>