Conceptos básicos del lenguaje C

Seminario de Sistemas Embebidos - FIUBA

Alan Kharsansky (akharsa@gmail.com)

Octubre - 2011

Índice I

Índice II

Introducción

El objetivo de esta clase es repasar los conceptos claves del lenguaje C que utilizaremos a lo largo del Seminario de Sistemas Embebidos. Para una referencia completa del lenguaje se recomienda consultar:

- **The C Programming Language:** Kernighan; Dennis M. Ritchie (March 1988)

En esta presentación solo se utilizarán aquellas funciones y estructuras del lenguaje que se utilizan con más frecuencia para la programación de microcontroladores. No es una definición completa del lenguaje desde ningún punto de vista.

Hello, World!

La estructura básica de un programa en C se puede ver en el siguiente ejemplo:

```c
#include <stdio.h>

int main() {
  // Este es un comentario de una linea

  printf("hello, world\n");

  return 0;

  /* Este es un comentario
  multilinea */
}
```

Hello, World!

Donde lo primero que nos encontramos es una directiva del pre-procesador: el Include.

```
#include <stdio.h>
```

Esta directiva, insertará de manera exacta el contenido del archivo stdio.h al comienzo de nuestro programa. Notar la diferencia de utilizar:

- **Paréntesis angulares** <>
- **Comillas dobles ""**

Hello, World!

Luego nos encontramos con la implementación de una función. En este caso el main que debe existir en toda aplicación escrita en lenguaje C. Una función se declara de la siguiente manera:

```
[tipo de dato de retorno] nombre([param1],[param2],...){
  // Implementacion
}
```

Que en este caso es:

```
int main(void){
  // Implementacion
}
```

Hello, World!

Y por último nos encontramos con una llamada a una función a la que se le pasa un parámetro, comentarios, y la sentencia return. Algunas cosas importantes a tener en cuenta:

- Los bloques de código deben ir entre llaves { }
- Las sentencias deben terminarse en punto y coma ;
- Unicidad del main()

Directivas del pre-procesador

Las más importantes y que utilizaremos con más frecuencia son:

```
#define NOMBRE [VALOR]
```

```
#include "ledDrivers.h"
```

```
#ifdef DEBUG
 printf("encendiendio maquina");
#else
// No imprimo nada
#endif
```

Cuidado! Las directivas de pre-procesador no se terminan en punto y coma.

Variables y tipos de datos

Las variables son los elementos son los elementos básicos contenedores de datos de un programa. Deben tener un tipo especifico según el contenido que almacenaran y con los operadores podremos actuar sobre ellas.

Tipos de datos nativos

Existen pocos tipos de datos "nativos" desde la primera versión del lenguaje. Luego se agregaron nuevos para facilitar su uso.

- **char:** puede almacenar un caracter (ASCII).
- **int:** puede almacenar un numero entero.
- **float:** almacena un numero real representado en punto flotante de precisión simple.
- **double:** similar al float pero de doble precisión.

Modificadores y calificadores

Los modificadores permiten alterar el tipo de variables de alguna u otra manera. Algunos comúnmente usados son:

- **signed** y **unsigned**
- **long** y **short**

Los calificadores son:

- **const:** significa que su contenido no se podrá alterar.
- **volatile:** le indican al compilador que tiene propiedades especiales para tener en cuenta en la optimización.

Declaración de variables

Se pueden utilizar las siguientes formas de declaración con y sin inicialización:

```
int valor;

unsigned int valor2,valor3;

float Kp = 0.454;

int a,b=5;

char letra = 'Z';
```

Alcance (scope)

Una variable se dice que puede ser:

Local si está declarada dentro de una función o bloque de código

Global si está declarada fuera de toda función.

Las variables globales pueden ser accedidas por cualquier función o bloque de código, mientras que las locales, solo dentro del entorno que fue creada.

Derivados

Existen también los tipos derivados. Estos son:

- **Arreglos (arrays)**
- **Estructuras**
- **Tipos enumerativos**
- **Punteros**

Arreglos

Los arreglos son contenedores de múltiples variables **contiguas** del mismo tipo. Son de acceso aleatorio y se los puede acceder fácilmente en cualquier posición. Su declaración es la siguiente:

```
char buffer[100];

int gains[4] = {10,20,-30,50};

char msg[] = {"Hello,world!"};
```

¿Cuantos elementos tendrá el arreglo msg?

