

Taller de programación: C - Parte 2

Autor: Luis A. Muñoz - UPC- 2020



Este documento resume las consideraciones que debemos de tomar al momento de definir funciones, así como trabajar con char y con strings.

Funciones definidas por el usuario

Un programa en C básicamente consiste en la edición de una función llamada `main`:

```
ini main()
{
    // Su código va aquí...
    return 0;
}
```

Para escribir un código que realice alguna tarea, se incluyen sentencias (o sea, eso que termina en un ";") que consisten en:

- Instrucciones del lenguaje C
- Funciones de las librerías

Por ejemplo, `break` es una *instrucción* porque realiza una acción sobre el código (detiene un lazo). Lo mismo sucede con `if` que afecta la secuencia de ejecución de las sentencias.

Por otro lado, `printf()` es una función de la librería `stdio.h` que debe ser incluida al inicio del programa para que pueda ser utilizada. Una función se reconoce porque siempre tiene `()` al final, así no contengan nada.

Entonces, una sentencia en C es una colección de instrucciones y funciones de la forma:

```
if (a % 2 == 0) print("%d es par\n", a);
```

Reconozca que términos son instrucciones y que otros son funciones. Este reconocimiento es importante para entender cómo es que se escribe un código de programación.

Ahora, eso de `a % 2 == 0` no resulta muy agradable porque aunque sabe lo que hace, no resulta tan claro qué es lo que está realizando. **Y esto es importante: ¿No le gustaría cambiar eso por algo más legible, como `es_par(numero)`?** Pienselo por un momento. Su código quedaría de esta forma:

```
if es_par(a) printf("%d es par\n", a);
```

Si compara ambas sentencias estará de acuerdo conmigo que esta última versión es mucho más clara. Si desea hacer esto debe de incuir una *nueva función* llamada `es_par()` que tomará un valor de entrada (estos se llaman *argumentos de una función*) y retornará un *valor de salida* que indique con un `0` si el parámetro de entrada no es par, y con un `1` si el parámetro de entrada es par.

Es decir, necesita una función definida por usted, el usuario. Esto no solo hace su código más legible, sino que establece una estrategia de solución a cualquier tipo de problema del tipo *divide y vencerás*, en donde un problema complejo se divide en problemas más simples para luego integrar todo en un solo proceso.

Entonces, un código es la solución de la función `main`, compuesta de sentencias que consistirán de instrucciones, funciones de las librerías y funciones definidas por el usuario.

Funciones: parametros de entrada y salida

Las funciones, en términos generales, toman valores que le son pasados como entradas y retorna *una* salida. Esto debe de especificarse al momento de plantearse una función.

Por ejemplo, para la función `es_par()` , ¿con qué tipo de dato necesitamos alimentar a la función? Para saber la paridad de un número, este debe de ser entero. Por otro lado, nuestra función nos responderá Si/No y esto en C lo definimos con los valores 1/0, que son valores enteros. Todo esto lo definiremos de la siguiente manera:

```
int es_par(int);
```

Esto es lo que se llama el *prototipo de una función*, es decir, la definición de sus entradas y salidas, así como el nombre de esta. La salida será el `int` que está a la izquierda del nombre, mientras que el `int` entre paréntesis será la entrada de esa función.

Nota: Una función puede no devolver nada. Para ese caso no se puede dejar en blanco al estilo `funcion()` , sino hay que especificar que no se va a devolver *nada* con la palabra `void` de la forma `void funcion()` .

Podemos tener una función que tome varios parámetros de entrada (separados por ",") pero no podemos especificar más de un parámetro de salida.

Considere el siguiente ejemplo:

```
In [ ]: #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int suma(int num1, int num2)
{
    return num1 + num2;
}

int main()
{
    int num1, num2;

    printf("num1: "); scanf("%d", &num1);
    printf("num2: "); scanf("%d", &num2);

    printf("%d + %d = %d\n", num1, num2, suma(num1, num2));

    return 0;
}
```

```
num1: 3
num2: 5
3 + 5 = 8
```

Observe que al principio del script se ha definido una función de la forma `int suma(int num1, int num2)` . Aquí se está resolviendo directamente la función (como si fuera la función `main` solo que tiene otro nombre y parámetros de entrada). Al finalizar, la función debe de devolver un valor entero (igual a la suma de los argumentos de entrada) y esto lo realiza con la instrucción `return` . *La función retorna un valor, no lo imprime* porque la salida de esta será la entrada de otra función, en este caso de la función `printf()` que será la encargada de imprimir el valor retornado por `suma()` .

Prototipo de la función

En el script anterior la función se ha escrito y resuelto antes de la función `main()` . Esto es así por que el compilador revisa el archivo de arriba a abajo y va decodificando cada una de las sentencias en el programa para luego convertirlas en comandos binarios del procesador (Assembler). Esto quiere decir que si se escribe la función *debajo* de la función `main()` , cuando llegue a la sentencia

```
`printf("%d + %d = %d\n", num1, num2, suma(num1, num2));`
```

no reconocerá la función `suma()` (¿ahora entiende porque hay que escribir `#include<stdlib.h>` al principio del programa?).

