UFS001C1
C Programming

Ian Johnson

Room 3P16

Telephone: extension 3167

Email: Ian.Johnson@uwe.ac.uk

http://www.cems.uwe.ac.uk/~irjohnson/ufs001c1.html

What is this module about?

The C Programming Language!

Books?

"***C Programming***", Tony Royce, Macmillan 1996

This is strongly recommended. You will need some form of C reference, this book is relatively cheap!

Other books:

"***Simple C***", Jim McGregor et. al., Addison-Wesley, 1998

I prefer Royce

"***The C Programming Language***", Kernighan, B.W. & Ritchie, D.M., Prentice Hall, 2nd edition 1988.

The classic C book written by the language inventors. Comprehensive & excellent - but expensive.

Assessment?

Two assignments, one due end of term 1, the other the end of term 2.

One examination

Tools & Environment

Working in 3P11/3P27

We are going to be using the standard unix tools under Linux.

Linux and all of the tools are Free! Installing & using linux at home is highly recommended!

A Short history of C and Unix

- Bell laboratories - the ancenstral home of C and UNIX.
- BCPL (due to Martin Richards) -- a user friendly OS providing powerful development tools developed from BCPL. Assembler tedious long and error prone.
- 1969 - The first UNIX system (Ken Thompson et. al.)
- 1970 - A new language ``B'' (due to Ken Thompson) for the first UNIX system (typeless).
- A totally new language ``C'' (Brian Kernighan & Dennis Ritchie) a successor to ``B''.
- 1972 - "The number of Unix installations has grown to 10 with more expected" *2nd Edition manual*
- 1973 - UNIX OS almost totally rewritten in ``C''.
- 1974 - Licensed to Universities for educational use
- 1978 - K&R - *de facto* standard from C's inventors.
- 1983 - ANSI standard C - formalised and agreed *de jure* standard.

Currently over 90% of UNIX (and Unix like operating systems) is written in C, as is the C compiler itself and all of the user commands.

A plethora of "Unixes", most commercial, (e.g. HP-UX, AIX, Solaris) are derived from SVR4, many others are derived from BSD. Some (e.g. Linux) are a little schizophrenic in this regard (e.g. SVR4 init with BSD ps).

Starting at C

What is C?

A **small** language that:

- makes extensive use of function calls
- has loose typing (cf. Pascal)
- is structured
- enables low level (bit level) programming easily
- supports pointers - extensive use of pointers for memory, array, structures and functions.
- is efficient
- is portable (C can be compiled on a variety of computers)

But:

- It has poor error detection which can scare the beginner. (This can be improved by setting compiler options, and using lint)
- It assumes the programmer knows what she is doing!
- It can easily be badly written (see next slide)!

What is C Not?

- C is not an object-orientated programming language
- C is not C++ or Java (although both are based on C)

Obfuscated C

The annual obfuscated C competition (http://www.ioccc.org) is a contest to see who can write the worst C.

Here is a previous winner, would you like to explain what it does, or fix it if a problem developed?

```
#include <stdio.h>

main(t,_,a)
char *a;
{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86, 0, a+1 )+a)):1,t<_?main(t+1, _, a
):3,main ( -94, -27+t, a
)&&t == 2 ?_<13 ?main ( 2, _+1, "%s %d %d\n"
):9:16:t<0?t<-72?main(_,
t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/
w#q#n+,/#{l,+,/n{n+\
,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K
w'K:'+}e#';dq#'l q#'+d'K#!/\
+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'
/+#n';d}rw' i;# ){n\
l]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw'
iK{;[{nl]'/w#q#\
n'wk nw' iwk{KK{nl]!/w{%'l##w#' i;
:{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
:t<-50?_==*a ?putchar(a[31]):main(-
65,_,a+1):main((*a == '/')+t,_,a\
+1 ):0<t?main ( 2, 2 ,
"%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc \
i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m
.vpbks,fxntdCeghiry"),a+1);}
```

What exactly does it do? (see next slide!)

