G52CPP
C++ Programming
Lecture 4

Dr Jason Atkin

Office Hours and Labs

- For lab questions please ask in the lab
  - If we need more time I can get lab helpers to help at other times too
- For coursework issues, we will have extra lab sessions with the lab helpers
- For course questions or other issues see me either:
  - After the Tuesday lecture (5pm outside LT3)
  - In office hours, 11-12noon Wednesday

Lectures so far

- Introduction
- Summary of what you should already know about C
- Pointer reminders were in the G52OSC lecture
  - Assigning a pointer to another copies the address – makes it point at the same thing

    ```
    char* p2 = p1; // p1 is a char*
    ```
  - & (address of) and * (dereference)
- More pointers + arrays

Arrays

- **Array elements are stored in consecutive areas of memory**
  - Very useful
- **No length is stored for an array**
  - If you need it, 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?

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**

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

Pointer and array similarities

- **Array names are pointers to the first element in the array**

```
char str[] = { 'H',
   'e','l','l','o','!',
   '\n‘, 0};
char* p = str;
```

   p has value 1000 here

- **Pointers can be treated as arrays**:

```
char c = p[4];
```

   c has value ‘o’

Address	Value	Name
1000	‘H’	str[0]
1001	‘e’	str[1]
1002	‘l’	str[2]
1003	‘l’	str[3]
1004	‘o’	str[4]
1005	‘!’	str[5]
1006	‘\n’	str[6]
1007	‘\0’	str[7]
1008	**1000**	**p**

Arrays allocate memory to store values, pointers do not

Aside: do not use variable sized arrays

- Variable length arrays are **NOT** valid in C++
  - Sadly, gcc on avon, bann etc will allow them in C++
- E.g.:

```
int myfunc( int iSize )
{
        char array[iSize];
        …
}
```

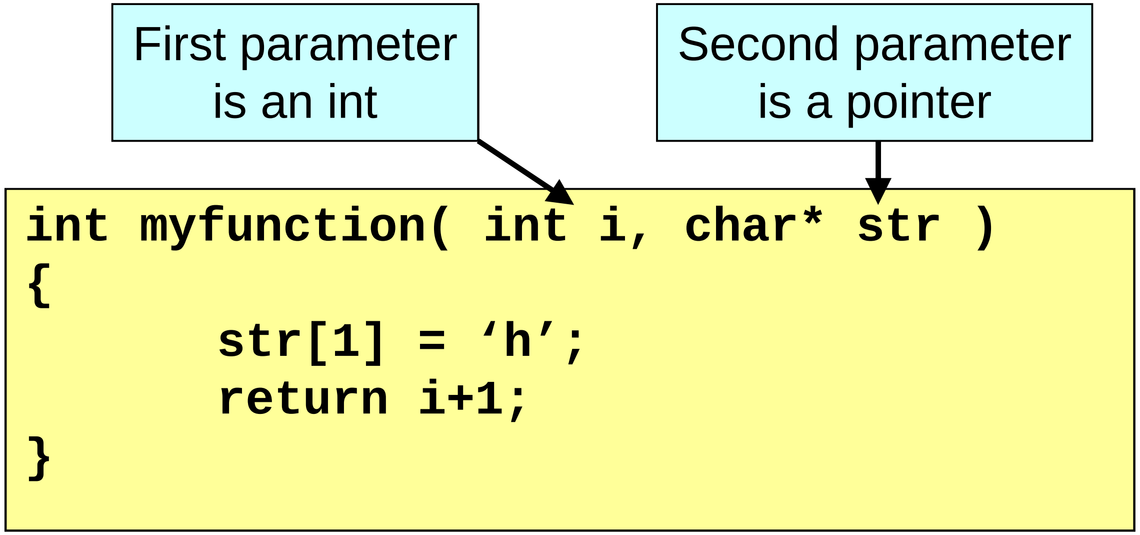
  - Size of array is not a constant, it depends upon the value of variable
- **You must use a numeric literal or a constant for a size**
  - You can use a `#define` to set it to a literal
- If you need variable size arrays, use `malloc()` or `new`
- Use: `g++ -pedantic myfile.cpp` to get a warning

