Resumen de C orientado al compilador CCS.

Estructura de un programa en C

Un programa en C debe contener como mínimo una función llamada ***main***, que es por donde se inicia siempre la ejecución. El principio y el final de la función se marca con unas llaves.

```
main(){
/* esto es un comentario */
printf("Hello world!");
}
```

Además puede tener otras funciones todas ellas definidas al mismo nivel (es decir, no existe el anidamiento de funciones).

```
main(){
printf("Hello world!");
otro_saludo();
}

otro_saludo(){
printf("Hola mundo!");
}
```

Estructura de una función

Cualquier función (incluida la función main) tiene la siguiente estructura:

tipo_devuelto nombre_función (nom_par1, nom_par2,...,nom_parN)

/*declaración del tipo de los parámetros */
tipo nom_par1;
tipo nom_par2;
. . .
tipo nom_parN;

{
/* declaración de las variables locales */

tipo variable1;
tipo variable2;
. . .
tipo variableK;

sentencias_del_cuerpo_de_la_función;
return valor;
}

- Todas las sentencias en C tienen que acabar con el carácter **;**
- **Declaración de parámetros y variables**
  - La declaración del tipo de cada parámetro tiene que hacerse antes de la llave de inicio de función.
  - La declaración de las variables locales se hace justo a continuación de la llave de inicio de bloque
  - Si nuestro programa necesitara usar variables globales, las tendríamos que declarar fuera de todas las funciones.
  - Las sentencias de declaración siempre son de la forma:
    tipo_dato nombre_var;

    Si tenemos mas de una variable/parámetro del mismo tipo, se pueden declarar en la misma sentencia, separando los nombres con el carácter **,**:
    tipo_dato nombre_var1,nombre_var2;

  - Las variables (locales o globales) se pueden inicializar en la misma sentencia de declaración:
    tipo_dato nombre_var=valor_inicial;

Algunos tipos de Datos en C (ANSI)

- Tipos de datos simples:
  - Carácter: **char**
    Enteros: **short**, **int**, **long**
    Coma flotante: **float**, **double**
  - Los tres tipos de enteros se diferencian en la cantidad de memoria que se usa para representar el dato. La especificación dice que el tamaño de un **short** debe ser menor o igual que el de un **int**, que a su vez debe ser menor o igual que un **long**.
  - Si antes del tipo de dato se pone la palabra clave **unsigned** el compilador supone que los valores de la variable serán siempre positivos.
  - La palabra clave **void** se utiliza para indicar *ningún tipo*, y a veces se utiliza como "tipo comodín". También sirve para indicar que una función no devuelve ningún valor.
  - En C no existe un tipo para los booleanos. El valor 0 se interpreta como falso, mientras que cualquier valor diferente de 0 se interpreta como cierto.

- Tipos de datos estructurados:
  - **tablas** (matrices, arrays)
    - Las variables de tipo matriz se declaran de la siguiente manera:

      tipo_elemento nombre_var[num_elementos];

    - El índice del primer elemento de una matriz es el **0**, y por lo tanto el último es **num_elementos-1**
    - Por ejemplo, la sentencia:
      int dni[8];
      Declara una matriz de 8 enteros
      Y las siguientes sentencias modifican el primer y el último

elemento de la variable:
dni[0]=4;
dni[7]=8;

- Es posible declarar tablas multidimensionales. Basta con añadir entre corchetes el número de elementos de cada dimensión. Por ejemplo, la siguiente sentencia declara una matriz de 8 x 10 enteros:
int matriz[8][10];

- **Estructuras**
  - Para definir una variable de tipo tupla se utiliza la palabra clave **struct** , a continuación se declara entre llaves el tipo de cada uno de los campos que se quiere poner en el patrón, y se termina con el nombre de la variable:
struct {
tipo1 campo1;
tipo2 campo2;
tipo3 campo3;
} nombre_var;

Por ejemplo:
struct {
char nombre[30];
int dni[8];
int edad;
} ficha_alumno_1;

  - También es posible definir patrones de estructuras que luego se pueden usar en la declaración de cualquier variable. La definición de patrones se tiene que hacer fuera de todas las funciones:
struct nombre_patron{
tipo1 campo1;
tipo2 campo2;
tipo3 campo3;
};

