C Programming for Operating Systems

Overview

Dr Alun Moon

Computing, Engineering and Information Sciences

2nd October 2012

Good Books

- Brian Kerngihan and Denis Ritche.
  The C Programming Language.
  Prentice Hall, second edition, 1988.
- Stephen Kochan.
  Programming in C.
  Samms, third edition, 2004.
- Steve Holmes.
  C programming.
  http://www2.its.strath.ac.uk/courses/c/
  C programming

Ineresting reads

Rob Pike.
Notes on programming in c, 1989.

L.W. Cannon et.al.
Indian hill c style and coding standards.

Henry Spencer.
How to steal code, or inventing the wheel only once.

Henry Spencer.
The Ten Commandments for C Programmers

C

Background

The C programming language has been associated with operating systems programming since its beginning [Kernighan and Ritchie]. The bond between C and Unix is very tight, C was invented to write Unix.
C is a powerfull tool well suited to embedded systems

Global Structure

The generally reccommended structure goes something like this :-

1. Include Header files
2. Declare constants
3. Declare global variables
4. Declare functions

```
#include <stdio.h>
const int life = 42;
int main(int argc, char *argv[] )
{
    printf("Hello world\n");
}
```

Header files and libraries

`#include <>`

- The include directive copies in the specified header file.
- This file contains
  - function declarations
  - constant definitions

  needed for using the related library.
- The project context will set up the approprate directories for local header files

Constants

ways of defining

defines often seen in older programs, provides a textual substitution for the constant

```
#define YEAR 365
#define PI 3.14159
```

emumerated constants provide a mechanism for creating sequential integer constants

```
enum { NULL, WAITING, RUNNING, STOPPED };
enum { MONDAY=1, TUESDAY, WEDNESDAY };
```

constant variables create a variable and mark it constant (read only)

```
const double e = 2.71828 ;
const char logo = '@';
```

Functions
prototypes

C needs functions and variables to be declared before their use. This is the role the header files provide, declaring functions from their libraries.

```
void *memcpy(void *dest, const void *src, size_t n);
```

A prototype tells the compiler that a function exists, what type and how many parameters it has and the return type.
The actual code for the function is elsewhere, later in the file, in another file, or already compiled in the libraries.
The IAR toolset may enforce the condition that *all* functions must have a prototype declared.

Functions
declare and define

A function is simply its prototype followed by the body of the function providing the code.

```
int weight(int x, int y)
{
    int z = x*x;
    return z/y ;
}
```

The `return` statement provides the returned value for the function.

language data structure elements

- arrays
- strings
- structures
- pointers

Arrays

```
float blocks[4];
int days[] = {31,28,31,30,31,30,31,31,30,30,31,31};
life[2] = 5;
```

- no bounds checking; the index can be any integer value.
- cannot determine the size of an array at runtime
- **Beware** running off the end of an array. . . *Here be dragons*

Arrays as function parameter

Arrays can be passed to functions

```
float data[100];
mean = sum(100, data)/100;
```

The array is passed by reference not value

```
float sum(int s, float d[])
{
    int  i;
    float t=0.0f;
    for(i=0 ; i<s ; i++)  t+=d[i];
    return t;
}
```

Strings
data structures

Strings in C are just arrays of character types, there is no string type.

```
char msg[] = "no space on device";
```

String operations (copying, comparison etc) are handled by functions from the standard C library (`string.h`);
Strings are stored as an array of their ASCII values, terminating in a zero (null) value

```
char hi[] = { 'h', 'e', 'l', 'l', 'o', '\0'};
```

Structures

Structures in C are a way of grouping data elements together, into a composite data type. Each field has a type and a name.

```
struct label {
    int x;
    int y;
    float size;
}
```

Once declared the structure can be used like any other data-type. Structures can be declared, initialised, passed to functions and returned as values from functions.

Given a variable declared as a structure

```
struct label q;
```

The fields can be accessed by their declared name.

```
q.x = q.y*2;
q.size = pi*sqrt(q.x*q.x + q.y*q.y);
```

Pointers

Pointers are the powerfull part of C,
and have the potential to be the most dangerous.

