Reglas básicas de la programación en lenguaje C

ISO - SO

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

0. Introducción

C es un lenguaje de programación de propósito general. Fue inventado por Dennis Ritchie del los Laboratorios Bell de AT&T con el fin de proporcionar un lenguaje de alto nivel a la programación sobre el sistema operativo UNIX. Hoy en día se utiliza también para muchas otras aplicaciones. Se le ha llamado “el lenguaje de programación de sistemas” porque es muy útil para escribir compiladores y sistemas operativos, ya que permite expresar el código de forma simple y concisa.

C posee tres tipos de datos básicos: enteros (int), caracteres (char) y números en coma flotante (float, double). Los enteros pueden ser calificados como cortos (short), largos (long) o sin signo (unsigned). Otros tipos se pueden derivar de éstos, creándolos mediante punteros, vectores, estructuras o uniones, pero no provee operaciones para manipular objetos compuestos.

Las construcciones de control son sencillas y familiares para programadores que ya conozcan Pascal, PL/I o Modula 2. Permiten agrupar sentencias, tomar decisiones (if-else), seleccionar entre varias posibilidades (switch), hacer bucles evaluando la condición al empezar (while, for) o al acabar (do), y salir inmediatamente de un bucle (break).

Las funciones pueden devolver valores de los tipos básicos o de los estructurados. Cualquier función puede llamarse recursivamente. Las variables locales de las funciones se crean nuevas en cada invocación. No se permite la definición anidada de funciones. En cambio, las variables pueden ser locales a una función, o globales y entonces visibles desde todo el programa.

Una fase de preprocesamiento previa a la compilación permite la sustitución de macros, la inclusión de otros ficheros fuente y la compilación condicional.

El lenguaje C va acompañado de una librería con funciones para llamar al sistema operativo, dar formato a las entradas y salidas, gestionar la memoria, manipular strings, etc...

1. Estructura del programa

La estructura más usual de un programa en C es la siguiente:

```
/* los comentarios comienzan por “/*” y acaban con “*/” */
#include archivo /* inclusión archivos cabecera */                    (punto 3)
#define CONST /* declaración de constantes*/                          (punto 4)
tipo variable; /* declaración de variables globales */                (punto 5)
tipo funcion (argumentos); /* declaración de cabeceras de funciones */ (punto 7.3)
main()
{
tipo variable; /* declaración de variables locales */                 (punto 5)
cuerpo del programa                                                   (punto 6)
}

tipo funcion (argumentos) /* declaración de funciones */
{
tipo variable; /* declaración de variables locales */                 (punto 5)
cuerpo de la función                                                  (punto 6)
return (resultado);
}
```

`main()` es una función como las demás que pueden aparecer en el programa, lo que la caracteriza es que es la primera que se ejecuta.

2. Comentarios

Para poner comentarios se sigue el siguiente formato:

`/*` *comentario* **`*/`**

Un comentario puede ocupar varias líneas. No se pueden poner comentarios anidados; se considera comentario todo lo incluído entre **`/*`** y **`*/`**.

3. Inclusión de archivos

Es elegante construir un programa de forma modular a partir de varios ficheros con declaraciones de funciones y variables, el cuerpo principal del programa, las funciones auxiliares, etc... Cuando se necesite, se puede incluir un fichero fuente mediante la primitiva del preprocesador **`#include`**, utilizando una linea por cada archivo.

Para utilizar una función de librería es necesario incluir el archivo cabecera donde está declarada la función y también se puede encontrar declaraciones de constantes y tipos. Por convención, los nombres de los archivos cabecera que contienen declaraciones acaban con la “extensión” .h , pero no es obligatorio.

Las sentencias para hacer la inclusión son:

`#include <`*nombre_archivo.h***`>`**
`#include "`*nombre_archivo.h***`"`**

Si se especifica explícitamente el nombre completo del camino al indicar el nombre del archivo, el compilador sólo buscará el archivo con ese pathname completo.

Diferencia de las dos sintaxis: si se encierra el nombre del archivo entre comillas, primero se buscará el archivo en el directorio de trabajo actual, y si no lo encuentra, buscará en los directorios estándar de include. Si el nombre está encerrado entre paréntesis angulares sólo busca el archivo en los directorios de include estándar, pero en ningún momento se recorre el directorio de trabajo actual. Estos directorios estándar son **/usr/include** o **/include**, si no se especifican otros directorios con las directivas **`-Idir`** o **`-I`** del compilador.