Estructuras

Las estructuras son tipos de datos creados por el usuario que contienen uno o más tipos de datos (nativos o creados por el usuario). Por ejemplo, si queremos almacenar diferentes datos de una persona, sería útil posiblemente tener un tipo de dato "persona" que tenga los siguientes atributos:

- nombre
- edad
- altura
- peso

Estructuras

Para declarar este tipo de datos nuevo, se debe utilizar la siguiente sintaxis:

```
typedef struct{
  char name[100];
  unsigned int age;
  float height;
  float weight;
}user_t;
```

Y luego usarlas como cualquier otro tipo de datos:

```
user_t User;
user_t admins[10];
```

En realidad, el typedef y el struct no necesariamente deben ir juntos, pero por comodidad se lo suele utilizar así.

Tipos enumerativos

Los tipos enumerativos solo admiten una cantidad finita de posibles valores. Se los suele utilizar cuando se busca que una variable solo pueda tomar determinados valores como ser: TRUE y FALSE, los días de la semana, colores, etc. Para declararlos se usa la siguiente sintaxis:

```
typedef enum {TRUE, FALSE} boolean_t;
```

El compilador utilizará alguna representación para el TRUE y otra para el FALSE, pero el programador no debe preocuparse por ello. Para declarar una variable se puede usar el siguiente fragmento de código:

```
boolean_t showDebug = FALSE;
```

Punteros

Los punteros son variables que no contienen datos sino posiciones de memoria, muy comúnmente de otras variables. Para declararlas se utiliza el símbolo asterisco * y se declaran de la siguiente manera:

```
int * p1;       // p1 es un puntero a un int
char * msg;     // msg es un puntero a un char
```

Su uso y sus detalles se verán en las siguientes secciones

Tamaño de los tipos de datos

Es lógico pensar que si un char debe contener un caracter ASCII entonces el mismo debe poder almacenar 8 bits.

Un float que cumpla con la norma IEEE-754 debe tener 32 bits para simple precisión y 64 bits para doble precisión.

El int generalmente se refiere a que usa la palabra del sistema entera. Entonces es esperable que en un microcontrolador de 32 bits, el int ocupe 32 bits. ¿Pero en uno de 8?

Tamaño de los tipos de datos

En general este problema no es tomado muy en cuenta cuando se diseña software para PC, pero en embebidos esto puede traer problemas de portabilidad. Los tamaños de las variables no son necesariamente respetadas por los compiladores, por eso surge en en el standard C99 el **stdint.h**

stdint.h

En el archivo de cabecera stdint.h se declaran los siguientes tipos de datos de tamaño exacto de bits y que son respetados por todos los compiladores C99:

Tipo	Singado	Bits	Bytes
int8_t	Signed	8	1
uint8_t	Unsigned	8	1
int16_t	Signed	16	2
uint16_t	Unsigned	16	2
int32_t	Signed	32	4
uint32_t	Unsigned	32	4
int64_t	Signed	64	8
uint64_t	Unsigned	64	8

Operadores

Los operadores permiten operar, modificar y consultar el valor de las variables. Una manera de categorizarlos es por su función:

- aritméticos
- de asignación
- lógicos y relacionales
- operadores de bits
- de acceso
- pperadore de punteros

Operadores aritméticos

Estos son: +, -, *, /, %, ++ y - -
Por ejemplo:

```
int m = -45;
int b = 8;
int x = 10;
int y;

y =  m*x + b;
b++;
m--;
```

Ver los operadores ++ y - - como post y pre incremento

Operadores de asignación

Estos son: = , += , -= , *= , /=, %=
Por ejemplo:

```
a = 10;
b = c*a;
d += 1;
d -= 10;
```

Lógicos y relacionales

Estos operadores se evalúan y pueden devolver solo verdadero o falso mediante un 1 o un 0 respectivamente.

Relacionales: $>$, $>=$, $<$, $<=$
Lógicos: $==$, (iguales), $!=$ (distintos)
Negado: !

Operadores de bits

Los operadores de bit operan bit a bit de cada variable, sin importar su tipo.

Estos son: & (AND), | (OR), $\wedge$ (XOR), << (left shift), >> (right shift), $\sim$ (NOT)

```
int flags;

flags = flags & ~0x05;
flags  = flags | 0x01;
```

También son validos —=, &=, etc.

Operadores de acceso

Los operadores punto (.) y llaves ([ ]) permiten acceder a miembros de una estructura y de un arreglo respectivamente. Por ejemplo:

```
user_t user;
float coefs[10];

user.age = 43;
user.height = 1.89;

coef[0] = user.age * user.height;
```

Operadores de punteros

Los operadores * y & son utilizados para punteros.

- *: "lo apuntado por"
- &: "la dirección de memoria de"

Por ejemplo:

```
int a = 10;
int * pa;

pa = &a;

*pa = 20;
```

¿Cuánto vale a al finalizar la ejecución del código?

Precedencia de operadores

¿Cuánto vale la siguiente expresión?

$$3 * 4 + 5$$

¿17 o 27?

