***Memoria (Memory)***

En la semana 2, hablamos sobre la memoria y cómo cada byte tiene una dirección, o identificador, para que podamos referirnos a dónde se almacenan realmente nuestros datos. (*En el bloque 4, apartado “6. Punteros y paso por referencia” ampliamos este tema mediante un ejemplo*)

Resulta que, por convención, las direcciones para memoria usan el sistema de conteo ***hexadecimal,*** o **base-16**, donde hay 16 dígitos: 0-9, y A-F como equivalentes a 10-15. Pongamos algo de contexto. Si usted recuerda, nuestro sistema de conteo inicial fue el ***binario,*** o **base-2,** era *base-2* porque cada bit podía asumir, como mucho, dos posibilidades de dígitos los cuáles eran: *0 y 1* (más nada).

Bueno, resulta que, en un sistema ***hexadecimal*** las posibilidades son mayores por cada espacio de memoria: los valores que puede tomar cada *dígito* no se limitan solamente a unos (1s) o ceros (0s); sino que, como ya lo planteamos, pueden tomar valores que van desde el 0 hasta el 9 y continuan con el A hasta el F (siendo considerados *valores decimales* también), en su orden alfabetico. Si se da cuenta, si contabilizamos cúantas posibilidades de dígito hay desde 0 a 9 y, posteriormente, desde A hasta F... *nos daría un número equivalente de* ***16 posibles (base-16)*** *y no sólo de 2 posibles como lo es inicialmente un sistema binario (base-2) ni sólo 10 posibles como lo es un sistema decimal que va de 0 a 9 (base-10).*

***base-16: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A(10), B(11), C(12), D(13), E(14), F(15);*** valores totales como posibilidades de valor por cada espacio de memoria individual, dígito, en un sistema ***hexadecimal. Así fue establecido convencionalmente por el hombre.***

El valor decimal de, por ejemplo, **0F** en un sistema ***hexadecimal*** sería equivalente a ***15***; el valor decimal de ***0*** es 0 y el valor decimal de ***F***, siendo considerado un dígito también dentro de un sistema *hexadecimal*, sería equivalente a 15.

¿Qué seguiría después de ***0F?***

Como después de ***F*** en el sistema ***hexadecimal*** no hay más nada; y, si quisieramos representar el valor decimal que sigue; es decir, ***16...*** debemos entonces incrementar en una unidad nuestro primer espacio de memoria: siendo que ***0*** pase a ser ***1;*** y, el espacio de memoria donde se ubica ***F,*** se restaure a su partida inicial: ***0.*** Como estamos ante un incremento de valores (en una unidad decimal extra para ser precisos), entonces el valor decimal de ***10 (que son dos dígitos)*** en *hexadecimal*es: ***16. 10*** en *hexadecimal*sería precisamente lo que viene después de ***0F*** *(que, recuerde, tiene un valor decimal de* ***15****).*

*Dicho esto,* y conociendo el valor decimal de cada posibilidad de dígito en un sistema hexadecimal, siendo que por cada dígito hay 16 posibilidades de valor para ser tomadas: *que van de 0-9 a A-F (y no 2 posibilidades como en un sistema binario, 0 y 1), tenemos que: Con dos dígitos, podemos tener un valor máximo de* ***FF, ó 16^1 \* 15 + 16^0 \* 15 = 240 + 15 = 255 (ó 16 \* 16 = 256)****, que es el mismo valor máximo con 8 bits -o un byte- de binario* ***(2 \* 2 \* 2 \* 2 \* 2 \* 2 \* 2 \* 2 = 256).******Entonces, dos dígitos en hexadecimal pueden representar convenientemente el valor de un byte -8 bits- en binario.*** Cada dígito en hexadecimal, con 16 valores, se asigna a cuatro bits en binario (*2 \* 2 \* 2 \* 2 = 16*).

