G52CPP
C++ Programming
Lecture 5

Dr Jason Atkin

Don’t panic

- At the start there is a lot to take in
- And it all fits together so it’s hard to follow without the other parts
- It should drop into place as we progress

- The assessments are at the end – to allow you time to assimilate the information
- **Don’t** worry yet
- **Do** practice and do the examples

Previous lectures

- `&` operator : address-of
- Addresses are stored in pointers
- Copying a pointer:
  - Copy points to the same thing
  - I.e. the address is copied
- Can pass a pointer into or return one from a function
- `*` 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[]`

This lecture

- Pointer arithmetic
  - strlen
  - strcpy

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 number of elements different that they are (1+number of elements between them)
- Number of bytes different, divided by size of 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
- Local, global and static variables
- Variable shadowing