G52CPP
C++ Programming
Lecture 3

Dr Jason Atkin

E-Mail: jaa@cs.nott.ac.uk

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

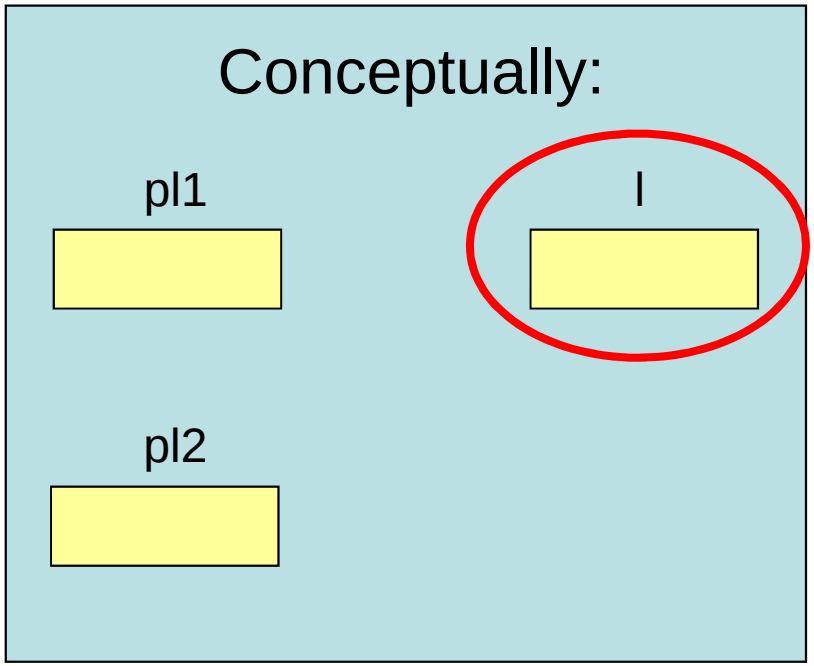
- **`char*`** is a pointer to a **`char`**, but can sometimes be treated as a string

- 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 *should* **`#include`** the right header file to use them

Pointer example

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

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



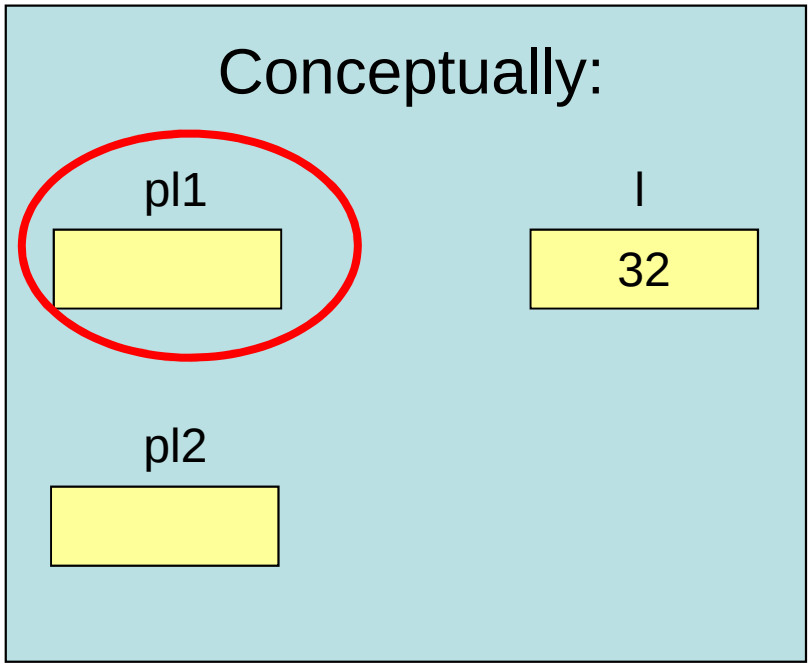
Actually: (example addresses)