En el sistema ***Hexadecimal,*** como máximo, podemos comprender 6 dígitos en conjunto. Por ejemplo, el sistema de color *RGB* utiliza convencionalmente hexadecimal para describir la cantidad de cada color. ***000000*** en hexadecimal representa 0 para cada uno de los colores rojo, verde y azul, dando como resultado un color ***negro.*** Y ***FF0000*** sería 255, o la mayor cantidad posible de ***rojo.*** ***FFFFFF*** indicaría el valor más alto de cada color, combinándose para ser el ***blanco*** más brillante. Con diferentes valores para cada color, podemos representar millones de colores diferentes.

*dato: para la memoria de nuestra computadora, también usaremos hexadecimal para cada dirección o ubicación (no un sistema binario), así se estableció de forma convencional (quizás porque resume las cosas).*

*Es facil confundir un sistema de conteo por otro; por ejemplo, no saber cuando se está hablando de un sistema decimal o de un sistema hexadecimal. Pues bien, para evitar este tipo de conflictos, lo ideal es que por escrito indiquemos que un valor está en hexadecimal prefijándolo con* ***0x:*** *como en* ***0x10, donde el valor, que realmente es 10 (0x*** *es sólo un indicativo que refiere a hexadecimales****) es igual a 16 en decimal, en lugar de 10. Recuerde que 10, que son dos dígitos, en Hexadecimal corresponden a un valor decimal equivalente a 16.***

***Direcciones (Addresses)***

Para hablar de direcciones en C, es preciso saber de "parametro de dirección” y "parametro de indirección".

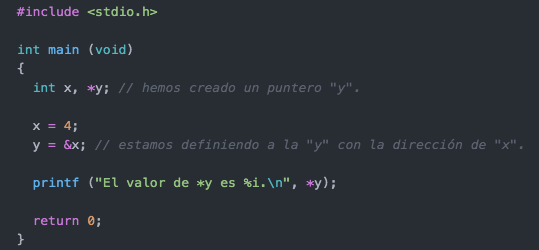
***Parametro de dirección (dirección de operador “&”):*** **tienen la función de indicar u obtener la dirección (o identificador) donde se guarda o almacena una variable, y su valor o dato, dentro de la memoria del computador.** El simbolo de Ampersand **"&"** que usamos en "scanf" es un buen ejemplo: lo que hace, justamente, es decirle al valor de entrada (pasado por tecla) que se guarde o almacene en una dirección: esa dirección sería la dirección de la variable *x* seleccionada con ampersand "&". Digamos, por ejemplo “&x”.

***Parametro de indirección:*** Este, contrario al "parametro de dirección" que se representa simbolicamente con un ampersand **"&",** es representado por un asterisco **"\*"**. La variable que opera con “\*” se llama: **"Puntero".**

**Básicamente lo que hace un puntero es irse el valor del que tenga (o guarde) la dirección de una variable; es decir, es una variable que contiene la dirección de algún otro valor (que mire dentro de una dirección de memoria en particular). Más fácil: la variable que guarda a la variable que tenga el ampersand "&".**

En resumen, “&” le dice en qué dirección se encuentra una variable; y, el “\*” le dice: “vaya a esa dirección”. Estos parametros trabajan conjuntamente porque requieren de especificar “direcciones” para su funcionalidad.

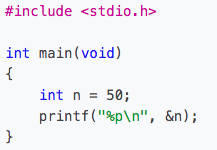
Por ejemplo:



*Sin embargo (y en primer lugar), si deseamos, podemos conocer también propiamente la dirección que tiene, dentro de la memoria del computador, una variable.*

Por ejemplo, si queremos llamar propiamente la dirección de una variable en una función, el especificador de formato para la **dirección** de cualquier variable es: **%p *(p de* *pointer, puntero*)**; es decir, si queremos llamar la dirección de una variable, cualquiera que sea *(la dirección que tiene especificamente dentro de toda la memoria del computador);* y, suponiendo que nuestra variable de interés se llama, por ejemplo, “x”... al ser llamada su dirección en una función tipo *printf* es necesario indicar el especificador de formato preciso para ella, el cual es: %p; y, además, dejar claro a qué variable en especifica se le quiere llamar su dirección, en este caso, nos referimos a la dirección de la variable “x”; para eso, para llamar a la dirección de “x” dentro de una función debe hacerlo así: **&x**.

