

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○○	Causes of Error ○○○○○○○○○○○○○○○○	Projects ○○○○
--------------------	-------------------------------	----------------	---------------------	-------------------------------------	------------------

## C for Science

### Lecture 2 of 5

Sam Bott

<http://wwwf.imperial.ac.uk/~shb104/c>  
s.bott@imperial.ac.uk

15<sup>th</sup> February 2014

**Imperial College**  
London

Sam Bott
C for Science

C Shorthand ●○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○○	Causes of Error ○○○○○○○○○○○○○○○○	Projects ○○○○
--------------------	-------------------------------	----------------	---------------------	-------------------------------------	------------------

Last Week...

- C is a language for creating fast, portable programs.
- We use an IDE to write source, compile, link and debug our C programs.
- The basic structure of a C program has been demonstrated and used.
- There are two categories of number in C: integers and floating point numbers.
- We have seen how logic and statements can control the flow of a program.
- `printf` and `scanf` will write and read from the console respectively.

Sam Bott
C for Science

C Shorthand ●○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○○	Causes of Error ○○○○○○○○○○○○○○○○	Projects ○○○○
--------------------	-------------------------------	----------------	---------------------	-------------------------------------	------------------

C Shorthand

## Terse Code

There are shortcuts in the C language that allow for concise code.

- 1 Incrementing by 1: Pre-increment, and post-increment.  
`++i` Increment `i` by 1, then use it.  
`i++` Use `i`, then increment it by 1.
- 2 Increment by another variable.  
 Normal code: `sum = sum + v[i];`  
 Terse code: `sum += v[i];`

An example:

```
for (i=0; i < N; i++)
    sum += v[i];
```

Sam Bott
C for Science

C Shorthand ●○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○○	Causes of Error ○○○○○○○○○○○○○○○○	Projects ○○○○
--------------------	-------------------------------	----------------	---------------------	-------------------------------------	------------------

C Shorthand

## More Shorthand

```
--i;           decrement i by 1.
sum -= v[i];   means sum = sum - v[i];
sum *= v[i];   means sum = sum * v[i];
sum /= v[i];   means sum = sum / v[i];
sum %= 2;      means sum = sum % 2;
```

Other operators can also be abbreviated this way.

### Inline if - The Ternary Operator

The following code:

```
if (r1 > r2) { maxr = r1; }
else maxr = { r2; }
```

can be abbreviated:

```
maxr = (r1 > r2) ? r1 : r2;
```

Sam Bott
C for Science

C Shorthand ○○○	Functions ●○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○	Causes of Error ○○○○○○○○○○○○○○○	Projects ○○○
--------------------	-------------------------------	----------------	--------------------	------------------------------------	-----------------

Defining Functions

## Defining Functions

The C language only provides essential functionality, meaning a lot of functions need to be written yourself. Here are a few general rules for functions:

- Functions cannot define other functions within them.
- An optional single value can be returned.
- All arguments to functions are passed by value and remain unaffected by the function.
- Passing pointers to functions allows them to “return” multiple variables.

Sam Bott  
C for Science

C Shorthand ○○○	Functions ●○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○	Causes of Error ○○○○○○○○○○○○○○○	Projects ○○○
--------------------	-------------------------------	----------------	--------------------	------------------------------------	-----------------

Defining Functions

## Declarations vs Definitions

### Function Declarations

These tell the compiler about the *existence* of a function, which then allows us to call it. A declaration ends with a ;.

```
int quad_roots (double A, double B, double C,
               double * r1, double * r2);
```

### Function Definitions

The code making up the function is supplied to the compiler. A function can only be defined once. A definition contains braces { and }:

```
int quad_roots (double A, double B, double C,
               double * r1, double * r2)
{ ... }
```

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○●○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○	Causes of Error ○○○○○○○○○○○○○○○	Projects ○○○
--------------------	-------------------------------	----------------	--------------------	------------------------------------	-----------------

Using Functions

## An Example: Quadratic Equation Solver

As a worked example we write a function to solve the quadratic equation:

$$Ax^2 + Bx + C = 0 \quad A, B, C \in \mathbb{R}$$

Our quadratic solver will:

- Take the three doubles A, B and C as arguments.
- Solve the quadratic and return an `int` signifying to the caller the type of answer available:
  - 1 A = 0, we have a linear equation.
  - 0 There are two distinct real roots.
  - 1 We have a pair of complex conjugate roots.
  - 2 Both roots are real and identical.