Y luego, al declarar las variables se puede usar el nombre del patrón:
struct nombre_patron nombre_var1, nombre_var2;

  - Para acceder a cada campo de una variable de tipo tupla se usa el carácter **.**: *ficha_alumno_1.edad=18;*

- **strings**
El tipo *string* no existe como tal. Un string es un array de carácteres en el que se marca el final con el carácter nulo (código ASCII 0: '\0').
El lenguaje C ofrece muchas funciones de manipulación de strings, pero para usarlas correctamente debemos estar seguros de que los arrays que se le pasan tienen la marca de fin (si no es asi, el comportamiento de las funciones es indefinido).

- **Definición de nuevos tipos**
  El programador puede definir nuevos tipos mediante la sentencia *typedef*. Esta sentencia se tiene que situar fuera del ámbito de todas las funciones:

  typedef nuevo_tipo definicion;
  typedef boolean short;

Constantes en C

- Las constantes de tipo carácter en C se representan encerradas entre comillas simples (''). Por ejemplo, para inicializar una variable de tipo carácter con el valor de la letra *A*:
  char variable='A';

- Las constantes de tipo string en C se representan encerradas entre comillas dobles (""):
  printf("Hello world");
  Hay que tener presente que al poner una tira de caracteres entre comillas dobles el compilador añade al final el carácter nulo.

- Se pueden definir constantes en C mediante la directiva del preprocesador *#define*:
  #define NOMBRE_CTE valor
  #define MAX_VALOR 200
- **Enum**

  Permite definir una secuencia de constantes de forma consecutiva.

enum ADC_SOURCE {an0, an1, an2, an3, bandgap, srefh, srefl, itemp, refa, refb};

Tipos de variables en CCS:

unsigned define un número de 8 bits sin signo
unsigned int define un número de 8 bits sin signo
int define un número de 8 bits sin signo
char define un número de 8 bits sin signo
long define un número de 16 bits sin signo
long int define un número de 16 bits sin signo
signed define un número de 8 bits con signo
signed int define un número de 8 bits con signo
signed long define un número de 16 bits con signo
float define un número de 32 bits en punto flotante
short define un bit
short int define un bit

Es decir en este compilador sí existe una variable de tipo bit.

u8 define un número de 8 bits sin signo
u16 define un número de 16 bits sin signo

#byte directiva para hacer referencia a una posición interna de memoria en particular

Si se añade el prefijo **const** quedará la variable en la ROM (no ocupará RAM).
No están permitidos punteros a constantes. **short** es un tipo especial utilizado para generar código muy eficiente para las operaciones de I/O. No se permiten las matrices de **short** ni los punteros a **short**.

Ej:

```
  tabla char[8];
  int x,y;
  signed int j=3;
  int mat[4][4];
  const display char[4]= {1,2,3,4};
  #byte  status = 3                              // las directivas no acaban en ;
  #byte  port_b = 6
…
  x=mat[0][1];
```

Algunas particularidades de las estructuras en CCS:

```
struct lcd_pin_map {
  boolean enable;
  boolean rs;
  boolean rw;
  boolean unused;
  int     data : 4;
 } lcd;
```

Si ponemos expresión_constante después de un identificador, determina el número de bits que utilizará dicho identificador. Este número puede ser 1,2, ...8.

Ej:

```
#byte  status = 3
#byte  port_b = 6
struct  {
  short int r_w;
  short int c_d;
  int no_usado : 2;
  int dato    : 4; }  port_a;
#byte port_a = 5
...
port_a.c_d = 1;
```

Operadores en C

- **Aritméticos**:
  - suma: i = i + 1;
  - resta: i = i - 1;
  - multiplicación: i = i * 1;
  - división: i = i / 1;
  - resto: i = i % 1;
  - autoincremento: i++; ó ++i;
  - autodecremento: i--; ó --i;
- **Asignación**:
  - Asignar a una variable un valor: a=b+c;
  - Asignar a una variable el resultado de operar esa variable con un valor: a+=4; a-=4; a*=4; a/=4; a%=4;
- **Relacionales**:
  - mayor que/mayor o igual que: i>4; i>=4;
  - menor que/menor o igual que: i<4; i<=4;
  - igual que/diferente que: i==4; i!=4;
- **Lógicos**:
  - and: a && b;
  - or: a || b;
- **Manipulación de bits**:
  - and bit a bit: a & b;
  - or bit a bit: a | b;
  - or exclusivo bit a bit: a ^ b;

Control de ejecución

Cuando el cuerpo de un bucle o de un condicional está formado por mas de una sentencia es necesario marcar el principio y el final del cuerpo con las llaves. Si sólo está formado por una sentencia las llaves son opcionales.