4. Declaración de constantes y macros

Para definir constantes se sigue el formato:

`#define` identificador *valo*r

El compilador substituirá el identificador por el valor, cuando lo encuentre en el archivo fuente. En general, las constantes se escriben totalmente en mayúsculas, aunque el preprocesador no impone ninguna norma al respecto. Es sólo por claridad.

Permite también definir macros, en las cuales se sustituye sus parámetros por los del punto en que se la invoca.

Ejemplos:
`#define` TRUE 1
`#define` FALSE 0
`#define` ERROR_MSG "Error en la lectura\n"
`#define` IMPRIMIR(x) printf(“%d “,x)

5. Declaración de variables globales y locales

El formato para definir variables es el siguiente:

*tip*o lista_variable**s** ;

La *lista_variables* es el conjunto de nombres de variables separadas por comas.
Las variables pueden declararse:
a) Dentro del bloque de código de las funciones o del main. Son variables locales y por ello, sólo se pueden utilizar en este entorno.
b) En la definición de los parámetros formales de las funciones. Tienen la misma utilización que las variables locales.
c) Fuera de todas las funciones, incluido el main. Son variables globales y se pueden usar desde cualquier parte del programa.

5.1. Tipos de datos

5.1.1. Tipos de datos elementales

Nombre	Descripción
`int`	entero
`char`	carácter
`float`	número de coma flotante (con punto decimal y/o exponente**)**

Ejemplo:
```
int a, b, c ;
float x ;
```

5.1.2. Tipos de datos estructurados

Declaración de tablas

El formato general para la declaración de una tabla unidimensional es:

*tip*o *nombre_var* **`[`***tamaño***`];`**

Esta sentencia declara una variable *nombre_var*, donde *tip*o indica el tipo de los elementos de la tabla, y *tamañ*o el número de elementos que contiene la tabla. Esta tabla se indexará desde 0 hasta “tamaño-1”. Un tipo especial de tablas son los "strings", que son vectores de carácteres que acaban con el carácter ASCII 0.
Ejemplo:

```
int vector[10];
char mens1[5], mens2[80] ;
```

Para acceder a un elemento de una tabla, el formato es: *nombre_var***`[`***posición***`].** También se pueden definir tablas multidimensionales, siguiendo el esquema:

*tip*o *nombre_var* [*tam_dim_1*][*tam_dim_2*] ... [*tam_dim_N*];

en el que *tam_dim_i* es el tamaño de la dimensión i-ésima. Ejemplo:

```
int a, b, num[3][10][4];
```

Declaración de tuplas

El formato general de una definición de tupla es:

```
struct nombre_tipo_de_estructura {
```

tipo nombre_campo_1;
.....
.....
tipo nombre_campo_N;
} lista_variables;

Se puede omitir el *nombre_tipo_de_estructura*, o bien, la *lista_variables*, pero no los dos. Ejemplo:

```
struct cuenta {
        int cuenta_num;
        char cuen_tipo;
        char nombre[80];
        float saldo;
} cliente;
```

Ahora podríamos declarar otra variable del mismo tipo:

```
struct  cuenta   nuevocliente;
```

Si se omite la *lista_variables*, se debe poner el ";" detrás de la llave de fin de tupla. Ejemplo:

```
struct fecha {
        int dia;
        int mes;
        int año;
};
```

Para acceder a un elemento de una tupla se usa el siguiente formato:nombre_variable.nombre_campo . Ejemplo:

```
fecha  f_actual;          /* declaración */
x = f_actual.mes          /* y así se referencia */
```

5.1.3. Tipos de datos definidos por el usuario

En términos generales un nuevo tipo se define como:

```
typedef tipo nombre_nuevo_tipo;
```

donde *tipo* se refiere a un tipo de datos estándar existente o previamente definido. Ejemplo:

```
typedef  float  altura;
altura  hombres[100], mujeres[100];
```