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○●○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○	Causes of Error ○○○○○○○○○○○○○○○	Projects ○○○
--------------------	-------------------------------	----------------	--------------------	------------------------------------	-----------------

Using Functions

## The Code

One possible function prototype is:

```
int quad_roots (double A, double B, double C,
               double * r1, double * r2);
```

- The variables A, B and C are unchanged by `quad_roots`.
- We need to return two doubles (the roots of the equation), thus we take in pointers `double *r1` and `double *r2`.
- C90 does not allow for complex number types (C99 does support them), so we have to think a little bit about the complex number case.

Sam Bott  
C for Science

C Shorthand  
000

**Functions**  
0000●0000000000

Arrays  
0000

Debugging  
000000

Causes of Error  
0000000000000000

Projects  
0000

Using Functions

Code Snippet for Calling quad\_roots

```

...
int main()
{
    double A, B, C, root1, root2;
    int quad_case;
    ...
    quad_case = quad_roots(A, B, C, &root1,
                          &root2);

    switch(quad_case)
    {
        case -1: linear equation
    }
}

```

Sam Bott

C for Science

C Shorthand  
000

**Functions**  
0000●0000000000

Arrays  
0000

Debugging  
000000

Causes of Error  
0000000000000000

Projects  
0000

Using Functions

Code Snippet for quad\_roots

```

int quad_roots(double A, double B, double C,
              double * r1, double *r2)
{
    double d;

    /* linear case */
    if (A == 0.0)
    {
        *r1 = -C/B;
        return -1;
    }

    /* compute the discriminant */
    d = B*B-4.0*A*C;
}

```

Sam Bott

C for Science

C Shorthand  
000

**Functions**  
00000●0000000000

Arrays  
0000

Debugging  
000000

Causes of Error  
0000000000000000

Projects  
0000

Structuring Functions

The Stack

Let's consider this example function.

```

int hasRealRoots(double A,
                double B, double C)
{
    double d = B*B-4.0*A*C;
    if (d < 0) return 0;
    return 1;
}

```

- We need space to hold a copy of A, B and C.
- We need space to store our computed d.
- When we've finished, we need to get back to the calling function.

This is achieved by using a *stack*.

Sam Bott

C for Science

C Shorthand  
000

**Functions**  
00000●0000000000

Arrays  
0000

Debugging  
000000

Causes of Error  
0000000000000000

Projects  
0000

Structuring Functions

The Stack - Rough Sketch (Stack Frames)

The diagram illustrates the stack structure during function calls. It consists of a vertical stack of frames:

- Top frame (yellow):** Empty space above the current function's frame.
- hasRealRoots frame (green):** Contains 'Locals of hasRealRoots', 'Arguments of hasRealRoots', and 'Return to quad\_sol'.
- quad\_sol frame (blue):** Contains 'Locals of quad\_sol', 'Arguments of quad\_sol', and 'Return to main'.
- main frame (grey):** Contains 'Locals of main' and 'Arguments of main'.

- Consider the case where we have main, which calls `quad_sol`, which in turn calls `hasRealRoots`.
- We add and remove items from the stack as the program executes.
- Adding/removing items from the stack takes very little time.
- The stack is fixed in size, if we go over the top ("smash the stack"), our program crashes.

Sam Bott

C for Science

C Shorthand  
ooo
Functions  
oooooooo●ooooo
Arrays  
ooo
Debugging  
ooooo
Causes of Error  
oooooooooooooooo
Projects  
ooo

Structuring Functions

## Recursive Functions

As C uses a stack by default when calling functions, we are able to write functions that call themselves. These are called *recursive functions*.

### An Example: Computing the Factorial

$$n! = \prod_{i=1}^n i, \quad 0! = 1, \quad n \in \mathbb{N}.$$

Lends itself to be coded up as a recursive function.

### A Tougher Example: Fibonacci Numbers

$$F_n = F_{n-1} + F_{n-2}, \quad F_0 = F_1 = 1.$$

A naïve implementation of this will kill the stack, and take a very long time to execute.

