Introduction to C/C++

With some HPC added for fun ☺

Martin Vickers
mjv08@aber.ac.uk

What I am going to try to teach

- The ‘C/C++’ Programming Language
  - Rather than just C++, which is almost identical except for a few key additions (These will be covered at the end)
- High Performance Computing
  - We all want our programs to run faster, especially when you have a deadline
- A little bit of UNIX and Shell scripts

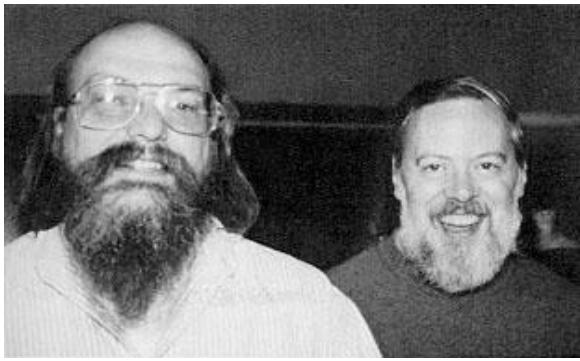
Mainly because it’s easier to work with and I like it ☺

What I am going to conveniently forget about..

- Header files
  - Important for large programs
- We are using a C++ compiler which means some things won't work in C alone
- Real concurrent programming. e.g. locks, semaphores etc.

History of C

- Developed around 1969-1973 at AT&T Labs to be used with UNIX by Dennis MacAlister Ritchie
- Features derived from ‘B’
- Procedural Language
- The addition of struct made it very powerful
- Much of UNIX kernel is in C



Types of C

- Kernighan and Ritchie (K&R)
  - 1978 book *The C Programming Language*
  - The first 'informal' C specification
  - Introduced; standard libraries, long int, compound operators (e.g. i *= 10)
- ANSI C (American National Standards Institute)
  - To establish a standard C
  - Later adopted by ISO (International Organization for Standardization)

Types of C

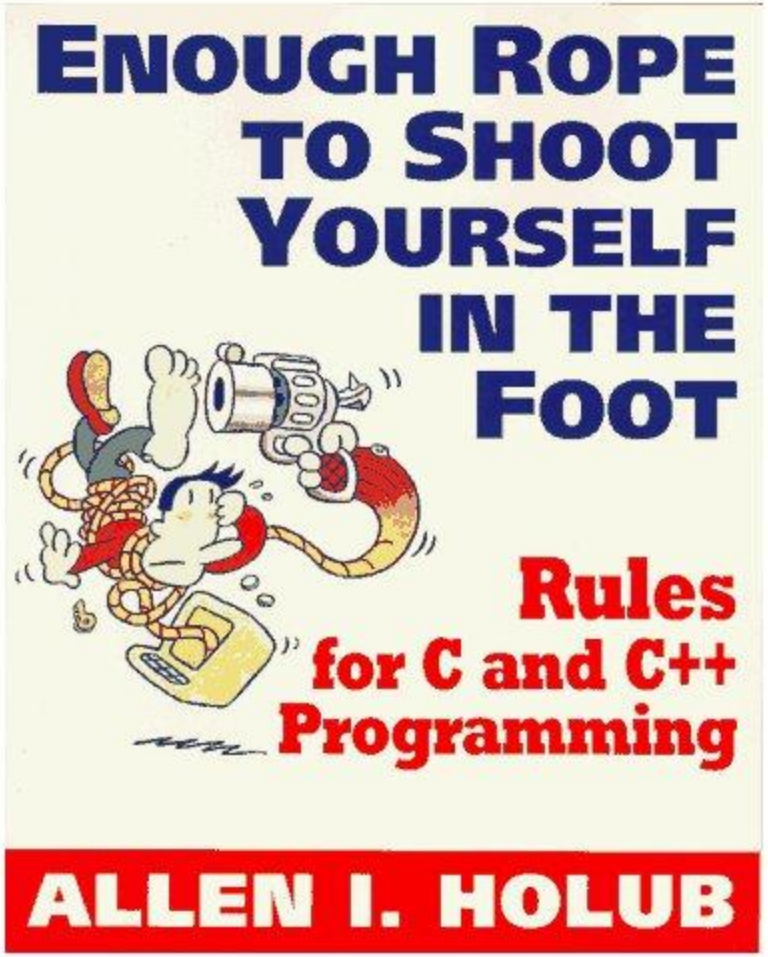
- C99
  - An updated version of ANSI C
  - New features (none of which are important for this lecture series)
  - Been the standard ever since even though C1X has been on the drawing board since 2007

Why is C important?