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?**



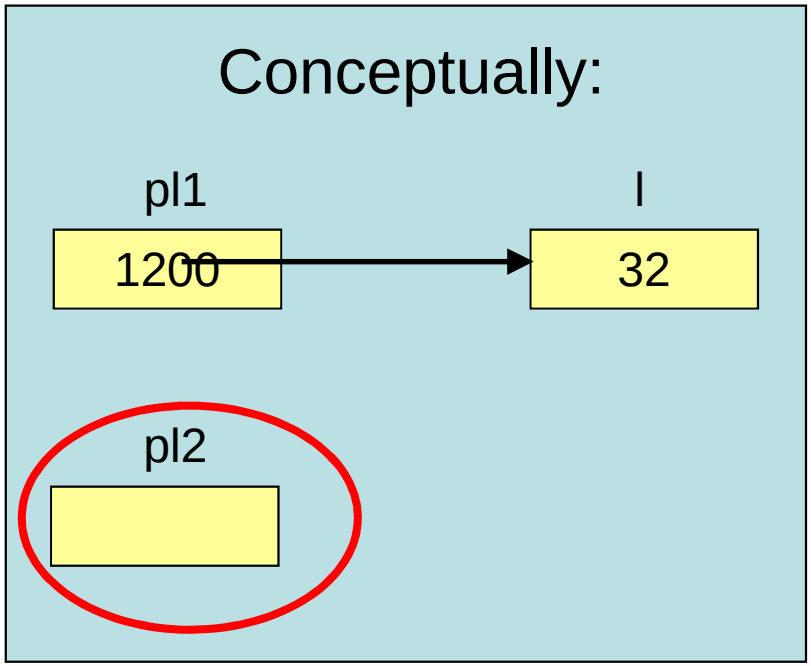
Actually: (example addresses)

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?**



Actually: (example addresses)

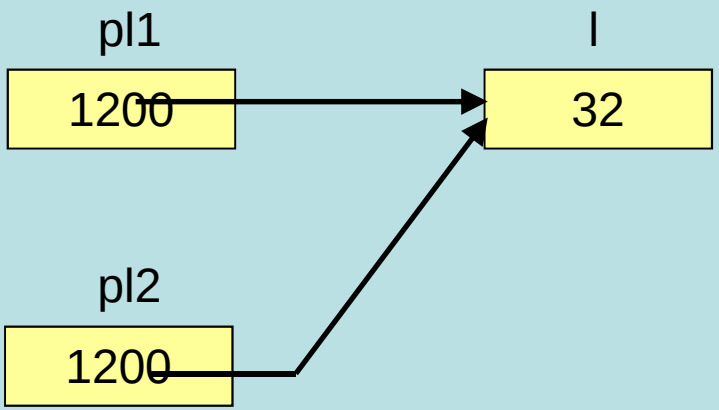
Address	Name	Value
1200	l	32
5232	pl1	1200
6044	pl2	

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:



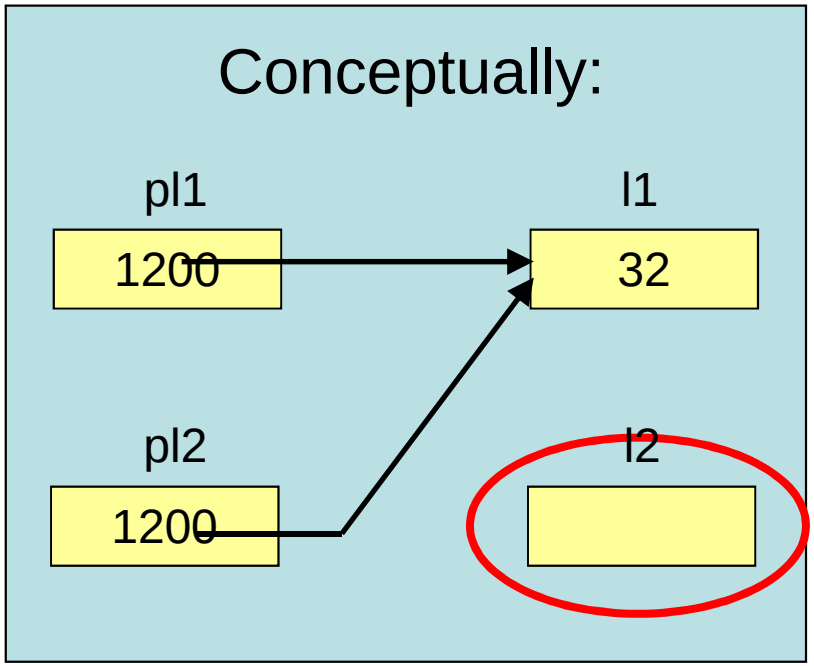
Actually: (example addresses)

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`?