Sam Bott  
C for Science

C Shorthand  
ooo
Functions  
oooooooo●ooooo
Arrays  
ooo
Debugging  
ooooo
Causes of Error  
oooooooooooooooo
Projects  
ooo

Structuring Functions

## Computing the Factorial

```

1  #include <stdio.h>
2
3  int NFact(int N)
4  {
5      if (N>1) return N*NFact(N-1);
6      return 1;
7  }
8
9  int main()
10 {
11     int n;
12     printf("Enter n:");
13     scanf("%d", &n);
14     printf("%d! = %d\n", n, NFact(n));
15     return 0;
16 }
```

Sam Bott  
C for Science

C Shorthand  
ooo
Functions  
oooooooo●ooooo
Arrays  
ooo
Debugging  
ooooo
Causes of Error  
oooooooooooooooo
Projects  
ooo

Structuring Functions

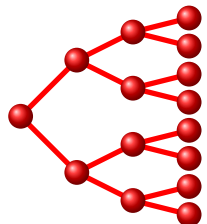

## Computing Fibonacci Numbers

```

int BadFib(int N)
{
    if (N < 2) return 1;
    return (BadFib(N-1) +
            BadFib(N-2));
}

int utilf(int a, int b, int n)
{
    if(n < 1) return b;
    return utilf(b, a+b, n-1);
}

int GoodFib(int n)
{
    return utilf(0, 1, n);
}
```

Sam Bott  
C for Science

C Shorthand  
ooo
Functions  
oooooooo●ooooo
Arrays  
ooo
Debugging  
ooooo
Causes of Error  
oooooooooooooooo
Projects  
ooo

Code Structure

## Imperative versus Functional Programming

Two programming techniques are popular in C:

### Imperative

- Very long functions.
- Lots of global variables.
- Very few function calls.

### Functional

- Lots of small functions.
- Each function has a clearly defined rôle.
- Global variables avoided as much as possible.

I would encourage leaning towards the latter, a good program will contain traits from both styles.

Sam Bott  
C for Science



C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○○○	Arrays ○○●○○	Debugging ○○○○○○○	Causes of Error ○○○○○○○○○○○○○○○○○○	Projects ○○○○
--------------------	---------------------------------	-----------------	----------------------	---------------------------------------	------------------

Data Arrays

## Accessing Array Elements

**Arrays in C are indexed from 0!**

```
#include <stdio.h>

int main()
{
    int primes[6] = {2, 3, 5, 7, 11, 13};

    printf("first prime = %d\n", primes[0]);
    printf("next prime = %d\n", primes[1]);

    return 0;
}
```

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○○○	Arrays ○○●○○	Debugging ○○○○○○○	Causes of Error ○○○○○○○○○○○○○○○○○○	Projects ○○○○
--------------------	---------------------------------	-----------------	----------------------	---------------------------------------	------------------

A Little More on Pointers

## A little more on pointers

**Reminder**

- Declared using: `type * ptrVar;`
- Variable to pointer (pointer *referencing*): `ptrA = &A;`
- Pointer to variable (pointer *de-referencing*):  
`*ptrA = newVar;`

**In addition**

Pointers are memory addresses, and as such allow arithmetic!

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○○○	Arrays ○○○●○	Debugging ○○○○○○○	Causes of Error ○○○○○○○○○○○○○○○○○○	Projects ○○○○
--------------------	---------------------------------	-----------------	----------------------	---------------------------------------	------------------

A Little More on Pointers

## Pointer Arithmetic

- Different data types in C are different sizes.
- Pointers are usually declared with a type (i.e. `int *`, `float *`, `double *`).

**Relation to Arrays**

Given the array `myArray` and an integer `index`, the following is true:

```
myArray[index] = *(myArray + index)
```

- And this is the reason array indices start at 0...

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○○○	Arrays ○○○○○	Debugging ●○○○○○	Causes of Error ○○○○○○○○○○○○○○○○○○	Projects ○○○○
--------------------	---------------------------------	-----------------	---------------------	---------------------------------------	------------------

Debugging Techniques

## Debugging

- Once you've designed, typed up and successfully compiled your program, the difficult part begins: debugging!
- Problems in the code are usually either easy to locate or stubbornly elusive.

