G52CPP
C++ Programming
Lecture 4

Dr Jason Atkin

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

Previous 2 lectures

- `&` operator : address-of
- Addresses are stored in pointers
- Copying a pointer:
  - Copy points to the same thing
  - I.e. the address is copied
- `*` operator : de-reference the pointer
  - Get/use the thing pointed at
- C-string : array of `char`s with a 0 at end
- `int argc` and `char* argv[]`

What does this code do?

- The following code will compile:

```
char c = 'H';
printf( "%s\n", &c );
```

- **`printf()`** wants a **`char*`**, **`&c`** is a **`char*`**
- Surely this is OK isn't it?
  - Afterall, it does compile!

- **Q: What do you think the output is?**

This lecture

- More pointers
  - Array names are pointers to first element
  - Pointers can be treated as arrays
  - Pointer casting and printing
  - Pointer arithmetic
- Functions:
  - Declarations and definitions
- Passing pointers as parameters

Arrays and pointers

Reminder: arrays

Arrays can be of basic types, pointers, objects, functions, … (as we will see later)

Arrays can be initialised or uninitialised

```
short sa[12];
char* pca[3];
long la[2] = { 4, 1 };
char ca[3] = {'o','n','e'};
```

Array length is **not** stored

No bounds checking is performed

Array initialisation…

- Creating an initialised array:
  - You can specify initial values, in {}
  - E.g. 4 **short**s, with values 4,1,0,0

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

  - E.g. 5 **char**s, with values ‘o’, ‘n’, ‘e’, 0, 0

```
char chararray[5] = {'o','n','e'};
```

- If you specify an initialisation list with insufficient elements, the remaining elements will be zeroed

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**

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

Remember this 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!

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:

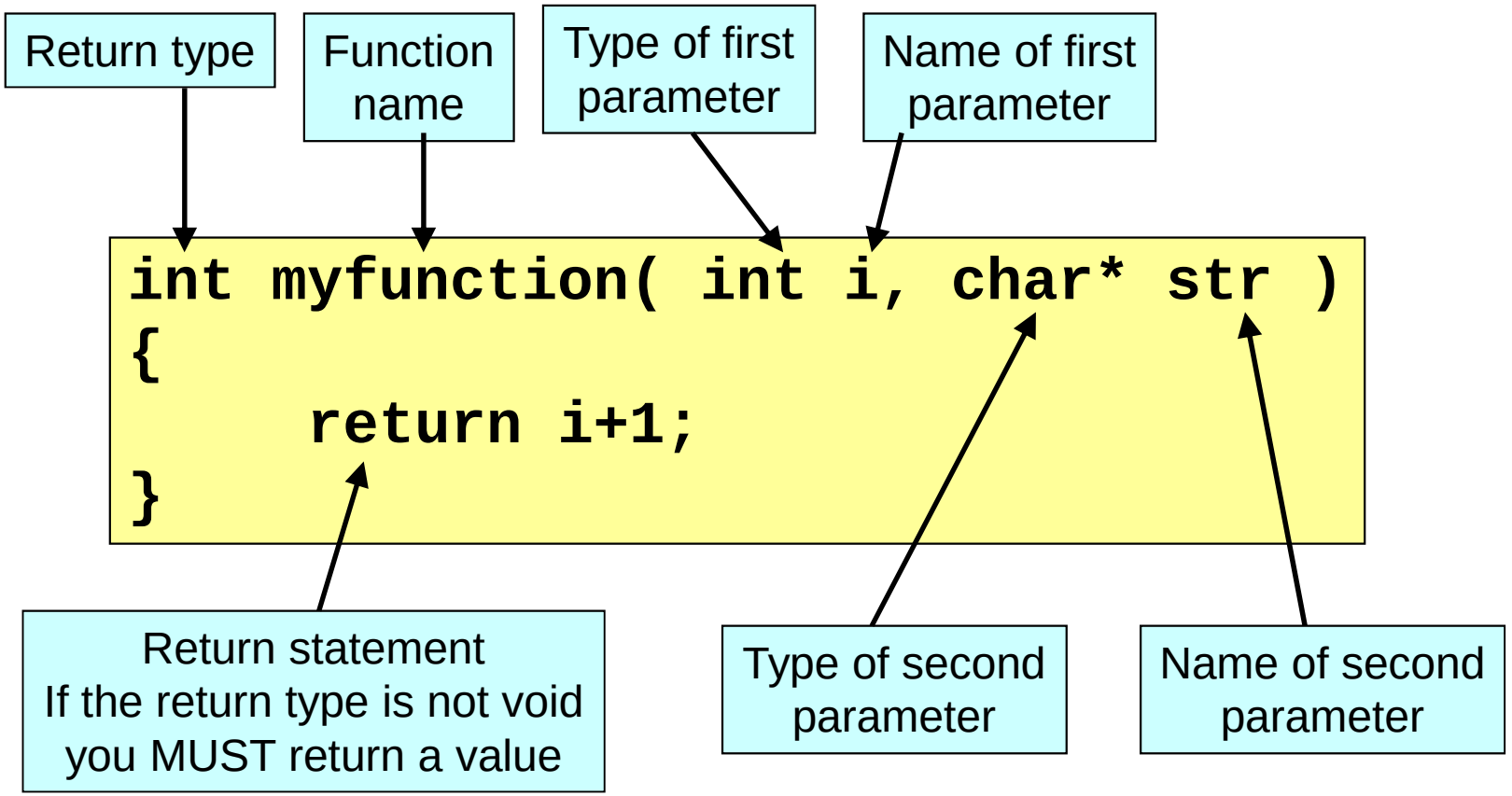
```
char c1 = 'h';
char* pc2 = &c1;
printf("%p ",(void*)pc2);
printf("%p\n",(void*)&c1);
```