La definición de tipos mediante **typede**f es especialmente útil para definir tuplas, que pueden ser escritas como:

```
typedef  struct {
        tipo nombre_campo1;
                .....
                .....
        tipo nombre_campoN;
} nuevo_tipo;
```

Ejemplo:

```
typedef struct {
        int dia;
        int mes;
        int año;
} fecha;
fecha   aniversario;        /* La variable aniversario es de tipo fecha */
```

5.1.4. Punteros

El formato general para una variable de tipo puntero es:

tipo_apuntado *nombre_puntero;

Ejemplo:

```
char *p;        /* p contendrá la dirección de un espacio de memoria de tamaño carácter */
char p[];       /* Equivalente a la anterior */
int a, *temp, b;
```

Operadores relacionados con punteros:

& Retorna la dirección de memoria de su operando.

***** Indica el contenido de memoria apuntada por su operando.

P.ej.: p = &a;

Por ejemplo, teniendo en cuenta que "=" es la instrucción de asignación:

```
*p = 'a';       /* el carácter “a” se almacena en la posición apuntada por p */
temp = &a;      /* temp contiene la dirección de la variable a */
a = 3;          /* a := 3 */
b =*temp;       /* b contiene el valor de a */
```

Cuando el *tipo_apuntado* es una tupla, para acceder a un campo de la tupla a través del puntero, la sintaxis es:

nombre_puntero -> nombre_campo

Ejemplo:

```
struct ropa {
        char fabricante[20];
        int coste;
        char descripción[40];
} *p;
p -> coste = 1000;      /* Acceso al campo coste a través del puntero p */
```

IMPORTANTE: La declaración de un puntero **no** lleva asociada la reserva de espacio para el tipo de datos apuntado. Así, en el ejemplo anterior, antes de acceder al campo 'coste' deberíamos haber reservado espacio en memoria para una variable de tipo ropa y hacer que p apunte a ella.

5.2. Inicialización de variables en la declaración

El esquema para la inicialización de variables en la declaración es el siguiente:

tipo nombre_variable = valor;

Ejemplo:

```
int a = 3, b;
char c = ´p´, p;
int  tabla[10] = {1,2,3,4,5,6,7,8,9,10};
char cadena[5] = "hola"; otra_cadena[10] = "pepito";
```

IMPORTANTE: La inicialización de una cadena almacena automáticamente, tras el último carácter de la cadena, el carácter nulo ´**0**´ que indica final de string. En los ejemplos, en la quinta posición (cadena[4]) y en la séptima posición (otra_cadena[6]). Las funciones que manipulan strings esperan que acaben en ´**0**´ (por ejemplo, strlen). No está permitida la asignación, en ejecución, de una cadena de caracteres, hay que usar una función de copia como strcpy. Ver el anexo acerca de los strings o cadenas de caracteres al final de este documento.

6. CUERPO DEL PROGRAMA

6.1. Operadores

6.1.1. Operadores aritméticos

Operador	Propósito
+	adición
-	sustracción
*	multiplicación
/	división
%	módulo

6.1.2. Operadores unarios aritméticos

Operador	Propósito
-	menos unario
++	autoincremento
- -	autodecremento
sizeof	retorna el tamaño de su operando en bytes

6.1.3. Operadores relacionales

Operador	Propósito
<	menor que
<=	menor o igual que
>	mayor que
>=	mayor o igual que
==	igual que
!=	distinto que

Estos seis operadores se utilizan para formar expresiones lógicas que representen condiciones que pueden ser *cierta*s o *falsa*s. La expresión resultante será de tipo entero, ya que *fals*o se representa por el valor 0 y *cier*to por distinto de 0. Ojo con asignar cuando realmente se quería comparar: “if (a = b) ...” es una asignación e “if (a == b) ...” es una comparación.

6.1.4. Operadores lógicos

Operador	Propósito
&&	Y lógico
||	O lógico
!	NO lógico

No existe un tipo booleano; se trabaja con enteros. En este contexto un cero significa *falso* y cualquier valor distinto de cero, no sólo el 1, se interpreta como *cierto.*

6.1.5. Asignación

El operador de asignación es el operador =. El formato es el siguiente :

identificador = expresión;

Se permite hacer asignación múltiple del tipo:

i = j = 5.9 ;

6.1.6. Asociatividad y precedencia de operadores

Se muestra a continuación una relación de precedencia de los operadores de mayor a menor.