- Almost all operating system kernels are written in C
  - FYI – A ***kernel*** is the interface between hardware and software – e.g. It lets the programmer use the hardware
- You can work with the most fundamental parts of the computer using C
- As it is ‘low level’, it’s very **very** fast

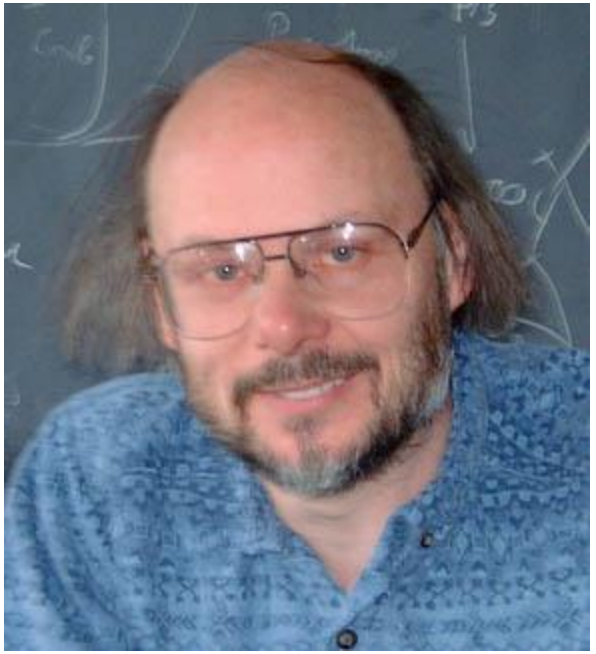
Downsides to C

- Compilation standards are often an issue
- It does things you don't expect
  - It's your fault, not the language!!!!
- You can really mess things up in C



Where did C++ come from?

- Bjarne Stroustrup at Bell Labs
- Originally called “C with Classes”
- Developed in 1979, later became C++ in 1983
- Has enhancements such as classes, overloading, multiple inheritence, exception handling, polymorphism (very cool ☺) and much more

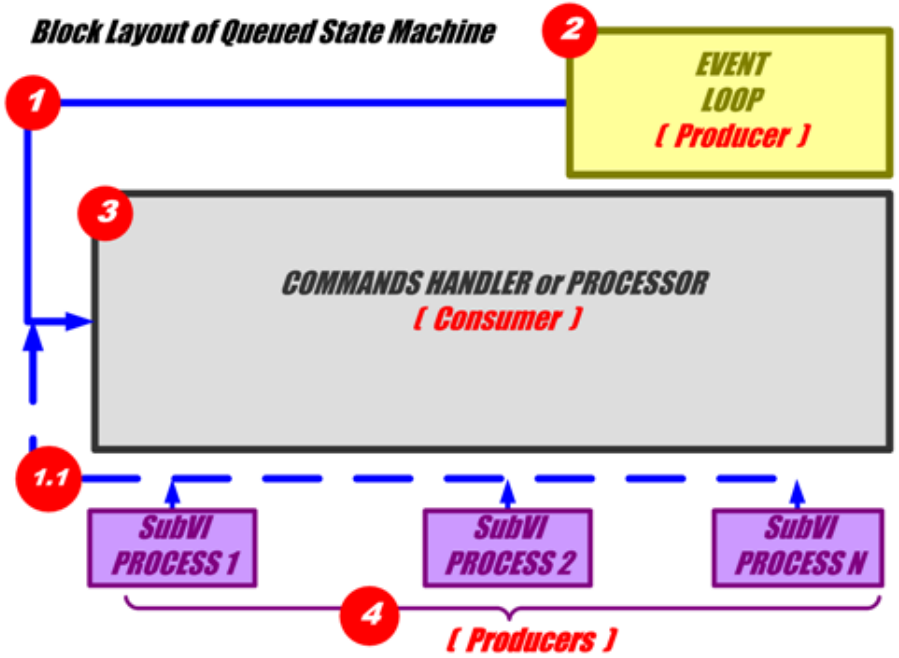


High Performance Computing (HPC)