**Easier Problems**

- Program crash/fault at the same point every run.

**Nastier Problems**

- Numerical output differs to what is expected.
- Program crashes seemingly randomly.

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ●○○○○	Causes of Error ○○○○○○○○○○○○○○○○	Projects ○○○
--------------------	-------------------------------	----------------	--------------------	-------------------------------------	-----------------

Debugging Techniques

## Debugging Techniques

In increasing order of difficulty:

### Create Verbose Output

- A few strategically placed `printf` statements can prove to be helpful, but they have to be read (by a human):
  - too few and you miss the problem,
  - too many and they rapidly become useless.
- Straightforward to implement (and to comment out).

### Code Defensively

- Consider specialised test cases.
- Write code to test intermediate results.
- Use the `assert` macro.

### Use a Debugger

For those non-trivial problems.

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○●○○	Causes of Error ○○○○○○○○○○○○○○○○	Projects ○○○
--------------------	-------------------------------	----------------	--------------------	-------------------------------------	-----------------

Assertion Checking

## Assertion Checking

In the header `<assert.h>`, the macro `assert` is defined. It has the following syntax:

```
assert(logical_expression);
```

If *logical\_expression* evaluates to false (zero) then:

- Program execution stops immediately.
- An error message is sent to *stderr* (the console) stating the line number where the assertion failed.

```
assert( a != 0); /* a should never be zero */
```

### Switching it off

Placing `#define NDEBUG` before all the `#include <assert.h>` statements de-activates assertion checking.

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○●○○	Causes of Error ○○○○○○○○○○○○○○○○	Projects ○○○
--------------------	-------------------------------	----------------	--------------------	-------------------------------------	-----------------

Assertion Checking

## Assertion Checking - An Example

```

1 #include <stdio.h>
2 #include <assert.h>
3 /* the sqrt function is much better than this... */
4 double squareRoot(double N)
5 {
6     double x = 1.0;
7     int loop;
8     /* negative numbers are not allowed! */
9     assert(N >= 0.0);
10    if (N == 0.0) return 0.0;
11    for (loop = 0; loop < 10; loop++)
12        x = (x*x+N) / (2.0*x);
13    return x;
14 }
15
16 int main()
17 {
18     double square;
19     printf("Enter a non-negative number:");
20     scanf("%lg", &square);
21     /* SHOULD HAVE: if (x < 0.0) ... */
22     printf("Square root = %g\n", squareRoot(square));
23     return 0;
24 }
```

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○●	Causes of Error ○○○○○○○○○○○○○○○○	Projects ○○○
--------------------	-------------------------------	----------------	--------------------	-------------------------------------	-----------------

Compile-time Logic

## The C Preprocessor - Conditional Compilation

We have already seen the `#include` and `#define` statements. Conditional statements are also possible:

```

1 #include <stdio.h>
2
3 int main(int argc, char ** argv)
4 {
5     #ifdef NDEBUG
6         printf("Assertions DISABLED\n");
7         printf("%d arguments\n", argc);
8     #else
9         printf("Assertions ENABLED\n");
10    #endif
11    return 0;
12 }
```

This logic is performed at *compile time*.

Sam Bott  
C for Science

C Shorthand  
000

Functions  
0000000000000000

Arrays  
0000

Debugging  
00000●

Causes of Error  
0000000000000000

Projects  
0000

Using a Debugger

## Using a Debugger

- Microsoft's Visual Studio contains a brilliant interactive debugger.
- GNU debugger (`gdb`) is also very good, and is essential for certain scenarios.
- Programs need to be compiled with debug information.
- Running a program straight through a debugger will show you the line of code that crashed it (if it crashes).

### Interactive Analysis of Running Code

- Program execution can be paused at *breakpoints*.
- Functions can be *stepped into*, *stepped over*, or *stepped out from*.
- Variables/arrays can be *watched*.

Interactive debugging is very tricky at first, but soon becomes invaluable for isolating subtle problems.

Sam Bott

C for Science

C Shorthand  
000

Functions  
0000000000000000

Arrays  
0000

Debugging  
000000

Causes of Error  
●0000000000000000

Projects  
0000

Maths

## Common Problems

- 1 Misuse of Power!  
 $x^2$  should always be written `x*x`, *NOT* `pow(x, 2.0)`.  
 $\sqrt{x}$  should be written `sqrt(x)`, *NOT* `pow(x, 0.5)`.
- 2 Integer Division  
Remember a fraction such as  $1/3$  is equal to zero. To get a floating point fraction, this should be rewritten `1.0/3.0`.