Aunque escribir las funciones antes de la función principal `main()` funciona, no es la mejor practica de programación. Lo mejor es *prototipar la función* antes de la función `main` . Un prototipo de la función es la descripción operativa de la función, es decir cual es su nombre, cuantos y que tipo de argumentos de entrada tiene y que valor retorna. Entonces escribiríamos antes de la función `main()` :

```
int suma(int, int);
```

Esto no solo es más ordenado sin que permite entender como es que funciona el proceso de compilación. Ahora, al compilar el programa, el compilador recorre la definición anterior y será suficiente para poder reconocer la sentencia `printf("%d + %d = %d\n", num1, num2, suma(num1, num2));` ya que tanto `num1` como `num2` son variables de tipo `int` y el resultado de esta función será mostrada por el `printf` con el carácter de conversión `%d` lo que es correcto porque la función `suma()` retorna un `int` .

O sea, el compilador no analiza la calidad del código o si este funciona o no; lo que hace es ver si todos los tipos de datos coinciden entre si. Utilizando un programa llamado *parser* corta el código donde haya ":" y el compilador va uniendo las costuras por los tipos de datos que pasan de una función a otra, conectando direcciones de memoria de lectura y escritura hasta construir un código binario.

En el siguiente script se ha prototipado la función `resta(int, int);` y esta se resuelve *debajo* de la función `main()` . Note que el prototipo de la función termina con ":" porque es una sentencia.

```
In [ ]: #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

// PROTOTIPO DE FUNCIONES
int resta(int, int);

int main()
{
    int num1, num2;

    printf("num1: "); scanf("%d", &num1);
    printf("num2: "); scanf("%d", &num2);

    printf("%d - %d = %d\n", num1, num2, resta(num1, num2));

    return 0;
}

// FUNCIONES
int resta(int num1, int num2)
{
    return num1 - num2;
}
```

```
num1: 10
num2: 3
10 - 3 = 7
```

El arte de trabajar con funciones: de lo general a lo particular

Tiene un problema de programación: escriba un programa que imprima todos los números primos en el rango de 1 a 100. ¿Por donde empieza?

La mayoría de estudiantes de programación contestarán por resolver con un código cómo hallar los números primos. Esta aproximación no es más que otra forma de expresar el problema original por lo que no se ha avanzado nada (y probablemente no se llegue a ningún lado).

Entonces reduzcamos el problema: ¿cómo puedo saber si un número es primo? Entonces intentaremos resolver ese problema en la función `main()` para que sea parte de una operación condicional con un `if` que estará dentro de un lazo `for` que barrerá los 100 primeros dígitos. Así puesto no se ve más sencillo.

Aquí es que hay que pensar en funciones. ¿Y si se escribe una función que halle los números primos entre 1 y 100? Pero eso es lo que va a hacer la función `main()`, ¿no? ¿Entonces qué tal una función que me retorne con un 1 o 0 si un número es primo o no? Eso tiene mucho más sentido.

¡Entonces resolvemos la función primero! No.

El truco para trabajar con funciones es prototiparlas y asumir que funcionarán en el futuro y utilizarlas en la función `main()` como si ya estuvieran resueltas. Esto puede sonar extraño pero es una buena práctica de programación, ya que el flujo de la programación va a la par con la del compilador: lo importante es la definición.

Entonces: ¿cuál sería el prototipo de la función que me diga si un número es primo o no? Primero un buen nombre: `es_primo()`. Luego: ¿qué parámetros de entrada tendrá? Un valor entero y retornará 1 ó 0. Entonces queda de la siguiente forma:

```
int es_primo(int);
```

Listo. Ahora liberemos el poder de las funciones. ¿Eso funciona? Supongamos que sí: le colocamos un número entre los paréntesis y me dirá si o no dependiendo de si ese número es primo. Entonces `main()` queda bastante simple:

```
int main()
{
    for (int n = 1; n < 100; n++) {
        if (es_primo(n)) {
            printf("%d\n", n);
        }
    }
}
```

¿Sencillo, o no? Un lazo `for` de 1 a 100 que verifica si el número es primo utilizando la función `es_primo()`. Si es así imprimimos el número y si no iteramos al siguiente. No solo tenemos separado el código en módulos sino que hemos ganado claridad (¿hay algo más claro que `if (es_primo(num))`?). Es decir, que cosa es más clara:

```
if (num % 2 == 0)
```

contra:

```
if (es_par(n))
```

Las funciones aclaran el código, facilitan su lectura y convierten un problema complicado en uno mas sencillo. Ahora mi problema original de imprimir un listado de todos los números primos en el rango de 1 a 100 se ha reducido a saber si un número es primo o no.

include <stdio.h>

include <stdlib.h>

include <string.h>

include <time.h>

include <math.h>

```
// PROTO TIPO DE FUNCIONES int es_primo(int);  
  
int main() { for (int n = 1; n < 100; n++) { if (es_primo(n)) { printf("%d\n", n); } } }  
  
// FUNCIONES int es_primo(int num) { if (num < 2) { return 0; } else if ((num == 2) || (num == 3)) { return 1; } else  
{ for (int div = 2; div <= (int)sqrt(num); div++) { if (num % div == 0) { return 0; } } }  
  
return 1;  
}
```

