

C for Science

Lecture 3 of 5

Sam Bott

<http://www2.imperial.ac.uk/~shb104/c>
s.bott@imperial.ac.uk

17th April 2013

Imperial College
London

Last Week...

- Terse Code is used for concise code.
- Functions structure our code fragments into reusable routines.
- Arrays hold multiple values of one type.
- Debugging code allows us to find problems in our code or input.
- Scope - variable lifetime - has been introduced.
- Projects allow us to make a program from multiple `.c` files.

Bytes (or `char`s)

- We've mentioned `char`s briefly so far, and used them extensively whilst drawing little attention to them.
- `char` is the smallest data type in C, `sizeof(char) = 1` byte (this is explicitly stated in the standard).
- We can perform integer computations using `char` and `unsigned char`.
- The most common use for `char` is as part of a string.
- We can assign single letters as follows:

```
char letter = 'a';
```

where we use single quotes.

- An array of `char`s is specified using double quotes:

```
char * name = "Sam B";
```

ASCII Table

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	\0	SOH	STX	ETX	EOT	ENQ	ACK	\a	\b	\t	\n	VT	\f	\r	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	' '	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

- Characters 0x00 to 0x1f (31) are *non-printable*.
- Characters 0x80 (128) to 0xff (255) are *extended characters*.

Demo of char

```
1 #include <stdio.h>
2
3 int main()
4 {
5     char * name = "Grace";
6     printf(name); /* not recommended, but allowed*/
7     printf("\nname    = %s\n", name);
8     printf("name[0] = %c = %d\n", name[0], name[0]);
9     return 0;
10 }
```

Gives the following output:

Grace

name = Grace

name[0] = G = 71

Some useful char Functions

For chars

<code>isalpha(c)</code>	True (non-zero) if <code>c</code> is from A-Z,a-z
<code>isdigit(c)</code>	True if <code>c</code> if from 0-9
<code>isalnum(c)</code>	<code>=(isalpha(c) isdigit(c))</code>
<code>islower(c)</code>	True if <code>c</code> is from a-z
<code>isupper(c)</code>	True if <code>c</code> is from A-Z
<code>d=tolower(c)</code>	Convert to lowercase (if <code>isupper(c)</code>), otherwise it returns <code>c</code>
<code>d=toupper(c)</code>	Convert to uppercase (if <code>islower(c)</code>), otherwise it returns <code>c</code>

Layout of a String in Memory

Given: `char * string = "A string in C!";` In memory this looks like:

%c	A		s	t	r	i	n	g		i	n		C	!	\0
%d	65	32	115	116	114	105	110	103	32	105	110	32	67	33	0

- All strings in C are terminated by 0 (or '`\0`').
- `char` values of 0-127 correspond to *ASCII* codes, their use should be relatively consistent between different compilers.
- All other values correspond to *extended ASCII* codes, the representations of which vary considerably between compilers.

Some useful String Functions

```
strlen(s)
```

Returns the number of characters pointed to by `s`, the trailing `NULL` (`'\0'`) is excluded!

```
strncpy(dest, source, length)
```

Copies a maximum of `length` characters from `source` to `dest`.

```
int strcmp(s1, s2, length)
```

Compares a maximum of `length` characters of `s1` and `s2`. Note that `strcmp` returns 0 (usually false) for equality!

A String Demo

```

1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4  #include <ctype.h>
5
6  int main()
7  {
8      unsigned int loop;
9      char * string = "A string in C!", * copy;
10     printf("strlen(string) = %d\n", strlen(string));
11     copy = (char *) calloc(strlen(string)+1, 1);
12     if (!copy)
13     {
14         fprintf(stderr, "Couldn't allocate buffer!\n");
15         return -1;
16     }
17
18     strncpy(copy, string, strlen(string));
19     printf("strncpy(string, copy) = %d\n",
20           strcmp(string, copy, strlen(string)));
21
22     for (loop = 0; loop < strlen(copy); loop++)
23         copy[loop] = toupper(copy[loop]);
24
25     printf("modified copy = \"%s\"\n", copy);
26     printf("strcmp(string, copy) = %d\n",
27           strcmp(string, copy, strlen(string)));
28
29     free(copy);
30     return 0;
31 }

```

Results from String Demo

The program on the previous slide gives the following output:

```
strlen(string) = 14
strncmp(string, copy) = 0
modified copy = "A STRING IN C!"
strncmp(string, copy) = 32
```

- Note that `strncmp` returns 0 for *equality* and non-zero otherwise.
- Case insensitive string comparisons can be made using:
`strnicmp`.