Actually: (example addresses)

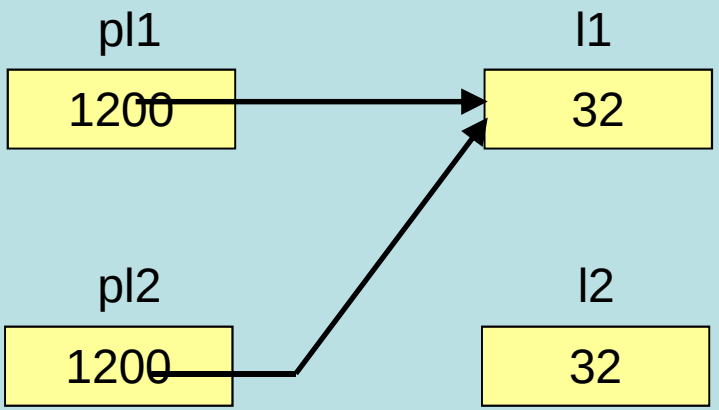
Address	Name	Value
1200	l1	32
5232	pl1	1200
6044	pl2	1200
6134	l2	

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 **at** address **`pl2`**)

Conceptually:



Actually: (example addresses)

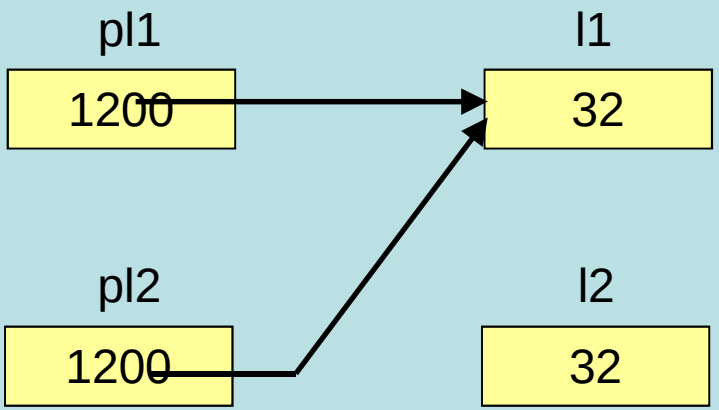
Address	Name	Value
1200	l1	32
5232	pl1	1200
6044	pl2	1200
6134	l2	32

Dereferencing example

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

Q: What does this do?

Conceptually:

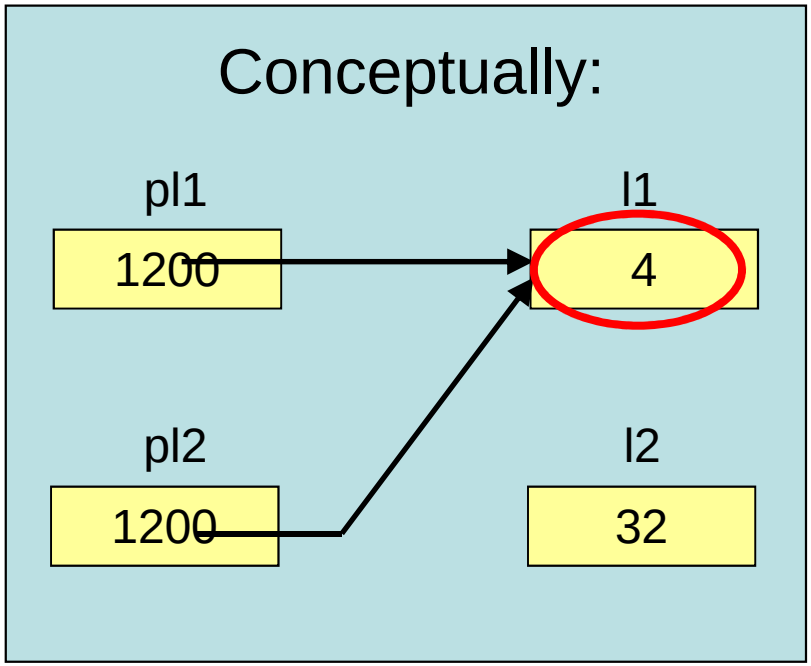


Actually: (example addresses)

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`



Actually: (example addresses)

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

Lecture Outline

- Arrays
  - Only one-dimensional arrays
  - Fit well with pointers
- char* and C-strings
- argv and argc

An introduction to (1D) arrays

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 **`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

Simple array creation (2)

