Co-funded by the
Erasmus+ Programme
of the European Union

Sistemas Embebidos

Lenguaje C

Introducción

- Creado en los 70s por Dennis Ritchie.
- Lenguaje de bajo nivel que permite el control completo de los recursos de hardware.
- El código se escribe en dos tipos de archivos:
  - Header files: archivo de declaración de variables y funciones, con extensión .h, que luego se incluye en los source files.
  - Source files: archivo de definición de variables y funciones (la implementación), con extensión .c.

Variables enteras

Tipo	Tamaño	Rango
char	1 byte	-128 to 127
unsigned char	1 byte	0 to 255
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
int	2 o 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 o 4 bytes	0 to 65,535 or 0 to 4,294,967,295
long	4 o 8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	4 o 8 bytes	0 to 18446744073709551615

La función sizeof() permite determinar el tamaño en bytes de un tipo o variable.

Variables enteras - stdint.h

- Para eliminar ambigüedades y hacer el código más claro y fácil de portar, stdint.h define varios tipos de enteros con signo y tamaño claramente definidos.
  - int8_t
  - int16_t
  - int32_t
  - uint8_t
  - uint16_t
  - uint32_t

Variables flotantes

Tipo	Tamaño	Rango	Precisión
float	4 bytes	1.2E-38 to 3.4E+38	6 decimales
double	8 bytes	2.3E-308 to 1.7E+308	15 decimales
long double	10 bytes	3.4E-4932 to 1.1E+4932	19 decimales

Void

void es un tipo de retorno nulo que se puede usar en tres situaciones:

- Para indicar que una función no retorna ningún valor, se define de tipo void.
- Para explicitar que una función no toma ningún parámetro (no es necesario, y por defecto ya es así).
- Para definir un puntero de tipo void (void*) que indica que contiene una dirección pero no especifica el tamaño del destino al que apunta.

Literales

Los valores para distintos tipos de datos se llaman *literales* y pueden ser enteros, racionales, caracteres, cadenas de caracteres y enumeraciones.

1. Enteros
    - Decimal - 123
    - Hexadecimal - 0x123 (empieza con 0x)
    - Octal - 0123 (empieza con 0)
    - unsigned - 12345U (sufijo U o u)
    - long - 12345L (sufijo L o l)
    - unsigned long - 12345UL

Literales

2. Racionales
    - Punto flotante - Ej: 15.2
    - Mantisa y exponente - Ej: 45.2e-3 (notación científica)

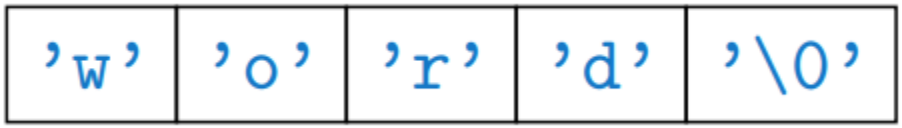
2. Caracteres
    - Los caracteres simples se representan con entre comillas simples - Ej ‘a’
    - El compilador convierte el caracter en el entero (int) correspondiente al código ASCII de ese caracter.

Literales

4. Cadenas de caracteres (strings)

   Una secuencia de caracteres entre comillas dobles. La cadena se almacena como un array de caracteres con terminación nula (null o '\0').

Por ejemplo "word" se almacena así:



Literales

5. Enumeración

   Permite asociar un nombre a un valor. Son siempre de tipo int, y por defecto comienzan en cero, pero se puede explicitar otro valor.

```
enum
{
    SPADES,
    CLUBS,
    HEARTS,
    DIAMONDS
};
```

```
enum
{
    SPADES = 10,
    CLUBS,  // valor es 11
    HEARTS = 13,
    DIAMONDS = 15
};
```

Funciones

- Las funciones se declaran y definen.
  - La declaración (o header o prototipo) de una función proporciona información acerca de cómo se debe llamar a la función, y es de la forma:

    ```
    type function_name(arguments);
    ```

    Permite modularizar el código y llamar una función que se encuentra en otro archivo.
  - La definición (o implementación) de la función contiene el código en sí de la función.