Input and Output in C - Streams

- All I/O in C is accomplished via file *streams*.
- This stems from C's UNIX roots (every device is a file).
- We have already seen `printf`, formatted output to console.
- C90 defines three streams which are initialised by default when a program starts.

<code>stdin</code>	Input from the keyboard. (read only)
<code>stdout</code>	Text console. (write only)
<code>stderr</code>	Another text console. (write only)

Why have `stderr`?

Stream Redirection

- Your console allows for a C program's streams to be *redirected* at the command line.
- This is completely transparent to the C program (it just reads and writes data oblivious to the source).
- One can redirect `stdout` using (`>`) as follows:

```
myprogram > output.txt
```

- `stderr` can be redirected using (`2>`):

```
myprogram 2> errorlog.txt
```

- `stdin` can be redirected using (`<`):

```
myprogram < input.txt
```

Having a separate *error stream*, allows for important messages to be filtered from data output.

printf and scanf - formatted output and input

These are automatically connected to `stdout` and `stdin` respectively. The more generalised functions are `fprintf` and `fscanf`. As an example:

```
fprintf(stdout, "Text to stdout...\n");  
fprintf(stderr, "Text to stderr...\n");
```

We can also read text from a stream:

```
fscanf(stdin, "%lf", &ptrDouble);
```

- Additional streams can be opened using `fopen`.

fopen - open a file stream

```
FILE * stream = fopen(char * filename, char * mode);
```

Working backwards, `mode` is a string telling C how to open the file:

mode	meaning
"r"	Open <code>filename</code> for reading. The file must exist or <code>NULL</code> is returned.
"w"	Open <code>filename</code> for writing, starting from the beginning of the file. The file will be created if it doesn't already exist. Any old data will be overwritten.
"a"	Open <code>filename</code> for writing, starting from the end of the file. File will be <i>appended</i> if it does exist and created otherwise.
"r+"	Open <code>filename</code> for reading and writing, starting from the beginning. If the file doesn't exist, <code>NULL</code> is returned.
"w+"	Open <code>filename</code> for reading and writing, starting from the beginning. If the file doesn't exist it's created.
"a+"	Open <code>filename</code> for reading and writing (append if exists).

fopen - II

```
FILE * stream = fopen(char * filename, char * mode);
```

filename

Must be a legal operating system file name. A safe option would be something similar to:

```
"results.txt"
```

Absolute Filenames

We can specify a full directory path to a filename:

```
data = fopen("C:\\TEMP\\mydata.dat", "w"); /* Windows */
data = fopen("/tmp/mydata.dat", "w");      /* UNIX */
```

Relative Filenames

It is much safer to omit the full path:

```
data = fopen("mydata.dat", "w"); /* works on most os' */
```

fopen - III

```
FILE * stream = fopen(char * filename, char * mode);
```

- Here `stream` is a pointer to a `FILE` data type.
- `stdin`, `stdout` and `stderr` are also pointers (which are opened automatically for us).
- Sometimes it is not possible to open a file (it may not exist, or belong to another user etc...), in this case a special pointer value `NULL` (which is equal to zero) is returned.

NULL pointers

A `NULL` pointer is a special value, usually signalling failure. This can be checked for:

```
stream = fopen("illegal/\\filename.txt", "w");
if (!stream)
{
    fprintf(stderr, "Unable to open file\n");
    ...
}
```


fclose - Closing a file stream

- Any file streams `fopen`'ed should be closed using:

```
fclose(stream);
```

- File read/writes may go to an internal buffer before they go to the disk (to speed things up).
- An explicit file close ensures that any pending data is written to disk.
- For the lazy there is:

```
fcloseall();
```

- Do *NOT* call `fclose` on any of the following:

```
stdin, stdout & stderr
```

Moving around a file - `rewind`, `ftell` and `fseek`

```
rewind(stream);
```

Start the next read/write from the beginning of the file.

```
fpos = ftell(stream);
```

Get an `int` telling us how far we are in the file (from the beginning).

```
fseek(stream, fpos, SEEK_SET);
```

Move `fpos` bytes into the file (counted from the beginning).

```
fseek(stream, fpos, SEEK_CUR);
```

Move `fpos` bytes forward into file (from current position).

```
fseek(stream, fpos, SEEK_END);
```

Move `fpos` bytes from the end of the file.

File Types

Files in C can contain data in two formats:

- 1 As text, where each byte is interpreted as an ASCII character mapping.
- 2 Or as raw *binary* data, the meaning of which is left to the program to determine.

