75.41 95.15 95.12 Algoritmos y Programación II Curso 4
TP0 - Split

6 de septiembre de 2021

1. Introducción

Al trabajar con texto, separar cadenas de caracteres por un delimitador es una de las operaciones mas comunes. El lenguaje de programación **C** no cuenta con esta primitiva para el manejo de strings y por lo tanto vamos a tener que implementarla nosotros mismos. El objetivo de este trabajo optativo es introducirnos en el mundo del manejo de la memoria dinámica en **C** y de paso servir como repaso de algoritmia básica de la materia anterior. El código elaborado en este TP optativo será de utilidad al elaborar el próximo TP.

2. Cadenas de caracteres en C

C no cuenta con una primitiva para representar cadenas de caracteres (strings). Para representar strings en **C** se utilizan vectores de tipo **char** y una convención. La convención indica que los strings finalizan con un caracter de valor 0. Esto significa que en **C**, cualquier secuencia de bytes terminada con un valor 0 puede tomarse como un string. Recordemos algunos conceptos básicos antes de seguir avanzando.

2.1. Tipos de dato

C, a diferencia de los lenguajes de **tipado** dinámico, posee verificación estricta de tipos. Esto implica que cada variable utilizada debe declarar un tipo. El tipo declarado nos indica qué rango y tipo de valores se pueden almacenar en dicha variable. Los tipos básicos que provee **C** son:

- **char**: 1 byte, almacena valores numéricos enteros de 0 a 255
- **int**: Generalmente 4 bytes. Almacena valores numéricos enteros entre **INT_MIN** e **INT_MAX**.
- **float** y **double**: 4 y 8 bytes respectivamente. Almacenan valores numéricos en punto flotante (reales).

Además de estos tipos básicos existen modificadores que pueden afectar los rangos numéricos que pueden ser almacenados. Por ejemplo existe el modificador **unsigned**, que no permite que los valores almacenados sean negativos. Entonces, a partir de estos tipos básicos y utilizando modificadores se puede lograr una gran variedad de tipos diferentes.

2.2. Arrays

Los **arrays** (que muchas veces solemos llamar **vectores**), son colecciones de elementos del mismo tipo. Por ejemplo, tener una variable de tipo **array de enteros** de tamaño 10 significa que la variable puede almacenar 10 **int**. Estos 10 enteros pueden ser accedidos directamente mediante un subíndice y adicionalmente se encuentran acomodados secuencialmente en la memoria (uno al lado del otro, sin espacios de por medio).

```c
//Declaro una variable diez_enteros que puede almacenar 10 ints
int diez_enteros[10];

//asigno el valor 999 al primer entero del array
diez_enteros[0] = 999;

//asigno el valor -534 al último entero del array
diez_enteros[9] = -534;
```

2.3. Texto y tabla ASCII

Como se puede observar, sólo hablamos de tipos de dato que almacenan valores numéricos. ¿Cómo representamos caracteres y texto? Si asignamos un número a cada letra del abecedario, podríamos utilizar los tipos de datos que tenemos disponibles para representar texto. Para lograr que esto funcione, claro, hayq que ponerse de acuerdo en nua convención de qué número corresponde a cada letra. Esta convención existe y la podemos ver reflejade en la tabla **ASCII**, que en definitiva es una tabla que nos permite conocer la equivalencia entre un número y el caracter que representa.