La función parte separando el caso de los números menores a 2 que no son primos (`return 0`). Luego si el número es 2 o 3 indicará que sí son primos (`return 1`). Para los siguientes valores debemos de considerar si tienen factores primos (o sea, si son divisibles por otros números) utilizando una división por tentativa entre 2 y la raíz del número. Si se encuentra un divisor del número, entonces no será primo. Por eso la sección:

```
if (num % div == 0) {
    return 0;
}
```

Ya que hay un divisor y por lo tanto `num` no es un número primo. Si se acaba al lazo y barre con todos los posibles divisores, entonces ha agotado todas las posibilidades y ningún valor es divisor de `num`, así que este valor debe de ser primo (el `return 1` que esta al final).

Punteros: la parte fea de C

Generaciones de estudiantes tienen muy feos recuerdos de una cosa llamada *punteros* (no ha visto una verdadera cara de asco hasta que le pregunte a un estudiante de programación que son los "punteros a punteros"). Pero la idea es bastante sencilla:

- Un puntero es una variable que apunta a otra.
- Como esta "apuntando a una variable" que está en memoria, lo que almacena es una dirección de memoria.
- Entonces, los punteros guardan direcciones de memoria.

Pero eso es solo el comienzo. Considere el siguiente código:

```
In [ ]: #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    int a = 100;
    int ptr_a;

    ptr_a = &a;

    printf("Valor de a: %d\n", a);
    printf("Direccion de a: %d\n", &a);
    printf("Valor de ptr_a: %d\n", ptr_a);

    return 0;
}
```

```
Valor de a: 100}
Direccion de a: 6487576
Valor de ptr_a: 6487576
```

Se ha declarado la variable `a` y se le ha asignado el valor de 100. Recuerde que la variable `a` tiene una dirección de memoria asociada a la que puede tener acceso con el operador `&`. Se declara una variable entera `ptr_a` que guardará la dirección de `a`:

```
int ptr_a;
ptr_a = &a
```

Ahora, imprimimos la variable, su dirección y la valor que guarda la misma dirección. Hasta ahora nada raro.

Ahora, pongamos un `*` en la variable que guarda la dirección de `a` y luego la asignación:

```
int *ptr_a
ptr_a = &a
```

Ese `*` hace la mágia...

```
In [ ]: #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int main()
{
    int a = 100;
    int *ptr_a;

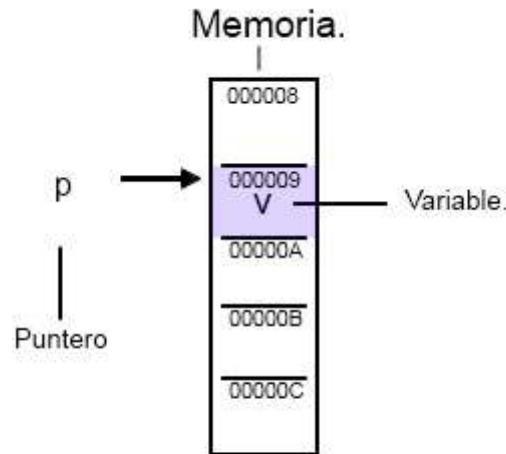
    ptr_a = &a;

    printf("Valor de a: %d\n", a);
    printf("Direccion de a: %d\n", &a);
    printf("Direccion de a: %d\n", ptr_a);
    printf("Valor de ptr_a: %d\n", *ptr_a);

    return 0;
}
```

```
Valor de a: 100
Direccion de a: 6487572
Direccion de a: 6487572
Valor de ptr_a: 100
```

Debe leer ese * como si fuese una flecha que apunta a algo. Esa es la forma de definir un puntero. Un puntero es una variable que apunta a otra. Si apuntara a un variable de tipo int deberá definirse como int ; si apuntara a una variable tipo float , el puntero deberá ser float (esto quedará claro al revisar los arreglos). Al momento de hacer la asignación ptr_a = &a se esta asignando la dirección de la variable al puntero. Si imprime ptr_a (es decir, lo que almacena el puntero) verá la dirección de a (lo mismo que &a). Pero si imprime *a (es decir, a qué esta *apuntando* el puntero ptr_a) verá el valor de la variable a .



`p = 0x000009h`

`0x000009h = Posición en memoria
que ocupa v`

Esto es grande... no parece pero lo es. Porque ahora puedo saber que hay en una posición de memoria si tengo su dirección. Es como conocer el teléfono de una variable. Y eso es todo lo que hay que saber sobre los punteros... por el momento.

