LANGUAGE OF THE 'C'

STEVEN GOODWIN

A language so synonymous with computing history and Unix it's very name is the epitome of the elite. These articles for the beginner, teach you the fundamentals of 'ANSI C', as well as providing interest snippets from under the hood of the compiler.

When Stephen Hawkin wrote his best-selling work 'A Brief History of Time', he remarked that his editor informed him that every equation would cut the sales of his book in half. In a similar fashion, my editor informed me that every paragraph I wrote without a piece of code would lose half my readership. So, in the interests of keeping everyone happy, here is the obligatory 'Hello,World' program.

```
1    #include <stdio.h>
2
3    int main(int argc, char *argv[])
4    {
5        printf("Hello, World\n");
6        return 0;
7    }
8    /* A variation of the standard 'Hello
World' problem - seen everywhere */
```

Type the above code into your least unfavourite text editor (omitting the line numbers), save it as 'myfirsttime.c' and in a shell type,

```
gcc myfirsttime.c
./a.out

or, if you prefer,

gcc myfirsttime.c -oMyProgram
./MyProgram
```

Should you have the development environment installed correctly, Linux should pop back with the phrase 'Hello, World'.

What's Going On?

The simplicity of the example above belies the fact it contains the vital ingredients that make 'C', 'C'. Namely,

- The pre-processor
- The compiler
- The linker

The compilation process is built from three major steps, executed automatically by 'gcc' or 'make', in the above order. However, the whole process is just called 'compilation'; it is very rare to explicitly run any specific step (known as a *pass*). Only when there are problems would you refer to any specific step, e.g. "Help!!!! My program won't link!". See BOXOUT: GCC, for more information on the workings of this process.

The Pre-processor

Any line in 'C' that begins with the hash mark (#) actually belongs to the pre-processor, not the compiler. The common instructions (called *directives*) are *include* (as in our example), *define* (to create macros) and *ifdef* (which, along with *endif*, support conditional compilation). It runs immediately before the compiler, and gets to modify, fix, adapt and generally fiddle about with the source text.

By including the header file *stdio.h* (STanDard Input/Output) we may use the contents of that file within our program, as if we'd wrote it ourselves. In this case, we will use 'printf' to output text to stdout. i.e. the screen. By including the name in brockets ('<' and '>') we indicate that this file is stored (by default) in '/usr/include' with all the other system header files. If the file were stored in the current directory, we would use double quotes (") for the name. Relative paths can be used with the name (as are absolute paths, but these are not recommended for portability).

The pre-processor will also strip out comments from your source code - and like any good programmer you will have several of them in your code to strip out, right? For example,

```
8       /* A variation of the standard 'Hello
World' problem - seen everywhere */
```

Examples of the pre-processor:

```
#include <stdio.h>
#include "myprogram.h"
#define VERSION_NUM 1.2
#define MAX_TABLE_SIZE 32
#ifdef _WIN32
    printf("I refused to be➥
compiled under Windows➥
!\n");
#endif
```

Comments begin with the '/*' token, and ignore everything up to (and including) the first '*/' it sees, which ends the comment. They can not, therefore, be nested (since the second '/*' will have already been ignored), but the comment may extend over several lines.

```
/*
   A comment 'box'
   over four lines
*/
```

The Compiler

The mother! The daddy! This takes our '.c' file (replete with pre-processing modifications, macro substitutions, and so on) and turns it into a binary file. Not an executable *elf* file, though. A binary. It is called an *object file*, and has a '.o' extension. When compiling single files (with gcc) it is usually deleted once compilation has finished. With larger projects (using make files), they are kept to improve speed, since only out of date '.o' files will be re-built. Library files are also '.o' files, but (naturally) these are not deleted after use!