This Lecture

- Functions:
  - Declarations and definitions
  - Passing pointers as parameters
- char* and C-strings
- argv and argc

Passing pointers as parameters

Parameters can be pointers



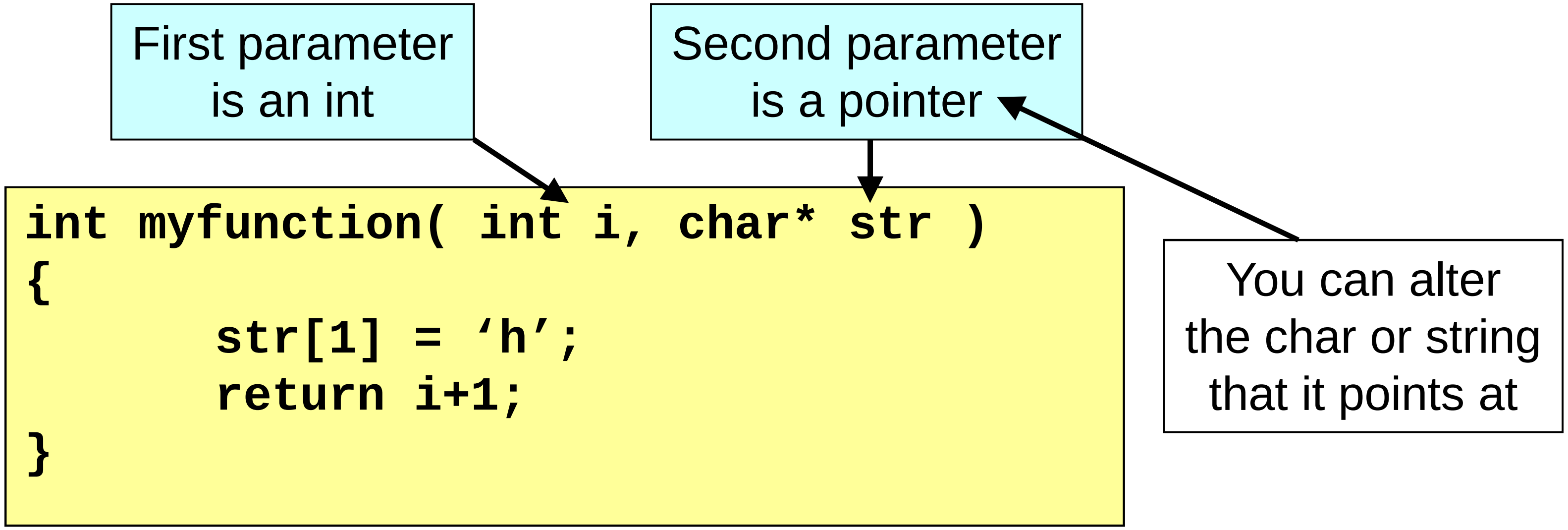
First parameter is an int

Second parameter is a pointer

```
int myfunction( int i, char* str )
{
        str[1] = ‘h’;
        return i+1;
}
```

- Each parameter has a single type, so may be one ‘thing’
- A **copy** of the ‘thing’ is stored in the memory for the parameter
  - – i.e. the function gets its own copy!
  - – Of a variable (incl pointer), literal value, etc

Parameters can be pointers



First parameter is an int

Second parameter is a pointer

```
int myfunction( int i, char* str )
{
        str[1] = ‘h’;
        return i+1;
}
```

You can alter the char or string that it points at

- If you want to **alter** something that is external to a function from within a function, you need to **refer to the thing** itself, **not a copy of it**:
  - Easy way is to pass a pointer to it
  - A copy of a pointer will point to the same thing
    - i.e. It will copy the address rather than the thing pointed at
    - Thus you can change the thing at that address