Parametros por valor o por referencia

¿Que cosa he ganado con los punteros? Recuede el prototipo de la función:

```
int suma(int, int);
```

Cuando llamemos a la función tendremos que reemplazar los argumentos definidos como `int` por valores del mismo tipo al momento de llamar a la función de la forma:

```
int a = 10, b = 5;  
suma(a, b);
```

Esto se conoce como *pasar parámetros por valor* y suena bastante lógico. Pero hay otra forma de hacer lo mismo. Recuerde: una variable tiene un valor y una dirección de memoria. ¿Y si le pasamos a la función las direcciones de memoria de las variables `a` y `b`. Así no haremos una copia de estas variables en la memoria (y eso es hacer uso eficiente de un recurso escaso en un sistema microprocesado con 8Kb de RAM, por ejemplo) sino que le decimos a la función: "alli te paso los teléfonos de las variables para que los ubiques y sepas cuantos valen". Entonces, llamaremos a la función de la forma:

```
int a= 10, b = 5;  
suma(&a, &b);
```

Ahora, la función no recibirá los valores `10` y `5` (es decir, no estamos pasando los parámetros por valor) sino las direcciones binarias donde se encuentra el número `10` y el número `5` (es decir, estamos *pasando los parámetros por referencia*). ¿Y que objeto nos permite tomar direcciones de variables y recuperar lo que hay en esta dirección? Un puntero:

```
int suma(int*, int*);
```

Ahora el prototipo de la función define que la función espera recibir dos direcciones de memoria y trabaja con los punteros para obtener el valor de lo que está en las direcciones recibidas como parámetros.

```
In [ ]: #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

// PROTOTIPO DE FUNCIONES (pasar por referencia)
int suma(int *, int *);

int main()
{
    int num1, num2;

    printf("num1: "); scanf("%d", &num1);
    printf("num2: "); scanf("%d", &num2);

    printf("%d + %d = %d\n", num1, num2, suma(&num1, &num2));

    return 0;
}

// FUNCIONES
int suma(int *num1, int *num2)
{
    // Recibo punteros y * permite saber que hay en estas direcciones
    return *num1 + *num2;
}
```

num1: 10
num2: 5
10 + 5 = 15

Como retornar varios parametros

Ok. Los punteros me permiten ahorrar 8 bytes de memoria... no parece mucho. Pero ahora podemos superar un grave inconveniente de las funciones: pueden recibir muchos parametros de entrada, pero solo pueden retornar un valor.

Suponga que quiere escribir una función que tome dos coordenadas cartesianas y las quiere convertir en coordenadas polares (o lo que es lo mismo, convertir un número complejo a polar). Esto requiere que la función reciba dos argumentos de entrada pero también que retorne dos valores de salida (el módulo y el ángulo del número complejo).

Y aqui es donde los punteros salen a relucir. Considere el siguiente prototipo:

```
void cart2pol(int, int, float*, float*);
```

Esto define una función que convierte de "cartesiano a polar" y no retornará ningún valor de salida (de allí el tipo de salida `void` que significa en inglés "vacío"), pero recibirá cuatro parámetros de entrada: los dos primeros pasados por valor (las coordenadas del plano cartesiano) y dos parámetros adicionales pasados por referencia (ambos de tipo `float`) que le servirán a la función como las direcciones de memoria donde escribirá los resultados. Entonces, se le está pasando a la función los datos que necesitará para hacer los cálculos y las dos direcciones para que retorne sus salidas (por eso `void` ya que formalmente no retornará un "valor"). Entonces, al llamar a la función se escribirá:

```
cart2pol(rel, img, &mod, &ang);
```

Note que al llamar a la función no hay algo como `var = cartspol(...)` porque esta no retornará nada. Los datos resultantes de la función estarán en las variables `mod` y `ang` que han sido pasadas como parámetros de entrada (que seguirán siendo *de entrada* pues se ingresan los valores de direcciones).

El siguiente código resume todo lo anterior:

```
In [ ]: #define _USE_MATH_DEFINES
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>

// PROTOTIPO DE FUNCIONES (pasar por referencia)
void cart2pol(int, int, float *, float *);

int main()
{
    int rel, img;
    char ch;
    float mod, ang;

    printf("Ingrese un numero complejo [a+bi]: ");
    scanf("%d%c%d%c", &rel, &ch, &img, &ch);

    cart2pol(rel, img, &mod, &ang);

    printf("Modulo: %.4f\nAngulo: %.4f\n", mod, ang);

    return 0;
}

// FUNCIONES
void cart2pol(int rel, int img, float *mod, float *ang) {
    *mod = sqrt(pow(rel, 2) + pow(img, 2));
    *ang = (180/M_PI) * atan2(img, rel); // resultados a grados
}
```

Ingrese un numero complejo [a+bi]: 3+4j

Modulo: 5.0000

Angulo: 53.1301

¿Que es eso de `#define _USE_MATH_DEFINES`? Es la activación de un atributo que habilita el uso de los macros de la librería `math.h` que incluye etiquetas para algunas constantes numéricas como `M_PI` para el valor de π (las funciones trigonométricas de `math.h` operan con radianes tal y como Dios manda, por lo que se convierten los resultados a grados).

Note por otro lado que la función no tiene la instrucción `return` ya que no retorna nada.

Letras en C: el tipo char

Hasta ahora hemos tratado con variables que contienen números tanto enteros como reales. Pero no con las variables de tipo `char`. Formalmente, un `char` es un valor numérico que representa una letra bajo un estandar de codificación llamado ASCII. Este define los códigos para cada letra del alfabeto inglés estandar (y que define valores de hasta 7 bits). El compilador reserva 1 byte (8 bits) para estos datos.