Funciones

- Ejemplo:

```
// Declaración
int suma(int a, int b);

// Definición
int suma(int a, int b)
{
    return a + b;
}
```

Estructura básica

Todo programa debe contener una función main(), que es el punto de entrada.

La función main en un sistema embebido suele ser de tipo void y no tener argumentos:

```
void main()
{
    printf("Hello World\n");
}
```

Programa de ejemplo

```c
int calculo(int a); /* function header/prototype */

void main()         /* main function */
{
    int v = 10;     /* variable declaration */
    int r;
    r = compute(v); /* function call */
}

int calculo(int a)  /* definition of the function */
{
    int b = 10 + a; /* function body */
    return b;       /* function return value */
}
```

Variables - Scope

El scope define la visibilidad y vida de las variables. A grandes rasgos, las variables pueden ser auto (de automatic), globales y/o estáticas.

auto - Cualquier variable local (dentro de una función) es por defecto automática.

- Si no se inicializa, su valor es aleatorio.
- “Vive” sólo durante la ejecución del bloque en el que está definida.
- Es visible sólo dentro del bloque en el que está definida.

Variables - Scope

global - Si la variables se define fuera de cualquier función (generalmente encima de la función main), se dice que la variable es global

- Si no se inicializa, su valor por defecto es 0.
- “Vive” durante todo el programa en una dirección fija de memoria.
- Es visible en todo el archivo .c en el que es definida.

static - Una variable local se puede definir con el prefijo static. En ese caso, tiene la visibilidad de una variable local (solo dentro del bloque) pero vive todo el programa (como si fuera global).

Variables - Scope

```
int varGlobal = 10;          // varGlobal vive todo el programa

void test()
{
    static varStatic = 1;    // varStatic vive todo el programa pero solo es visible dentro de test()
    int i=1, j=2;            // Se declaran e inicializan i y j
    ...
    {
        float a=7, j=3;      // Se declara una nueva variable j
        j = j + a;           // Aquí j es float
        ...                  // La variable int j no es visible aquí
        ...                  // La variable i si
    }

    j = 3;                   // Se modifica j int
    ...                      // Las variables float a y j ya no existen
}
```

Expresiones

- Las expresiones son los bloques que permiten realizar un operación o cálculo en base a entradas.
- Una expresión se compone de operandos y operadores.
- Puede estar compuesta por:
  - Literales
  - Variables
  - Operadores unarios y binarios
  - Llamados a funciones
- Ejemplos:

```
10 * x + 1
funcion(x)
(a && b == 1)
```

Operadores

Nivel	Operadores	Descripción
1	`() [] -> .`	Acceso a un elemento de un vector y paréntesis

2	`+ - ! ~`
	`* &`
	`++ --`
	`(cast) sizeof`
		Signo (unario), negación lógica, negación bit a bit
		Acceso a un elemento (unarios): puntero y dirección
		Incremento y decremento (pre y post)
		Conversión de tipo (*casting*) y tamaño de un elemento

3	`* / %`	Producto, división, módulo (resto)
4	`+ -`	Suma y resta
5	`>> <<`	Desplazamientos
6	`< <= >= >`	Comparaciones de superioridad e inferioridad
7	`== !=`	Comparaciones de igualdad
8	`&`	Y (*And*) bit a bit (binario)
9	`^`	O-exclusivo (*Exclusive-Or*) (binario)
10	`|`	O (*Or*) bit a bit (binario)
11	`&&`	Y (*And*) lógico
12	`||`	O (*Or*) lógico
13	`?:`	Condicional
14	`= *= /= %= += -= >>= <<= &= ^= |=`	Asignaciones
15	`,`	Coma

Control de flujo - if

```
if (expression1)
    expression1_true
else if (expression2)       // Opcional
    expression2_true
else                        // Opcional
    expression_else

// Si se necesita ejecutar más de una expresión, se usan corchetes
```

Control de flujo - switch

```
switch (condicion)
{
    case opcion_a:
        expression_if_a;
        break;

    case opcion_b:
        expression_if_b;
        break;

    default:                    // Opcional
        expression_else;
}
```

Control de flujo - goto

```
void funcion()
{
    if (condicion)
        goto final;

    // Código

    if (condicion2)
        goto final;

    // Código

final:
    // Código para terminar bien la función
}
```

Solo usar en este tipo de casos. Otros usos derivan en spaghetti code.