- **Bucles**

bucle **while**:

```
while
(expresion) {
sentencias;
}
```

bucle **do-while**:

```
do {
sentencias;
}while
(expresion);
```

bucle **for**:

```
for (inicializaciones; expresion;
actualizaciones) {
sentencias;
}
```

  - Comentarios sobre el **for** :
    - **inicializaciones**: estas sentencias se ejecutan antes de entrar por primera vez en el bucle.
    - **expresion**: esta es la condición que se debe cumplir para entrar en el bucle. Se comprueba antes de iniciar una nueva iteración.

        - **actualizaciones**: estas sentencias se ejecutan al final de cada iteración, antes de comprobar si se debe entrar de nuevo en el bucle.

- **Condicionales**

if, caso 1	**if, caso 2**	**if, caso 3**	***elección múltiple***
if (expresion) *{* *sentencias;* *}*	*if (expresion)* *{* *sentencias;* *}* *else {* *sentencias;* *}*	*if (expresion) {* *sentencias;* *}* *else if (expresion)* *{* *sentencias;* *}*	*switch (expresion){* *case etiqueta1:* *sentencias;* *break;* *case etiqueta2:* *sentencias;* *break;* *case etiqueta3:* *sentencias;* *break;* *default: sentencias;* *}*

    - Comentarios sobre el **switch**:
        - La etiqueta *default* es opcional, y se utiliza cuando se quiere hacer un tratamiento en caso de que la expresión no coincida con ninguna de las etiquetas anteriores.
        - La sentencia *break* se utiliza para forzar la salida del switch una vez se han ejecutado las sentencias del caso correspondiente. Si no se pone, la ejecución continuará por las sentencias del siguiente caso (hasta que se encuentre un break).

Punteros

- Un puntero es una variable que almacena una dirección de memoria. Los punteros se utilizan en C, entre otras cosas, para implementar el paso de parámetros por referencia, para gestionar la reserva dinámica de memoria, y para manipular matrices.
- Para declarar una variable de tipo puntero se debe especificar de que tipo es el dato que se almacenará en la dirección de memoria que contendrá el puntero:

  *int *p_int;*

  Esta sentencia declara una variable de nombre *p_int* (por lo tanto, tendrá una posición de memoria asignada), que va a contener una dirección de memoria, en la que se almacenará un entero.

- Si tenemos una variable de tipo puntero, es posible acceder a su propio valor (como en el caso de cualquier otra variable):
  p_int=&i;

Esta sentencia utiliza el operador *&* para obtener la dirección de memoria en la que se guarda la variable *i* y se la asigna a la variable *p_int*.

- Pero también es posible acceder mediante una *indirección* a la posición de memoria que indica su valor.
  **p_int=4;*

  Esta sentencia obtiene el valor que guarda *p_int*, y accede a la posición de memoria que indica ese valor (es decir, accede a la posición de memoria en la que se guarda la variable *i*) para dejar un 4.

- Antes de utilizar el operador de *indirección* sobre un puntero, hay que asegurarse de que está inicializado de forma correcta. Esto es, el valor que contiene se corresponde con una dirección de memoria válida (que se ha reservado estática o dinámicamente), y además es la dirección a la que interesa acceder.

- La manera de inicializar correctamente un puntero es asignándole la dirección de memoria de otra variable, o asignándole el resultado de alguna de las funciones de reserva de memoria dinámica (*malloc*, *calloc*,...). Estas funciones reciben como parámetro la cantidad de memoria que interesa reservar, marcan como válida esa cantidad y devuelven la dirección de memoria base de la nueva región.