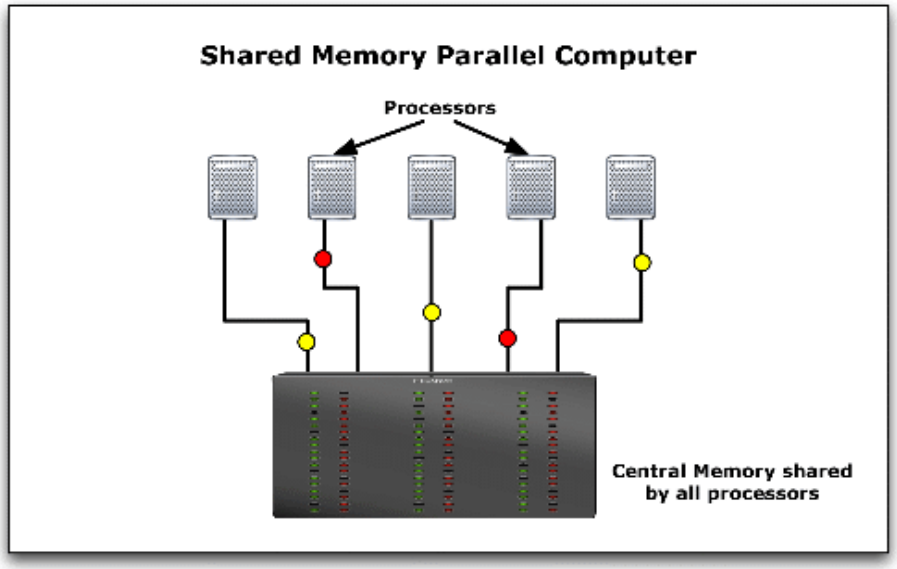
- Big Computers
- Cost a lot of money
- Lots of memory, processors and storage
- Very powerful
- ....we have one in the basement (holly)
  - A Beowulf cluster

HPC – CPUs and Memory

Single CPU and Memory

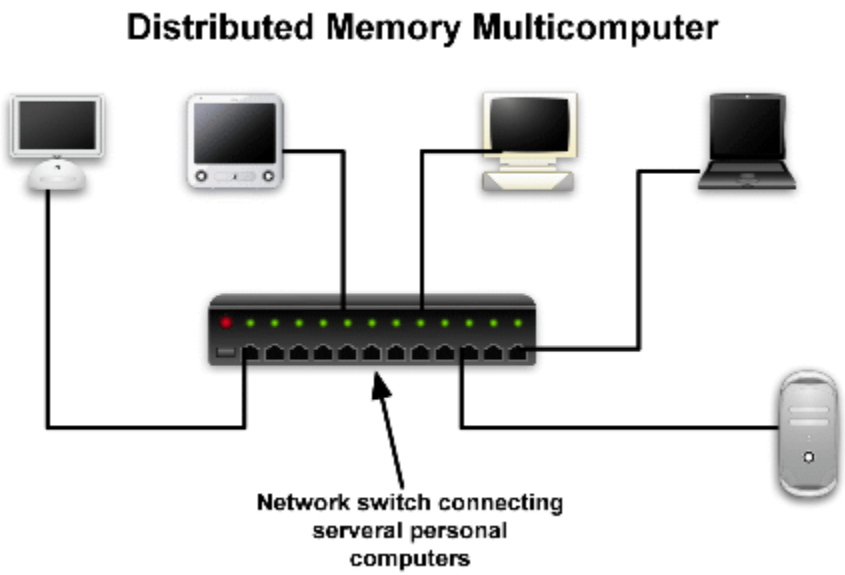


Multiprocessor with Shared Memory



HPC – CPUs and Memory (cont.)

