G52OSC
OPERATING SYSTEMS AND CONCURRENCY

Basics of C and Pointers

Dr Jason Atkin

Last lecture

- Overview of module
- The importance of concurrency
  - Data can change when you didn't change it
  - Your process can be interrupted at any time
  - Non-atomic changes can be interrupted
    - E.g. i++
  - Sometimes adding more threads does not help the speed
    - The location of data in memory can matter
      - Even when data itself is not shared!
  - You need to know what you are doing

This Lecture

- #include and #define
  - Conditional compilation
- Pointers
  - Address of operator
  - Copying/assigning pointers
  - Dereferencing
  - Arrays
  - C-style strings

Really simple program

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("Hello world!\n");
    return 0;
}
```

More realistic program

```
#define WIN32_LEAN_AND_MEAN
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_RUNS 2

volatile DWORD dwTotal = 0;
```

```
DWORD WINAPI my_function(
       LPVOID lpParam )
{
   for ( int i = 0;
      i < 1000000;  i++ )
   {
      dwTotal++;
   }
   return 0;
}
```

```
int main(int argc, char* argv[])
{
   int iTN = 0;
   dwTotal = 0;
   for ( iTN = 0;
         iTN < NUM_RUNS;
      ++iTN )
   {
      my_function( … );
   }
   printf("Total %d\n",dwTotal);

   printf( "Press RETURN" );
   while ( getchar() != '\n' )
      ;
    return 0;
}
```

THE C/C++ PRE-PROCESSOR

#include

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("Hello world!\n");
    return 0;
}
```

The C/C++ Preprocessor

- Runs BEFORE passing code to the compiler
  - Compiler will only see the code after the pre-processor has changed it
- It affects statements beginning with `#`
- Examples:
  - `#include`
  - `#define, #undef`
  - `#if, #ifdef, #ifndef, #else, #endif`
  - `#pragma`

#include

- **Replaces this statement by the text of the specified file**
  - For example, to include function declarations
- E.g. **`#include <stdio.h>`**
  - Include the file with standard input/output function declarations in it (e.g. **`printf`**)
  - Looks in the directories on the include path
  - **Normally used for system header files**
  - Note: C++ standard header files may differ – but same effects
- E.g. **`#include "myheader.h"`**
  - The **`""`** usually means look in the project path as well as the main include path
  - **Normally used for your own, project-specific header files**
- Do not confuse with Java's '**`import`**':
  - **`import`** defines the packages to look in for resolving class names (more like the C++ keyword **`using`**, but still different)
  - **`#include`** replaces the line, potentially with function declarations

#define (macros)

- An **semi-intelligent** '*find and replace*' facility
- Often considered ***bad*** in C++ code but useful in C

- Example: define a 'constant':
  - `#define MAX_ENTRIES 100`
  - Replace occurrences of "`MAX_ENTRIES`" by the text "`100`" (without quotes), e.g. in:

```
if ( entry_num < MAX_ENTRIES ) { … }
```

- **Remember:** Done by the pre-processor!
  - E.g. **NOT** actually a **definition** of a **constant**
- 'Constant' `#define`s usually written in CAPITALS

Conditional compilation

- You can remove parts of the source code if desired
  - Done by the pre-processor (not compiled)

- E.g. Only include code if some name has been defined earlier (in the code or included header file)

```
#ifdef <NAME_OF_DEFINE>
    <Include this code if it was defined>
#else
    <Include this code if it was not defined>
#endif
```

- To include only ‘if not defined’ use **`#ifndef`**

PRINTF()

printf()

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("Hello world!\n");
    return 0;
}
```

The printf function

- Reminder: **`printf`** is declared in '**`stdio.h`**'
  - **`#include <stdio.h>`** so compiler knows what it is
- **`printf`** will output **formatted** text
- It uses tags (starting with '%') which are replaced by the supplied parameter values, **in order**
- Examples:

```
int i = 50;
char* mystring = "Displayable string";
printf( "Number: %d\n", i );
printf( "String: %s\n", mystring );
printf( "%d %s\n", i, mystring );
```

POINTERS AND ADDRESSES

Variables

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    int i = 4;
    const char* s = "String";
    printf("%d %s\n", i, s);
    return 0;
}
```

Variables: size and location

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?