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

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

  - E.g. 3 **char**s, 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 };
```

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

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:

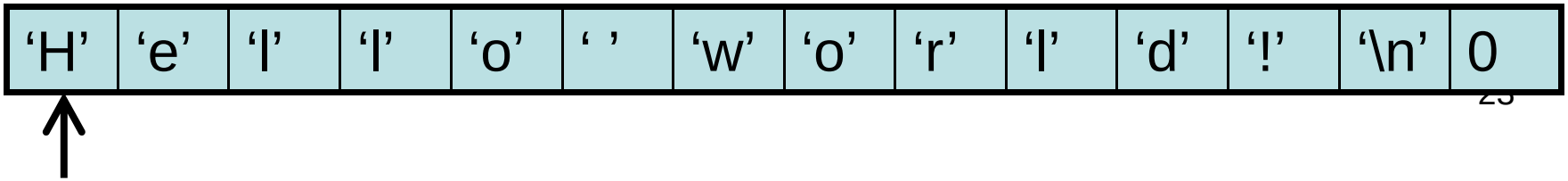
```
char ac[] = {'c','+','+','c',
             'h','a','r','\0'};
char* str = ac;
char c = str[4]; // c gets value ‘h’
```

- 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**

char* and C-String

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

- We have treated **`char*`** as a ‘string’
- In fact it 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:
  - 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 <cstring> to use them
- ***You need to know these and what they do***
- 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

Initialisation of a char array

- You can ***initialise*** a char array from a string, so the following are equivalent:

```
char c1[] = "Hello";
char c2[] = {'H','e','l','l','o','\0'};
```

- **This is a special case for char arrays**
- It is different to:

```
char* c3 = "Hello";
```

  - Which creates a POINTER, not an ARRAY
  - A ‘little’ confusing

Would this code work?

```
#include <cstdio>

int main()
{
   char c1[] = "Hello";
   char c2[] = { 'H', 'e', 'l', 'l', 'o', 0};
   char* c3 = "Hello";

   c1[0] = 'A';
   c2[0] = 'B';
   c3[0] = 'C';

   printf( "%s %s %s\n", c1, c2, c3 );
   return 0;
}
```

Example

```
#include <cstdio>

int main()
{
   char c1[] = "Hello";
   char c2[] = { 'H', 'e', 'l', 'l', 'o', 0};
   char* c3 = "Hello";

   c1[0] = 'A';
   c2[0] = 'B';
//c3[0] = 'C'; // Would probably segmentation fault

   printf( "%s %s %s\n", c1, c2, c3 );
   return 0;
}
```

- But it would compile!

Important!

Not all **`char*`**s are C-Strings

- **This is important to remember**
- **A C-string is a `char*` which points to an array of characters with a 0 to mark the end**

- Note: The parameter for **`main()`**

  `char* argv[]`

  IS an array of C-strings
- There is no way to know this from the parameter type, but we **know** (from other information) that **`main`** always gets passed an array of C-Strings

`argc` and **`argv`**

The “Hello World” Program