- Multiple CPU and Memory
  - Essentially lots of computers hooked up together with one in charge
- Also known as a Beowulf cluster
- Simply put, lots of computers can be a High Performance Computer



Right...before we program

- UNIX ☺
  - The command line (it's so pretty)
  - Everything runs from the command line
- Editor
  - EMACS, VI, nedit, pico
- Compiler
  - g++, gcc (there are lots more...and they don't all produce the same output)

Hello IMAPS!

- The most basic C program you will write
- Things to think about;
  - `#include <iostream>`
  - `int main( void ) //` you may see just `main()`
  - `cout <<`
  - Lines end in ;
- g++ program.c (simples)
- Produces exec file called a.out
  - Can run `g++ program.c -o myProgram` to produce something else
- To run a.out (or whatever you've typed using -o)
  - `./a.out`

Variables

- Nothing new really;
  - Four common primitive types
    - `char, int, float` and `double`
    - These can be signed or unsigned
      - `unsigned` are non-negative values
      - `signed` are half/half (if you declare `signed int`, it is the same as simply `int`)
    - Sizes
      - `long` or `short` (gives a larger range – architecture specific)
  - Assignment
    - `int a = 10;`
    - `int b = a;`

A little more about primitive types

- Integer types – whole numbers
  - `bool //0 or 1, true of false - 1 bit`
  - `char //8 bits`
  - `short //16 bits`
  - `int //16 bits`
  - `long //32 bits`
- Can be signed or unsigned

An aside - Binary

- Just in case you didn't know;
  - A binary zero or one is called a bit
  - There are 8 bits in a byte
  - A byte is the smallest addressable space in memory
  - Read binary from right to left (an unsigned char)

```
128  64  32  16  8  4  2  1
 0    1   1   0  0  0  1  1  = 99
```

  - It gets more complicated with floating points, HEX etc.

More primitives

- Floating point types
  - `float //32 bit`
  - `double //64 bit`
  - `long double //128 bit`
- And you can make your own!!!
  - We will come to that later
- Let's see how big types are;
  - `sizeof(variable);`

Arithmetic Operators

- Again, nothing new
  - `+ - * /`
  - Extras such as;
    - `()`
    - `%` modulus
    - `++` and `--`(increment and decrement) e.g. `a++` or `b--`
- Orders of precedence
  - Look it up, but it is pretty much what one would expect (well at least at this level)

Conditional and Logical Operators

- Comparing variables
- More stuff you probably know..
  - Conditional
    - equal to == (e.g. `a == b`)
    - not equal to != (e.g. `a != b`)
    - greater/less than < > (e.g. `a > b`)
  - Logical
    - not ! (e.g. `!a`)
    - and && (`a && b`)
    - Or || (`a || b`)

Putting it together with `if` statements

```
if(thing){
  //do something
} else if (other_thing) {
  //do something different
} else {
  //well it's something else
}
```

- This is an if statement...the bread and butter of procedural programming

Putting it together with `if` statements

- CODE EXAMPLE, I suppose we could write something live in a lecture....

Loops

- While loop (will only run if condition is TRUE)

```
while(something){
  //keep doing
}
```

- Do while (will ALWAYS do the loop once)

```
do{
  //keep doing
} while(something);
```

Loops (cont.)