Address of : &

- We can ask for the address of a variable
  - And we can ‘write it down’ somewhere
  - This is like asking where someone lives
- Use the **`&`** operator in C/C++
- E.g.: If we have:

```
long longvalue = 345639L;
```

- Then: **`&longvalue`** is the address where the variable **`longvalue`** is stored in memory
  - Like the address of a person in a street/town
- Now we just have to store the address…

Pointers

- `*` 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`

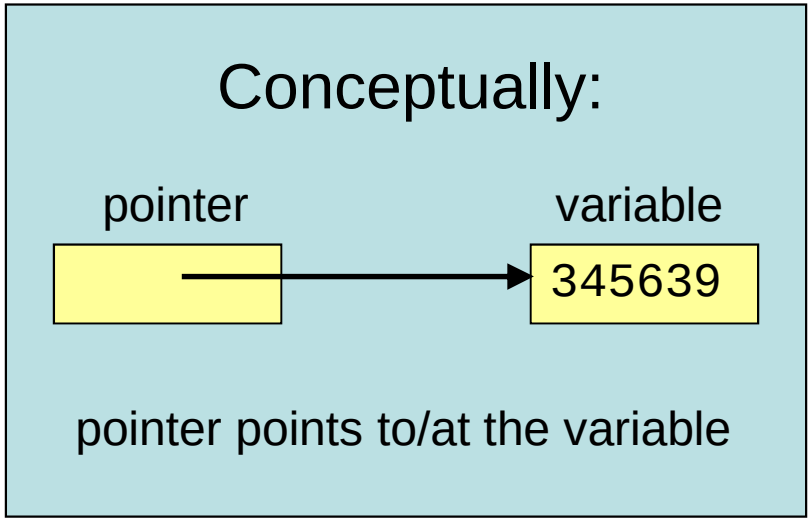
  `int*` is a pointer to an `int`

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

- 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

The concept of a pointer



- 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

Putting & and * together

- Example:
  - Create a long variable

```
long l = 345639L;
```

  - Take the address and store it in a `long*` variable
    - i.e. in a **pointer** to a **long**

```
long* pl = &l;
```

Conceptually:

pl points to/at l

Actually: (example addresses)

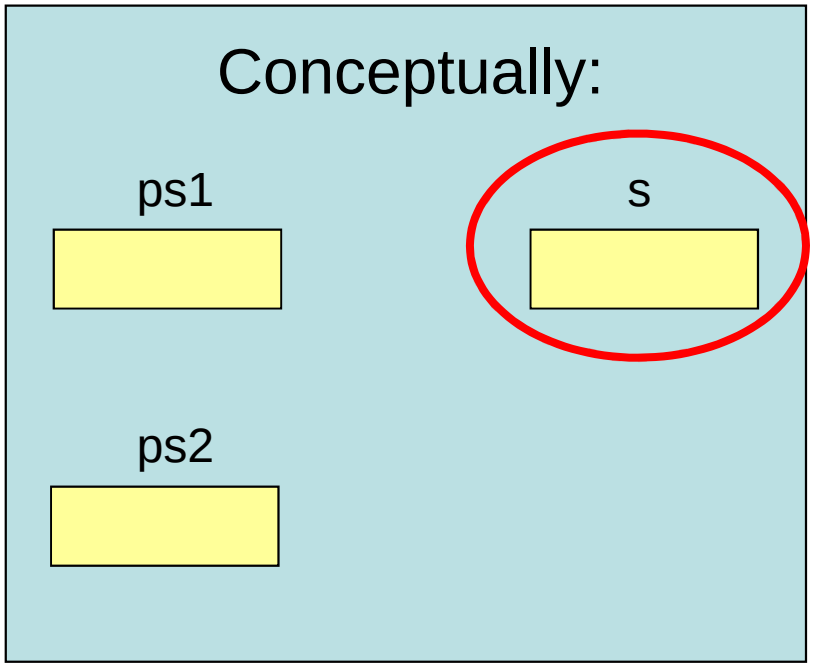
Address	Name	Type	Value
1000	l	long	345639
3056	pl	long*	1000