These files are referred to as text files and binary files respectively. All the I/O covered so far applies to text files.

Each format has it's advantages and disadvantages.

Text versus Binary Files

Text Files

- **Pro** - Can be read by almost any program on almost any computer. (And by humans!).
- **Con** - Inefficient - Numbers can take up much more space than necessary.

Binary Files

- **Pro** - Accurate and efficient.
- **Con** - Incompatible - Binary data can be very difficult to read by other programs, and incredibly difficult between processor architectures.

Variable sizes

- Different data types take up different amounts of memory.
- Example `float` is smaller than `double`.
- The `sizeof` keyword gives a type's size (in bytes).

```
#include <stdio.h>
```

```
int main()  
{  
    printf("sizeof(float) = %d\n", sizeof(float));  
    printf("sizeof(double) = %d\n", sizeof(double));  
    printf("sizeof(int) = %d\n", sizeof(int));  
    printf("sizeof(short) = %d\n", sizeof(short));  
    return 0;  
}
```

Handling Binary Files

Binary files are opened using `fopen`, we need to add "b" to the file mode though:

```
stream = fopen("data.dat", "rb");
```

Data is read from and written to a binary file using `fread` and `fwrite`. For example:

```
double values[10] = {0.0};  
/* we have "stream" an open binary file */  
read = fread(values, sizeof(double), 10, stream);  
if (read != 10)  
{ read failed }
```

Please see `<stdio.h>` for more details.

Permanent versus Temporary Files

- Sometimes we may not wish for a file to be permanent.
- Temporary files provide a means for storing intermediate computations.
- We open a temporary file using the `tmpfile()` function.

Example

```
stream = tmpfile();  
if (!stream)  
{  
    /* handle failure */  
}  
/* fread, fwrite, fprintf, fscanf */  
fclose(stream);
```

Dynamic Allocation

The heap

- C has set aside a special block of memory known as the *heap*.
- Memory can be requested from the heap.
- If memory is available, it is provided, otherwise a request is made to the operating system.
- The implementation details of the heap are “undefined”.

sizeof

- We need to know how much memory to request.
- The `sizeof` keyword returns the size (in bytes), of a variable or data type.

How to Borrow Memory

We use the `malloc` function (defined in `<stdlib.h>`) to request memory and `free` to return it to the heap.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #define NCOUNT 10
4 int main()
5 {
6     int * numbers, loop;
7     numbers = (int *) malloc(sizeof(int)*NCOUNT);
8     if (!numbers)
9     {
10         fprintf(stderr, "Unable to allocate memory\n");
11         return -1;
12     }
13     for (loop = 0; loop < NCOUNT; loop++)
14         numbers[loop] = loop;
15
16     for (loop = 0; loop < NCOUNT; loop++)
17         printf("numbers[%d] = %d\n", loop, numbers[loop]);
18
19     free(numbers);
20     return 0;
21 }
```

How to Borrow Memory - Details

```
void * malloc(size_t Size);  
void free(void * Memory);
```

What is `void *`?

- `void *` is a pointer to an unknown data type, it consists only of an address.
- We can explicitly cast from `void *` to any other pointer type.
- Any pointer type can be implicitly casted to a `void *`.
- A value of `NULL` usually means that a function encountered an error.

What is `size_t`?

`size_t` is defined to be an unsigned integer type large enough to hold numbers returned by `sizeof`.

How to Borrow Memory - More Details

Two more, useful memory management functions are:

<code>realloc</code>	Modifies the size of a memory block.
<code>calloc</code>	Allocates memory, and sets every byte to 0.

Some common heap problems

- Allocating memory is slow and should be done as few times as possible.
- Memory allocation can fail, every `malloc/calloc` should be checked (or at the very least `asserted`).
- Every `malloc/calloc` should have a corresponding `free`, memory is *leaked* if it's not `freed` after use.

1-D Example: Fibonacci (again!)