Nota: Un dato tipo `char` se especifica con comillas simples '''. No utilice comillas dobles. Esta es una fuente de error muy común en el código.

En la secuencia de valores `char` estan los números como signos de escritura (es decir, 0 y '0' son cosas diferentes), los signos de puntuación, las letras mayúsculas y minúsculas y caracteres especiales (como '\n'). Una cosa a recordar es que en la secuencia de letras en la codificación ASCII, las distancia que separa el grupo de las letras mayúsculas de las minúsculas es de 32 valores:

```
A B C ... ----- a b c ...
```

Entonces, si se realiza la operación `'A' + 32`, se obtendrá `a`, y si se hace la operación `'a' - 32` se obtendrá `A`. Podemos definir una función que convierta un carter en minúscula a mayúscula. Este tomará el carácter y verificará si está en el rango de las letras minúsculas. De ser así le restará 32 y retornará el resultado. De lo contrario retorna el dato de entrada igual como ingreso.

```
In [12]: #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

char to_upper(char);

int main()
{
    for (char letra='!'; letra <= 'b'; letra++) {
        printf("%c ", to_upper(letra));
    }
    printf("\n");

    return 0;
}

char to_upper(char c) {
    if (c >= 'a' && c <= 'z') {
        return c - 32;
    } else {
        return c;
    }
}
```

! " # \$ % & ' () * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @ A B C D E F G
H I J K L M N O P Q R S T U V W X Y Z [\] ^ _ ` A B

Note que la verificación del rango de minusculas `if (c >= 'a' && c <= 'z')` opera con las letras como si fueran valores enteros.

strings: una colección de chars

Un `string` se define como una colección de caracteres, algo que se conoce formalmente como un *arreglo de caracteres*. No es un tipo de datos, es una colección de datos. Un `string` se especifica con comillas dobles "" y ocupa una secuencia consecutiva de posiciones de memoria. Si un `string` tiene 10 caracteres, este ocupará 11 posiciones de memoria pues la última posición está reservada para un carácter especial llamado `NULL` ('\0'). Esto indica en qué momento ha terminado la secuencia.

Esto es importante ya que el control de la longitud de una cadena es controlada por el programador. Cuando se define una colección de datos se utilizan los caracteres []. Por ejemplo:

```
char nombre[] = "Alan Brito"
```

Si no se especifica el número de caracteres entre los [], el compilador le asignará el valor 11 para este caso, ya que la palabra tiene 10 caracteres más el carácter '\0'.

Considere el siguiente script:

```
In [13]: #include <stdio.h>
#include <stdlib.h>
#include <string.h> // strlen, strcat, strrev, strcmp, strcpy
#include <time.h>
#include <ctype.h>

int main()
{
    char nombre[] = "Elvio Lado";

    for (int i = 0; i < 11; i++) { // sizeof, strlen
        printf("nombre[%d] = %c\n", i, nombre[i]);
    }

    return 0;
}
```

```
nombre[0] = E
nombre[1] = l
nombre[2] = v
nombre[3] = i
nombre[4] = o
nombre[5] =
nombre[6] = L
nombre[7] = a
nombre[8] = d
nombre[9] = o
nombre[10] =
```

En un `string`, cada uno de los caracteres ocupa una posición de memoria, aunque se puede utilizar un *puntero local* que inicia en 0 para el primer carácter. Este es el *índice* del arreglo, en este caso, el índice del `string`. El lazo `for` gestiona el valor de este índice para tener acceso a cada uno de los caracteres con el formato `string[i]`.

Un error típico con el uso de los `string` es el error por *desboramiento*. Por ejemplo:

```
In [ ]: #include <stdio.h>
#include <stdlib.h>
#include <string.h> // strlen, strcat, strrev, strcmp, strcpy
#include <time.h>
#include <ctype.h>

int main()
{
    char nombre[] = "Elvio Lado";

    for (int i = 0; i < 20; i++) { // sizeof, strlen
        printf("nombre[%d] = %c\n", i, nombre[i]);
    }

    return 0;
}
```

```

nombre[0] = E
nombre[1] = l
nombre[2] = v
nombre[3] = i
nombre[4] = o
nombre[5] =
nombre[6] = L
nombre[7] = a
nombre[8] = d
nombre[9] = o
nombre[10] =
nombre[11] =
nombre[12] = ó
nombre[13] =
nombre[14] =
nombre[15] =
nombre[16] = ñ
nombre[17] = !!
nombre[18] = í
nombre[19] =

```

Se observa que el programa no falla y el compilador no retorna un error. Si el lazo `for` supera la longitud de la cadena se siguen leyendo las siguientes posiciones de memoria. Por lo tanto es necesario conocer la longitud de la cadena. Algunas funciones útiles para la gestión de cadenas están disponibles en la librería `string.h`. Por ejemplo, la función `strlen()` retorna la longitud de una cadena. La función `sizeof()` también hará algo parecido pues retorna el número de bytes de un objeto y en este caso el arreglo de caracteres tiene un tamaño, pero retornará un numero igual a `strlen() + 1` pues estará considerando el carácter '`\n`'.