pl's value is the address of l

Pointer example

```
short s = 965;
short* ps1 = &s;
short* ps2 = ps1;
```

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



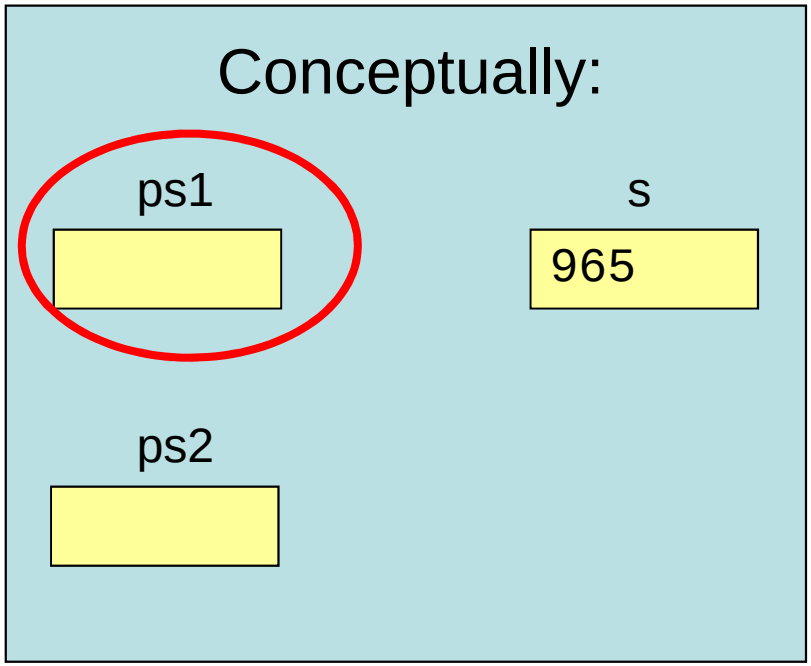
Actually: (example addresses)

Address	Name	Value
3000	s	
5232	ps1	
6044	ps2	

Pointer example

```
short s = 965;
short* ps1 = &s;
short* ps2 = ps1;
```

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



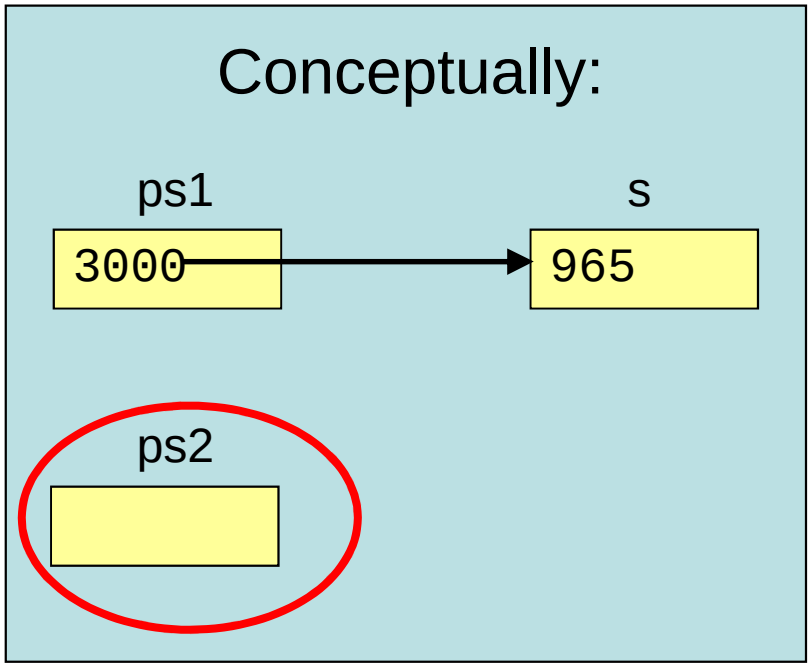
Actually: (example addresses)

Address	Name	Value
3000	s	965
5232	ps1	
6044	ps2	

Pointer example

```
short s = 965;
short* ps1 = &s;
short* ps2 = ps1;
```

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



Actually: (example addresses)