The Linker

The final hurdle. It combines 'myfirsttime.o' and any specified *libraries* into a single executable. That is, it *links* them together. By default, the library file /usr/lib/libc.so is also included - since this contains standard C routines (such as printf) - as is the all-important start-up code: /usr/lib/crt1.o. If you were writing a program using mathematics (with functions like sin or cos), you would use '#include <math.h>' in your source file, and link in the maths library (/usr/lib/libm.so) with,

```
gcc mathscode.c -lm
```

Execution always begins with a function called 'main'. So, if you haven't created a 'main' function, the linker will complain with an 'undefined reference to 'main'' error. You will also get this error if you use the 'math.h' header file (which says 'I wish to use the 'sin' function), but then fail to link the appropriate (libm.so) library (which says 'here is the code for the 'sin' function I told you about earlier').

A Break From The Old Routine

Every 'C' program is made up from routines, called *functions*. Even the library code (like 'printf') is implemented as functions. There are no built-in functions with 'C' (see BOXOUT: Design Philosophy), and there is no difference in the way we call a library function, compared to one of our own.

Listing 2 uses three invocations of a function in 'C':

The function name can consist of any combination of alphanumeric characters, or the underscore ('_'), but must begin with a letter or underscore. The latter is not recommended as some systems use underscored names for their own macros, variables and functions. Case is important, because the two variables 'count' and 'COUNT' are seen differently by the compiler. They must also be unique (in the first 32 characters), and must not clash with any of 'C' reserved words (see BOXOUT: Reserved Words).

If functions could only process information they would still be useful. But not very. Like a Von Neumann architecture, it requires input and output. This is achieved with *parameters* (sometimes called *arguments*) and a *return type*, respectively.

All functions require that a return type, and parameter list is given. However, as seen in the example above, the word 'void' can be used to indicate that nothing comes out of the 'Banner' function - and nothing goes in.

Into the Gap

When a function is called, data is passed into it with *parameters*. As many as you like, as long as each is separated by a comma. Each parameter consists of a type (to describe its size and structure) and a name (or *identifier*, so it can be referenced).

All parameters in 'C' are passed 'by value', so should the identifier get changed within the function (intentionally, or otherwise) then the data held in calling function will not get changed because only the *value* was passed to the function.

We Can Work it Out

Now to get information *out* of a function. One of the limits we have to live with for the moment is that we can only *return* one piece of data. This is because 'C' only supports a single *return type*. We'll look at ways around this in future articles.

Declaring this return type is done by prefixing the function name with the, er, type you want to return. You then indicate what value you want to return by using the 'C' instruction, er, *return*. (This 'C' lark's not as hard as it looks, is it?)

In listing 4, line 16 gives the value of '100' back to 'main' which then (at line 7) stuffs it into (the more technical term is 'assigns it to'!) the variable 'iNum'. Some people will place brackets around the value to return, but this is a hang over from K&R C dialect and no longer necessary.

Note that if the function is declared as having a return type you **must** provide one, otherwise the compiler will generate an error. Should there be a case where an appropriate value could not be returned (e.g. size of a file that doesn't exist), then allow the return type to include an error code.

In cases where there is no return type (e.g. the function begins 'void', as with 'void LeaveGap(int

Listing 2

```
1    #include <stdio.h>
2
3    void Banner(void);
4    void LeaveGap(int
iLines);
5
6    int main(int argc, ch
ar  *argv[])
7    {
8        Banner();
9        LeaveGap(2);
10       return 0;
11   }
12
13   void Banner(void)
14   {
15       printf("My
program\n");
16   }
17
18   void LeaveGap(int
iLines)
19   {
20   int i;
21
22       for(i=0 ; i <
iLines ; i++)
23           printf("\n");
24   }
```

Listing 3

```
1    #include <stdio.h>
2
3    void ParamChange(int
iNum);
4
5    int main(int argc, ch
ar *argv[])
6    {
7    int iNum;
8
9        iNum=10;
10       printf("iNum =
%d\n", iNum); /* iNum=10 */
11       LeaveGap(iNum);
/* Passes the value of iN
um - 10 - not iNum
12   itself */
13       printf("iNum =
%d\n", iNum); /* iNum=10 */
14       return 0;
15   }
16
17   void ParamChange(int
iNum)
18   {
19       iNum = 0;
20   }
```