Sam Bott

C for Science

C Shorthand  
000

Functions  
0000000000000000

Arrays  
0000

Debugging  
000000

Causes of Error  
●0000000000000000


Projects  
0000

Floating Point Numbers

## Floating Point Numbers (IEEE 754 Standard)

(from the previous lecture)

On my machine, a `float` (single precision) looks like:



It consists of three parts, the *sign bit*( $b$ ), the *biased exponent*( $e$ ) and the *fraction*( $f$ ). We break down a number  $x$ :

$$x^{\text{float}} = (-1)^b \times 2^{e-127} \times (1 + f \times 2^{-23}), \quad \begin{matrix} 0 < e < 255 \\ 0 \leq f \leq 2^{23} - 1 \end{matrix}$$

We have three special numbers,  $-\text{Inf}$  ( $-\infty$ ),  $\text{Inf}$  ( $\infty$ ) and  $\text{NaN}$  (Not a Number).

For `double` (double precision) we have:

$$x^{\text{double}} = (-1)^b \times 2^{e-1023} \times (1 + f \times 2^{-52}), \quad \begin{matrix} 0 < e < 2047 \\ 0 \leq f \leq 2^{52} - 1 \end{matrix}$$

Sam Bott

C for Science

C Shorthand  
000

Functions  
0000000000000000

Arrays  
0000

Debugging  
000000

Causes of Error  
●0000000000000000

Projects  
0000

Floating Point Numbers

## Floating Point Number Analysis

In `<float.h>`, there are some useful quantities:

Quantity	Float	Double
Maximum Value	<code>FLT_MAX</code>	<code>DBL_MAX</code>
Minimum Value	<code>FLT_MIN</code>	<code>DBL_MIN</code>
Max Decimal Exponent	<code>FLT_MAX_10_EXP</code>	<code>DBL_MAX_10_EXP</code>
Min Decimal Exponent	<code>FLT_MIN_10_EXP</code>	<code>DBL_MIN_10_EXP</code>
$\epsilon$	<code>FLT_EPSILON</code>	<code>DBL_EPSILON</code>

### Floating point $\epsilon$

$\epsilon$  is the smallest (in magnitude) number such that:

$$1.0 + \epsilon \neq 1.0$$

Sam Bott

C for Science



C Shorthand  
○○○

Functions  
○○○○○○○○○○○○○○○○○○○○

Arrays  
○○○○

Debugging  
○○○○○○○

Causes of Error  
○○○○●○○○○○○○○○○○○

Projects  
○○○○

Floating Point Numbers

## Floating Point Accuracy

- Some numbers can be represented in floating point exactly: e.g.  $2^i$ , any integers that fit in the significand (mantissa).
- Most numbers need to be approximated, e.g.  $\sqrt{2}$ ,  $\pi$ .
- One overlooked example is  $0.1$ !
- It is possible (though rare) to get exact answers from floating point arithmetic
- Relative errors of  $\approx 10^{-15}$  for `double` and  $\approx 10^{-6}$  for `float` are considered to be very good.
- Multiplication and division generally preserve relative error (but can take us outside the floating point range).

Sam Bott

C for Science

C Shorthand  
○○○

Functions  
○○○○○○○○○○○○○○○○○○○○

Arrays  
○○○○

Debugging  
○○○○○○○

Causes of Error  
○○○○●○○○○○○○○○○○○

Projects  
○○○○

Floating Point Numbers

## The Largest Source of Floating Point Errors

Addition and subtraction are the largest contributors to floating point error.

### The Golden Rule

**Do not subtract two very similar floating point numbers!**

(This leads to “*catastrophic cancellation*”).

Sam Bott

C for Science

C Shorthand  
○○○

Functions  
○○○○○○○○○○○○○○○○○○○○

Arrays  
○○○○

Debugging  
○○○○○○○

Causes of Error  
○○○○●○○○○○○○○○○○○

Projects  
○○○○

Casting

## Casting

Casting can either be *implicit* or *explicit*.