```

1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int main()
5  {
6      double * fibs;
7      unsigned int nfibs=0, loop;
8      while (nfibs < 2)
9      {
10         printf("How many Fibonacci numbers are needed (>1)?\n");
11         scanf("%u", &nfibs);
12     }
13
14     fibs = (double *) malloc(sizeof(double)*nfibs);
15     if (!fibs) /* malloc failed? */
16     {
17         fprintf(stderr, "Unable to allocate memory!\n");
18         return -1;
19     }
20
21     fibs[0] = 1.0; fibs[1] = 1.0;
22     for (loop = 2; loop < nfibs; loop++)
23         fibs[loop] = fibs[loop-1]+fibs[loop-2];
24
25     for (loop = 0; loop < nfibs; loop++)
26         printf("fib[%u] = %lg\n", loop, fibs[loop]);
27
28     free(fibs);
29     return 0;
30 }

```

Pointers to Pointers: `**ptr`

- Pointers in C are very flexible, to the extent that we can form pointers to pointers!
- This is known more formally as the “level of indirection”.
- Tensors of arbitrary rank (Fortran 95 is limited to 7) can be formed easily in C.
- The most useful example in C being matrices.

Matrices in C

Matrices in C are commonly represented by the `double **` type. This means “a pointer to a pointer to a `double`”.

Pointers and Arrays

We saw the connection between arrays and pointers in the last lecture. Given the array:

```
double ar [4] = {1.0, 2.0, 3.0, 4.0};
```

The elements of this array can be accessed as follows:

```
* (ar) == ar[0] == 1.0  
* (ar+1) == ar[1] == 2.0  
* (ar+2) == ar[2] == 3.0
```

Pointer referencing is also supported:

```
ar == &ar[0]  
(ar+1) == &ar[1]  
(ar+2) == &ar[2]
```

Allowed Pointer Operations

Declaration: `double * pA, * pB;`

Assignment: `pA = &var;`

Increment: `pA = pA + 1;`

Decrement: `pA = pA - 1;`

Difference: `gap = pA - pB;`

Comparison: `if (pA == pB)`

De-referencing: `*pA = val;`

Fixed Size Two-Dimensional Arrays

We can declare arrays of dimension higher than one, as follows:

```
double a[2][3] = {{1.0, 2.0, 3.0},
                  {2.0, 3.0, 4.0}};
```

Where the elements of `a` are denoted as expected:

<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>
<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>

To access the top left element:

```
myVal = a[0][0]; /* equal to 1.0 */
```


Fixed Size Two-Dimensional Arrays II

```
1  #include <stdio.h>
2  #define COLS 3
3
4  void printArray(int matrix[][COLS], int rows)
5  {
6      int i, j;
7      for (i = 0; i < rows; i++)
8      {
9          for (j = 0; j < COLS; j++)
10             printf("%d ", matrix[i][j]);
11         printf("\n");
12     }
13 }
14
15 int main()
16 {
17     int matrix[2][COLS] = {{1, 2, 3},{4, 5, 6}};
18     printArray(matrix, 2);
19     return 0;
20 }
```

Fixed Size Two-Dimensional Arrays III

We have seen an example of a two dimensional array:

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

- In memory it is arranged as follows:

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]
---------	---------	---------	---------	---------	---------

(i.e. as a one dimensional array).

- Fixed size arrays are very inflexible as they require the dimensions to be “hard coded”.
- They are allocated from the stack thus large arrays may cause problems.
- *Dynamically allocated* arrays overcome these restrictions.

Constructing Matrices with Pointers

```

1 double ** makeMatrix(unsigned int rows, unsigned int cols)
2 {
3     unsigned int i;
4     double ** matrix;
5
6     matrix = (double **) malloc(rows * sizeof(double *));
7     if (!matrix) return NULL; /* failed */
8
9     for (i = 0; i < rows; i++)
10    {
11        matrix[i] = (double *) malloc(cols*sizeof(double));
12        if (!matrix[i])
13            return NULL; /* lazy, we should really free
14                           all the memory allocated above */
15    }
16
17    return matrix;
18 }

```

Accessing Matrix Elements

Usage pattern for `makeMatrix`

```
double ** matrix = makeMatrix(rows, cols);
for (i=0; i < rows; i++)
    for (j=0; j < cols; j++)
        matrix[i][j] = 0.0;
free the matrix
```

- Accessing the dynamically allocated array looks identical to the fixed size ones, but “under the hood” things are a little different:

```
matrix[row][col] = (*(matrix + row) + col)
```

- The `makeMatrix` code on the previous slide contained a lot of `malloc` statements, is there a better way to allocate a matrix? (yes!)

A Better Way of Allocating Matrices

```
1 double ** allocMatrix(unsigned int rows, unsigned int cols)
2 {
3     double ** matrix;
4     unsigned int i;
5
6     matrix = (double **) malloc (rows*sizeof(double *));
7     if (!matrix) return NULL; /* failed */
8
9     matrix[0] = (double *) malloc (rows*cols*sizeof(double));
10    if (!matrix[0])
11    {
12        free(matrix); /* we don't need matrix any more */
13        return NULL; /* failed */
14    }
15
16    for (i = 1; i < rows; i++)
17        matrix[i] = matrix[i-1] + cols;
18
19    return matrix;
20 }
```

Why is allocMatrix Better?

- allocMatrix only uses 2 mallocs whilst, makeMatrix uses $\text{cols} + 1$.
- Meaning there are fewer points of failure (we only check two pointers for NULL),
- It is much easier to free a matrix allocated with the allocMatrix function, all we need to do is:

```
void freeMatrix(double ** matrix)
{
    free(matrix[0]);
    free(matrix);
}
```

Case Study: Matrix Addition

Let's define some utility functions to:

- 1 Allocate memory for the matrix (`allocMatrix`) - **done**,
- 2 Free a matrix (`freeMatrix`) - **done**,
- 3 Print a matrix (`printMatrix`),
- 4 Create a random matrix (`randomMatrix`),
- 5 Add matrices together (`addMatrices`)

We drive all these functions using a `main` function.

printMatrix and randomMatrix

```
1 void printMatrix(double ** matrix, unsigned int rows,
2                  unsigned int cols)
3 {
4     unsigned int i, j;
5
6     for (i = 0; i < rows; i++)
7     {
8         for (j = 0; j < cols; j++)
9             printf("%8.5lf ", matrix[i][j]);
10        printf("\n");
11    }
12 }
```

```
1 void randomMatrix(double ** matrix, unsigned int rows,
2                   unsigned int cols)
3 {
4     unsigned int i, j;
5     for (i = 0; i < rows; i++)
6         for (j = 0; j < cols; j++)
7             matrix[i][j] = (double)rand() / RAND_MAX;
8 }
```


addMatrices

```
1 void addMatrices(double ** matrixA, double ** matrixB,
2                 double ** matrixR,
3                 unsigned int rows, unsigned int cols)
4 {
5     unsigned int i, j;
6     for (i = 0; i < rows; i++)
7         for (j = 0; j < cols; j++)
8             matrixR[i][j] = matrixA[i][j]+matrixB[i][j];
9 }
```

The main function

```

1  int main()
2  {
3      unsigned int rows, cols;
4      double ** matrixA, ** matrixB, **matrixC;
5      printf("Enter rows cols: ");
6      scanf("%u %u", &rows, &cols);
7
8      matrixA = allocMatrix(rows, cols);
9      matrixB = allocMatrix(rows, cols);
10     matrixC = allocMatrix(rows, cols);
11
12     if (!matrixA || !matrixB || !matrixC)
13     { /* a little lazy, but it does the job */
14         fprintf(stderr, "Unable to allocate matrices!\n");
15         return -1;
16     }
17
18     randomMatrix(matrixA, rows, cols); randomMatrix(matrixB, rows, cols);
19     addMatrices(matrixA, matrixB, matrixC, rows, cols);
20
21     printf("\n\nmatrix A = \n");
22     printMatrix(matrixA, rows, cols);
23     printf("\n\nmatrixB = \n");
24     printMatrix(matrixB, rows, cols);
25     printf("\n\nmatrixA + matrixB = \n");
26     printMatrix(matrixC, rows, cols);
27
28     freeMatrix(matrixC); freeMatrix(matrixB); freeMatrix(matrixA);
29     return 0;
30 }

```

Results

Enter rows cols: 4 4

matrix A =

0.00125	0.56359	0.19330	0.80874
0.58501	0.47987	0.35029	0.89596
0.82284	0.74660	0.17411	0.85894
0.71050	0.51353	0.30399	0.01498

matrixB =

0.09140	0.36445	0.14731	0.16590
0.98853	0.44569	0.11908	0.00467
0.00891	0.37788	0.53166	0.57118
0.60176	0.60717	0.16623	0.66305

matrixA + matrixB =

0.09265	0.92804	0.34062	0.97464
1.57353	0.92557	0.46937	0.90063
0.83175	1.12449	0.70577	1.43013
1.31227	1.12070	0.47023	0.67803

Summary

- A `char` is the smallest addressable data type (1 byte). It can be used to represent a text character and a small number (−128 to 127) interchangeably.
- In addition to the three default streams (`stdin`, `stderr`, `stdout`), you can open and interact with file streams by opening them with `fopen`.
- Memory can be allocated dynamically using `malloc`. This is useful for arrays or matrices of an unknown, varying or large sizes.
- Considering the structure of the data in memory (and using the examples here) enables you to allocate elaborate data structures more efficiently.
- `free` is used to release previously allocated memory back to the stack; forgetting to 'free' memory causes memory-leaks.