Output from the previous program:

```
On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in a pear tree.

On the fourth day of Christmas my true love gave to me
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the fifth day of Christmas my true love gave to me
five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the sixth day of Christmas my true love gave to me
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the seventh day of Christmas my true love gave to me
seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the eighth day of Christmas my true love gave to me
eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the ninth day of Christmas my true love gave to me
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the tenth day of Christmas my true love gave to me
ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the eleventh day of Christmas my true love gave to me
eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the twelfth day of Christmas my true love gave to me
twelve drummers drumming, eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.
```

Teaching (and learning) any programming language (particularly your first) is always difficult to start with.

To become proficient you will need to work your way through ***Royce*** or an alternative book.

In the early weeks, complete the random number exercises and the exercises from Royce (if you haven't bought a copy, check with a fellow student what they're doing.

THE ONLY WAY TO LEARN TO PROGRAM IS TO PROGRAM!

To get the most from C you will need to know what functions are available in the standard libraries.

Starting at the beginning …

… (or rather *main*)

All C programs consist of functions. Execution of a C program begins with the function *main()*

It therefore follows that all C programs must have at least one function, and one & only one called *main()*.

A function is declared in C as:

```
Return_Type Function_Name(Arguments)
  {
     /* function contents */
  }
```

Functions can call other functions and make use of their return value, or pass then data to work with as arguments.

This is how we can reuse other people's code, and modularise our own.

All C compilers depend on libraries of standard functions to be useful, since the language itself is very small.

All libraries have an associated header file which contains prototypes for the functions, (so the compiler can check that you're using them correctly), together with data structure definitions, and other useful definitions.

We can declare both the Return_Type & Arguments to be ***void,*** A Special case where we don't wish to pass arguments or return a value.

```
int main(void)
  {
    return 0;
  }
```

Is probably the worlds simplest C program. Some compilers will allow you to declare main as void, but gcc will warn you!

Comments

```
/* this is a comment */
```

Comments should add meaning to a program, and aid understanding.

```
day = day + 3 % 7 /* add 3 to day modulo 7 */
```

Does not. If you can read C, the comment is pointless, if you can't – why are you trying to edit this program?

Try to explain how and why you are doing something.

```
day = day + 3 % 7 /* skip the weekend */
```

Beware:

```
/* Comment 1
/* Comment 2
   End of comment 2 */
   End of Comment 1 */
```

is illegal, although many compilers will not be concerned.

```
// This is a C++ comment **not** a C comment
```

Some compilers will let you get away with this others will not!

(*Hint for coursework: this is a pet hate of mine!*)

Many C compilers today also compile C++.

Some determine which is which by file extension (.C, or .cpp for C++), others guess.

Some accept C++ comments, others do not.

AGAIN: DO NOT USE C++ COMMENTS IN C!

Types

All functions return a value of a specified type, (although this may be void, i.e. none). If none is stated, then you have implicitly declared that function to return an int.

All variables have a specific type.

The type of a variable tells the compiler how much memory to allocate to hold the variable, and what it will be used for.

Older C compilers allowed assignment between incompatible types. Modern compilers do not!

Type checking allows the compiler to more carefully evaluate the correctness of your code. You can make the compiler stricter through using optional switches (e.g. -pedantic - more later)

The basic types follow, more later!

- int An integer value (i.e. a whole number). int's in C are of the most convenient size for the CPU architecture, typically today 32 bit.

- char a byte (8 bit) sized integer, convenient for holding ASCII characters (which require 7 bits).

- float a single precision floating point number.

- double a double precision floating point number (more accurate but requiring usually twice the memory.

we can also have "`short int`" or `short` and "`long int`" or `long`

Unlike some languages (e.g. occam2) which define the exact size of a variable, C does not.

Instead:

16bit <= short <= int <= long
long >= 32bit

long double is also allowed (but very rarely seen!)

All integer types can be *unsigned*

Unsigned char	0 … 255
char	-128 … 127

We also have `register`, `static` & `volatile` type modifiers which we'll discuss later.

