G52CPP
C++ Programming
Lecture 3

Dr Jason Atkin

Revision so far…

- **C/C++ designed for speed, Java for catching errors**
- Java hides a lot of the details (so ***can*** C++)
- Much of C, C++ and Java are very similar

- C++ **`bool`** values are like Java booleans
  - ints can be used, 0 means false, non-zero (or 1) means true
- Sizes of C/C++ types can vary across platforms
- C provides a powerful library of functions
  - You can use them in C++
  - You *should* **`#include`** the right header file to use them
  - C++ has different formats for some C header files

- *You can split your code across multiple files*
  - *But things need to be declared before you use them*

This lecture

- We will go through most of these slides very quickly so that we can spend time seeing what actually happens in a demo
  - Hopefully you attended the Thursday G52OSC lecture

- Variables, addresses and arrays
- Where things are laid out in memory
- How arrays are laid out
- Running off the end of an array

Pointers and addresses

Reminder from G52OSC : Variables

Every variable has:

A name:	In your program only
An address:	Location in memory at runtime
A size:	Number of bytes it takes up
A value:	The number(s) actually stored

Does it matter:

1) Where a variable is stored?
2) How big a variable is?

Variables and memory

- C/C++ let you find out:
  - Where variables are in memory
  - How big they are
- In Java we don't care
  - In C/C++ we **MAY** care
  - We can take advantage of this for faster code
- I am going to use the kind of table on the right (in yellow) throughout these examples (& later lectures)
- Assume all variables are local variables – defined within some function

Example, local variables:

```
short s1, s2;
long l1, l2;
char c1,c2,c3,c4;
```

Address	Name	Type	Size
1000	s1	short	2
1002	s2	short	2
1004	l1	long	4
1008	l2	long	4
1012	c1	char	1
1013	c2	char	1
1014	c3	char	1
1015	c4	char	1

IMPORTANT WARNINGS

- Addresses in diagrams are for illustration only
- Actual positions of data in memory depend upon
  - Compiler
  - Operating system
  - Whether optimisation is turned on
- For example, you cannot assume:
  - That local variables will be in adjacent areas in memory
  - The ordering of the bytes in a multiple byte data type
- **DO NOT RELY ON POSITIONS OF DATA**
  - **UNLESS YOU KNOW THEY ARE FIXED**
  - There are **some** guarantees (within arrays and structs)

Reminders

- Use the **&** operator in C/C++ to get the address in memory of a variable
- Store this in a pointer
- E.g.: If we have:

```
long longvalue = 345639L;
long* pl = &longvalue;
```

- Note: It may be a good idea to put a p in front of variable name if it is a pointer, pp for pointer to pointer, etc. I don't mind but you may find it helps when it gets complex. Be consistent!

Reminder: Pointers

- `*` after the type it points to is used to denote a pointer
  - i.e. a variable which will hold the address of some other variable
- Examples:

  `char*` is a pointer to a `char`

  `int*` is a pointer to an `int`

  `void*` is a generic pointer, an address of some data of *unknown* type (or a 'generic' address)

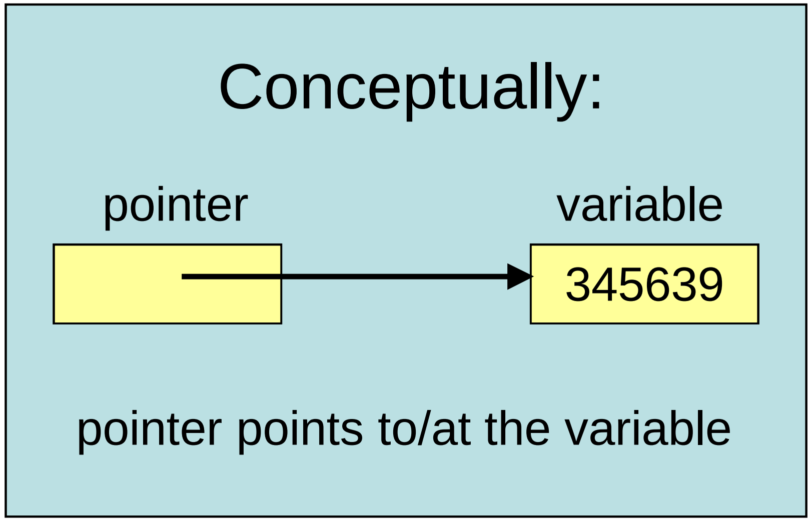
  `char**` is a pointer to a `char*`

  `int***` is a pointer to an `int**`

- Remember two things about pointers:

1. The **value** of the pointer is an **address** in memory
2. The **type** of the pointer says what **type** of data the program should **expect** to find at the address

Reminder: the concept



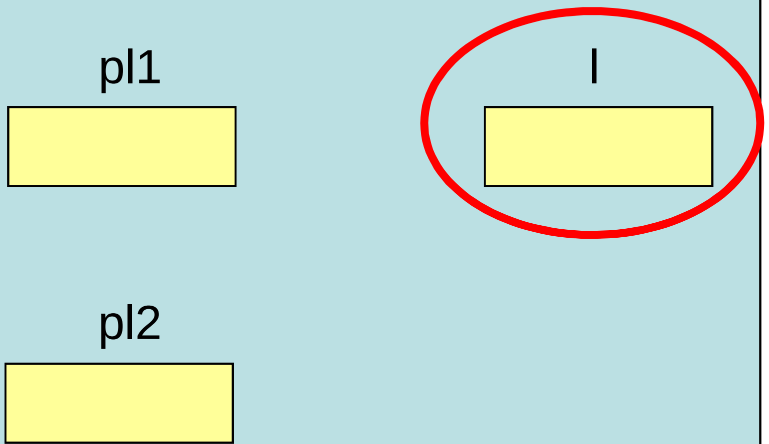
- You can think of pointers whichever way is easier for you
  1. As an **address** in memory and a **type**
  2. As a way of **pointing** to some other data, and a record of what type of data you think the thing pointed at is

Pointer example

```
long l = 32;
long* pl1 = &l;
long* pl2 = pl1;
```

- **Q: What goes into the red circled parts?**

Conceptually:



Actually: (example addresses)

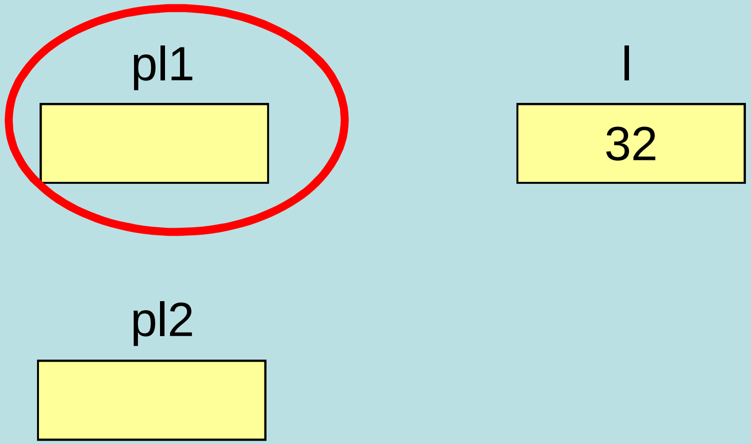
Address	Name	Value
1200	l	
5232	pl1	
6044	pl2	

Pointer example

```
long l = 32;
long* pl1 = &l;
long* pl2 = pl1;
```

- **Q: What goes into the red circled parts?**

Conceptually:



Actually: (example addresses)

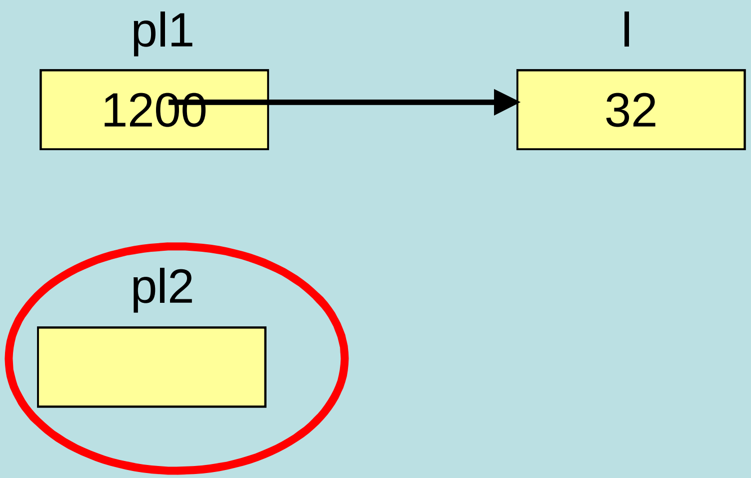
Address	Name	Value
1200	l	32
5232	pl1	
6044	pl2	

Pointer example

```
long l = 32;
long* pl1 = &l;
long* pl2 = pl1;
```

- **Q: What goes into the red circled parts?**

Conceptually:



Actually: (example addresses)

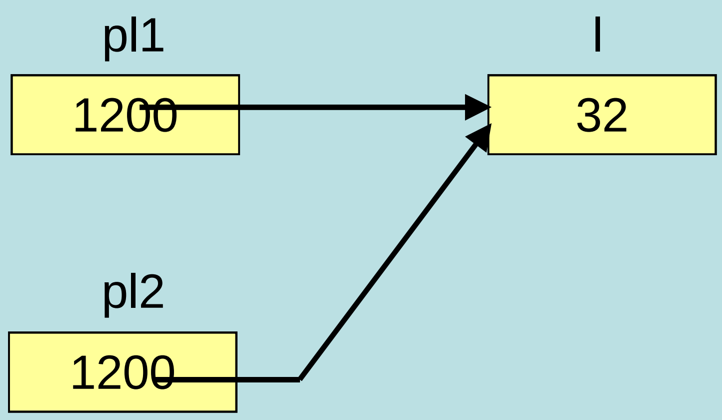
Address	Name	Value
1200	l	32
5232	pl1	1200
6044	pl2	

Pointer example

```
long l = 32;
long* pl1 = &l;
long* pl2 = pl1;
```

- **Assigning one pointer to another means:**
  - **It points at the same object**
  - **It has the same address stored in it (i.e. the same value)**

Conceptually:



Actually: (example addresses)

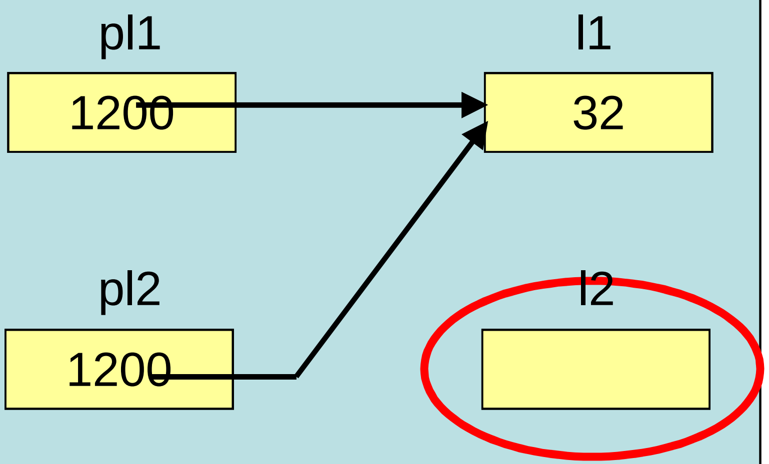
Address	Name	Value
1200	l	32
5232	pl1	1200
6044	pl2	1200

Dereferencing example

```
long l1 = 32;
long* pl1 = &l1;
long* pl2 = pl1;
long l2 = *pl2;
```

- **What goes into the red circled parts?**
  - Hint: What is `*pl2`?

Conceptually:



Actually: (example addresses)

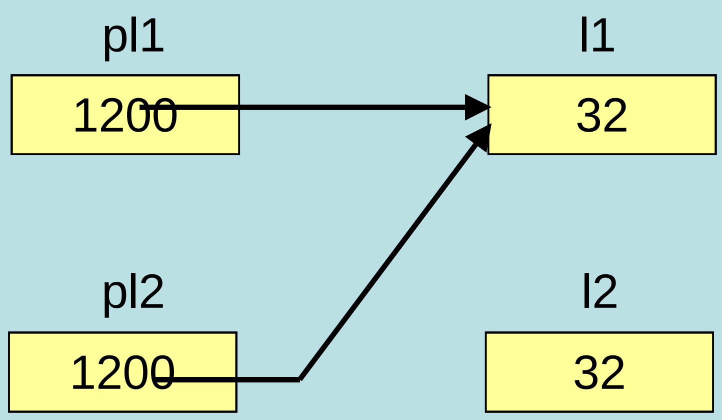
Address	Name	Value
1200	l1	32
5232	pl1	1200
6044	pl2	1200
6134	l2	

Dereferencing example

```
long l1 = 32;
long* pl1 = &l1;
long* pl2 = pl1;
long l2 = *pl2;
```

- So, we can access (use) the value of `l1` without knowing it is the value of variable `l1` (just the value **<u>at</u>** address `pl2`)

Conceptually:



Actually: (example addresses)