- Output is in hexadecimal
- Example output:

```
0012FF73 0012FF73
```

Functions

Functions in C



- Functions in **C** are global, not class members
- You structure your code using files not classes

Identifying functions in C

- In C a function is **identified** by its **name**
  - The name must be unique
- In C++ (and Java) the types of parameters are also considered (function overloading)
- This example is NOT valid in C89/C90
  but is valid in C++, or Java

```
int multiply( int a, int b )
        { return a*b; }
long multiply( long a, long b)
        { return a*b; }
```

Declarations and definitions (1)

- Functions should be declared before they are called (so compiler can warn about errors)
- Definitions are also declarations
- One trick is to define functions in reverse order

order_functions.cpp

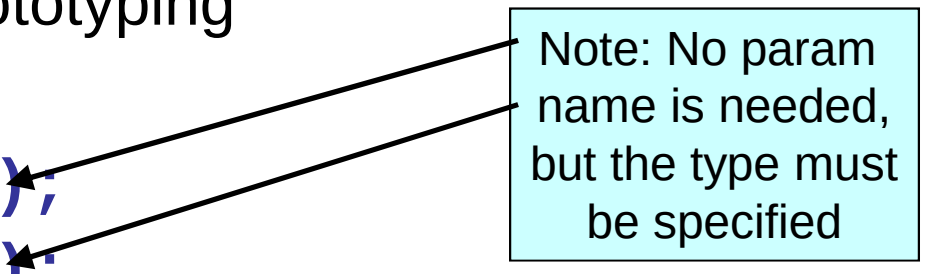
```
int myfunc2()
    { return 1; }
int myfunc1()
    { return myfunc2(); }
int main( int argc, char* argv[] )
    { return myfunc1(); }
```

Declarations and definitions (2)

- Otherwise, declare functions before usage
  - Called function prototyping
- e.g.:

```
int myfunc1(int);
int myfunc2(int);

int main( int argc, char* argv[] )
    { return myfunc1(argc); }
int myfunc1( int i1 )
    { return myfunc2(i1) + 1; }
int myfunc2( int i2 )
    { return 1 + i2; }
```