Uso de punteros para paso de parámetros por referencia

- En C el mecanismo de paso de parámetros es por valor. Es decir, se hace una copia de la variable que se pasa por parámetro, y el código de la función trabaja con esa copia. Por lo tanto, cualquier modificación que la función haga sobre el parámetro se pierde una vez la función acabe.
  Si interesa que esas modificaciones se conserven una vez que se sale de la función, necesitamos una manera para que el código de la función acceda a la posición de memoria en la que tenemos la variable que se pasa como parámetro.

- Esto se implementa declarando el parámetro como un puntero que apunte a la variable que nos interesa, y usando dentro de la función el operador de indirección para acceder a ese dato.

```
int funcion(param1,param2)
int *param1;
char *param2;
{
int i=*param1+10;
. . .
*param1=4;
*param2='a';
. . .
}
```

Esta función recibe dos parámetros que quiere modificar, por lo tanto en lugar de

pasarle los datos, se le tiene que pasar la dirección de memoria donde se guardan esos datos. Y para acceder a ellos utiliza el operador de indirección *.

- Por su parte, en la llamada a la función hay que pasar la dirección de memoria de los datos, por ejemplo usando el operador de dirección &.

```
main(){
int i=17;
char c='b';
. . .
funcion(&i,&c);
. . .
}
```

Uso de punteros en la gestión de memoria dinámica

Siempre que sea necesario utilizar memoria dinámica para almacenar una estructura de datos, se puede declarar la variable como un apuntador al tipo de la estructura, y retrasar la inicialización hasta que se reserva la memoria. Cuando es posible hacer esta reserva, se asigna a la variable que se había declarado la dirección de memoria reservada, para que sea posible el acceso.
Por ejemplo, si se quiere implementar una lista de tamaño variable, cada vez que haya que añadir un nodo, habrá que reservar memoria para el nuevo nodo antes de inicializarlo. Suponiendo que habeis definido el tipo nodo_t:

```
nodo_t *nodo_aux;
. . .
nodo_aux=malloc(tamanyo_nodo);
inicializar_nodo(nodo_aux);
insertar_nodo(lista, nodo_aux);
. . .
```

Uso de punteros en la manipulación de matrices

- El nombre de una variable de tipo matriz es también un puntero al primer elemento de la matriz. Pasar una matriz como parámetro consiste en pasar la dirección del primer elemento (sino, habría que copiar todos los elementos, y para matrices grandes sería un desperdicio de tiempo y de memoria).
Por lo tanto, una función que tiene una matriz como parámetro puede ser de cualquiera de las dos maneras siguientes:

```
funcion(param)          funcion(param)
char param[];           char *param;
{                       {
. . .                   . . .
param[0]=4;             param[0]=4;
```

```
. . .                    . . .
}                        }
```

Utilizando punteros para acceder a la parte más o menos significativa de una variable utilizando typecast.

```
#define hii(x)  ( * ((unsigned int *)(&x)+1))
#define loo(x)  ( * ((unsigned int *)(&x)))
```

Uso de ficheros cabecera

- Los *ficheros cabecera* se utilizan para agrupar en ellos definiciones que son necesarias desde varios ficheros fuente. De esta manera, si se modifica una de estas definiciones no es necesario recorrer todos los ficheros fuente, sino que es suficiente con modificarla en el fichero cabecera.
El nombre de este tipo de ficheros suele terminar con el sufijo **.h**
- La manera de hacer visibles las definiciones del fichero desde todos los fuente donde se necesitan, es mediante la directiva de preprocesador **include**

#include <nombre.h>
#include "nombre.h"

Cuando el preprocesador se encuentra con esta directiva, la substituye por el contenido del fichero que se le indica (*nombre.h*). De manera, que el efecto es el mismo que si se hubiera escrito directamente en ese punto el contenido del fichero.

- Si el nombre del fichero se escribe entre comillas dobles, el preprocesador lo busca en el directorio actual de trabajo, y si no lo encuentra da error. Si por el contrario, el nombre se escribe entre paréntesis angulares, el preprocesador lo busca en los directorios estándar de *includes*, si no lo encuentra lo busca en el directorio actual de trabajo, y si tampoco lo encuentra allí da error.