For Listing 2

3&4	**As a Prototype**	Tells the compiler that there is *going to be* a function in the code called 'Banner', but it has yet to appear in the source. It gives the compiler enough information allowing it to be used, pending the implementation - which may be in this '.c' file. Another '.c' file. Or a '.o' library file. You should always include prototypes for your functions, and naturally, the parameters should always match, however it is not necessary to include the name (iLines). Sometimes, prototypes are prefixed with the reserved word 'extern', meaning that this is the prototype for a function that resides in another file (*extern*al to this one).
8&9	**As a Function Call**	Makes a call to the 'Banner' code, and once complete continues with the statement immediately following it. All function calls must include brackets, even if there are no parameters to pass. A function name *without* brackets, although valid, is actually a pointer to the memory location where the function is stored!
13-24	**As a Definition**	The code! The braces indicate the start and end of the function definition. Between them lie the instructions, each ending with a semi-colon. See BOXOUT:Layout.

GCC

gcc allows you (the user) to review the compilation process. Each stage can be processed with it's own program, and generates its own intermediate output file. Although not particularly useful for the beginner, it is sometimes interesting to see how the path from source to executable occurs. Note: gcc also gives you access to the assembler-generated output from the compiler; a step I do not explicit consider here.

To see these intermediate files, build your code with, gcc -save-temps myfirsttime.c
Although I refer to 'libc.so' as a file used by linker, it is not always (ever?) a binary file. It is, in fact, a linker script indicating the true location of the shared library (or libraries). Usually /lib/libc.so.6'.
(*) GCC doesn't actually use this file for pre-processing. It is included to demonstrate the method.

iLines)') you may still use 'return' to leave the function early - you just omit any return data.

```
void Banner2(void)
{
        /* Do stuff here */
        return;
        /*
** Any code here is never called - but being a
** compiled language, must still be valid
syntax.
*/
}
```

Functions that do not return a type are sometimes categorised as procedures. From a stylistic point of view, procedures should be named depending on what *they do*. Whereas functions should be named by what *they return*.

Exile On Main Street

Earlier, we said that *main* was just a function like any other. Is that really true?

I'm afraid it is! Although we never call it (*), Linux does. Through the shell. It prepares the two parameters for main (argc and argv) from the arguments you type at the prompt. It also takes the return value (Listing 5, line 9) from the program and passes it to the shell as the exit code.

The 'argc' parameter describes the number of parameters passed to this program on the command line. 'argv' is a pointer (you'll have to be patient to learn about these beauties!) to an array (you'll have to be less patient to learn about these!) containing each of the arguments.
Run listing 5 with:

```
./a.out Hello World
```

and you should see:

```
0 : ./a.out
1 : Hello
2 : World
```

In addition to the *real* arguments (which follow the quoting conventions for the shell you are using), there is one surprise. The name of the program as argument zero. Replete with path.

And the exit code? When a program completes, this value is returned to the shell indicating its status. Zero (0) is used for 'completed successfully', while one (1) indicates a failure. The header file <stdlib.h> creates two text macros, EXIT_SUCCESS and EXIT_FAILURE that indicate this, and they can be *return*ed from 'main', instead of the values 0 and 1.

There is no good reason for not including an exit code (at least 'return 0;') at the end of your 'main' function. It is certainly better (and more standards compliant) than declaring your function as 'void main(void)', which gcc will warn against, but still allow.

There was a case, years ago, when a programmer (writing assembler, not C) created a one line program - with a bug in it! The bug being he didn't return an exit code, confusing the OS, causing it to take the next number it saw (which he had not placed) and use it. Don't fall into the same trap!

(*) It is *technically* possible to call main from inside your program (recursively) because it is just a function! However, very few people do. And even fewer in real-world environments.

Wind Of Change

Variables: the proverbial lifeblood of a program. They run through the veins (functions) passing information, handling sort data, and holding your frag count in Quake! We have already seen variables used in the example above, but I didn't dwell on them. Now I will!