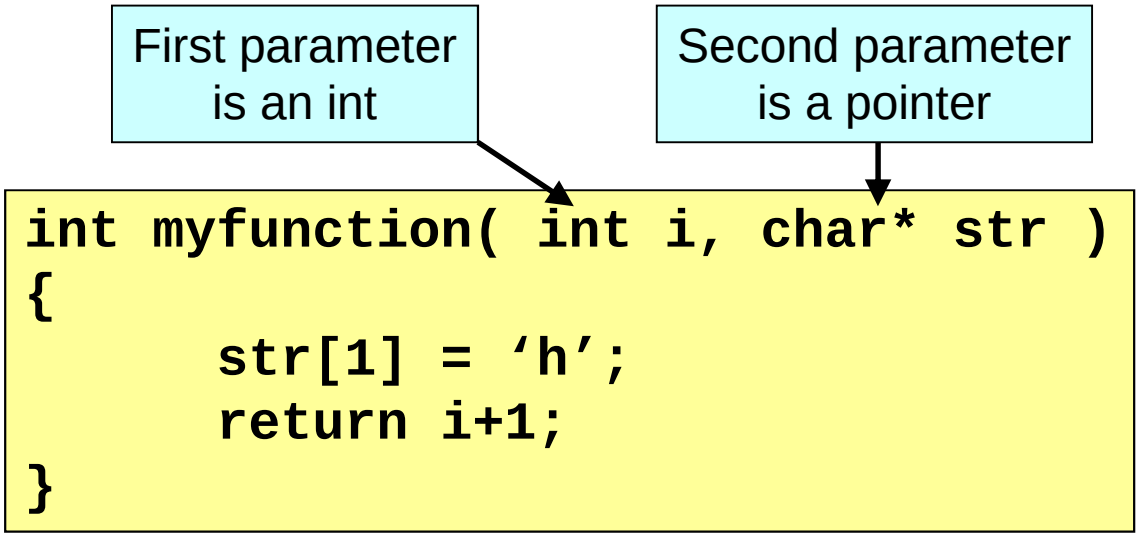
prototype.cpp

Function declarations

- **You must declare functions before use**
  - But definitions are also declarations
- Declarations usually go in header files
  - **`cstdio`** has many standard i/o function declarations
  - **`cstring`** has many string function declarations
  - You will *usually* have one header file per .c or .cpp file
    - Containing ***declarations*** of everything in the file that should be available from outside the file (i.e. functions & variables in C, also classes in C++)
- Function declarations specify only:
  - Function name
  - Return type
  - Type(s) of parameter(s)

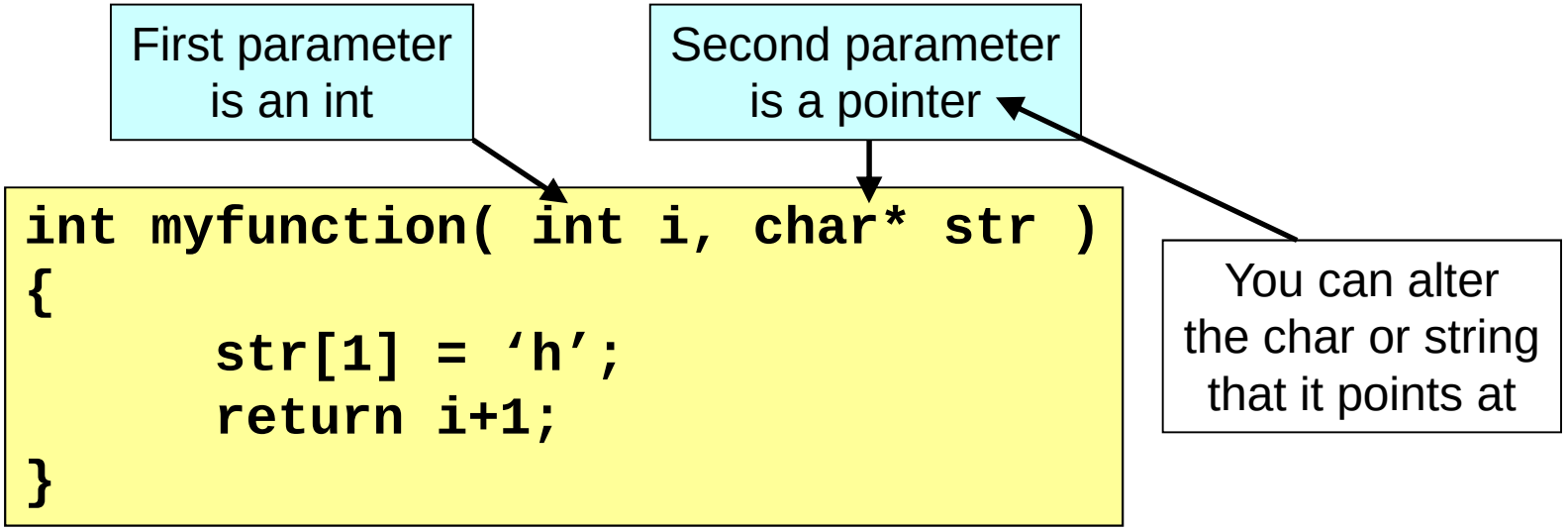
Passing pointers as parameters

Parameters can be pointers



- 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



- 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;
}
```

pass_by_ref.cpp

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

Pointer arithmetic

Pointer arithmetic, by example

- E.g.:

```
char ac[] = {'c','+','+','c',
             'h','a','r','\0'};
char* pc = ac;
printf( “%c\n”, *pc );
```

- **Q1: What is the output of the printf?**

Pointer arithmetic, by example

- E.g.:

```
char ac[] = {'c','+','+','c',
             'h','a','r','\0'};
char* pc = ac;
printf( "%c\n", *pc );
```

- We can increment pc:

```
pc++;
```

- **Q2: What do you *think* `pc++` does?**

Pointer arithmetic, by example

- E.g.:

```
char ac[] = {'c','+','+','c',
             'h','a','r','\0'};
char* pc = ac;
printf( “%c\n”, *pc );
```

- We can increment pc:

```
pc++;
```

- **Q3: What do you think this outputs?**

```
printf( “%c\n”, *pc );
```

Similarly, with **`shorts`**

- E.g.:

```
short as[] = { 1, 7, 9, 4 };
short* ps = as;
printf( “%d\n”, *ps );
```

- **Q1: What is the output of the printf?**

Similarly, with **`shorts`**

- E.g.:

```
short as[] = { 1, 7, 9, 4 };
short* ps = as;
printf( "%d\n", *ps );
```

- We can increment **`ps`**:

```
ps++;
```

- **Q2: What do you think `ps++` does?**

Similarly, with **`shorts`**

- E.g.:

```
short as[] = { 1, 7, 9, 4 };
short* ps = as;
printf( “%d\n”, *ps );
```

- We can increment ps:

```
ps++;
```

- **Q3: What do you think this outputs?**

```
printf( “%d\n”, *ps );
```

Pointer increment

- Incrementing a pointer increases the value of the address stored by an amount equal to the size of the thing the pointer **thinks that** it points at
- i.e. **the type of the pointer matters**
- This allows moving through an array using a pointer

```
char str[] = {…}
char* p = str;
p++; // p==1001
char c = *p; //'e'
```

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

Pointer decrement

- Decrementing a pointer decreases the value of the address stored by an amount equal to the size of the thing the pointer **thinks that** it points at
- **Be very careful about array bounds!**

```
short as[8] = {…};
short* p = as;
p--; // p==998
short s = *p;//??
```

Address	Value	Name
998	?	?
1000	234	as[0]
1002	839	as[1]
1004	1	as[2]
1006	743	as[3]
1008	938	as[4]
1010	2342	as[5]
1012	0	as[6]
1014	3425	as[7]
1016	1000	p

Pointer Arithmetic Summary

- Pointers store addresses
  - You can increment/decrement them (++,--)
    - Changing the address that is stored
  - You can also add to or subtract from the value of a pointer
  - They move in **multiples of the size** of the **type that they THINK they point at**
  - e.g.: If a **`short`** is 2 bytes, then incrementing a **`short*`** pointer will add 2 to the address
  - This is very useful for moving through arrays

Finally: subtracting pointers

- If you subtract one pointer from another (of the same type) then the result is the count of the **number of elements** between them +1
- Or the number of bytes, divided by the size of an element

```
short as[8] = {
   234,839,1,743,938,
   2342,0,3425 };
short* p1 = &(as[3]);
short* p2 = &(as[5]);
int i = p2 – p1;
```

Address	Value	Name
998	?	?
1000	234	as[0]
1002	839	as[1]
1004	1	as[2]
1006	743	as[3]
1008	938	as[4]
1010	2342	as[5]
1012	0	as[6]
1014	3425	as[7]
1016	1006	p1
1020	1010	p2

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.

Example 2: strlen() revisited

- **`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 mystrlen2( char* str )
{
   char* temp = str;
   while ( *temp )
      temp++;
   return temp-str;
}
```

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

When you subtract a pointer from another (of the same type), the result is the number of elements difference between them

Implementing strcpy

How we could implement strcpy

```
char src[] = {'C','',
     's','t','r',0};
char dest[7];
strcpy( dest, src );
```

```
char* mystrcpy(
   char* dest, char* src )
{
   char* p = dest;
   char* q = src;
   while ( *p++ = *q++ )
      ;
   return dest;
}
```

Address	Value	Name
1000	‘C’	src[0]
1001	‘ ’	src[1]
1002	‘s’	src[2]
1003	‘t ‘	src[3]
1004	‘r’	src[4]
1005	0	src[5]
6000	?	dest[0]
6001	?	dest[1]
6002	?	dest[2]
6003	?	dest[3]
6004	?	dest[4]
6005	?	dest[5]
6006	?	dest[6]

Note: *p++ is equivalent to *(p++) (post-increment has higher precedence)

Reminder: Operator Precedence

- Operators are evaluated in a specific order
  - Highest operator precedence applies first
- Examples (highest to lowest, not complete)

Increasing precedence

(), [], ++, --	Grouping, array access, post increment/decrement
++, --, *, &	Pre-increment, dereference, address of (right to left)
*, /, %	Multiplication, division, modulus
+ -	Addition, subtraction
<, <=, >, >=	Comparison
==, !=	Comparison: equal to, not equal to
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
? :	Ternary conditional
=, +=, -= etc	Assignment and '… and assign' (right to left)

Next lecture

- The Stack and Stack Frames
- The C / C++ Pre-processor
- Compiling and Linking Multiple Files