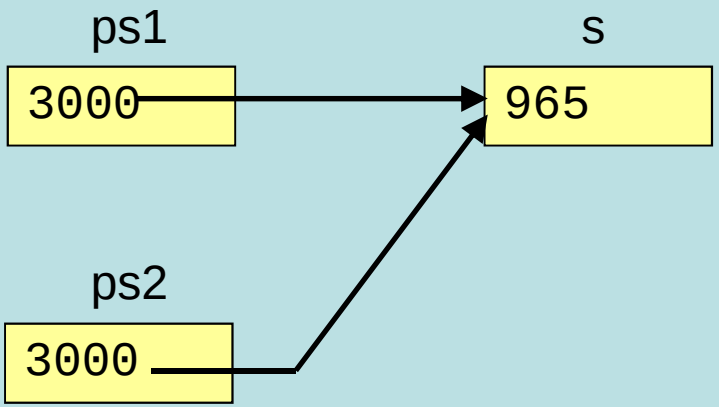
Address	Name	Value
3000	s	965
5232	ps1	3000
6044	ps2	

Pointer example

```
short s = 965;
short* ps1 = &s;
short* ps2 = ps1;
```

- **So, 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
3000	s	965
5232	ps1	3000
6044	ps2	3000

Dereferencing operator : *

- The `*` operator is used to access the ‘thing’ that a pointer points at
- For example: define a `char` and `char*`

```
char c1 = 'h';
char* pc2 = &c1; // pc2 is a pointer to c1
```

- Ask for the value of the thing pc2 points at

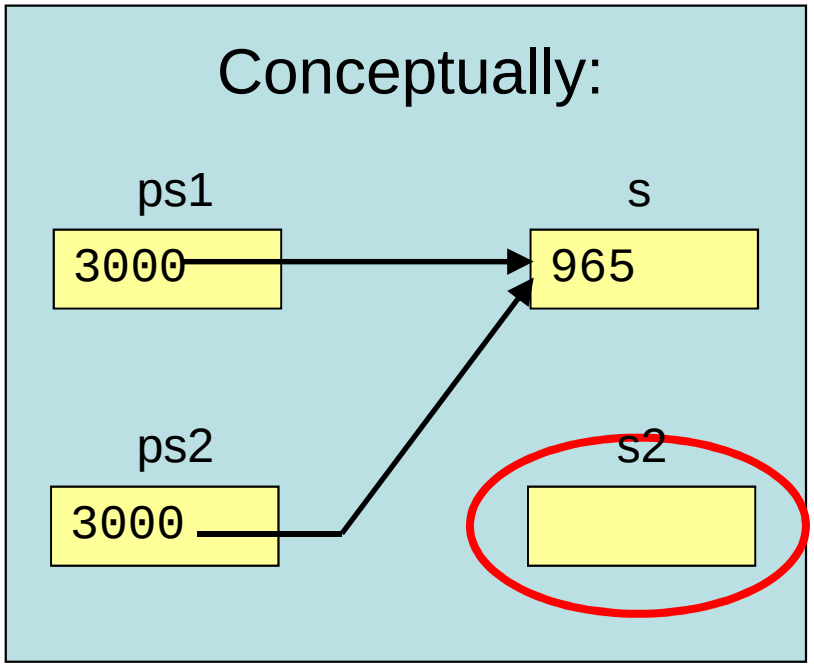
```
char c3 = *pc2;  // *pc2 is thing pointed at
```

- Thinking in terms of pointers holding addresses…
  - `pc2` is a `char*`, so it is the address of a char
  - `*pc2` is the `char` pointed at, i.e. `c1`!
  - So, `*pc2` is (now) another name for `c1`

Dereferencing example

```
short s1 = 965;
short* ps1 = &s1;
short* ps2 = ps1;
short s2 = *ps2;
```

- **What goes into the red circled parts?**
  - Hint: What is ***ps2**?