Example: pointer parameter

```
void AlterCopy( int icopy )
{
   icopy = 2;
}
void AlterValue( int* picopy )
{
   *picopy = 3;
}
int main( int argc, char* argv[] )
{
   int i = 1;
   printf( “Initial value of i is %d\n”, i );
   AlterCopy( i );
   printf( “After AlterCopy, value of i is %d\n”, i );
   AlterValue( &i );
   printf( “After AlterValue, value of i is %d\n”, i );
   return 0;
}
```

Java makes the decision for you

- **Java object references act like pointers**
  - They reference (point to) the same object, rather than a copy
- Consider the following Java code:

```
public static int main()
{
  int i = 42;
  MyClass ob = new MyClass();
  myFunc( ob, i );
}
static void myFunc( MyClass ob, int i )
{
  i = 23; // Does not affect the i in main.
  ob.set…( … ); // References the same ob as in main
}
```

- Here a reference to the object is passed, not the object itself

Summary of parameter passing

- To allow a function to alter a variable, pass its address
  - i.e. a pointer to it
  - The value of the ***pointer / address*** is copied
  - Note: Can also use references (C++ only, later lecture)
- To just provide data, you can pass the value
  - But passing the address may sometimes be quicker, less data to copy for big objects
- e.g. When you pass a '**`char*`**' to a function, the function can alter the contents of the string pointed at
  - *Through* the pointer
- **`strcpy()`** uses this to copy a string

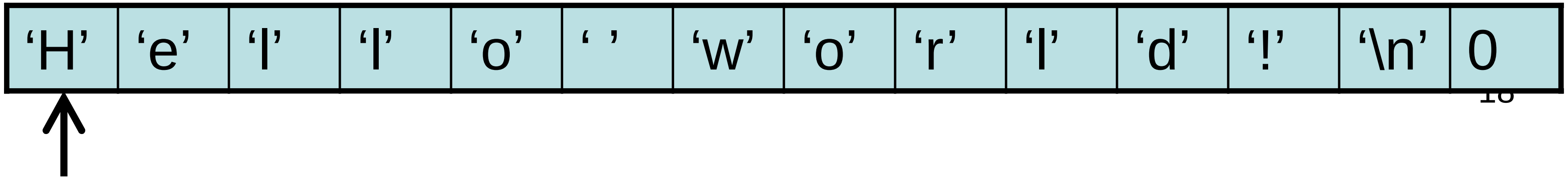
The return statement

- Functions can return only ONE value
- **The returned value is copied!**
- The value may be:
  - a basic type (e.g. `int`)
  - a pointer (or C++ reference, see later)
    - The address is copied (same for references)
  - a struct, union (see later) or object (C++ only)
    - The struct, union, object etc is copied
- May create a temporary variable in calling function, to store the returned value