Paso de parámetros al main

Es posible pasarle parámetros al programa desde la línea de comandos. Para ello se considera que la función main tiene dos parámetros llamados **argc** y **argv**.

```
main (argc,argv)
int argc;
char *argv[];
{
. . .
}
```

argc es un entero que contendrá el número de parámetros que ha recibido el programa.

Y **argv** es un array de strings, que contiene en cada posición uno de esos parámetros. El nombre del ejecutable también se considera un parámetro, así que argv[0] contendrá siempre ese nombre.

Compilación de un programa sencillo escrito en C en un entorno Unix

La manera más simple de compilar un programa escrito en C, es mediante el comando cc: *cc fichero_fuente*
Este comando generará (si no hay ningún error de compilación en el fichero fuente) un ejecutable al que llamará *a.out*. Si queremos que el ejecutable tenga un nombre diferente podemos utilizar la opción -o: *cc -o nombre_ejecutable fichero_fuente*
La documentación de soporte a la práctica 2 contiene más información sobre el proceso de generación de ejecutables.

Algunas funciones de la librería de C

- **Funciones para la manipulación de strings.**
  Recordad que un string es un array de caracteres que tiene como marca de final el carácter '\0'. Si utilzáis un array de caracteres que no tenga este carácter, el comportamiento de estas funciones será erróneo y aleatorio.
  También debéis tener presente que "a" indica constante de tipo string, mientras que 'a' indica constante de tipo carácter. Así, para almacenar la constante "a" se necesitaría como mínimo un array de 2 posiciones (para 'a' y para '\0').
  - size_t strlen (char *s1): devuelve el tamaño del string s1 (no cuenta el '\0').
  - char *strncpy (s, ct, n): copia hasta n caracteres de la cadena ct a s; devue lve s. Rellena con '\0' si ct tiene menos de n caracteres.
  - char *strcat (s, ct): concatena la cadena ct al final de la cadena s; devuelv e s.
  - char *strncat (s, ct, n): concatena hasta n caracteres de la cadena ct a la c adena s, terminando con '\0'; devuelve s.
  - int strcmp (cs, ct): compara la cadena cs con ct; devuelve <0 si cs0 si cs>ct.
  - int strncmp (cs, ct, n): compara hasta n caracteres de la cadena cs con la c adena ct; devuelve <0 si cs0 si cs>ct
  - char *strchr (cs, c) : devuelve un puntero a la primera aparición de c en cs o NULL si no está presente.
  - char *strrchr (cs, c): devuelve un puntero a la última aparició n de c en cs, NULL si no está presente.
- **Otras funciones útiles de C:**
  - int sprintf(char *s, const char *format, ... /* [args] */): deja en el string s el resultado de combinar con el formato indicado por format el resto de argumentos
  - char *strerror (int errnum): devuelve el string representativo del tipo de error indicado por errnum.
  - int atoi (const char *s): convierte el string s en entero.
  - double atof (const char *s): convierte el string s en double.
  - void *malloc(size_t size): valida una zona de memoria de tamaño size, y devuelve su direcció, o NULL si no hay memoria disponible.
  - void free (void *p): libera la zona de memoria a la que apuntaba p

Otras funciones útiles en CCS

…. Consultar manual compilador C CCS

Algunos errores comunes en la programación en C

- El primer elemento de un array en C tiene como índice 0. Por lo tanto, las posiciones válidas de un array de 10 elementos serán de la 0 a la 9. Si intentamos acceder a la posición 10, nos estaremos saliendo de rango, y estaremos accediendo a lo que en esa dirección de memoria haya. El compilador **no** avisa de este error. Será en tiempo de ejecución cuando se produzca el fallo, que dependerá del estado de la memoria en ese momento (esto es: el tipo de error es aleatorio).
- Es importante, distinguir entre los operadores = e == y no confundirlos.
- Al compilar un programa y encontrar errores, es recomendable corregirlos de uno en uno en el orden en que se producen, pues la corrección de un error puede hacer que desaparezcan unos cuantos o que aparezcan otros que antes no se daban .
- Una variable de tipo puntero contiene una dirección de memoria. Antes de acceder a esa dirección tenemos que asegurarnos de que es válida (es una zona de memoria que hemos reservado, ya sea porque contiene una variable que hemos declarado, o porque es una zona que hemos reservado con una función de reserva de memoria dinámica).
- Si a continuación de la sentencia de control de un bucle o de la cabecera de una función se pone un ';', el compilador interpreta que ese bucle o esa función tienen una única sentencia (y por eso no hay llaves) y que esa sentencia es la sentencia nula. Por ejemplo:

  while (i< LIM);

  Es un bucle infinito, por mucho que después del ';' se añada lo que lo que se pretende que fuera el cuerpo del bucle.