Precedencia de operadores

Todos sabemos que primero se evalúa lo que esta entre los operadores + y - y luego estos últimos. Pero, esta sintaxis ¿Qué pregunta?

```
if (x & MASK == 0x55)
```

Si no sabemos que orden de precedencia (evaluación) tienen los operadores, no podemos estar seguros de que se va a evaluar x&MASK y luego se preguntará si es igual a 0x55 o primero se preguntara si la mascara MASK es 0x55 y luego se hace una AND con x.

Precedencia de operadores

La solución más segura es utilizar siempre paréntesis en las expresiones para evaluarlas. Esto no agrega ningún peso (overhead) adicional al código, y permite estar seguros de la evaluación.

```
if ( (x & MASK) == 0x55 )
```

Precedencia de operadores

El estandar C tiene definida una tabla de precedencias:

Operators	Associativity
() [] − > .	left to right
! ∼ ++ −− + - *(type)sizeof	right to left
*, /, %	left to right
+,−	left to right
<<, >>	left to right
<, <=, >, >=	left to right
== !=	left to right
&	left to right
∧	left to right
—	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ∧= —= <<= >>=	right to left

Control de flujo

Mediante el control de flujo, el programador puede hacer que el código ejecute de manera condicional o repetitiva diferente a una ejecución secuencial pura.

Estructuras condicionales - If

La estructura IF permite evaluar la condición de verdad de una expresión (que devuelve verdadero (1) o falso (0)). Su sintaxis en su forma más general es:

```
if (CONDICION1){
    \\ Bloque a ejecutar si se cumple la condicion 1
}else if(CONDICION2){
    \\ Bloque a ejecutar si se cumple la condicion 2
}else if(CONDICION3){
    \\ Bloque a ejecutar si se cumple la condicion 2
}else{
    \\ Bloque a ejecutar si no se cumplen las condiciones 1,2 ni 3
}
```

Estructuras condicionales - If

Por ejemplo:

```
if ( (a >= 65) && (a<=90){

   printf("la letra es mayuscula");

}else if ( (a >= 97) && (a<=122){

   printf("la letra es minuscula");

}
```

```
if (pulsador() == TRUE){
   prenderLed();
}else{
   apagarLed();
}
```

Estructuras condicionales - Switch

El comando Switch permite comparar una variable contra diferentes valores **enteros** de manera más compacta. Su sintaxis es:

```
switch (expression) {
    case expr:
        statements
    case expr:
        statements
    default:
        statements
}
```

Para terminar un bloque, se utilza el comando **break** pero esté puede ser omitido si se quiere que dos condiciones utilicen el mismo código.

Estructuras condicionales - Switch

Ejemplo:

```
switch (lastCommand) {
    case 'a':
    case 'A':
        // Hacer algo
        break;
    case 'b':
    case 'B':
        // Hacer algo
        break;
    default:
        // No es ningun comando reconocido
}
```

Para terminar un bloque, se utiliza el comando **break** pero esté puede ser omitido si se quiere que dos condiciones utilicen el mismo código.

Estructuras repetitivas - While

El While permite ejecutar un bloque de código siempre que se cumpla una condición. Su sintaxis es:

```
while(CONDICION){
    // codigo a ejecutar
}
```

Estructuras repetitivas - While

Por ejemplo:

```
while(!pulsador()){
    prenderLed();
    delay(100);
    apagarLed();
    delay(100);
}
```

Estructuras repetitivas - For

La estructura for permite ejecutar un código de manera similar al while, siempre que se cumpla una codición pero agrega la posibilidad de incluir una inicialización que se realizará una sola vez, y una operación luego de cada ciclo. Su sintaxis es:

```
for ( INIT ; CONDICION ; ACCION ){
    // Bloque de codigo a ejecutar
}
```

Estructuras repetitivas - For

Por ejemplo:

```
for (i = 0 ;  i < 10 ; i++ ){
    printf("\%i\r\n",i);
}
```

Funciones

Las funciones permiten dividir tareas grandes en pequeñas partes. Algunas características de las mismas son:

- Permiten la modularización
- Separan la implementación del uso

Funciones

En el lenguaje C, las funciones pueden y/o deben tener:

- un nombre único
- un tipo de valor de retorno
- parámetros

Por ejemplo:

```
int value;

value = readAdcChannel(4);
```

readAdcChannel es el nombre de una función que recibe un solo parámetro (aparentemente entero) y devuelve por nombre un valor.

Funciones

El **prototipo** de una función nos permite saber todos los datos necesarios para el usuario de la función. Para este caso podría ser:

```
int readAdcChannel(int);
```

Y la implementación de la misma:

```
int readAdcChannel(int chan){

  if ((chan>=0) \&\& (chan<=9){
    return ADCVal;
  }else{
    return -1;
  }

}
```

Funciones - Retorno

El retorno es la salida de una función:

- Cuando una función se dice que devuelve un valor por "nombre" significa que su nombre (más los parámetros) se evalúan tomando un valor.
- El tipo de dato de retorno puede ser cualquier tipo de dato valido en el sistema (nativos y/o definidos por el usuario).
- La función debe usar la instrucción **return** para devolver un resultado.