*Veamos esto en un ejemplo:*

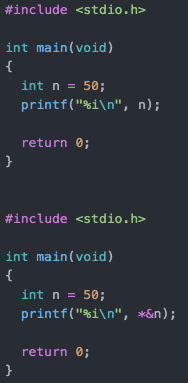


Lo que imprimiría este programa en C sería la dirección, dentro de toda la memoria del computador, donde se encuentra ubicada la variable de tipo entero “n”. No imprime “50”, no. Imprime la dirección de esa variable; es decir, “0x7ffd80792f7c” para este caso.

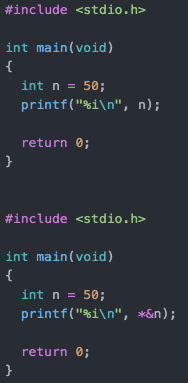
*Lo anterior, saber la ubicación exacta de una variable dentro de la memoria del computador, realmente no tiene mucha utilidad o relevancia; sin embargo, nos será muy útil más adelante el uso de las direcciones: saber donde se ubica una variable para luego ir hacia ella o hacia su valor (como lo hicimos, por ejemplo, en nuestro primer programa de este apartado: “direcciones”).*

De hecho, si juntamos los dos parametros simultaneamente, tal que así: ***“\*&”*** en una variable (o para llamarla), realmente lo que estamos haciendo es averiguar la dirección de la variable de interés -y en primer lugar (con &); y, luego, yendo a ella (con \*); es decir, simplemente podíamos llamar a la variable en sí misma y daba igual: nos ubicariamos directamente en su dirección con su respectivo valor. *Para que se entienda mejor...*

*Esto:*



*Es lo mismo que esto:*



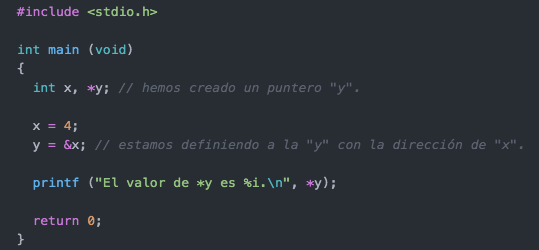
***Más sobre los punteros***

La variable *puntero* siempre se declara con el asterisco (\*); y, como interesa que tome el valor de otra variable, por medio de su dirección, generalmente se inicializa con otra variable compuesta por un parametro de dirección; por ejemplo,

*int x = 4;*

***int \*y = &x;***

Ahora, para llamar un **puntero** dentro de una función, *para que corra éste en función de la dirección de la variable a la que está* *apuntando*, observe de nuevo este ejemplo:

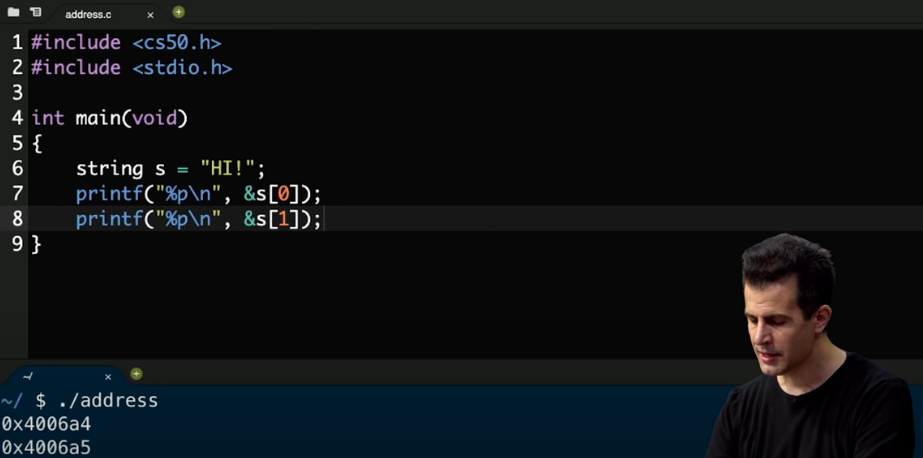


Dentro de la función se llama a la variable puntero **(\*y)**; en este caso, la función que la llama es ***printf***; es decir, queremos que se imprima el valor que se encuentra en la dirección de la variable que está siendo apuntada, literalmente, por la variable puntero **\*y**; es decir, que se imprima entonces el valor que conserva la variable **x.**