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	*[NULL]*	48	30	110000	60	**0**	96	60	1100000	140	`
1	1	1	1	*[START OF HEADING]*	49	31	110001	61	**1**	97	61	1100001	141	**a**
2	2	10	2	*[START OF TEXT]*	50	32	110010	62	**2**	98	62	1100010	142	**b**
3	3	11	3	*[END OF TEXT]*	51	33	110011	63	**3**	99	63	1100011	143	**c**
4	4	100	4	*[END OF TRANSMISSION]*	52	34	110100	64	**4**	100	64	1100100	144	**d**
5	5	101	5	*[ENQUIRY]*	53	35	110101	65	**5**	101	65	1100101	145	**e**
6	6	110	6	*[ACKNOWLEDGE]*	54	36	110110	66	**6**	102	66	1100110	146	**f**
7	7	111	7	*[BELL]*	55	37	110111	67	**7**	103	67	1100111	147	**g**
8	8	1000	10	*[BACKSPACE]*	56	38	111000	70	**8**	104	68	1101000	150	**h**
9	9	1001	11	*[HORIZONTAL TAB]*	57	39	111001	71	**9**	105	69	1101001	151	**i**
10	A	1010	12	*[LINE FEED]*	58	3A	111010	72	**:**	106	6A	1101010	152	**j**
11	B	1011	13	*[VERTICAL TAB]*	59	3B	111011	73	**;**	107	6B	1101011	153	**k**
12	C	1100	14	*[FORM FEED]*	60	3C	111100	74	**<**	108	6C	1101100	154	**l**
13	D	1101	15	*[CARRIAGE RETURN]*	61	3D	111101	75	**=**	109	6D	1101101	155	**m**
14	E	1110	16	*[SHIFT OUT]*	62	3E	111110	76	**>**	110	6E	1101110	156	**n**
15	F	1111	17	*[SHIFT IN]*	63	3F	111111	77	**?**	111	6F	1101111	157	**o**
16	10	10000	20	*[DATA LINK ESCAPE]*	64	40	1000000	100	**@**	112	70	1110000	160	**p**
17	11	10001	21	*[DEVICE CONTROL 1]*	65	41	1000001	101	**A**	113	71	1110001	161	**q**
18	12	10010	22	*[DEVICE CONTROL 2]*	66	42	1000010	102	**B**	114	72	1110010	162	**r**
19	13	10011	23	*[DEVICE CONTROL 3]*	67	43	1000011	103	**C**	115	73	1110011	163	**s**
20	14	10100	24	*[DEVICE CONTROL 4]*	68	44	1000100	104	**D**	116	74	1110100	164	**t**
21	15	10101	25	*[NEGATIVE ACKNOWLEDGE]*	69	45	1000101	105	**E**	117	75	1110101	165	**u**
22	16	10110	26	*[SYNCHRONOUS IDLE]*	70	46	1000110	106	**F**	118	76	1110110	166	**v**
23	17	10111	27	*[ENG OF TRANS. BLOCK]*	71	47	1000111	107	**G**	119	77	1110111	167	**w**
24	18	11000	30	*[CANCEL]*	72	48	1001000	110	**H**	120	78	1111000	170	**x**
25	19	11001	31	*[END OF MEDIUM]*	73	49	1001001	111	**I**	121	79	1111001	171	**y**
26	1A	11010	32	*[SUBSTITUTE]*	74	4A	1001010	112	**J**	122	7A	1111010	172	**z**
27	1B	11011	33	*[ESCAPE]*	75	4B	1001011	113	**K**	123	7B	1111011	173	**{**
28	1C	11100	34	*[FILE SEPARATOR]*	76	4C	1001100	114	**L**	124	7C	1111100	174	**|**
29	1D	11101	35	*[GROUP SEPARATOR]*	77	4D	1001101	115	**M**	125	7D	1111101	175	**}**
30	1E	11110	36	*[RECORD SEPARATOR]*	78	4E	1001110	116	**N**	126	7E	1111110	176	**~**
31	1F	11111	37	*[UNIT SEPARATOR]*	79	4F	1001111	117	**O**	127	7F	1111111	177	*[DEL]*
32	20	100000	40	*[SPACE]*	80	50	1010000	120	**P**					
33	21	100001	41	**!**	81	51	1010001	121	**Q**					
34	22	100010	42	**"**	82	52	1010010	122	**R**					
35	23	100011	43	**#**	83	53	1010011	123	**S**					
36	24	100100	44	**$**	84	54	1010100	124	**T**					
37	25	100101	45	**%**	85	55	1010101	125	**U**					
38	26	100110	46	**&**	86	56	1010110	126	**V**					
39	27	100111	47	**'**	87	57	1010111	127	**W**					
40	28	101000	50	**(**	88	58	1011000	130	**X**					
41	29	101001	51	**)**	89	59	1011001	131	**Y**					
42	2A	101010	52	*****	90	5A	1011010	132	**Z**					
43	2B	101011	53	**+**	91	5B	1011011	133	**[**					
44	2C	101100	54	**,**	92	5C	1011100	134	****					
45	2D	101101	55	**-**	93	5D	1011101	135	**]**					
46	2E	101110	56	**.**	94	5E	1011110	136	**^**					
47	2F	101111	57	**/**	95	5F	1011111	137	**_**					

Figura 1: La tabla ASCII, ~~robada~~ sacada de wikipedia

Entonces por ejemplo, la tabla nos dice que para representar el caracter '@' se usa el valor 64 y que el valor 99 representa la 'c' minúscula. Los valores de 0 a 31 y el 127 son especiales y se conocen como **caracteres de control** y se utilizan para representar diferentes cosas. Por ejemplo el caracter 10 o 0x0A (representación hexadecimal), o \n (código especial de escape reconocido por **C**) representa un final de línea en sistemas tipo **Unix**. En particular el valor 0 es el que utilizamos como convención para marcar el final de un string.