### Implicit Casting

Conversion where there is no ambiguity (i.e. to a “bigger” data type) can be done automatically:

```
double x = 5; /* conversion from int to double */
double fEps = FLT_EPSILON; /* float to double */
```

### Explicit Casting

If we wish to force a type conversion we place the destination type in brackets before the source variable:

```
oldtype oldData = ...
newtype newData = (newtype) oldData;
```

Explicit casting should be avoided if possible.

Sam Bott

C for Science

C Shorthand  
○○○

Functions  
○○○○○○○○○○○○○○○○○○○○

Arrays  
○○○○

Debugging  
○○○○○○○

Causes of Error  
○○○○●○○○○○○○○○○○○

Projects  
○○○○

Precedence

## Operator Precedence and Associativity

From K&R2:

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * & (type) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
?:	right to left
= += -= *= /= %= &= ^=  = <<= >>=	right to left
,	left to right

Sam Bott

C for Science

C Shorthand  
000
Functions  
0000000000000000
Arrays  
0000
Debugging  
000000
Causes of Error  
0000000●000000
Projects  
0000

Precedence

## Operator Precedence and Associativity - Examples

`a - b * c / d`

- 1 `*` and `/` are carried out before `-` – due to precedence.
- 2 `*` is carried out before `/` due to (left to right) associativity.

`if (x & MASK == 0)`

- `==` has a higher precedence than `&` so is executed first!
- To get what we originally intended, parentheses are needed:

```
if ((x & MASK) == 0)
```

If in doubt

Put brackets around things...

Sam Bott

C for Science

C Shorthand  
000
Functions  
0000000000000000
Arrays  
0000
Debugging  
000000
Causes of Error  
0000000●000000
Projects  
0000

Keywords

## C Keywords

The following keywords are recognised by all C compilers as special commands. These words should not be used for variable names, function names etc.

<code>auto</code>	<code>break</code>	<code>case</code>	<code>char</code>
<code>const</code>	<code>continue</code>	<code>default</code>	<code>do</code>
<code>double</code>	<code>else</code>	<code>enum</code>	<code>extern</code>
<code>float</code>	<code>for</code>	<code>goto</code>	<code>if</code>
<code>int</code>	<code>long</code>	<code>register</code>	<code>return</code>
<code>short</code>	<code>signed</code>	<code>sizeof</code>	<code>static</code>
<code>struct</code>	<code>switch</code>	<code>typedef</code>	<code>union</code>
<code>unsigned</code>	<code>void</code>	<code>volatile</code>	<code>while</code>

Sam Bott

C for Science

C Shorthand  
000
Functions  
0000000000000000
Arrays  
0000
Debugging  
000000
Causes of Error  
0000000●000000
Projects  
0000

Keywords

## Preprocessor Keywords

- We also have the following preprocessor keywords:

<code>#include</code>	<code>#define</code>	<code>#undef</code>
<code>#if</code>	<code>#ifdef</code>	<code>#ifndef</code>
<code>#elif</code>	<code>#else</code>	<code>#endif</code>
<code>#error</code>	<code>#line</code>	<code>#pragma</code>

Sam Bott

C for Science

C Shorthand  
000
Functions  
0000000000000000
Arrays  
0000
Debugging  
000000
Causes of Error  
0000000●000000
Projects  
0000

Scope

## Scope: The Accessibility of Variables

Every variable in C has, associated with it, a *scope*. This defines how the variable can be accessed by functions in C. Some of the scoping rules are:

- All variables declared in the normal way inside a function are *local* to that function.
- Local variables can only be changed within the function they are defined, *unless*:
  - A pointer to a local variable may be passed to a function, extending the scope of that variable.
  - They are declared to be `extern` (more on this later).