- For loop – my favourite loop ☺

```
for(initialisation; condition; update){
  //do something
}
```

- Example (this will execute 10 times);

```
for(int i = 0; i < 10; i++){
  //do something
}
/*a quick note, this loop wouldn't work
  in C, as you need to declare i outside
  the loop*/
```

Loops.c

Example code

Infinite loop

- DANGER WILL ROBINSON!
- An infinite loop can be caused on purpose or by mistake
  - On purpose

```
while(1){
}
```

  - Bit of a (intentionally daft) mistake

```
int i;
for (i = 1; i > 0; i++) {
  //loop code
}
```

Note: In reality, when `i` gets to the size limit of an `int`, it'll stop

Break and Continue

- break; //exits to the nearest outer loop
- continue; //skips to the test condition of loop
- This example will show loops
  - adv_loops.c //while with break
  - adv_loops2.c //do while with break

Switch statement

- Kind of like an if statement...

```
switch(variable){
case 1:
   //do something when variable = 1
case 2:
   //do something else when variable = 2
default:
   //if none of the others, do something
}
```

Functions

- Break up the program a little

```
#include <iostream>
void myFunction(){
   printf(“Hey, look at me!! I am a function \n”);
}
int main( void ){
   myFunction(); //calls the function
}
```

- `main()` has to be at the bottom of your program

Let's take a minute to think about.....

- Commenting your code

```
/*Everything inside here will be a comment even
  if it's multiple lines*/
int a = 0; //for the rest of the line
```

- Make you code look pretty ☺

  Example

Arrays

- These can get very complicated, but we will start simple....

```
#include <stdio.h>
int main( void ){
   int anArray[10];
   for(int i = 0; i < 10; i++){
      anArray[i] = i; //populate array
   }
}
```

Another way to declare an array

```
#include <stdio.h>
int main( void ){
   int anArray[3] = {1, 2, 3};
   for(int i = 0; i < 10; i++){
      anArray[i] = i; //populate array
   }
}
```

- array.c example

Command line arguments

- This is why we love unix
- Allows program variables to be defined at run time

How to use it

```
#include <iostream>
int main(int argc, char *argv[]){
   cout << argv[1] << "\n"; //print out 1st arg
   cout << argv[0] << "\n"; //program name
   cout << argc << "\n"; //number of args
}
```

- What does char *argv[] actually mean?
  - Basically you have a 2D array...more on this later

So now we have the argument....well it's a `char`

- We need to change it to something else

```
#include <iostream>
#include <stdlib.h>
int main(int argc, char *argv[]){
    int anArray[argc-1];
    for(int i = 1; i < argc; i++){
        anArray[i-1] = atoi(argv[i]);
    }
    for(int i = 0; i < (argc-1); i++){
        std::cout << “Contains ” << anArray[i] << ”\n”;
    }
}
```

Some things to note..

- Notice how we suddenly have to be careful about what our iterator is doing.
- The atoi and others belong to stdlib.h
- `atoi() //converts a char array to int`
- `atol() //...to long integer`
- `atof() //...to double`
- More on strings later

I'm sick of writing `std::cout` So we can use namespaces ☺

- It's a C++ thing, so wouldn't work in C
- At the top of the page you can write;

```
using namespace std;
```

- This allows you to use commands within `std` without having to mention it every time

Since I've mentioned std, what is it?

- This is the ANSI Standard Library
- It contains lots of functions written for C and C++ (since most of the C stuff is old, it's still there but has been renamed)
  - e.g. cstring and string
- http://www.cplusplus.com/reference/
- To use these, you must;

```
#include <header> or #include <name>
```

Some libraries you may wish to use

- `#include <iostream>`
  - We've been using already for `cout`
- `#include <math.h>` or `#include <cmath>`
  - Gives you all your mathematical operators
- `#include <stdio.h>`
  - This is for C++ input/output
  - `printf("This is useful \n" ); //an alternative to cout`

A note on Architecture

- 32 or 64 bit
- Two implications; memory and word size
  - But there are more....
- Compilation is important
- Working with primitive types in memory – use `sizeof()` to be safe (you've already used this)