Funciones - Parámetros

Una función comúnmente recibirá parámetros con los cuales operar. Estos parámetros pueden ser de cualquier tipo y una función puede tener la cantidad de parámetros que uno necesite. Conceptualmente existen dos maneras de pasarle un parámetro a una función:

- Por copia
- Por referencia (punteros)

Funciones - Parámetros por copia

Cuando se dice que el parámetro se pasa por copia, el compilador **copia** los valores que el usuario ingresa al llamar a la función dentro del **stack** de la misma. Por eso los parámetros dentro de la función son **locales**. Por ejemplo:

```
int a = 5, b = 8;

printf("\%i", calcularCoef(a,b));
```

Donde la función sería por ejemplo:

```
int calcularCoef(int x, int y){
  return (x*y)/(x*x+y*y);
}
```

la variable "a" se copia a "x" y la variable "b" se copia a "y". "x" e "y" son variables locales de la función y modificarlas no afecta el valor de "a" y "b".

Funciones - Parámetros por puntero

Cuando se dice que el parámetro se pasa por puntero, la función recibe la dirección de memoria del dato y no una copia del mismo. Este método tiene algunas características importantes:

- Si el dato es grande (una estructura por ejemplo) esta no se copia entero **ahorrando memoria de stack y tiempo de ejecución**
- Es posible modificar el contenido de la variable "externa" ya que trabajo sobre la ubicación de la misma
- Es la única manera de pasar el contenido de un array de forma sencilla como parámetro
- Nos permite tener más de un valor de retorno

El último punto introduce el concepto de **parámetros de salida** que puede resultar un poco confuso.

Funciones - Parámetros de salida

Si necesitamos que una función devuelva más de un valor, entonces la única manera (sin devolver una estructura que contenga varios campos) es utilizar parámetros de salida. En el siguiente ejemplo se muestra como:

```
int sensorValue, sensorStatus;
char sensorName[20];
int retVal;

retVal = getSensorData(8,sensorName,&sensorValue,&sensorStatus);
```

Y el prototipo de la misma será:

```
int getSensorData(int num, char * name, int * val, int * stat);
```

Funciones - Parámetros de salida

La implementación podría ser algo similar a:

```
int getSensorData(int num, char * name, int * val, int * stat){

  int valBuff, statBuff;
  char nameBuff[20];
  int ret;

  // Codigo que lee los datos de un sensor externo y lo guarda en var

  if ( ret>0){
    strcpy(nameBuff,name);
    *val = valBuff;
    *stat = statBuff;
    return ret;
  }else{
    return -1;
  }
}
```

Punteros y arrays

Cuando se declara un arreglo de cualquier tipo, el nombre del arreglo es automáticamente **un puntero al primer elemento**.
Por ejemplo:

```
char welcomeMsg[] = {"System is starting..."};

UARTSend(welcomeMsg);
```

Y la función UARTSend podría tener esta forma:

```
void UARTSend(char * m){
  int i=0;

  while ( *(m+i)!='\0'){
    UARTputc(*(m+i));
    i++;
  }
}
```

Punteros a estructuras

Recordemos la estructura creada user_t y supongamos una armemos una función que imprime sus campos con cierto formato:

```
typedef struct{
  char name[100];
  unsigned int age;
  float height;
  float weight;
}user_t;

user_t admin={"Pedro Gonzales", 42, 1.75, 83.5};

printUserInfo ( &admin);
```

Punteros a estructuras

La función printUserInfo podría tener la siguiente forma:

```
printUserInfo ( user_t * u){
    UARTSend( (*u).name );
}
```

O lo que es lo mismo:

```
printUserInfo ( user_t * u){
    UARTSend( u->name );
}
```

Strings

Los strings o cadenas de caracteres, como tipo de dato en si no existen. Pero si hay un convención en cuanto a su utilización y decimos que:

"Un String en C es simplemente una cadena de caracteres (chars) terminadas en el carácter '\0' (0 decimal)".

En C, al declarar un array de chars e inicializarlo, automáticamente el compilador le agrega un 0 al final. Es decir:

```
char msg[]={"Hola"};
```

Es un arreglo de 5 elementos y el 5to es el 0. Al no ser un tipo nativo, se deben usar funciones para trabajar con strings ya sea para comparar, asignar, concatenar, etc. Estas funciones están incluidas en la biblioteca string de C y su archivo de cabecera es el **string.h**.

FIN