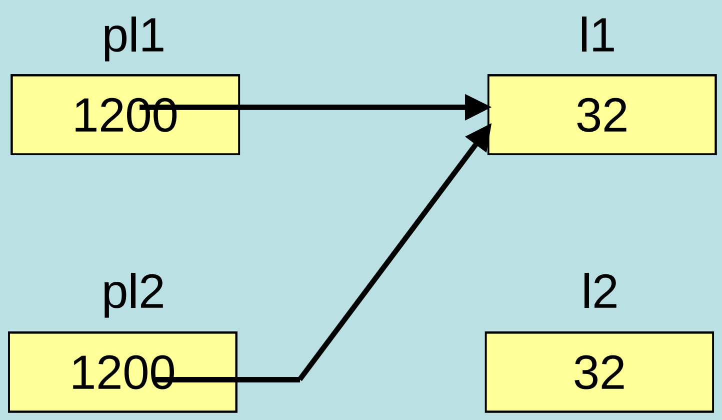
Address	Name	Value
1200	l1	32
5232	pl1	1200
6044	pl2	1200
6134	l2	32

Dereferencing example

```
long l1 = 32;
long* pl1 = &l1;
long* pl2 = pl1;
long l2 = *pl2;
*pl1 = 4;
```

Q: What does this do?

Conceptually:



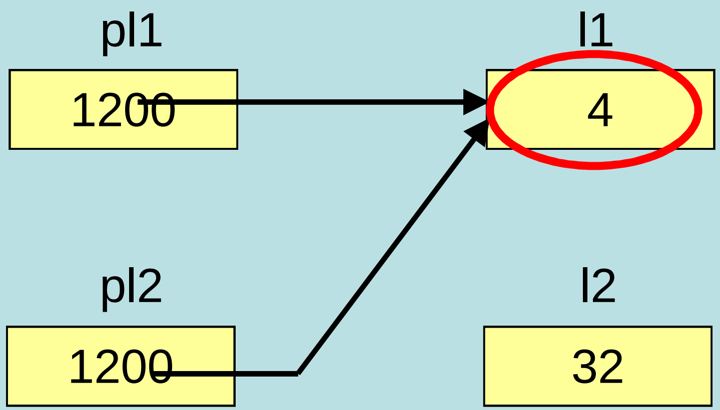
Actually: (example addresses)

Address	Name	Value
1200	l1	32
5232	pl1	1200
6044	pl2	1200
6134	l2	32

Dereferencing example

- ‘`*pl1 = 4`’ changes the value pointed at by `pl1`
- We can change the thing pointed at without knowing what variable the address actually refers to (just ‘change the value at this address’)
- The value of `l1` changed without us mentioning `l1`

Conceptually:



Actually: (example addresses)

Address	Name	Value
1200	l1	4
5232	pl1	1200
6044	pl2	1200
6134	l2	32

Important: Uninitialised Pointers

- In C and C++, variables are NOT initialised unless you give them an initial value
- Unless you initialise them, the value of a pointer is undefined
  - Always initialise all variables, including pointers
  - You can use NULL
- Dereferencing an unitialised pointer has undefined results
  - Could crash your program (likely)
  - Could crash your computer (less likely)
  - Could wipe your hard drive? (unlikely)

Pointer casting and printing

You can cast pointers

- You can cast a pointer into a different type

```
char c1 = 'h';
char* pc2 = &c1;
int* pi4 = (int*)pc2;
```

- The address stays the same in C
  - **There are certain C++ cases where the address may change – ignore these at the moment**
- You are just telling the compiler to expect a different type of data to be at the address
- **Dangerous?** e.g. You are telling the compiler to act as if an `int` is at the location given by `pc2`, but the type of `pc2` says it is actually a `char`

You can print an address

- **%p** in **printf** means expect a (**void***) pointer as the parameter value to replace the **%p** with
- E.g:

```
char c1 = 'h';
char* pc2 = &c1;
printf("%p ",(void*)pc2);
printf("%p\n",(void*)&c1);
```

- Output is in hexadecimal
- Example output:

```
0012FF73 0012FF73
```

1D arrays

Reminder: Uninitialised arrays

- Create an uninitialised array:
  - Add the square brackets [] at the **end** of the variable declaration, with a size inside the brackets
  - e.g. array of 4 **`char`**s: **`char myarray[4];`**
  - e.g. array of 6 **`short`**s: **`short secondarray[6];`**
  - e.g. array of 12 **`char*`**s: **`char* thirdarray[12];`**

- Values of the array elements are unknown!
  - **NOT** initialised!
  - Whatever was left around in the memory locations