Categoría del operador	**Operadores**	**Asociatividad**
Operadores unarios	- ++ -- ! sizeof	derecha
Multiplicación, división y resto	* / %	izquierda
Adición y sustracción	+ -	izquierda
Operadores relacionales	< <= >= >	izquierda
Operadores de igualdad	== !=	izquierda
Y lógica	&&	izquierda
O lógica	||	izquierda

6.2. Sentencias de control

A continuación citaremos algunas sentencias de control que pueden aparecer en los programas; en tales citas utilizaremos la expresión *sentencia*s para referirnos indistintamente a un grupo de sentencias, a una sentencia simple, o a la sentencia vacía (ninguna sentencia). Cuando se trate de una sentencia múltiple, este grupo de sentencias deberán ser encerradas entre llaves.

6.2.1. Sentencias alternativas

IF

```
if (condición) sentencias;
[else if (condición) sentencias;]
[else if (condición) sentencias;]
.............
[else sentencias;]
```

Ejemplo:
```
if (c==´ ´)     blancos++;
else if ((c>=´0´) && (c<=´9´))digitos++;
else if ((c>=´a´) && (c<=´z´))letras++;
else  otros++;
```

SWITCH

```
switch (variable)
{
case constante1 :cjto_sentencias
                 break;
case constante2 : cjto_sentecias
                 break;
      .........
[default: cjto_sentecias]
}
```

Ejemplo:
```
switch (car)
{
      case ´r´: minusc ++;
      case ´R´:rojo++;
```

```
                    break;
            case ´b´: minusc ++;
            case ´B´: blanco++;
                    break;
            case ´a´: minusc ++
            case ´A´:azules++;
                    break;
            default:otros++;
    }
```

En este ejemplo se puede ver que hay ´case´ que no tienen break; esto significa que se ejecutarán todas las sentencias que haya hasta encontrar el primer break.

El default es el conjunto de sentencias que se ejecutarán si el valor de la variable no coincide con ninguna de las constantes.

6.2.2. Sentencias repetitivas

WHILE

```
while (condición) sentencias;
```

Ejemplo: Cálculo de la media de los elementos de un vector.

```
#define MAX 10
main()
{
int vector[MAX] = {1,7,50,23,25,42,19,17,35,9};
int i=0, suma=0, media;
while (i < MAX)        suma = suma + vector[i++];
media=suma / MAX;
}
```

DO - WHILE

```
do sentecias while (condición) ;
```

Las sentecias se ejecutarán como mínimo una vez, ya que esta estructura corresponde a **repetir** en pseudocódigo, en la que la condición no se evalúa más que al final de cada iteración del bucle.

Ejemplo: Cálculo de la frecuencia de un elemento de un vector.

```
#define MAX 10
#define ELEMENTO ´a´
main()
{
        char   vector[MAX] = {´a´,´n´,´i´,´d´,´e´,´s´,´t´,´n´,´o´,´m´};
        int i=0, frec=0;
        do
                if (vector[i++] == ELEMENTO)          frec++;
        while (i < MAX);
}
```

FOR

```
for (inicialización; condición; incremento) sentencias;
```

Inicialización es una sentencia de asignación que inicializa la variable de control del bucle. *Condición* es la

expresión que comprueba la variable de control del bucle cada vez, para determinar cúando salir del bucle. *Incremento* define la manera en que cambia la variable de control, y se ejecuta después de la sentecia.

Equivale por tanto a :

```
inicialización
while (condición)
{
        sentencias
        incremento
}
```

Ejemplo: suma de dos vectores.

```
#define TAM 10
main()
{
float a[TAM] = {-10.0, -8.0, -6.5, 4.3, 4.0, 3.0, 2.0, 0.0, 1.9, -2.5};
float b[TAM] = {-2.3, -4.9, 3.0, 0.9, 1.0, 3.5, -1.3, 0.8, -0.9, 1.0}, c[TAM];
int i;
for (i = 0; i < TAM; i++)        c[i] = a[i] + b[i];
}
```

