BREVE INTRODUCCIÓN A C

Elaborado por Juan Pablo Contreras y Natalia Valencia.

Algunas generalidades sobre C

Java se desarrolló basado en la sintaxis básica de C, por lo cual las estructuras de control (**if**, **while**, etc.) y los tipos de datos primitivos son parecidos. Una diferencia entre los dos consiste en que para C el tipo de datos **char** es un tipo de datos numérico; en este sentido, es equivalente al tipo de datos **byte** de Java, y el tipo de datos **char** de Java no tiene un equivalente como tal en C. Adicionalmente, C tiene tipos numéricos con signo (**signed**) y sin signo (**unsigned**); los números sin signo almacenan valores estrictamente positivos. Esto se aplica a todos los tipos de datos: **signed int** y **unsigned int**, **signed char** y **unsigned char**, etc. Si un tipo de datos se declara sin especificar el signo, se supone que es signed: **int** es igual que **signed int**.

Otra diferencia notoria con Java es el tipo de datos **bool**: C no tiene valores lógicos como tales; las condiciones lógicas en C son valores numéricos: si el valor es cero, quiere decir falso, si el valor es diferente de cero, quiere decir cierto. Sin embargo, en C se dispone del archivo de encabezado **<stdbool.h>**, el cual provee un tipo de datos **bool** y las dos constantes **true** y **false**.

En C, el concepto de apuntador es muy importante; es un lenguaje muy basado en apuntadores. Un apuntador es una variable que contiene la dirección de otra; es una variable que se refiere a otra.

En C los programas se estructuran con base en procedimientos o funciones, esto a diferencia de la programación orientada por objetos donde los programas se estructuran alrededor de las clases. En consecuencia, un programa en C está constituido por un conjunto de procedimientos o funciones que se invocan entre ellos; se podría decir que la estructura viene dada por cómo fluye el control, y no por cómo se relacionan los datos entre ellos.

En algunos lenguajes se diferencia entre *procedimientos* y *funciones*: un procedimiento realiza acciones pero no retorna un valor; una función realiza acciones y retorna un valor. En el caso de C, esta diferencia no se refuerza; en consecuencia una función puede ser utilizada como un procedimiento, en cuyo caso el valor que retorna se desecha. En C procedimientos y funciones se declaran de la misma forma; un procedimiento es una función que retorna **void**, es decir, que no retorna nada. En general, C no refuerza mucho la diferencia entre expresiones e instrucciones, y hay algunas construcciones que tienen un poco de las dos cosas (como los operadores ++ y – y el operador de asignación mismo.

Estructura de un programa C – Main

```
/*Este es un comentario tipo c. También se puede usar //. */

#include <stdio.h>
/* stdio.h es una librería; una librería es un conjunto de funciones ya compiladas y listas para usar. #include es una directiva para informarle al compilador que el programa va a usar las funciones de una cierta librería. */
```

```
int miFuncion1(int , char *, float []);
/* En C es conveniente declarar prototipos de las funciones.
Un prototipo es una declaración del encabezado de la función
(sin el cuerpo). El anterior es un prototipo de función que
recibe un entero por valor, un apuntador a un char, un
arreglo de flotantes y retorna un entero. */

void miFuncion2(void);
/* Esta función no recibe ningún parámetro ni retorna valor
alguno, lo cual es denotado por la palabra void. */

main()
/* Un programa en C es un conjunto de funciones. La función
main es aquella en la cual inicia la ejecución del programa*/
{
      int a, b;
/* Esta es una declaración de variables del tipo entero. */

/* Aquí van las instrucciones del main. */
      printf("Este es un programa en C\n");
/* printf() es  una función de la librería stdio.h*/
}


int miFuncion1(int d, char *tPtr, float t[])
/* Implementación de miFuncion1 con la variable entera d, el
apuntador tPtr (a char) y el arreglo de float t */
{
/* Aquí van las instrucciones de la función miFuncion1. */
}

void miFuncion2(void);
{
/* Aquí van las instrucciones de la función miFuncion2. */
}
```

Alcance

El alcance sirve para saber en qué región del código la declaración de una cierta variable está activa. Si la declaración se realiza en un bloque de código entre llaves (es decir, entre { y }), el alcance es la región que va entre las llaves.

Sin embargo, existen un tipo de variables llamadas *globales*: estas deben ser declaradas en la parte de arriba del archivo o en una parte que no va entre llaves y el alcance se establece en todo el código (a partir del punto de declaración).