Bucles - while, do/while

```
while (expresion)
{
    // Código
}
```

La expresión se evalúa al principio.

Puede no ejecutarse nunca.

```
do
{
    // Código
}
while (expresion)
```

La expresión se evalúa al final.

Se ejecuta siempre al menos una vez.

Bucles - for

```
for (expresion_inicial; condicion; actualizacion)
{
    // Código
}
```

La ejecución se da de la siguiente forma:

1. Se ejecuta expresión_inicial
2. Se evalua condición.
    a. Si condición es true, sigue al siguiente paso
    b. Si condición es false, se termina el for
3. Se ejecuta el código dentro del for
4. Se ejecuta la actualización
5. Se vuelve al paso 2.

Variables - Arrays

Se pueden definir arrays de cualquier tipo de dato especificando el tamaño del array entre [].

Por ejemplo:

```
int a[10];  // Definición de un array de 10 enteros

a[0] = 1;   // Asigna el valor 1 al primer lugar del array
```

Variables - Dirección en memoria

```
                    (0x6D0B)
uint16_t var = 27915;
```


```
La dirección de var es
      0xA0001570
```

Punteros

& indica la dirección de una variable
* de una dirección devuelve el contenido

`uint16_t a = 27915;`		
a	La variable	Vale 27615
&a	La dirección donde está almacenada	Vale 0xA0001570

`uint16_t* p = &a;`		
p	Variable que “apunta” a la dirección de a	Vale 0xA0001570
*p	Valor que hay en la dirección a la que apunta	Vale 27615
&p	Dirección de un puntero (“puntero doble”)	Alguna dirección

Punteros

Ejemplo:

```
uint8_t a, b;
uint8_t *p0, *p1;

p0 = &a;    // La dirección de a es asignada a p0
*p0 = 1;    // Equivalente a “a = 1”
p1 = &b;    // La dirección de b es asignada a p1
*p1 = 2;    // Equivalente a “b = 2”
p0 = p1;    // p1 pasa a apuntar también a b
*p0 = 0;    // Equivalente a “b = 0”
```

Punteros a funciones

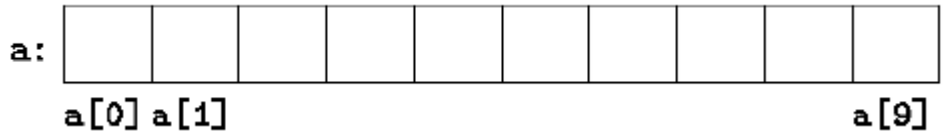
```
// Definición de un puntero llamado pfunc
// que va a apuntar a una funcion de tipo void
// que toma dos parámetros int8_t
void (*pfunc)(int8_t *, int8_t *);

// pfunc apunta a la función permutar
*pfunc = permutar;

// llamada a la función permutar a través del puntero pfunc
(*pfunc)(a,b);
```

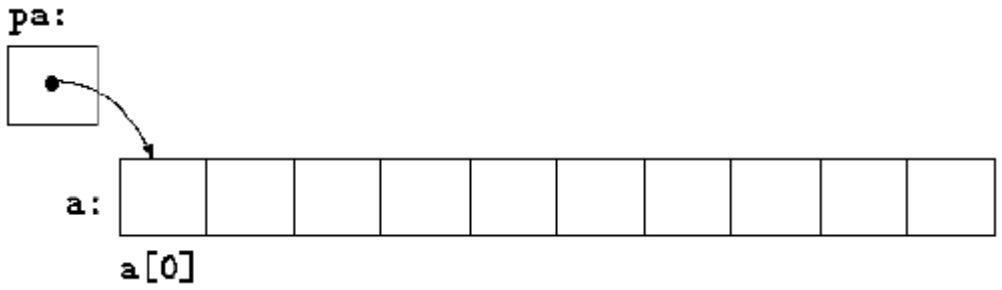
Punteros y arrays

```
char a[10], *pa;
```