The minimum & maximums are defined in the standard header file <limits.h>

On my home linux system, this can be found at:

`/usr/include/limits.h`

Here is a sample:

```
/* Minimum and maximum values a `signed char' can
hold.  */
#  define SCHAR_MIN   (-128)
#  define SCHAR_MAX   127

/* Maximum value an `unsigned char' can hold.
(Minimum is 0.)   */
#  define UCHAR_MAX   255

/* Minimum and maximum values a `signed int' can
hold.  */
#  define INT_MIN(-INT_MAX - 1)
#  define INT_MAX2147483647

/* Maximum value an `unsigned int' can hold.
(Minimum is 0.)   */
#  define UINT_MAX    4294967295U

/* Maximum value an `unsigned long int' can hold.
(Minimum is 0.)   */
#  if __WORDSIZE == 64
#   define ULONG_MAX  18446744073709551615UL
#  else
#   define ULONG_MAX  4294967295UL
#  endif
```

The last example illustrates the size issue nicely. Note the difference in maximum value of an unsigned long on a 64bit as against 32bit architecture.

If want to check a value is in range, we should use these constants, e.g. INT_MAX.

If we want to find out the size of a variable, we can use the C function sizeof(), or work it out from the constant.

SIGNED <something> MAX = 2^{n-1} (where n is the number of bits).

Standard header (or include files)

C depends on libraries to be useful. The language itself is very small.

All libraries have an associated header file, which prototypes functions (so that usage can be checked), contains data structure definitions, and other useful definitions (e.g. the maximum size of data types (limits.h) as discussed above).

the notation:

#include <*headerfile.h*>

says include *headerfile.h* from the system location of header files. On a unix system, this is often /usr/include by may not be.

Where are include files kept for Visual Studio?

If you create your own header files then you can reference them as

`#include "myheader.h"` (if it is in the current directory)

or

```
#include "../include/ myheader.h"

#include "/home/staff/irj/p1/myheader.h"
```

Variables

variable names in C start with a letter, and can contain upper and lower case letters, digits and the underscore "_" character.

C is case sensitive, C and c are not the same!

To declare an integer variable:

```
int my_integer_variable;
```

Convention in C is that lower case names are used for variables, all upper case names for defined constants:

```
int my_integer_variable = INT_MAX;
```

It is acceptable to name variables with mixed case:

e.g.

```
int CountVar77
```

Generally variables names should be carefully chosen to communicate with any reader of your program …

… however many C programs will use `i, j, k` as the name of subscript or count variables for traditional reasons.

Constants

Whilst C has a `const` type modifier, it is rarely used:

```
const double pi = 31.4159;
```

far more frequently, C programmers will use the pre-processor directive:

```
#define PI 3.14159
```

These behave differently. You can think of #define as copying the text 3.14159 into your program to replace every occurrence of PI.

Constants have a default type, without a decimal point this is integer

1000 is an int
1000L (or l) is a long
1000U (or u) is an unsigned int

Integers can also be specified in octal (base 8) or hexadecimal (base 16). A leading 0 implies octal, a leading 0x implies hexadecimal.

With a decimal point, constants are double precision floating point.

3.14159 is a double
3.14159F (or f) is a float

doubles can also be represented in exponent form, e.g. 1e3 is 1000 stored as a double.

Character constants are written as 'a'. We can also specify a character escape as:
'\ooo' (3 octal digits)

or

'\xhh' (2 hex digits)

Remember, characters are treated as numbers!

```
char x;

x = 48   x = '\x30'   x = 0x30     x = '0'
```

all give x the same value.

In short, C characters are 8 bit (1 byte) integers, usually signed, which are generally used to store ASCII characters.

We also have some predefined special characters:

(See next slide)

\b	backspace	\a	alert (bell)
\n	newline	\r	carriage return
\t	tab	\v	vertical tab
\\	\	\?	?
\"	"	\f	formfeed
\'	'	\0	the null character

As well as others (see for example K&R section A2.5.2, p 193)

Strings

String constants such as

```
"Hello World"
```

are specified in double-quotes.

It is vitally important to remember that "a" and 'a' are NOT the same.

"a" requires 2 bytes of memory, one for the ASCII code for a, one for a null character.