Pero volvamos a la función `strlen()`. ¿Cuál es el proceso que le permite a la función obtener el tamaño de una cadena? Porque recorre toda la cadena contando cuantos pasos debe saltar entre el primer carácter y el carácter '`\0`'. Es decir, que a la función `strlen()` se le pasa la dirección de memoria del primer carácter para que pueda hacer el cálculo. Esto es un proceso automático: cuando se define un `string` de la forma:

```
char nombre[] = "Elvio Lado"
```

la variable `nombre` tiene la dirección que ocupa en memoria `nombre[0]`, es decir, la letra 'E' inicial. Esto será de capital importancia para entender como funcionan los arreglos.

Ahora considere el siguiente código:

```
In [4]: #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ctype.h> // toupper, tolower, isalpha, isdigit, isalnum, ispunct, is
space

int main()
{
    int i = 0;
    char nombre[] = "Elvio Lado";    // string

    while (nombre[i] != '\0') {
        printf("%c", toupper(nombre[i]));
        i++;
    }
    printf("\n");

    return 0;
}
```

ELVIO LADO

Esta vez tenemos un lazo `while` que condiciona las iteraciones mientras el carácter que está siendo leído con el índice `i` no sea '\0'. (Pruebe reemplazar el carácter '\0' por NULL y verá que el programa también responderá correctamente, pero el compilador generará un Warning que indica que no se puede *comparar un puntero con un entero*. Esto deja claro que `nombre[i]` es un puntero).

Ahora, recuerde aquello de *pasar parámetros por referencia*. La función `toupper()` es parte de la librería `ctype.h`, que tiene funciones para ser utilizadas con un `char`. Esta función convierte una letra en mayúsculas ("to upper", a mayúsculas). Recibe un puntero, es decir, la dirección de memoria donde está un carácter. Esta función hace lo mismo que hicimos en nuestra función `to_upper()` que definimos líneas arriba (pero aquí pasamos los parámetros por valor).

Entonces (y esto es importante), cuando se trabaja con los `strings`, se trabaja con posiciones de memoria y con parámetros pasados por referencia de forma automática. Esto permite entender porque las funciones de la librería `string.h` tienen una forma de uso tan caprichosa.

Primero: una cosa que suele confundir es la imposibilidad de escribir la siguiente instrucción:

```
char nombre[11];
nombre = "Elvio Lado";
```

La segunda línea fallara. Ahora que sabe que `nombre` es realmente la dirección de memoria del primer elemento de ese arreglo de 11 caracteres, entenderá que no se puede hacer esa asignación.

Es por eso que tenemos la función `strcpy()` que copia una cadena en otra. Entonces escribimos lo siguiente:

```
nombre = strcpy("Elvio Lado");
```

Y nos llevamos la triste sorpresa que no funciona. ¿Por qué? Pieselo un poco antes de continuar...

Pues porque las funciones no pueden devolver más de un valor y una cadena tiene muchos valores tipo `char`. Entonces se tiene que pasar la dirección del `string` resultante como parámetro de entrada. La forma correcta de llamar a la función `strcpy()` es :

```
strcpy(nombre, "Elvio Lado");
```

Esto funcionará correctamente. Esta es una fuente de confusión muy común entre los estudiantes de C (y algunos un poco más expertos), pero esto queda claro y explicado (y no hay que tener que memorizar nada) si se tiene claro lo que es pasar parámetros por referencia y que los `string` (y arreglos en general) solo tienen la información de la posición de memoria del primer elemento.

```
In [ ]: #include <stdio.h>
#include <stdlib.h> // atof, atoi
#include <string.h> // strlen, strcat, strrev, strcmp, strcpy
#include <time.h>

int main()
{
    int i;
    char nombre[25], apellido[25];
    char nombre_completo[50];

    printf("Ingrese su nombre: ");
    scanf("%s %s", nombre, apellido);

    strcat(nombre, " ");
    strcat(nombre, apellido);

    printf("Nombre: %s\n", nombre);

    strrev(nombre);
    printf("Nombre Invertido: %s\n", nombre);

    return 0;
}
```

Ahora puede entender lo que hacen las funciones de los `string`. ¿Qué hace `strcat()`? Concatena dos cadenas (las une en una sola). En este caso la cadena `nombre` se junta con la cadena '' y el resultado se guarda nuevamente en `nombre`. Luego se hace la misma operación entre `nombre` y `apellido` y todo se guarda nuevamente en `nombre`.

Lo mismo pasa con `strrev()` que invierte una cadena. Se le pasa un solo parametro: el `string` a invertir y lo que retornará será el mismo `string` invertido y no uno nuevo; el `string` se invierte en sus mismas posiciones de memoria.

La mejor forma de pedir información al usuario: `scanf` con `%s`

Ahora que sabemos todas estas cosas de los `string`, podemos volver a darle una mirada a `scanf` y la forma de pedirle datos al usuario. Considere lo siguiente:

```
Ingrese el radio: radio
```

Estas cosas suceden...