```c
#include <stdio.h> /* C file */

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

C version

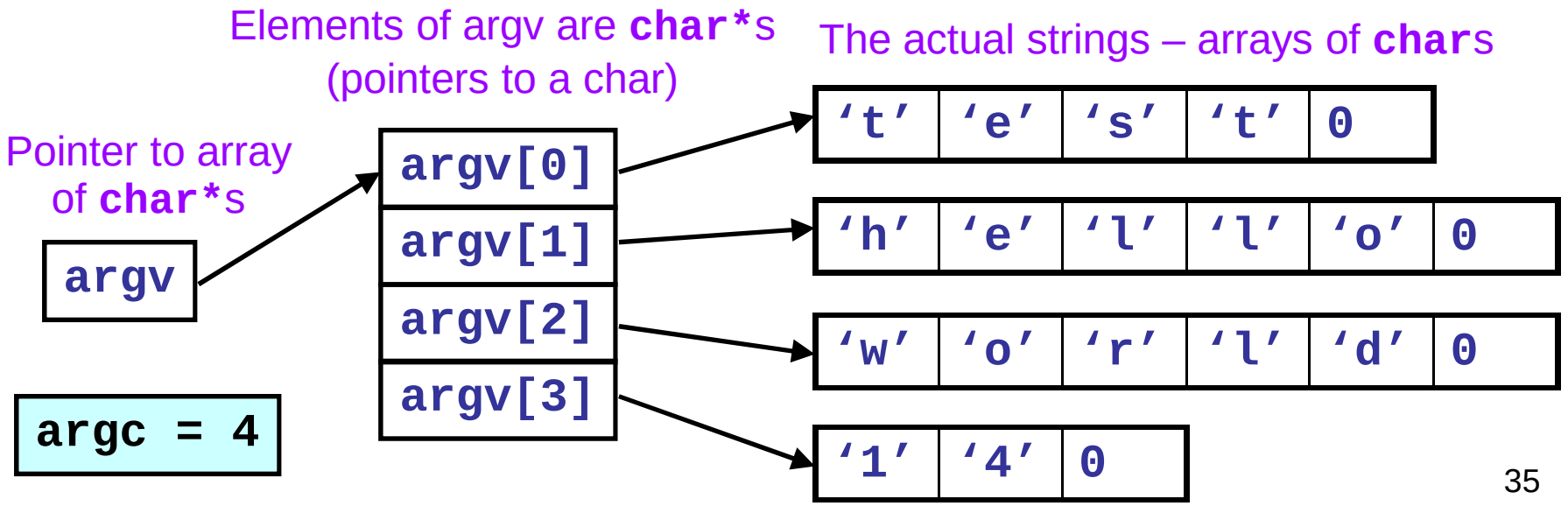
```cpp
#include <cstdio> /* C++ file */

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

C++ version

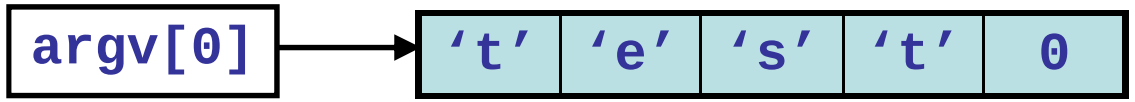
Command line arguments

- **`int main(int argc, char *argv[])`**
- **`argc:`** count of arguments – including the filename
- **`argv[]`**: array of **`char*`**s
- **`argv[i]:`** a **`char*`** pointing to an array of chars
- To get a character from an array, use **`[]`** (or **`*`** to get first)
- e.g. command line: '**`test hello world 14`**'



Use of command line args

- What can we do with command line arguments?
- Treat them as a string:
  - e.g.

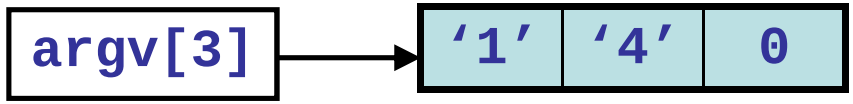



```
printf( "Filename was %s\n", argv[0] );
```

- Extract a character from them:
  - e.g.


```
char* param = argv[1];
printf( "%c,%c,%c\n", param[0], *param, param[1]);
printf( "%c, %c\n", *argv[1], argv[1][0] );
```

- Convert a string *(not a char!)* to an integer
- e.g.




```
int iVal = atoi(argv[3]);
```

main()

- You don’t need to declare the parameters for main

```
int main()
```

- You can declare argv as:

```
char** argv
```

  – instead of

```
char* argv[]
```

  – The two forms are equivalent

  – Both forms are pointers to pointers

Determining string length

Example: strlen()

- **`int strlen( char* str )`**
  - Get string length, in chars
  - Check each character in turn until a '\0' (or 0) is found, then return the length
  - Length excludes the '\0'

```
int mystrlen( char* str )
{
   int i = 0;
   while ( str[i] )
      i++;
   return i;
}
```

Address	Name	Value
1000	str[0]	'C'
1001	str[1]	' '
1002	str[2]	's'
1003	str[3]	't '
1004	str[4]	'r'
1005	str[5]	'i'
1006	str[6]	'n'
1007	str[7]	'g'
1008	str[8]	'\0', 0

Remember from lecture 2, integers can be used in conditions
Value 0 means false, non-zero means true.

Summary

Pointers are important

- If you understand pointers, many other things will make sense
- Do not worry if it is not entirely clear now
  - But please go through these slides until it is
- Pointers are not complex
  - Just remember that they just store an address of something else
  - And the type of thing that they point at
  - I.e. They point to something else

Arrays

- You can easily create arrays
  - Initialised or uninitialised
- **Array elements are stored in consecutive areas of memory**
  - Very useful – see next lecture
- **No length is stored for an array**
  - If you need it you need to store it or work it out
- **No bounds checking is performed when you use an array**
  - The compiler **trusts** you, so why waste time checking up on you?

Next lecture

- More pointers
  - Pointers can be treated as arrays
  - Pointer casting and printing
  - Pointer arithmetic
- Functions:
  - Declarations and definitions
- Passing pointers as parameters