Variables in 'C' have more restrictions than their counterparts in script languages. Notably, they *must* have a specific *type* that is declared *before* they are used. Also, once given a type (say, int) they can not change it. Ever. This allows the compiler to provide a more optimal storage method (say the heap, or a specific register) which in turn produces faster code.

```
int iNumberOfNames;
long iMaximumNumberOfNames = 16;
float fRadiusOfCircle, fCircumference;
unsigned char cMiddleInitial;
```

Above, we have declared five variables. The name given to each is arbitrary; any combination of numbers, letters (either case) and the underscore may be used (as with functions, above). However, for stylist reasons, I prefix each name with a single

letter indicating it's type, be it *i*ntegral, *f*loating point or a character.

When declared, the variable has no value. This doesn't mean the variable is zero. It means it has no *discernible* value – i.e. it is filled with garbage and could be anything. It is therefore imperative that all variables are initialised before use, perhaps in the declaration (as shown with 'iMaximumNumberOfNames' above), or in a line on its own using the copy-cat syntax of, (also see BOXOUT: Constant Values)

```
iMaximumNumberOfNames=16;
```

Most types may be considered signed, or unsigned. Signed means it is capable of storing a sign (either + or -) with the number. Unsigned means there is no sign, and therefore always positive. By default, types are signed (except in the case of 'char's, which is compiler-specific!) It is generally considered bad practise, however, to write code that *relies* on all chars being 'unsigned' (or vice-versa).

This sign/unsigned feature can cause problems. Should you store a character from the input stream (say) in a 'char', and the system you are using has 'unsigned char's then the end of file token (EOF, which equals -1) can not be correctly identified. FWIW, gcc defaults to 'signed char's, so you should have no such problems. But it always best to avoid the possibility altogether, and use a variable with enough space for all possible outcomes – *and* an error code.

Table 1 indicates the range of values possible with each data type to help you choose the appropriate data type for any given occasion.

Some programmers will write 'long int'. Since 'int' is considered the default type, this actually means 'long'. And when a programmer writers 'short int' – they mean short.

For portability between machines, your code should not rely on the size of any type (although 'int' is large enough to be used for most purposes). Since this can be difficult, many programmers will define their own types called 'WORD' or 'int32_t', implying the variables' size, with the code. These types area used in exactly the same way as the built-in types like 'int' and 'float', and are defined with the lines:

```
typedef unsigned short WORD;
typedef signed int int32_t;
```

Then, should the platform change, only these 'typedef' instructions need to change. There are examples of this in */usr/include/sys/types.h* of most distributions.

Finally, there is one special type you should be aware of: *size_t*. Any variable declared with this type has enough bits in it (literally) to reference any address in memory (and consequently any array that can fit *into* memory). It is usually an int. However, in an embedded system where the processor is, say, a Z80 (making the natural size for an int, 16 bits) with an extended memory, able to access a megabyte, the size_t type would have to be 32 bits long. Like I said, you should be *aware* of it, even if it doesn't make too much sense at the moment.

Not all types are typical

'C' may have types, but it is only *weakly*-typed. So, if you have two variables,

```
int iNum = 10;
float fVal = 12.5f;
```

The compiler will not stop, and not usually warn, you when mixing incompatible data types with code like:

```
iNum = 0.4f;
iNum = fVal * 7;
fVal = iNum / 3;
```

It is possible to predict the outcomes of these expressions, since 'C' has rules for these things. Am I going to tell you what they are? No! I'm going to tell you not to do it!

So there!

You're not local, are you?

Time for another example.

Here we have demonstrated two types of variable. *Local* and *global*. The variables declared inside the function braces (lines 7 & 14) are termed local, since they only belong to code inside the function; between the braces. This is termed its *scope*. The variable outside all the functions, at line 3, is termed global. i.e. it has *global scope*, and as such can be accessed by any of the functions.

If a variable is declared at local, and global scope (as in the example above), the local variable will 'hide' the global one making is inaccessible.