Si:

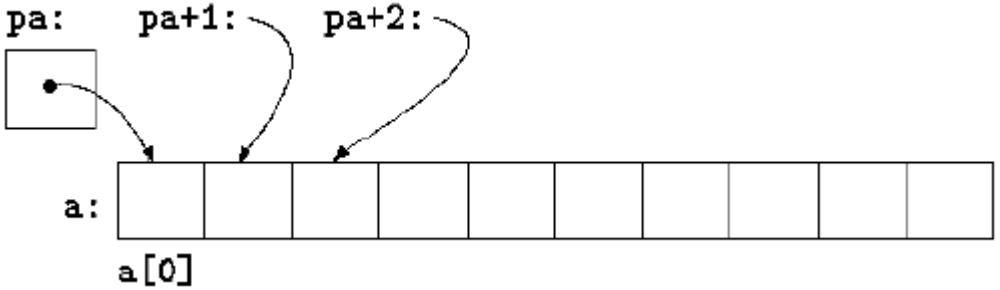
```
pa = a;
```



Entonces:

**pa es el valor de a[0]*

**(pa+1) es el valor de a[1]*



Strings

C no tiene un tipo de dato string.

En cambio, se usan arrays de char terminados en '\0'

```
char str[] = hola;
```

Como son arrays, y no un tipo de dato, no se puede operar directamente con strings.

```
char str1[] = "hola";
char str2[10];

str2 = str1;  // MAL
```

Como sucede con cualquier array, el nombre es solo un puntero al primer lugar del array.

Strings

Una forma de copiar str1 en str2 es, por ejemplo:

```
void strcpy(char* dest, char* src)
{
    while (*src != '\0')
    {
        *dest = *src;
        dest++;
        src++;
    }

    *dest = '\0';
}
```

Strings

Existe una biblioteca estándar de C, llamada string.h que resuelve la mayoría de las operaciones con strings por medio de funciones.

Ejemplos:

- strlen - devuelve el largo de un string
- strcpy - copia un string en otro
- strcmp - compara dos strings
- strcat - concatena dos strings
- strchr - busca un caracter dentro de un string

typedef

```
typedef char bool;

bool isActive;          char isActive;
```

~

```
typedef unsigned long time_t;

time_t timestamp;          unsigned long timestamp;
```

~

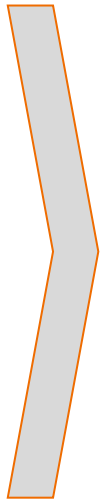
~

```
uint32_t timestamp;
```

Estructuras

Las estructuras (struct) permiten agrupar varias variables en un mismo grupo. Cada variable de la estructura se conoce como miembro de la estructura.

```
struct student
{
    uint8_t name[31];          //31 bytes
    uint8_t address[21];       //21 bytes
    uint32_t student_num;      //4 bytes
    uint32_t phone;            //4 bytes
    uint16_t scores[10];       //20 bytes
};
```



80 bytes

Estructuras

Para acceder a un miembro de una estructura, se usa el operador punto (.)

```
// Acceder al miembro phone de student1
student1.phone = 903456;

// Copiar "Pedro" en el miembro name de student2
strcpy(student2.name, "Pedro");

// Defino un array de students llamado classroom
struct student classroom[34];

// Accedo al miembro student_num de classroom[4]
classroom[4].student_num = 1234;
```

Estructuras y punteros

Se pueden definir puntero a estructuras, y acceder a los miembros de la estructura apuntada con el operador ->

```
// Defino una variable de tipo puntero a student llamada p
struct student *p;

// Hago que p apunte a student1
p = &student1;

// Las dos sentencias abajo son formas equivalentes de acceder
// a el miembro phone de la estructura a la que apunta p
p->phone;
(*p).phone;
```

Librerías estandar

	Define
stdbool.h	el tipo bool y las constantes true y false
stdint.h	los tipos enteros estándar (uint8_t, int16_t, etc.)
stdio.h	las funciones printf, sprintf, scanf, sscanf (entre otras)
string.h	funciones para manejo de arrays y strings
math.h	varias funciones matemáticas (cos, exp, log10, etc.)