OK A bit of UNIX goodness ☺

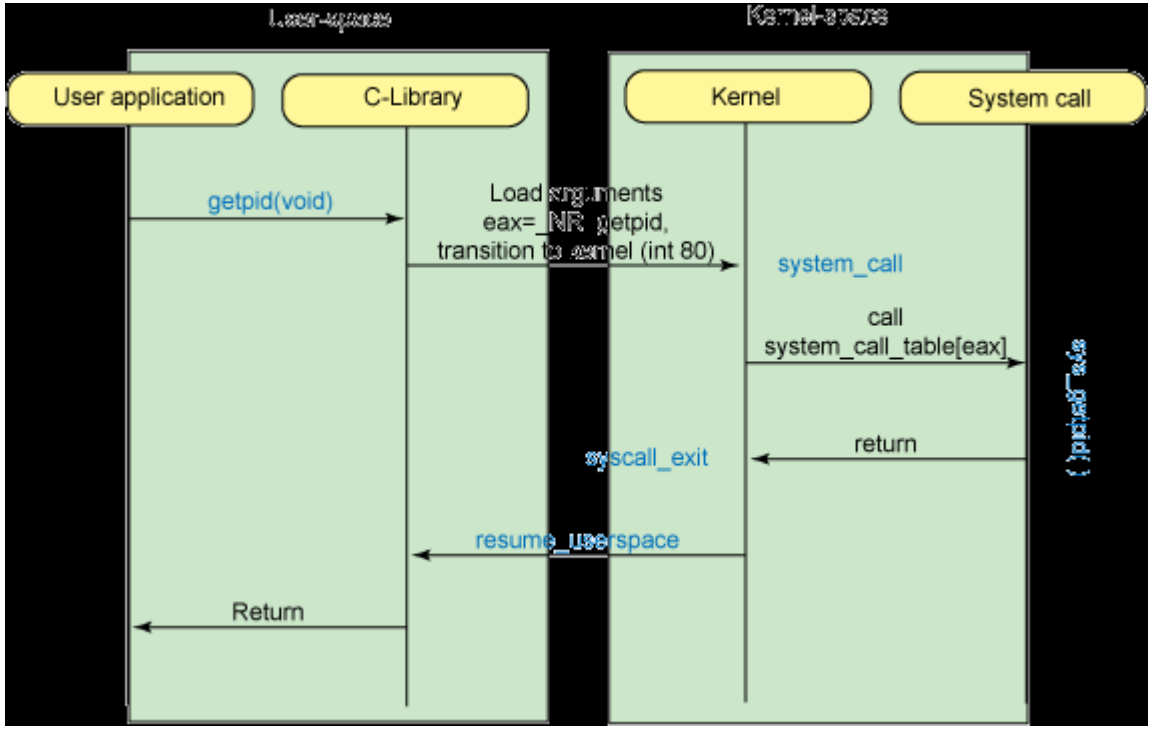
- You can do lots of things from the command line using BASH (Bourne Again SHell)
- A simple usage of this is to run a program multiple times (consider larger programs)
- We have loops in BASH;

```
for i in 1 2 3; do echo $i; done
```

It's a bit different from C/C++

What does your program actually do?

- This can be viewed by using `strace` in UNIX



OK, let's do a few exercises

- Write hello IMAPS – simply to make sure you can use an editor, the command line and g++
- Write a program to calculate the smallest of several values
  - User enters X values into the command line
  - Your program will find the smallest
  - Print it to the screen
- I will show you how to run multiple time from a data file

Reading in a file

- So far, we've taken user input from the command line (which is so pretty ☺)
- However…..there is a limit to how much you can put in the command line
  - E.g. on 32bit Linux kernal 2.6, max = 32,767
- So if we read it in from a file, we can have more ☺

fstream

- A simple version

```
string line;
ifstream myfile("myfile.dat");
while(!myfile.eof()){
  getline(myfile, line);
}
```

What if the file fails to read?