Global variables can be placed anywhere in the file, but since they are generally considered 'a bad thing', it is best to keep them together in a coders commune at the top of the file! You might want to adopt a naming convention to distinguish them and prevent the 'hiding problem', above. Perhaps by prefixing them with 'g_'.

Instructions

So far, I've referred 'instructions'; the 'printf instruction', the 'if instruction', the 'return instruction' and so on. In truth, they are something more than just instructions. They are statements. Each function comprises of braces, and a list of statements, where a statement can be anything from table 2.

Listing 4

```
1    #include <stdio.h>
2
3    int GetNumEntries(void);
4
5    int main(int argc, char *argv[])
6    {
7    int iNum;
8
9        iNum = GetNumEntries();
10       printf("iNum = %d\n", iNum);
11       return 0;
12   }
13
14   int GetNumEntries(void)
15   {
16       return 100;
17   }
```

Listing 5

```
1    #include <stdio.h>
2
3    int main(int argc, char *argv[])
4    {
5    int i;
6
7    for(i=0;i<argc;i++)
8    printf("%d : %s\n", I, argv[i]);
9        return 0;
10   }
```

Listing 6

```
1    #include <stdio.h>
2
3    int iVar=100;
4
5    void fn(void)
6    {
7    int iVar2=20;
8
9        printf("fn1 : iVar = %d, iVar2 = %d\n", iVar, iVar2);
10   }
11
12   int main(int argc, char *argv[])
13   {
14   int iVar=1, iVar2=2;
15
16       printf("main : iVar = %d, iVar2 = %d\n", iVar, iVar2);
17       fn();
18
19       return 0;
20   }
```

Reserved Words

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

Table 2 implies a few things. Let me clarify, with a brief Q & A.

Q. When writing an 'if', you don't need braces. True?

A. True. So long as you only have one statement, since the second one would be part of normal code, and not the 'if'.

Q. But my 'one statement' could be a compound statement; containing lots more statements. Really?

A. Yes. It's like asking a 'three-wish genie' for *another* three wishes. In programming 'C', that's allowed!

Q. Do the braces mean I can declare more local variables inside the 'if' statement, and not at the beginning of the function?

A. Yes! This refers back to the concept 'scope'. A variable can be declared after the brace, and will remain active until the end brace. So although you can use your newly created variable inside, say, an 'if' you can not use it outside.

```
if (iEntriesToSort > 0)
{
int a=1;
        /* Look, ma! I created a local variable in a strange place! */
}
/* 'a' is no longer valid, since it's gone out of scope */
```

Q. If you omit all three expressions from a 'for', and use the empty statement you can write code like: for(;;); What does that do?

A. Nothing. Forever. It sits and spins in an endless loop since there's no get out clause (exp2 above), and no code to jump out.

Q. Is there a use for the empty statement?

A. Yes. But be careful. If you were to write:

```
if (x>0);
        {
        /* X is five. Why is this always getting called? */
        }
```

you might fail to notice the semi-colon after 'if'. This counts as an (empty) statement, and therefore the open brace begins a new compound statement that will always execute.

Q. Ouch! Could that happen elsewhere?

A. Yes, but with slightly different results.

```
while(x>0);
        {
        x– –;
        }
```

This doesn't exit because the loop executes the empty statement while x is greater than zero, and since it never reaches the post-decrement, is shall always do so.

Q. So what's it good for them?

A. A unified syntax. (don't ask!) And easier to read code, when negative logic would make it less intuitive to the reader.

```
if (?some complicated expression?)
        ;
else
        DoStuff();
```

Design Philosophy and History

'C' was created in 1972 by Brian W. Kernighan and Denis M. Ritchie on the UNIX operating system,

Table 1

Type	Size	Unsigned Range	Signed Range	Notes
char	1	0 to 128	-128 to 127	A character. This may be signed, or unsigned, depending on your compiler.
short	2 *	0 to 65535	-32768 to 32767	A short integer.
int	4 *	0 to 4294967295	-2147483648 to 2147483647	Integer. Most common for loops and counters. It is also the size most fitting for your machine.
long	4 *	0 to 4294967295	-2147483648 to 2147483647	A long integer.
float	4	Always signed.	Using six digit precision	A floating point number in IEEE format. Numbers in floating point format are usually suffixed with 'f', i.e. 3.1415f to distinguish them with doubles.
double	8	Always signed. Using ten digit precision		Double precision, again from IEEE. Used when floating point numbers are not accurate enough.