Si en su programa escribió `scanf(%d)` pues tendrá un problema. Pero hay otro problema más grave aún:

```
Ingrese radio: 5.9
```

El cálculo no será correcto porque el `%d` que interpreta la entrada en el `scanf()` convertirá ese 5.9 en el número 5 (es decir, un `int`).

Lo ideal es recibir siempre un `string` y luego interpretarlo correctamente. Considere el siguiente código:

```
In [ ]: #define _USE_MATH_DEFINES
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <math.h>

float area_circ(float);
int es_dígito(char*);

int main()
{
    char input[25];

    do {
        printf("Ingrese radio: "); scanf("%s", &input);
    } while (!es_dígito(input));

    printf("Área: %.2f\n", area_circ(atof(input)));

    return 0;
}

float area_circ(float r) {
    return M_PI * pow(r,2);
}

int es_dígito(char * string) {
    for (int i=0; i < strlen(string); i++) {
        if (!isdigit(string[i])) {
            return 0;
        }
    }
    return 1;
}
```

La función `main` tiene un lazo `do while` que pide los datos y continua pidiendolos mientras no sea un ingreso correcto (en este caso mientras *no* sea un dígito (`!es_digito(input)`)). La función `es_digito()` es una función definida que justo hace lo que su nombre indica: retorna 1 si la cadena pasada como argumento de entrada es un "número escrito".

La ventaja de utilizar `scanf(' %s')` es que lo que el usuario ingresa por teclado se interpreta como un `string` para poder evaluarlo posteriormente. Así, ya no nos interesa si ingresa 'radio', '5' o '5.9'.

Ahora, la función `es_digito` recibe una cadena (es decir, un puntero para la dirección del primer carácter) que barrerá todos los caracteres que serán procesados por la función `isdigit()` que retorna 1 o 0 si el `char` ingresado como parámetro es un dígito en el rango '0' a '9' (y para suerte nuestra, incluye el carácter '.'). Si algunos de los caracteres no es un dígito (`if (!isdigit(string[i]))`) retorna 0. Si barre todos los caracteres y todos pasan la prueba, es un dígito.

Ahora, no sabemos si es un `int` o `float`, así que apostamos a seguro y suponemos que es un `float` por lo que utilizamos la función `atof()` (`ascii to float`) para convertir el `string` validado en un dato tipo `float`, para pasarlo a la función `area_circ` que retorna el área del círculo.

¡Así que hemos hecho que el control de los datos de entrada por parte del usuario sea indestructible!
Felicitaciones.

Proyecto: Tres en Raya

Queremos hacer un programa que juegue el Tres en Raya entre el usuario y la computadora. Vamos a basarnos en las cadenas de caracteres para resolver este problema.

Nuestro tablero tendrá la siguiente forma:

```
1 | 2 | 3
-----
4 | 5 | 6
-----
7 | 8 | 9
```

Y las jugadas las representaremos por una cadena de 9 caracteres. Por ejemplo, si nuestro tablero tiene la jugada:

```
0 |   | X
-----
| 0 |
-----
|   | X
```

Tendremos el siguiente estado del tablero:

```
board = "0_X_0___X"
```

Se han colocado unos "_" para indicar que son espacios en blanco. En el código están realmente en blanco.

Vamos a definir una función llamada `void show_board(char*)`. Esta función muestra el tablero utilizando una secuencia de `printf` con los datos de la cadena `board`. Pasamos el tablero por referencia de forma automática como argumento, como sucede con los arreglos o strings.

La función `show_board()` incluirá la instrucción `system("cls")`. Esta función (disponible gracias a la librería `conio.h`) nos permite ejecutar algunas instrucciones en la consola de Windows; en este caso, la instrucción "`cls`" de *clear screen*, por lo que se limpia la pantalla y el efecto es que los caracteres aparecen sobre el tablero que permanece inmóvil.

Vamos a definir dos funciones adicionales: `player_wins(char*)` y `pc_wins(char*)` que verifican si uno de los dos jugadores ha ganado. Estas funciones son bastante sencillas pues verifican todas las posibles combinaciones que hacen que gane uno de los dos jugadores. Pero al tenerlos como funciones hará que nuestro código sea más legible y fácil de resolver, ya que el lazo del juego tendrá la forma:

```

while (!player_wins() && !pc_wins()) {
    // El juego continua porque ninguno de los jugadores ha ganado
    .
    .
    .
}
```

En este código no nos vamos a complicar: el jugador humano siempre inicia con el carácter 'X'. Ingresa uno de los números de posición pos (entre 1 y 9) y esto modifica la cadena tablero al insertar un 'X' en la posición pos-1 (ya que la cuenta de la cadena inicia en 0). Luego la PC juega generando valores aleatorios entre 1 y 9 y verifica que esta posición no esté ocupada en el tablero (gracias a un lazo do while).

Una función adicional nos va a permitir el control de las posiciones válidas: `is_free(char*, int)` . Esta función nos retornará un valor 1 ó 0 si una posición en el tablero está libre.