En definitiva esta es la forma básica de almacenar texto en una computadora bajo el estándar **ASCII** (que no es el único pero sí el mas extendido). La representación **UTF8** por ejemplo es compatible con **ASCII** y utiliza los mismos primeros 128 códigos, pero luego utiliza múltiples bytes (de 2 a 4) para representar el resto de los caracteres o **codepoints**. La idea siempre es la misma. Los caracteres de texto son representados mediante valores numéricos. Cuando se trabaje con texto se está trabajando, a bajo nivel, con números.

2.4. Por fin strings

Volviendo un poco al principio, los strings no son mas que una colección de bytes (char) delimitado por un valor 0 (no confundir 0, a veces representado en el código como \0) con el caracter '0', que corresponde al valor 48. Entonces, aplicando los conocimientos que acabamos de repasar, podemos representar un string de forma básica en **C** haciendo lo siguiente:

```c
//declaro un array de 100 bytes
char mi_texto[100];

//Asigno caracteres a cada posición
//Notar que es posible usar la sintaxis de caracter con las comillas simples, tanto
//como utilizar los números directamente (aunque siempre que se pueda se recomienda
//utilizar la notación de caracteres, para que sea mas claro)
mi_texto[0] = 'H';
mi_texto[1] = 'o';
mi_texto[2] = 'l';
mi_texto[3] = 97; //97 es 'a'

//Hasta este punto, mi_texto no puede ser considerado un string, y no debe ser
//pasado a funciones que esperan un string, ya que no posee el 0 al final,
//y por lo tanto no es posible conocer dónde finaliza dicho string

//La siguiente operación es indefinida, y seguramente cause problemas de memoria
//ya que %s espera un string para mostrar por pantalla, pero mi_texto no lo es.
printf("%s", mi_texto);

mi_texto[4] = 0;
//A partir de este momento, mi_texto puede ser considerado un string.

//Esta otra sintáxis para definir strings utiliza punteros (no es tema de este repaso).
//Define un string otro_texto con el contenido "Hola". En este caso no es necesario
//poner el 0 explícitamente, ya que el compilador lo pone automáticamente.
//Como contrapartida, este string es estático, se encuentra en una zona de sólo
//lectura y no debe ser modificado (a menos que se quiera tener errores de memoria)
char* otro_texto = "Hola";
```

3. ¿Y qué hacemos con todo esto?

C99 (el estándar de **C** utilizado en la materia) cuenta con diferentes primitivas para trabajar con strings (como **strcpy**, **strcmp**, **strcat**, **strstr**, etc), pero no cuenta con una primitiva que reciba un string y un delimitador y devuelva un vector con los strings delimitados. Entonces la misión, si desean aceptarla (si, es un TP optativo, aunque en realidad después lo van a tener que implementar para el TP1), es implementar esta primitiva (**split**), utilizando memoria dinámica.

3.1. ¿Y cómo lo hacemos?

Básicamente hay que completar el archivo **split.c** con la implementación del algoritmo pedido. Tengan en cuenta los siguientes items:

- No se pueden asumir tamaños máximos: No vale decir 'Asumo que nunca me van a mandar un string mas grande que 5000 caracteres'. No. Usen memoria dinámica, que para eso está.
- Dicho esto, no todo en la vida es memoria dinámica. Úsenla dónde sea apropiado. El stack sirve para colocar variables locales, úsenlo.
- Modularización ante todo. No es tan importante en este punto si recorren dos veces el vector para resolver el problema. Es preferible hacer código claro que recorra dos veces el string a un código inentendible que recorre una sola vez.
- Tienen un **Makefile** que corre pruebas automatizadas para saber si su implementación cumple con lo básico que se pide. Aprovechenlo. Pueden agregar mas pruebas si lo creen necesario.
- La modularización es importante (¿Ya lo había dicho?). No me hagan un choclo.
- Los nombres de variable son importantes. Mejor **'int cantidad_de_substrings'** a **'int n'**.
- Los nombres de función también son importantes. Mejor **'buscar_posición_separador'** que **'buscar'**.

4. Entrega

La entrega deberá contar con todos los archivos que se adjuntan con este enunciado (todo lo necesario para compilar y correr correctamente). Dichos archivos deberán formar parte de un único archivo **.zip** el cual será entregado a través de la plataforma de corrección automática **Chanutron2021** para verificar si pasa correctamente las pruebas o no, y mas importante para este TP, para poder recibir alguna devolución de los correctores que los ayude a mejorar.