- It crashes and you don't know why
- So….

```
if(!myfile){
   cout << "The file didn't open\n";
   return -1;
}
```

- Did you ever wonder what the `int main()` part did?
  - It defaults to 0 is main executed without error
  - By returning -1, the program exited with error

Strings

- In C there are no strings, we have char arrays
- As such they are a pain to work with
- However luckily C++ has string.h ☺
- So can read out file line by line as a string
- Convert out string to the format we want;
  - int x = atoi(line);

Great, but what if the file contains many columns

- Told you strings were a pain….
- We need to split the line

```
char * split;
split = strtok(line, " ");
while(split !=NULL){
   cout << split << "\n";
   split = strtok(NULL, " ");
}
```

- string_split.c

The bubble sort algorithm

- Very simple (if a little inefficient)
- Sorts numbers into order
- It works like this;
  - Go through every element in the array
  - If the element is larger than the one to its right, then switch it
  - If you've switched it, store somehow that you've made a switch (you don't need to record what you've done)
  - Repeat until you complete a pass without switching anything

Exercise

- Implement bubble sort
- Tip: It's not that different from the smallest value program
- Program specific details
- `./a.out filename.dat 10000`
- The file contains 1 column of integers
- Print out file in the correct order
- The `time` command before the program tells you the time taken to run

Any optimisations?

- Observation:
  - The largest value, even if it is in the very first element of the array, will always be pushed to the end.
- So:
  - We would loop through all the elements the first time, then all but the end one the second time and so on.

Performance

- On 100,000 elements, run 3 times for each algorithm on Intel 2.5Ghz
- Bubble sort
  - 1m11.492sec 1m11.857sec 1m10.324sec
- Optimised Bubble sort
  - 53.253sec 51.658sec 53.383sec

We have lots of files to sort

- OK so we have 10 files to sort, how can we speed things up?
- BASH ☺

A note on HPC

- The easiest way to use HPC is to break the job down into multiple parts, and run it in several places
- Issues:
  - Keeping track of what you've done
  - What if a machine fails? – can you recover

Selection sort

- Go through array looking for the smallest number – 0 .. (size-1)
- Once found, swap the first element in the array with the smallest number
- Repeat for the next element in the array and so on
  - i .. (size-1)
- Performance 28.180sec 26.936sec 36.317sec

Pointers

I’ve been putting this off until now

- OK, so these are a pain, however they can be so powerful
- What is a pointer?

“A pointer is a reference to a place in memory”

Pointers (cont..)

- If we have an integer;

```
int foo = 10; //this is a 4byte variable (you know this)

int *ptr_foo; //this is a pointer to a variable (at the moment there is nothing in it)

ptr_foo = &foo; //sets the address of foo to the pointer

*ptr_foo = 42; //sets the integer that ptr_foo is pointing at to 42
```

Pointers (cont..)

- Confused yet?
- Anyone know why the size of ptr_foo is 8bytes where the size of foo is only 4bytes?
- Can anyone think of why pointers are useful?
- We'll leave pointers there for now and come back to them later

More on functions

- Let's do something useful with a function

```
#include <iostream>
using namespace std;
int square(int number){
    return (number * number);
}
int main( void ){
   int myNumber = 5;
   cout << "The square of " << myNumber << " is " <<
   square(myNumber) << "\n"; //calls the function
}
```

So we have ‘passed’ a variable to the function

- In the previous example, we *‘passed by value’*. Meaning that the value in the function is a copy of the value we passed in from `main()`.
- This is perfectly fine, however;
  - Copying takes time
  - Copying takes up more memory
- Is there another way?

Solution: *sigh* more pointers

- ‘Passing by reference’

```
#include <iostream>
using namespace std;
void square(int *number){
    *number = *number * *number;
}
int main( void ){
   int myNumber = 5;
   square(myNumber);
   cout << “The square of “ << myNumber << “ is “ <<
   mynumber << “\n”; //calls the function
}
```

- Anyone tell me what the output will be?

Structures

- I mentioned before that you can make your own data types;
- These are structures, let's take an example where you wish to write something to deal with 3D coordinates.
- I guess you could us a 3D array
- But every time you wish to work with a point, you need to pass three elements

Structures (cont..)

```
#include <iostream>
using namespace std;
struct point{
   int x;
   int y;
   int z;
};
int main(){
   point p1;
   p1.x = 10;
   p1.y = 20;
   p1.z = -10;
}
```

EVIL OBJECTS!!!!