El lazo se rompe si uno de los jugadores ha ganado y se tiene un lazo externo que controla si se quiere jugar otra partida. Aquí hay un truco bajo la manga:

Desea jugar otra partida [S/N]:

El usuario puede ingresar 's', 'S', 'n' o 'N', así que hay muchos casos a considerar. Esto se puede solucionar de la forma:

```

if (toupper(input[0]) == 'N') {
    quit_game = 1;
}
```

Esto convierte el primer char del string ingresado a mayúscula y lo compara con 'N'. Así, no importará si ingresó 'n' o 'N', 'no' o 'No'. Pero hay una línea anterior extraña que merece una explicación:

```
fflush(stdin);
```

Antes de ganar la partida el jugador ingresa su jugada y para hacerlo presiona un número y ENTER. Por razones técnicas que van más allá de nuestro alcance, este carácter ENTER se queda en el flujo de entrada de datos estándar (llamado `stdin`); esto es, el buffer del teclado. Entonces cuando llega la pregunta "Desea jugar otra partida [S/N]:" el juego arranca nuevamente porque ese ENTER ingresa al `scanf()`. Entonces se debe

```
In [ ]: #include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <conio.h>
#include <ctype.h>

// Prototipo de la funcion
void show_board(char*);
int player_wins(char*);
int pc_wins(char*);
int is_free(char*, int);

// main
int main()
{
    char board[] = "         ";
    int pos, quit_game = 0;
    char input[10];

    srand(time(NULL));

    // Mientras este jugando...
    while (!quit_game) {
        while (!player_wins(board) && !pc_wins(board)) {
            // ... se limpia la pantalla y se muestra el tablero
            show_board(board);

            // El jugador humano empieza primero y se controla
            // que ingrese una posicion libre
            do {
                printf("\n\nPosicion [1-9]: ");
                scanf("%d", &pos);
            } while (pos < 1 || pos > 9 || !is_free(board, pos));

            // Marcar la posicion con X
            board[pos-1] = 'X';

            // Juega L PC...
            do {
                pos = 1 + rand() % 9;
            } while (!is_free(board, pos));

            // Marcar la posicion con O
            board[pos-1] = 'O';
        }

        // Uno de los jugadores ha ganado
        show_board(board);

        if (player_wins(board)) {
            printf("\nUd. ha ganado!\n");
        } else {
            printf("\nLa PC ha ganado...\n");
        }
    }
}
```

```

    // Controlar si sale del juego
    fflush(stdin);
    do {
        printf("Desea continuar [S/N]: ");
        scanf("%s", &input);
    } while (toupper(input[0]) != 'S' && toupper(input[0]) != 'N');

    // Si ingresa N, salir del juego
    if (toupper(input[0]) == 'N') quit_game = 1;

    // Si ingresa S, Limpiar el tablero
    if (toupper(input[0]) == 'S') strcpy(board, "         ");
}

return 0;
}

// Funciones
void show_board(char *tab) {
    system("cls");
    printf("\n");
    printf("\t TRES EN RAYA\n");
    printf("\t ======\n\n");

    printf("\t %c | %c | %c\n", tab[0], tab[1], tab[2]);
    printf("\t ----- \n");
    printf("\t %c | %c | %c\n", tab[3], tab[4], tab[5]);
    printf("\t ----- \n");
    printf("\t %c | %c | %c\n", tab[6], tab[7], tab[8]);

}

int player_wins(char* tab){
    if (tab[0]=='X' && tab[1]=='X' && tab[2]=='X') return 1;
    if (tab[3]=='X' && tab[4]=='X' && tab[5]=='X') return 1;
    if (tab[6]=='X' && tab[7]=='X' && tab[8]=='X') return 1;
    if (tab[0]=='X' && tab[3]=='X' && tab[6]=='X') return 1;
    if (tab[1]=='X' && tab[4]=='X' && tab[7]=='X') return 1;
    if (tab[2]=='X' && tab[5]=='X' && tab[8]=='X') return 1;
    if (tab[0]=='X' && tab[4]=='X' && tab[8]=='X') return 1;
    if (tab[2]=='X' && tab[4]=='X' && tab[6]=='X') return 1;
    return 0;
}

int pc_wins(char* tab) {
    if (tab[0]=='O' && tab[1]=='O' && tab[2]=='O') return 1;
    if (tab[3]=='O' && tab[4]=='O' && tab[5]=='O') return 1;
    if (tab[6]=='O' && tab[7]=='O' && tab[8]=='O') return 1;
    if (tab[0]=='O' && tab[3]=='O' && tab[6]=='O') return 1;
    if (tab[1]=='O' && tab[4]=='O' && tab[7]=='O') return 1;
    if (tab[2]=='O' && tab[5]=='O' && tab[8]=='O') return 1;
    if (tab[0]=='O' && tab[4]=='O' && tab[8]=='O') return 1;
    if (tab[2]=='O' && tab[4]=='O' && tab[6]=='O') return 1;
    return 0;
}

int is_free(char* tab, int p){

```

```
if (tab[p-1] == ' ') {  
    return 1;  
} else {  
    return 0;  
}  
}
```