Note: The standards are vague about the exact number of bits used for each type. As long as 'short' is larger than 'char', and 'less than or equal' to the size of 'int', it doesn't matter. It is up to the compiler to choose suitable sizes for the target machine.

running on DEC PDP-11 (ask your father about these machines!). It is a portable, general-purpose, programming language (as opposed to Cobol, say, whose use is very much limited to the business sector) with 'an economy of expression?and a rich set of operators'. It is also a fairly low level language (since it can manipulate individual bits within a byte), mixed with high level features (of control structures, such as 'while' and 'for' loops) making it appear somewhat of a hybrid.

To maintain the generality of the language, much of the functionality, like input/output and file handling, was implemented in libraries - not included as part of the language. This meant the compiler was small, could be re-used easily and the language learnt quickly. Most of the libraries could be written in 'C' itself, allowing for greater portability.

FWIW, when Unix was re-written, it took up just over 13000 lines of system code! Only 800 were assembler. The rest was 'C'.

Pass	**Program**	**Input File(s)**	**Output File**
Pre-processor	cpp (*)	myfirsttime.c	myfirsttime.i
Compiler	gcc	myfirsttime.i	myfirsttime.s
Assembler	as	myfirsttime.s	myfirsttime.o
Linker	ld	*	a.out

*myfirsttime.o/usr/lib/crt1.o/usr/lib/libc.so

Table 2

;	An empty statement. Means do nothing.
exp;	An expression, ending with a semi-colon. Such as 'a = b*20;'.
exp;	A function call (with brackets, remember), also ending with a semi-colon. We'll see shortly why a function is considered an expression.
if (exp) stmt	Conditional execute stmt is exp is true (i.e. non-zero)
if (exp) stmt1 else stmt2	Conditional, with 'else' clause
while (exp) stmt	Pre-check loop. Continually execute stmt while exp evaluates to true
do stmt while(exp);	Post-check loop. As the standard 'while' loop, but guarantees that stmt will always execute at least once.

for(exp1;exp2;exp3) stmt	Equivalent to:
	exp1;
	while(exp2)
	{
	stmt
	exp3;
	}
	One, two, or all three expressions may be omitted from the 'for'.

switch(exp) stmt	See control structures in a later issue
return;	Leave a 'void' function
return exp;	Leave any non-void function with a value
goto label;	<cough!> <splutter!>
{ stmt_list }	A compound statement. You may, optionally, declare variables at the beginning of a compound statement. Each function declaration actually consists of a name and parameter list, followed by a compound statement.

Note: stmt means 'statement', any one from the table above. exp means expression.

Constant Values

When assigning a value to a variable, with a line like 'iNum = 3;' the value is termed a *constant*, since it can not change. There is more than one way to write it, however,

iNum = 1406;	Without any prefixes, the number is in base 10. Decimal. Like wot we used ta, guv!
iNum = 0x57E;	'0x' or '0X' gives us hex.
iNum = 02576;	Prefixing with a single '0' gives us octal. Therefore, only numbers 0-7 may be used.
iNum = 'A';	Single quotes around *a single letter* produce the ASCII value of that letter. In this, 65. This constant can be assigned to non-char types.
fNum = 0.1f;	The presence if a decimal point and a *suffix* of 'f' indicates a floating point number. Assigning a floating point number to an integer variable will cause truncation (i.e. it ignores the fractional part). All floating point constants *must* include the decimal point, otherwise it will interpret '12f' (say) as an integer but report an error when it means the 'f' hex digit.
fNum = 0.1;	Without the 'f', the compile treats the constant as a double precision number. Although not a bad thing, handling double precision numbers is more costly in terms of processing and code optimisation.