Reminder: Initialised arrays

- **Creating an initialised array:**
  - You can specify initial values, in {} (as in Java)
  - E.g. 2 `short`s, with values 4 and 1

```
short shortarray[2] = { 4, 1 };
```

- **You can let the compiler work out the size:**

```
char chararray[] =                    (size 8)
      {'c','+','+','c','h','a','r', 0 };
```

- **Note: If list too short: remaining elements zeroed**
  If list too long: compile time error

Reminder: Arrays in memory

- C-Arrays are stored in consecutive addresses in memory ***(this is one of the few things that you CAN assume about data locations)***
- **Important point:** From the address of the first element you can find the addresses of the others
- **Example: ->**

```
short s[] = { 4,1 };
long l[] ={100000,5};
char ac[] = {
  'c','+','+','c',
  'h','a','r',0};
```

Address	Name	Value	Size
1000	s[0]	4	2
1002	s[1]	1	2
1004	l[0]	100000	4
1008	l[1]	5	4
1012	ac[0]	‘c’	1
1013	ac[1]	‘+’	1
1014	ac[2]	‘+’	1
1015	ac[3]	‘c’	1
1016	ac[4]	‘h’	1
1017	ac[5]	‘a’	1
1018	ac[6]	‘r’	1
1019	ac[7]	‘\0’, 0	1

What we do and do not know…

- The addresses of elements **within** an array **are** consecutive
- The relative locations of **different arrays**, or **variables are NOT** fixed
- Example:

```
short s[] = { 4,1 };
long l[] ={100000,5};
char ac[] = {
   'c','+','+','c',
   'h','a','r',0};
```

- With a different compiler you may instead get a different ordering, or gaps

Address	Name	Value	Size
1000	ac[0]	‘c’	1
1001	ac[1]	‘+’	1
1002	ac[2]	‘+’	1
1003	ac[3]	‘c’	1
1004	ac[4]	‘h’	1
1005	ac[5]	‘a’	1
1006	ac[6]	‘r’	1
1007	ac[7]	‘\0’, 0	1
1020	l[0]	100000	4
1024	l[1]	5	4
1030	s[0]	4	2
1032	s[1]	1	2

Accessing an array element

- Exactly the same as in Java, use []
- E.g.:

```
char ac[] = {'c','+','+','c',
             'h','a','r', 0};
char c = ac[4];
```

- Using what we have seen of pointers:
- `char* pc1 = &(ac[0]);`
- `char* pc2 = &(ac[5]);`

Java vs C arrays : length

- A problem in C/C++ (not Java):

```
char ac[] = {'c','+','+','c','h','a',
   'r', 0};
char c = ac[4];
char c2 = ac[12];
```

← **OOPS!**

- How long is my array?
  - Java arrays include a length
  - C arrays do not. You could:

1. Label the last element with unique value?
2. Store the length somewhere?
3. If you can find the array size, work out the length

Java vs C arrays : bounds checks

- Java will throw an exception if you try to read/write beyond the bounds of an array

- C/C++ will let you read/overwrite whatever happens to be stored in the address if you read/write outside of array bounds
  - Checking would take time: speed vs safety
  - It assumes that you know what you are doing
  - The SHARP KNIFE vs SAFETY SCISSORS

Array names act as pointers

- The name of an array can act as a pointer to the first element in the array:

```
char ac[] = {'c','+','+','c',
             'h','a','r','\0'};
```

- These are equivalent:

```
char* pc3 = &(ac[0]);
char* pc3 = ac;
```

and make **pc3** point to the first element.

Note: **`&ac`** gives same value, different type

You can treat pointers as arrays

- Treating a pointer as an array:

```
char ac[] = {'c','+','+','c',
             'h','a','r','\0'};
char* str = ac;
char c = str[4]; // c gets value 'h'
```

- The **type of pointer** indicates the **type of array**
- The compiler trusts you
  - It assumes that you know what you are doing
  - i.e. it assumes that the pointer really has the address of the first element of an array
- **So if you are wrong, you can break things**

Demo

- Multiple files:
  - You can separate your C/C++ code into multiple files
  - #include header files into .c/.cpp files rather than compiling them on their own
- Compilation stages:
  - Pre-processor: handle #define, #include etc
  - Compile: convert each .c/.cpp file into object code
  - Linker: link the compiled files together

Next lecture

- Pointers and arrays
- char* and strings
- argc and argv
- More demos, to really understand it

- **NO LABS THIS WEEK**