- Un programa con una indentación clara nos ayudará a comprenderlo y a no desesperarnos cuando olvidemos una llave de cierre de algún a sentencia de control del flujo de ejecución.
- if (y=3) ….. → If (y==3) …..
- if ((y==3) & (j=7)) ….. → If ((y==3) & (j=7)) …..
- Asignación de matrices

  char entrada[4],salida[4];

  salida=entrada;

- Asignación de texto a cadenas

```
char texto1[20] = "Primer texto";
char texto2[20];
```

texto2=”Segundo texto”; → strcpy(texto2,”Segundo texto”);

Otras directivas del compilador:

#ASM y #ENDASM

```
int paridad (int dato)  {
int contador;
#asm
  movlw  8
  movwf  contador
  movlw  0
lazo:  xorwf  dato,w
  rrf    dato,f
  decfsz contador,f
  goto   lazo
  movwf  _return_
#endasm                                    }
```

#SEPARATE le dice al compilador que el procedimiento o función que sigue a la directiva será llevado a cabo por SEPARADO. Esto es útil para evitar que el compilador haga automáticamente un procedimiento en línea (INLINE). Esto ahorra memoria ROM pero usa más espacio de la pila. El compilador hará todos los procedimientos #SEPARATE, separados, tal como se solicita, aun cuando no haya bastante pila.
Ejemplo:

```
#separate
swap_byte (int *a, int *b) {
int t;
 t=*a;
 *a=*b;
 *b=t;
 }
```

Definiendo rutinas de interrupción en CCS:

Estas directivas especifican que la función que le sigue es una función de interrupción. Las funciones de interrupción no pueden tener ningún parámetro. Como es natural, no todas las directivas pueden usarse con todos los dispositivos. Las directivas de este tipo que disponemos son:
#INT_EXT INTERRUPCIÓN EXTERNA

#INT_RTCC DESBORDAMIENTO DEL TIMER0(RTCC)
#INT_RB CAMBIO EN UNO DE LOS PINES B4,B5,B6,B7
#INT_AD CONVERSOR A/D
#INT_EEPROM ESCRITURA EN LA EEPROM COMPLETADA
#INT_TIMER1 DESBORDAMIENTO DEL TIMER1
#INT_TIMER2 DESBORDAMIENTO DEL TIMER2
#INT_CP1 MODO CAPTURA DE DATOS POR CCP1
#INT_CCP2 MODO CAPTURA DE DATOS POR CCP2
#INT_SSP PUERTO DE SERIE INTELIGENTE(SPI, I2C)
#INT_PSP PUERTO PARALELO
#INT_TBE SCI DATO SERIE TRANSMITIDO
#INT_RDA SCI DATO SERIE RECIBIDO
#INT_COMP COMPARADOR DE INTERRUPCIONES
#INT_ADOF DESBORDAMIENTO DEL A/DC DEL PIC 14000
#INT_RC CAMBIO EN UN PIN Cx
#INT_I2C I2C DEL 14000
#INT_BUTTON PULSADOR DEL 14000
#INT_LCD LCD 92x

El compilador salta a la función de interrupción cuando se detecta una interrupción. Es el propio compilador el encargado de generar el código para guardar y restaurar el estado del procesador. También es el compilador quien borrará la interrupción (el flag). Sin embargo, nuestro programa es el encargado de llamar a la función ENABLE_INTERRUPT() para activar previamente la interrupción junto con el señalizador (flag) global de interrupciones.
Ej:

```
#int_ad
control_adc() {
  adc_activo=FALSO;
}
```

Compilación condicional:
Permite crear código que se compila sólo si está definida la etiqueta.

Ej:

```
#define FSS

#ifdef FSS

#endif
```