'a' requires 1 byte, to just hold the ASCII code for a.

Strings are arrays of characters with an ASCII NULL (value 0) as the last character.

Characters are single byte values.

A first program

As is traditional hello world

```
#include <stdio.h>  /*include standard C I-O header    */

int main(void)      /* declare a function called main, */
  {                 /* taking no arguments returning no
                       value                           */
    printf("Hello World\n");  /* use the library function
                                 printf to output our
                                 string                */
    return 0;       /* return from the function        */
  }
```

When we return from *main()*, we exit to the operating system.

When other functions return, control passes back to the calling function.

You should have already built this program!

Try compiling on the command line using:

```
gcc -pedantic -Wall –o hello hello.c
```

The compiler

A brief digression at this point.

The C compiler we are using is gcc. This compiler is both free and produces high quality code.

To find out everything about gcc, use the info system.

`info gcc` will get you started.

gcc supports many many options. (Use info or man to find out more)

Some important options you are likely to need shortly. are listed overleaf:

-g	Generate debugging information. If you can't see your source code in GVD, you've probably left this off!
-Wall	Warn about most potential problems/errors
-l*name*	Link with library lib*name*
-o *name*	Generate *name* as your programs output (executable). Without this option, you'll produce an executable called a.out.
-pedantic	Reject all programs than use ANSI/ISO forbidden extensions, and generate all warnings required by ANSI/ISO standards. This option is useful to begin with, but of limited value when you've gained confidence with the language
-ansi	Support the ANSI/ISO C standard

It is worth remembering that these are options!

Other programming tools

We've already met and used: gcc, emacs, gvd

- Lint (although uncommon on linux systems, http://lclint.cs.virginia.edu provides a link to splint a good free lint, together with several papers), is a program for "pulling the fluff" out of your C.

- Make - a very useful tool for managing complex development. See the tutorial on this module's web page.

- RCS - Revision Control System (Nigel's worksheet - available on the module web page - is an excellent start) CVS might also be worth a look.

- Emacs (tools/version control provides RCS support). ***Esc-X compile*** is happy to run make

Operators

All operators have precedence, (that is some are evaluated before others) and associativity (which direction they are evaluated in).

```
x = 3
```

associates right to left,
(the right hand side of the = is calculated first),
this should be commonsense, others might not be.

We'll revisit this shortly.

Arithmetic operators

Assignment	=				
Binary Arithmetic	+	-	*	/	as expected (both integer & real)
	%				modulo

Binary in this case means 2 operands!

Increment / Decrement (++ --)

x-- use x, then decrement
--x decrement x then use

Bitwise operators

(Section 5.10, page 239 Royce; Section 2.9 page 48 K & R)

These enable us to manipulate the individual bits of (almost always) integer values. (Often applied to `unsigned char`'s) Whilst C provides bitfield capabilities, these are not commonly used (since implementation dependent).