Actually: (example addresses)

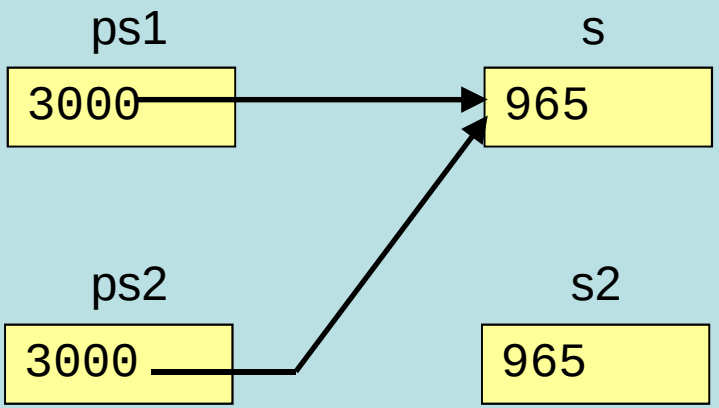
Address	Name	Value
3000	s1	965
5232	ps1	3000
6044	ps2	3000
6134	s2	

Dereferencing example

```
short s1 = 965;
short* ps1 = &s1;
short* ps2 = ps1;
short s2 = *ps2;
```

- So, we can access (use) the value of **`s1`** without knowing it is the value of variable **`s1`** (just the value **at** address **`ps2`**)

Conceptually:



Actually: (example addresses)

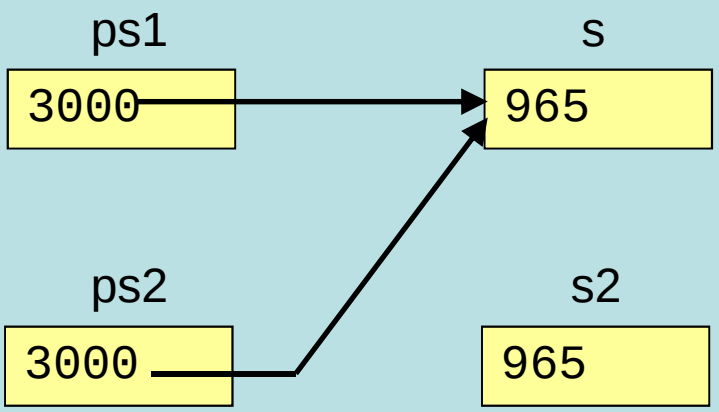
Address	Name	Value
3000	s1	965
5232	ps1	3000
6044	ps2	3000
6134	s2	965

Dereferencing example

```
short s1 = 965;
short* ps1 = &s1;
short* ps2 = ps1;
short s2 = *ps2;
*ps1 = 4;
```

Q: What does this do?

Conceptually:



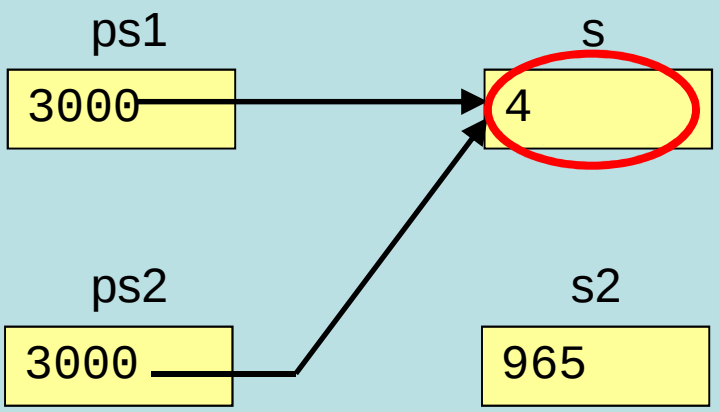
Actually: (example addresses)

Address	Name	Value
3000	s1	965
5232	ps1	3000
6044	ps2	3000
6134	s2	965

Dereferencing example

- '`*ps1 = 4`' changes the value pointed at by `ps1`
- 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 `s1` changed without us mentioning `s1`

Conceptually:



Actually: (example addresses)

Address	Name	Value
3000	s1	4
5232	ps1	3000
6044	ps2	3000
6134	s2	965

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)

ARRAYS AND STRINGS

Simple array creation (1)

- 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 `chars`: `char myarray[4];`
  - e.g. array of 6 `shorts`: `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
- Java would be something like:

```
byte [] myarray = new byte[4];
```

Simple array creation (2)

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

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

  - E.g. 3 **chars**, with values ‘o’, ‘n’ and ‘e’

```
char chararray[3] = {'o','n','e'};
```