Tipos de datos

TIPO	TAMAÑO EN BYTES	RANGO
short int	2	-32,768 -> +32,767
unsigned short int	2	0 -> +65,535
unsigned int	4	0 -> +4,294,967,295
int	4	-2,147,483,648 -> +2,147,483,647
long int	4	-2,147,483,648 -> +2,147,483,647
signed char	1	-128 -> +127
unsigned char	1	0 -> +255
float	4	
double	8	
long double	10	

En esta tabla hay que tener en cuenta que el tamaño en bytes de cada tipo de dato depende de la implementación de C y de la máquina, dado que las arquitecturas de cada máquina pueden diferir profundamente.

Operadores

Operadores aritméticos

+	Suma
-	Resta
*****	Producto
/	Cociente de una división
%	Resto de una división (módulo)

Operadores lógicos

!	Not (no lógico)
&&	And (y lógico)
||	Or (ó lógico)

Operadores binarios

&	And bit a bit
|	Or bit a bit)
~	Not bit a bit
^	Xor bit a bit

Operadores relacionales

==	Igual a
!=	No igual a
>	Mayor que
<	Menor que
>=	Mayor o igual que
<=	Menor o igual que

Operadores de corrimiento (de bits)

<<	Corrimiento a la izquierda
>>	Corrimiento a la derecha

La función de **a >> n** es desplazar los bits de la variable **a** tantos espacios como se le indiquen (**n**) a la derecha (se usa **<<** para desplazar a la izquierda; es decir, **a << n** mueve los bits de **a** a la izquierda **n** posiciones).

Por ejemplo: sea **ch** un **char** con contenido 00111000. Si aplicamos **ch >> 2**, produciría como resultado 00001110.

Operadores de dirección

Los datos en un computador están representados por dos ideas básicas, que encierran en sí el concepto de variable o constante. Estos conceptos son valor y dirección o ubicación dentro del espacio de direcciones de la memoria. La comprensión de esta realidad es crucial en el desarrollo de programas en ensamblador, y, dado que C es bastante cercano a la máquina, nos provee de ciertas herramientas que nos permiten trabajar con estos conceptos. Las direcciones en la IA32 se representan con números binarios de 32 bits, es decir, cualquier dirección es de 32 bits sin importar a qué dato direccione. Hay que aclarar que mientras que el ensamblador no nos provee de mecanismos de protección, C sí obliga que los apuntadores apunten a datos del mismo tipo. Para ver las facilidades de C, vamos a introducir la declaración de variables de tipo apuntador:

```
int * aPtr;
```

Esta instrucción declara un apuntador llamado **aPtr** que tiene la dirección de un entero. Si observan con atención, verán el uso de un operador fundamental en el manejo de apuntadores. Este operador es ***** y en el contexto de la declaración de variables que apuntan a otras variables significa: "Lo siguiente es un apuntador".

Ahora, una vez que tenemos el apuntador, ¿qué le asignamos? Supongamos que teníamos declarada una variable de tipo entero llamada **a** y que queremos asignarle a **aPtr** la dirección de **a**. Para esto, C nos provee del operador *ampersand* o **&**. Este operador es unario, como lo es el operador *****, y va a la izquierda de la variable de la que queremos obtener su dirección. Su semántica es "la dirección de…". Para ver su uso escribamos lo que deseamos:

```
aPtr = &a;
```

En este momento ya tenemos un apuntador en **aPtr** que apunta a **a**. Ahora, para saber el contenido de **a** usando **aPtr**, usamos el operador ***** en su otro uso, cuyo significado es: "el contenido de…". Supongamos que queremos asignar a un entero **b** definido al inicio del programa. Asignemos a **b** el contenido de **a** usando el apuntador **aPtr**:

```
b = *aPtr;
```

En este momento **b** ya tiene el valor de la posición de memoria apuntada por **aPtr**, o, en otras palabras, **b** tiene el valor de la variable **a**. Para los inquisitivos, también es posible definir apuntadores a apuntadores, usando ****** con los mismos usos que ya hemos aprendido. Este operador permite manejar la doble indirección.

Entrada – Salida

printf(): Escribe datos en la consola con el formato especificado.

scanf(): Función de entrada por consola con el formato especificado.

```
printf ("caracteres de impresión y escape", argumentos);
```

```
scanf (" caracteres de impresión y escape", argumentos);
```

Vectores

Arreglos unidimensionales

Los arreglos son secuencias de datos del mismo tipo que se guardan en posiciones contiguas de memoria. La forma general de declaración es: `tipo nomvariable [tamaño];` También podemos inicializar el arreglo al momento de declararlo.

Por ejemplo:

```
int i[10] = {1,2,3,4,5,6,7,8,9,10};
```

Los arreglos tienen que declararse para que el compilador reserve espacio en memoria para ellos. Una advertencia: C, a diferencia de Java, no comprueba los límites del arreglo, en consecuencia, podríamos sobrepasar el límite de un arreglo y escribir, sin querer, en alguna otra posición de memoria sin que el sistema reporte que eso ha ocurrido.