&	AND
|	OR
^	XOR
<<	left shift
>>	right shift
~	one's complement

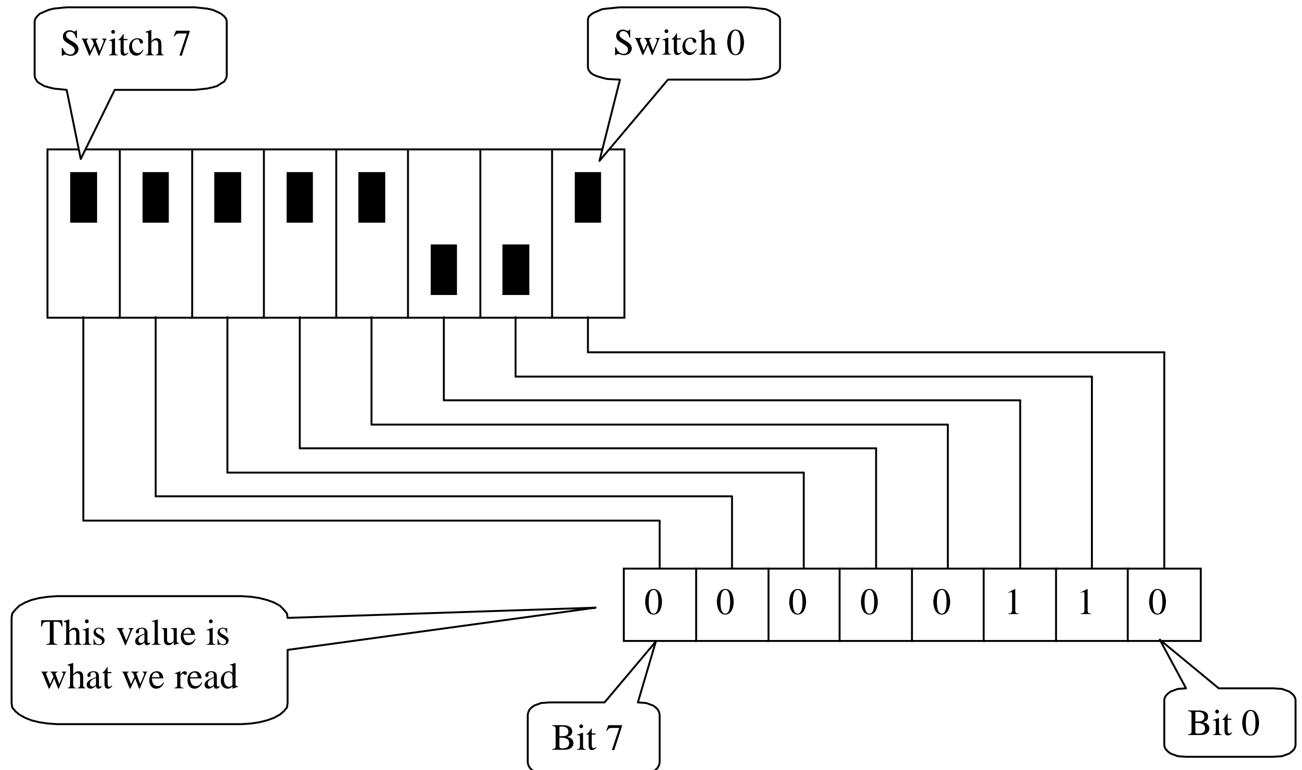
For low level programming and IO problems these are vitally important.

Using operating systems such as MS-DOS, we might use the `inport()` and `outport()` functions for memory mapped IO. (e.g. Marco - next term)

On Unix based operating systems, generally only kernel (as opposed to user) processes can directly access hardware resources. As a result, we usually have an abstraction, often file based. (e.g. working with the FAB's - we use standard read-write functions on file descriptors).

Imagine we have an 8bit IO-port (the FAB has 3 such ports). In our C program we'll have a declaration such as:

```
unsigned char portAvalue;
```



Again assuming the FAB and linux, we could obtain this value by using the `read()` function.

```
read(portA, &portAvalue, sizeof(unsigned char));
```

Our variable portAvalue now contains the above bit pattern (0x06).

We may wish to check whether switch 1 is set.

```
if (portAvalue & 0x02)
  printf("switch 1 is ON\n");
```

We may wish to test if switches 1 & 2 are both set:

```
if (portAvalue & (0x02 | 0x04))
```

We may wish to make our meaning more clear:

```
#define SWCH0    0x01
#define SWCH1    0x02
#define SWCH2    0x04
…
#define SWCH7    0x80

if (portAvalue & (SWCH1 | SWCH2))
```

We may wish to test all the switches in order:

```
for (i=0x01;i != 0; i << 1)
  if (portAvalue & i)
    printf("switch %d is ON\n",i-1);
```

Similarly with output (e.g. the LED's on the FAB):

We have 8 LED's (numbered 0 through 7)

To light the 3rd (number 2) with altering an others:

```
current_value = current_value | 0x04;
```

To turn it off, without altering other values:

```
current_value = current_value & 0xfb;
```

Note: `x = ~x`

will convert 0x04 to 0xfd & vice versa (assuming one is the current value of x)