char* and C-String

Reminder: 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



‘H’	‘e’	‘l’	‘l’	‘o’	‘ ’	‘w’	‘o’	‘r’	‘l’	‘d’	‘!’	‘\n’	0

Reminder: `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

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

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

C version

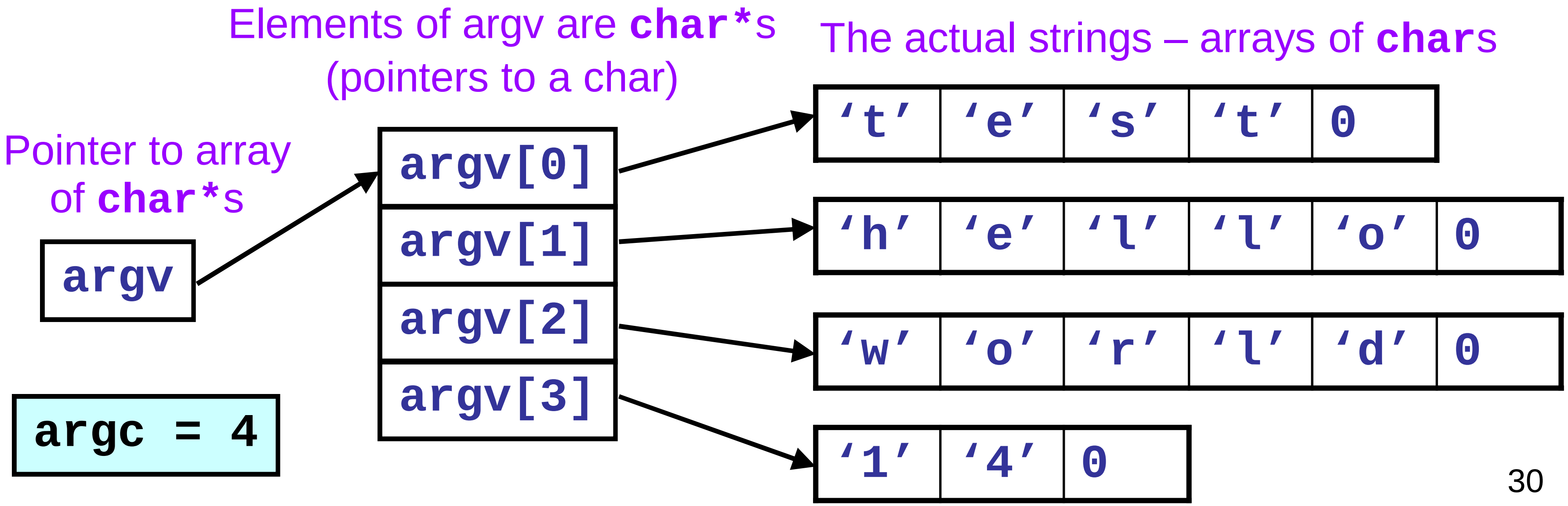
```
#include <cstdio> /* C++ file */

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`’



Elements of argv are **char***s (pointers to a char)

The actual strings – arrays of **char**s

Pointer to array of **char***s

argv

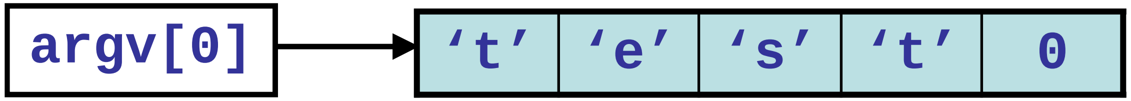
argc = 4

argv[0]
argv[1]
argv[2]
argv[3]

‘t’ ‘e’ ‘s’ ‘t’ 0
‘h’ ‘e’ ‘l’ ‘l’ ‘o’ 0
‘w’ ‘o’ ‘r’ ‘l’ ‘d’ 0
‘1’ ‘4’ 0

Use of command line args

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

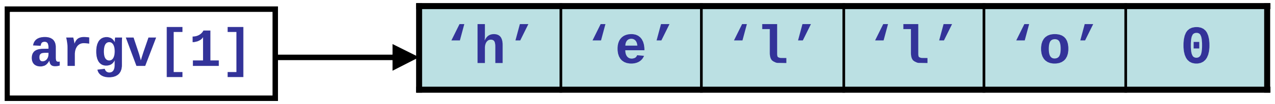


argv[0]

‘t’ ‘e’ ‘s’ ‘t’ 0

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

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

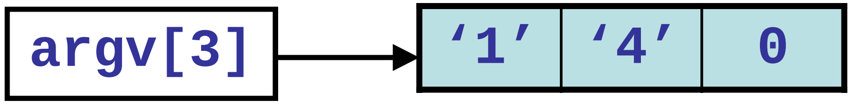


argv[1]

‘h’ ‘e’ ‘l’ ‘l’ ‘o’ 0

```
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.



argv[3]

‘1’ ‘4’ 0

```
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

- Pointer arithmetic
- Passing pointers as parameters