7. Funciones

7.1. Definición de una función

```
tipo nombre_función(tipo1 arg1, tipo2 arg2, ... ,tipoN argN)
{
declaración variables locales;
cuerpo función
}
```

O bien:

```
tipo nombre_función(arg1, arg2, ... , argN)
tipo1 arg1;
tipo2 arg2;
  ...
tipoN argN;
{
declaración variables locales;
cuerpo función
}
```

El *tipo* que aparece delante del nombre_función, es el tipo del valor que devuelve esta función. Si lo que se quiere implementar es un procedimiento en lugar de una función, el tipo devuelto se declara **`void`**. Asímismo, cuando no exista ningún parámetro formal, se deben poner simplemente los dos paréntesis **`( )`**.

Se devuelve un valor desde la función hasta el punto del programa desde donde se llamó, mediante la sentencia **`return`** *expresió*n. Si se omite la expresión, simplemente hace que se devuelva el control al punto de llamada.

Ejemplo:

```
int  factorial (int  n)
{
        int  i, fact = 1;

        if (n > 1)
          for  (i=2; i <= n; i++) fact = fact*i;
        return (fact);
}
```

Para hacer una llamada a la función anterior, sería por ejemplo:

```
a = factorial(3);
```

7.2. Paso de parámetros

El paso de parámetros en C es por valor, pero se puede simular una llamada por referencia utilizando un puntero como argumento. Este argumento formal tendrá la forma : ***tipo_*i** *arg_i , y a la hora de realizar la llamada, el parámetro real tendrá el formato : ***&param***

Ejemplo:

```
void   factorial (int  n, int  *fact)
{
        int i;

        *fact = 1;
        if (n > 1)
          for (i = 2; i <= n; i++) *fact = *fact * i;
}
```

Y un ejemplo de llamada a esta función:

```
int a;
factorial (3, &a);
```

7.3. Declaración de la cabecera de una función

Tendrá el siguiente formato:

tipo nombre_función *(tipo*1 arg1, ... ,*tipo*N argN);

O bien,

```
tipo nombre_función(arg1, arg2, ... , argN)
tipo1 arg1;
tipo2 arg2;
   ...
tipoN argN;
```

7.4. Paso de parámetros a la función main()

main ( ) es la función principal del programa, y también se le pueden pasar parámetros como a cualquier otra función, ya que no es más que la primera función a la que se llama dentro de un programa. El paso de parámetros se realizará desde el intérprete de comandos o desde la llamada al sistema que ejecuta un nuevo programa (exec).

Para ello se usan dos argumentos especiales predefinidos: **argc** y **argv.**

El parámetro **argc** es un entero que contiene el número de argumentos de la línea de comandos. Siempre tiene el

valor como mínimo de 1, ya que contabiliza también el nombre del programa como un parámetro. Siempre que utilizamos el valor de algún parámetro pasado al main, hay que verificar el valor de **argc**.

El parámetro **argv** es un vector de punteros a carácteres. Los punteros a carácteres apuntan a las cadenas que contienen los argumentos que se pasan en la línea de comandos.

El siguiente esquema indica cómo están definidos los argumentos **argc** y **argv** dentro del main :

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

Una alternativa a este esquema es la siguiente :

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

El paso de parámetros se realizará desde la línea de comandos de la siguiente manera:

nombre_programa param_1 param_2 **...** param_N

Ejemplo: A continuación se expone un programa cuyo ejecutable se llama exponenciacion, que eleva a un *exponent*e una cierta *base. Bas*e será el primer argumento desde la línea de comandos, y *exponent*e el segundo.

```
#include <stdlib.h>
main (int argc,char *argv[])
{
        int error, i, result = 1, base, exp;
        if (argc!=3)error=1;
        else
        {
                /* atoi es una función de conversión de tipos; dado un string,
                te devuelve el entero asociado */
            base = atoi (argv[1]);
            exp = atoi (argv[2]);
            if (exp > 0)
                for (i = 0; i < exp; i++)result = result*base;
            error = 0;
        }
return (error)
}
```

Una posible llamada al programa anterior podría ser:

exponenciacion 2 *3*

8. Escritura de datos, la función sprintf

Se pueden escribir datos en el dispositivo de salida estándar utilizando la función de librería **sprintf**.