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

```
long longarray[] =                    (size 3)
     {100000, 5, 543};
char chararray2[] =                   (size 8)
     {'c','+','+','c','h','a','r', 0 };
```

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

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:

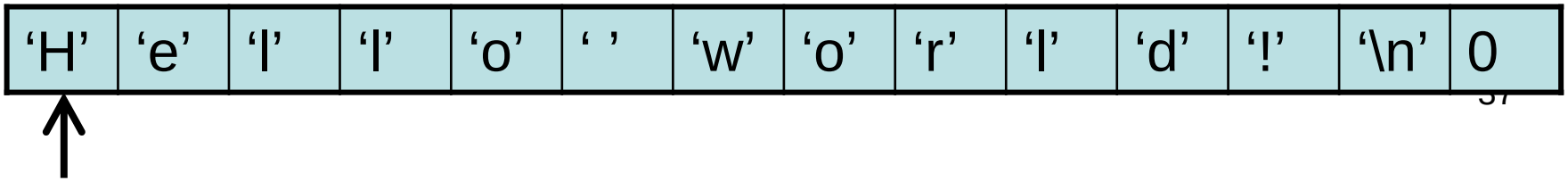
```
short s[] = { 4,1 };
long l[] ={100000,5};
char ac[] = {
   'c','+','+','c',
   'h','a','r',0};
```

- 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

C-string / **`char*`**

- **`char*`** is a pointer to a **`char`**/character
- **C-strings consist of an array of characters, terminated by a character value of zero**
  - The value zero is expressed by **`'\0'`**, or **`0`**
    - **NOT `'0'`!!!** (which is 48 in ASCII)
- Since arrays are in consecutive memory addresses, if we know the address of the first character in the array we can find all of the others



`char*` as a string?

- The **only** reason that a `char*` can act like a string is **by definition**:
  - It was **decided** by someone that strings would be an array of characters with a 0 at the end
  - But, consider the layout of an ASCII text file – it makes sense – this is the way that files are laid out
- There are various string functions in the C library
  - The string functions assume that, the `char*` is a pointer to an array of chars, with a value 0 at the end to mark the end of the array
- E.g.:
  - `printf()` to print a string
  - `strlen()` to determine the length of a string
  - `strcpy()` to copy a string into another string

Standard Library String Functions

- There are many string functions in the standard C library
- You should #include <string.h> to use them
- Examples:

`strcat(s1,s2)`	Concatenates string s2 onto the end of s1
`strncat(s1,s2,n)`	Concatenates up to n chars of string s2 to the end of s1
`strcmp(s1,s2)`	Compares two strings lexicographically
`strncmp(s1,s2,n)`	Compares first n chars of string s1 with the first n chars of string s2
`strcpy(s1,s2)`	Copies string s2 into string s1 ***(assumes room!)***
`strncpy(s1,s2,n)`	Copies up to n characters from string s2 into string s1. ***Again assumes there is room!***
`strstr(s1,ch)`	Returns a pointer to the first occurrence of char ch in string s1
`strlen(s1)`	Returns the length of s1
`sprintf(str,…)`	As printf, but builds the formatted string inside string str. ***ASSUMES THERE IS ROOM!!!***

String literals are arrays of chars

- Example:

```
char* str =
   "Hello!\n";
```

- We have 2 things:
  - A variable of type **`char*`**, called **`str`**
  - An array of chars, **with a 0 at the end** for the string

Address	Value	
10000	‘H’	72
10001	‘e’	101
10002	‘l’	108
10003	‘l’	108
10004	‘o’	111
10005	‘!’	33
10006	‘\n’	?
10007	‘\0’	0

Address	Variable	Value
2000	str	10000

You can manually create ‘strings’

1) Declare an array:

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

2) Get/store address of the first element:

```
char* pc = ac;
```

3) Pass it to **printf**:

```
printf(“%s”, pc);
```

or just use array name:

```
printf(“%s”, ac);
```

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

Next Lecture

- Threading and process fundamentals
- Creating a process using fork()
  - On Linux
- Creating a new thread
  - Windows and Linux
- What happens to our global variables?