Arreglos multidimensionales

C permite arreglos de más de una dimensión. La forma general de declaración de un arreglo multidimensional es:

```
Tipo nombre [a] [b] [c] ... [z];
```

Cadenas de caracteres

En C las cadenas de caracteres están basadas en arreglos de **char**. Los caracteres se almacenan en el arreglo uno detrás de otro a partir de la posición cero. Para indicar el fin de la cadena, el último carácter debe ser el carácter nulo (ASCII 0); esto implica que una cadena de `n` caracteres, ocupa `n+1` posiciones en el arreglo.

El carácter nulo en C se representa por `'\0'` (aunque, dado que los **char** son un tipo numérico, también se puede usar el número cero).

Estructuras

Una estructura es un "tipo derivado", es decir, es un tipo no primitivo de C que se construye a partir de tipos primitivos como int, float y otros. La sintaxis es como sigue:

```
struct nombreEstructura{
    tipo1  variable1;
...
    tipoM  variableM;
} variableTipoEstructura1,...,variableTipoEstructuraN;
```

Los *variableTipoNombreEstructuraX* pueden no existir, y son como "instancias sin métodos" con las características de la estructura. Para acceder a los datos dentro de las *variableTipoNombreEstructuraX* se usa el operador "." (como en Java), por ejemplo:

```
variableTipoEstructura1.variable1 = x;
```

Para crear otra "instancia" se usa lo siguiente dentro del cuerpo de una función (recuerden que **`main()`** es una función):

```
struct nombreEstructura variableTipoEstructura;
```

Referencias Principales[1]

- http://www.phim.unibe.ch/comp_doc/c_manual/C/CONCEPT/data_types.html
- http://www.cs.cf.ac.uk/Dave/C/
- http://www.cprogramming.com/
- http://www.its.strath.ac.uk/courses/c/tableofcontents3_1.html
- H. M. Deitel, P. J. Deitel. “Como programar en C/C++”

Programa de ejemplo

[1] Esta es una recopilación realizada a partir de diferentes páginas de Internet y otros medios con el fin único de servir como guía a los estudiantes del curso Fundamentos de infraestructura tecnológica [ISIS 1304].

El siguiente programa lee una serie de cadenas de caracteres en un vector de apuntadores a cadenas de caracteres. Después lee otra cadena y mira cuántas veces aparece esta cadena en el vector.

Después del ejemplo encuentra una serie de notas explicativas y algunos ejercicios.

Programa

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

bool comparar ( char *, char * );  //ver nota 1

int
main(int argc, char* argv[]) {
/* Suponemos cadenas de máximo 20 caracteres, por lo cual necesitamos
   vectores de char de 21 posiciones: 20 para los caracteres y uno para
   el carácter nulo del final. C no protege contra el desborde en los
   vectores: si se teclean más de 20 caracteres, los que sobren
   sobrescribirán otras variables en memoria con resultados impredecibles
*/

  char * v[5];   //ver nota 2
  char s[21];    //ver nota 3
  int i, total = 0;

  for ( i = 0; i < 5; i++){
    v[i] = (char *) malloc( 21 );     //ver notas 4 y 5
    printf( "Teclee cadena %d:", i ); //ver nota 6
    scanf( "%s", v[i] );              //ver nota 7
  }

  printf( "Teclee cadena para buscar:" );
  scanf( "%s", s );                   //ver nota 8

  for ( i = 0; i < 5; i++){
    if ( comparar( s, v[i] ) ){
      total++;
    }
  }

  printf( "La cadena %s esta %d veces en el vector\n", s, total );
  //ver nota 9

  return 0;  //ver nota 10

}

bool
comparar ( char * s, char * t ) {

  //Mientras que la primera cadena no se acabe
  while ( *s != '\0' ){  //ver nota 11
    if ( *s != *t )
      return false; //Si difieren en algún carácter, son diferentes
    else{
```

```
      s++;      //ver nota 12
      t++;
    }
  }

  //La primera cadena ya se acabó ...
  if ( *t != '\0' )
    return false; //... si la segunda no se ha acabado, son diferentes
  else
    return true;