Para poder utilizar esta función es necesario incluir la función de cabecera **stdio.h.**

En términos generales la función **sprint**f se escribe :

sprintf (puntero_a_cadena_resultado, "*cadena_de_control*", arg1, arg2, ... , argN)

donde *puntero_a_cadena_resultado* es el lugar donde se depositará el conjunto de caracteres resultado del la llamada a la función, *cadena_de_contro*l hace referencia a una cadena de carácteres que contiene información sobre el formato de la salida, y *arg1,...,argN* son argumentos que representan los datos de la salida.

*Códigos_de_contro*l de los datos de salida de uso común:

Codigo de control	**El dato es visualizado como...**
%c	un carácter
%d	un entero
%s	una cadena de carácteres
%f	un valor en coma flotante sin exponenete
%%	%

Para producir el salto de línea se utiliza el carácter \n y un tabulador se representa por \t.

Tras el formateo, se puede escribir en un fichero o dispositivo utilizando la llamada al sistema write().

Ejemplo:

```
/* fact.c */
main  (int argc, char *argv[])
{
char mens[80];
sprintf (mens, "El factorial de %d es %d\n", argv[1], factorial (atoi(argv[1]));
write (1, mens, strlen(mens));
}
```

Así, introduciendo en la línea de comandos:

```
$ fact 4
```

escribiría:

```
El factorial de 4 es 24
```

9. Anexo: Strings en el lenguaje C

En C, un “string” es un vector de caracteres que termina con un carácter especial (fin de string) que es el carácter NULL:

NULL == (char) 0 == ‘\0‘ == código ASCII 0

La declaración de un string, por tanto, será:

```
char buffer [MAX_BUFFER];
```

De esta forma hemos declarado y reservado memoria para un string de longitud MAX_BUFFER. Cuando se haya de pasar el string como parámetro, como que se hace por referencia, tendremos que pasar su dirección, que en C es el nombre del string (apunta al primer elemento del vector).

Este paso de parámetros por referencia es válido para cualquier tipo de vector, ya sea de caracteres, de enteros, de reales...

Ejemplo: - Función que recibe dos strings como parámetros y copia el uno sobre el otro:

```
strcpy   ( char o[], char d[])
/* o : string origen; d : string destino*/
{
int i;
for (i=0; o[i]!='\0'; i++)          d[i] = o[i];
}
```

En este caso tratamos el string como un vector donde la iésima posición del vector (o[i]) contiene el iésimo carácter del string. La función realiza un recorrido por el string hasta que encuentra el carácter NULL, señalización de su fin; entonces sale del bucle.

Otra forma de hacer lo mismo que en el ejemplo anterior:

```
strcpy   ( char *o, char *d)
/* o : string origen; d : string destino*/
{
while (*o)
        /* '*o' es el contenido de la dirección de memoria apuntada por 'o'.
        Cuando apuntado por esa dirección haya un carácter NULL, acabará el bucle */
        {
        *d = *o;
        d++;
        o++;
        }
}
```

Las instrucciones 'o++' y 'd++' incrementan el puntero en la medida en bytes de la variable a la que apunta (en nuestro caso incrementa en uno porque nuestro puntero apunta a carácter y un carácter ocupa un byte)

Llamada a la función anterior:

```
char origen[50], destino[50];
main()
{
 ....
 strcpy (origen, destino);
          ^^^      ^^^
        son punteros al primer elemento del vector ( origen == &origen[0] )
....
}
```

Por último, hay que diferenciar estas dos funciones:

sizeof (puntero) ---> retorna el tamaño del elemento que se le pasa como parámetro.

strlen (puntero) ---> retorna el tamaño del string apuntado por 'puntero'; la función recorre el vector hasta que encuentra el carácter NULL.

Ejemplo:

```
char buffer[100], *p;

/* Supongamos que el vector 'buffer' contiene: HOLA !\0 ..... (basura hasta el final). */

sizeof (p)==4 /* tamaño de un puntero */
sizeof (buffer)==100 /* tamaño del vector */

strlen(buffer) ----> retorna 6 (la longitud del string HOLA !)
```

Recordad que el espacio necesario para almacenar el string es 7 (siete) ya que el '\0' es un carácter más.