With great power, comes great resposibilty

Essentially a pointer holds the memory address of something.
They are most often found when passing large/complex parameters such as arrays and structures to functions.

basics

declaration uses the * operator.

```
int *np;
```

address of operator & is used to obtain the address of an object.

```
int number;
np = &number;
```

derefferencing the pointer retrieves the data at the pointers address

```
count = *np;
```

Arrays and pointers

Arrays and pointers have a very close relationship.

An array name is a pointer and a pointer can be used as an array!

Consider

```
int days[31];
int *p;

p = &days[0];
p = days;

p[3] = day[3];
```

Arrays as parameters

Pointers are the usual way to pass arrays as parameters to functions

```
char buffer[128];

strcpy("IO message 3\n\r", buffer);
```

```
int sequence(int n, int *d)
{
    int i;
    d[0] = d[1] = 1;
    for(i=2 ; i<n ; i++)
      d[i] = d[i-1] + d[i-2];
    return d[i-1];
}
```

A pointer can be modified

```
size_t msglen(char *b)
{
    char *a = b;
    while(*b) b++;
    return b-a;
}
```

An array name cannot have its address changed

```
char name[] = "some name or other";

name = foo ; /* NOT ALLOWED */
```

Structures and pointers

Pointers are handy for passing large structures as parameters. A pointer is more efficient.

```
struct boat {
    float loa;
    float beam;
    float draught;
}
struct boat dingy;

float displacement(struct boat *b)
{
    return b->loa * b->beam * b->draught ;
}
```

Pass by value and Pass by reference

```
int quadruple(int a)
{
    return a*4;
}
```

The *value* is coppied into the local variable. The original is unchanged.

```
void tripple(int *b)
{
    *b = (*b)*4;
}
```

when passing a pointer you can modify the original value outside the function;

Bit twiddling
Setting and clearing bits

A common task in Operating Systems (especially embedded ones) is the setting, clearing and testing of specific bits or groups of bits.

ASCII upper and lower case

In ASCII the difference between upper-case and lower-case letters is the value of bit 5 (counting from zero)

LEDs

A bank of LEDs may be attached to a port, each bit controlling a separate LED

logic identities

And

$A \land 0 = 0$	clears bit
$A \land 1 = A$	preserves bit

Or

$A \lor 0 = A$	preserves bit
$A \lor 1 = 1$	sets bit

Xor

$A \oplus 0 = A$	preserves bit
$A \oplus 1 = \neg A$	toggles bit

Each operator either preserves or changes a bit.

Masks

We need a mask, a value where each bit is set to

0 for each bit to be preserved

1 for each bit to be changed

The shift operators can simply be used here.

1<<0	gives 00000001_2
1<<2	gives 00000100_2
1<<5	gives 00100000_2

The mask can be used on an integer value using bitwise logical operations.

Masks

Set a bit

```
bit_fld |= (1 << n)
```

Clear a bit

```
bit_fld &= ~(1 << n)
```

Toggle a bit

```
bit_fld ^= (1 << n)
```

Test a bit

```
bit_fld & (1 << n)
```

Bit fields

A bit field is declared as a `signed int` or `unsigned int`. Following the member name by a colon and the number of bits it should occupy. Empty entries consisting of just a colon followed by a number of bits are also allowed; these indicate padding.

```
struct f
{
    unsigned int  flag : 1;
    signed int    num  : 4;
    : 3;
} g;
```

then

```
if (g.flag) g.num++ ;
```

Logic values

what to test for in `if`

In C 0 is false and non 0 is true.
So `while(n)` is a valid test.

Spot the potential for disaster

```
#define TRUE 1
#define FALSE 0
if ( isdigit(c)==TRUE )
```

On the first C compiler I used when you looked true had a value of -1

```
printf("true %d\n", (5<8) );
```

this reads better anyway

```
if(isdigit(c))
```