Sam Bott

C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○	Causes of Error ○○○○○○○○○○●○○	Projects ○○○
--------------------	-------------------------------	----------------	--------------------	----------------------------------	-----------------

Scope

## Scope: Example 1

```

1  #include <stdio.h>
2
3  void F1()
4  {
5      int i = 4;
6      printf("In F1(): I = %d\n", i);
7  }
8
9  int main()
10 {
11     int i = 2;
12     printf("In main(): I = %d\n", i);
13     F1();
14     printf("In main() again: I = %d\n", i);
15     return 0;
16 }

```

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○	Causes of Error ○○○○○○○○○○●○○	Projects ○○○
--------------------	-------------------------------	----------------	--------------------	----------------------------------	-----------------

Scope

## Scope: Example 2

```

1  #include <stdio.h>
2
3  void F1(int i)
4  {
5      printf("In F1(): I = %d\n", i);
6      i = 3;  /* what does this do? */
7  }
8
9  int main()
10 {
11     int i = 2;
12     printf("In main(): I = %d\n", i);
13     F1(i);
14     printf("In main() again: I = %d\n", i);
15     return 0;
16 }

```

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○	Causes of Error ○○○○○○○○○○●○○	Projects ○○○
--------------------	-------------------------------	----------------	--------------------	----------------------------------	-----------------

Scope

## Scope: Example 3

```

1  #include <stdio.h>
2
3  void F1(int * i)
4  {
5      printf("In F1(): I = %d\n", *i);
6      *i = 3;  /* what does this do? */
7  }
8
9  int main()
10 {
11     int i = 2;
12     printf("In main(): I = %d\n", i);
13     F1(&i);
14     printf("In main() again: I = %d\n", i);
15     return 0;
16 }

```

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○	Causes of Error ○○○○○○○○○○○○○○	Projects ●○○○
--------------------	-------------------------------	----------------	--------------------	-----------------------------------	------------------

Compiling from Multiple Files

## Projects and Makefiles

- It is possible (and encouraged) to build a program from multiple `.c` files.
- This maximises the portability of the code, and
- Speeds up compiling - if we only change one `.c` file we only need to recompile one file...
- Visual Studio manages programs in to so-called *projects*, and everything is done graphically.
- If using `gcc` at the command line there is a program called `make` which manages projects. Information for building programs is stored in a Makefile.

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○○	Causes of Error ○○○○○○○○○○○○○○○○○○	Projects ○○●○
--------------------	---------------------------------	----------------	---------------------	---------------------------------------	------------------

Compiling from Multiple Files

## A sample Makefile

```

CFLAGS = -O2 -DNDEBUG -Wall -ansi
LFLAGS = -lm
CC = gcc
CLEANFILES = fst.o matrixfunctions.o fst fst.exe

fouriersinetrans: fst.c matrixfunctions.c
    $(CC) fst.c matrixfunctions.c $(LFLAGS) -o fst

clean:
    touch $(CLEANFILES)
    rm $(CLEANFILES)

```

- This compiles `fst.c` and `matrixfunctions.c`.
- It then links them to produce `fst.exe` (MinGW) or `fst` (\*NIX).
- It has two rules `fouriersinetrans` (default) to build the program and `clean` to clean up all the compiled output.

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○○	Causes of Error ○○○○○○○○○○○○○○○○○○	Projects ○○●○
--------------------	---------------------------------	----------------	---------------------	---------------------------------------	------------------

Compiling from Multiple Files

## The C Preprocessor - How to #define Externally

We are not restricted to `#define` statements in source code.

### Visual Studio

In the Visual Studio “project properties” → “C/C++” → “Preprocessor” option we can specify preprocessor definitions.

### gcc

In gcc we can specify define statements in the command line as follows:

```
gcc myfile.c -DNDEBUG -o myprogram
```

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○○	Causes of Error ○○○○○○○○○○○○○○○○○○	Projects ○○●○
--------------------	---------------------------------	----------------	---------------------	---------------------------------------	------------------

Summary

## Summary

- Shorthand exists to allow us to create more concise code.
- Functions are used to structure, tidy and allow us to reuse code.
- Thought must be given to the stack when calling functions recursively.
- Arrays are data blocks of the same type.
- Debugging is the process of fixing code that is not giving the correct results.
- Variables can only be used within their 'scope' (shown with `{...}`).
- Multiple `.c` files can be used to create one program.

Sam Bott  
C for Science

C Shorthand ○○○	Functions ○○○○○○○○○○○○○○○○○○	Arrays ○○○○	Debugging ○○○○○○	Causes of Error ○○○○○○○○○○○○○○○○○○	Projects ○○●○
--------------------	---------------------------------	----------------	---------------------	---------------------------------------	------------------