}
```

Notas

Nota 1: esta tipo de declaraciones se llama *prototipo*. Es la forma de describirle al compilador una función que se va a utilizar pero todavía no se ha definido. En un prototipo se indica el tipo del resultado de la función, su nombre y el tipo de los parámetros que recibe.

Nota 2: `v[5]` denota que se trata de un vector de 5 posiciones. `char *` denota que el contenido de cada posición es un apuntador a un carácter. En síntesis, se trata de de un vector de apuntadores a caracteres; ahora, cada carácter puede ser el primer carácter de una cadena, en consecuencia, se puede ver como un vector de apuntadores a cadenas de caracteres.

Nota 3: este es un vector de char de 21 posiciones, luego puede almacenar cadenas de 20 caracteres (la posición 21 es para el carácter nulo).

Nota 4: note que `v` es un vector de apuntadores a cadenas, pero las cadenas en sí no existen todavía. La función `malloc` (*m*emory *alloc*ation) crea un espacio nuevo en memoria (es parecida al operador `new` de java). Su parámetro indica cuántos bytes se requieren, en nuestro caso, 21 bytes para almacenar cadenas de 20 caracteres.

Nota 5: la función malloc retorna un apuntador a `void` (`void *`), puesto que `v[i]` es de tipo `char *`, los tipos no coinciden y el compilador no permite efectuar la asignación. La expresión `(char *)` antes de malloc se denomina *casting*, y sirve para convertir entre tipos de datos; en el caso presente, convierte de `void *` a `char *`.

Nota 6: la función `printf` sirve para imprimir. No tiene un número fijo de parámetros, pero el primero siempre debe ser una cadena de caracteres. `printf` imprime directamente los caracteres que componen la cadenas, excepto los que son de la forma `%c`; estos últimos son caracteres de formato y en su lugar se imprime uno de los parámetros que viene después de la cadenas. Debe haber tantos caracteres de formato como parámetros después de la cadena. Hay una gran variedad de caracteres de formato, aquí utilizamos solo dos:

- `%s`: indica que el parámetro es un apuntador a una cadena de caracteres, y se encarga de imprimir la cadena.
- `%d`: indica que el parámetro es un número, y lo imprime.

Nota 7: la función `scanf` hace lo contrario: lee valores del teclado y los asigna a variables. Como `printf`, tiene un número variable de parámetros, y el primero siempre es una

cadena de caracteres. Los caracteres de formato también son de la forma `%c`, pero en este caso indican qué tipo de entidad se debe leer en el parámetro. El parámetro debe ser un apuntador a una variable del tipo adecuado. En nuestro caso, `v[i]` es un apuntador a una cadena de caracteres.

Nota 8: note que, en este caso, el parámetro de `scanf` es `s`. Para C un vector es un apuntador, en consecuencia, `s` también es un apuntador a una cadena de caracteres. Atención: `s[i]` es un char, pero `s` es un apuntador a char.

Nota 9: en las cadena de caracteres la secuencia `\c` indica un carácter no imprimible; por ejemplo, `\0` indica el carácter nulo, y `\n` indica cambio de línea.

Nota 10: la función `main` retorna valores al sistema operativo: 0 indica que el programa termino bien; un valor diferente de cero indica una terminación anormal.

Nota 11: `s` está apuntando a una cadena, luego `*s` es el carácter de la cadena al cual está apuntando.

Nota 12: `s` está apuntando a un cierto carácter de la cadena; `s++` incrementa el contenido de `s`, el cual es una dirección, lo cual implica que `s` queda apuntando al siguiente carácter de la cadena.

Ejercicios

1. ¿Qué ocurre si elimina la línea `#include <stdbool.h>` y compila el programa?
2. En el encabezado `string.h` se encuentra la función `strcmp` que compara dos cadenas de caracteres. En el ejemplo, incluya este encabezado, elimine la función `comparar` y remplace su uso por `strcmp`.
3. Cambie la declaración del vector `v` para que pueda tener hasta 100 cadenas. Antes del `for`, pregunte cuántas cadenas se van a teclear, y modifique el `for` acorde a esto. Si el número de entradas es mayor que 100 o menor o igual que cero, se rechaza y se vuelve a pedir
4. Modifique el programa anterior para que itere el número de veces indicado, pero, en cada iteración, pregunte por la cadena y el índice del vector donde debe leerla; es decir, el `scanf` debe leer una cadena y un entero `j`, y asignar la cadena a `v[j]`.
5. Modifique el programa anterior para que verifique si el entero `j` está en el rango válido; de no ser así, lo vuelve a pedir.
6. Note que, en `scanf`, el formato `%s` lee caracteres hasta que aparece un blanco o el carácter `'\n'`; en consecuencia, las cadenas no pueden tener blancos en su interior. Para superar este inconveniente, escriba una función `leerCadena` que lee caracteres hasta que aparece `'\n'`. Esta función debe tener como parámetro un apuntador a `char`. Para escribir esta función, puede usar la función `getchar( )`, la cual retorna un carácter del teclado.
7. Investigue qué hace la función `gets`.