***Un dato: las variables punteros ocupan 8 bytes.***

***Punteros & Strings***

Partamos de la base de que, en toda cadena de texto, su dirección dentro de la memoria corresponde a la misma que tiene el primer caracter de la cadena en cuestión. Por ejemplo, si declaro una cadena con nombre de “S” y la inicializo con “hi!”; tenemos que: ***string S = “hi!”***. La ubicación, dentro de la memoria, de la cadena ***S*** es equivalente a la dirección del primer caracter, “***h”***; tal que así: ***S*** está en la dirección ***0x123***, mientras que ***“h”*** igual: ***0x123.*** *Del resto, “i” estaría en 0x124 y “!” en 0x125.* Esto es así porque tiene sentido que, la ubicación o dirección dentro de la memoria, de la cadena de texto, se contabilice desde su punto inicial: su primer caracter. *Del resto, por la longitud propia del texto, ya sabremos cuánto espacio de memoria (en términos de bytes) asumiría en su conjunto el string o la cadena de texto según los caracteres que ésta contenga; Byte* por *byte,* *contiguos,* va ocupandose de la memoria cada *caracter* hasta llegar al último caracter del texto: Recuerde que cada caracter (variable tipo char) pesa un byte. Tal que así, por ejemplo:

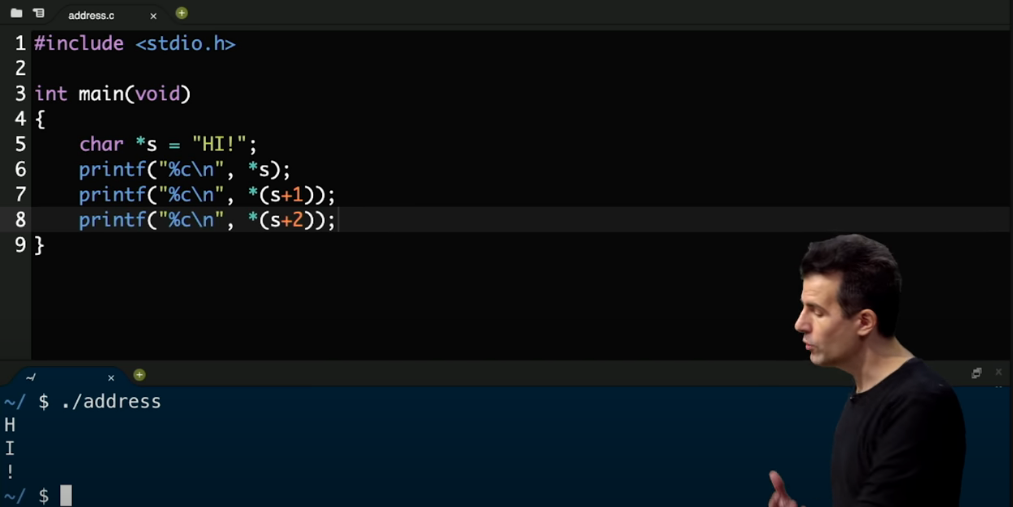
  
*Observe cómo varia la ubicación en una unidad, dentro de la memoria, al pasar del caracter s[0] a s[1]. Como estamos hablando del espacio de la* ***memoria*** *justamente; entonces, esa unidad adicional, se refiere al conteo de un* ***byte extra.***

Entonces, sí, en un nivel bajo, ***S*** es técnicamente una dirección; y si, técnicamente, es la dirección del primer byte (o caracter). Por lo anterior, puede pensar también de los ***Strings*** como simples punteros; es decir, como ir a la dirección de un primer caracter que se encuentra situado en algún lugar de la memoria del computador.

Dicho todo lo anterior, el tipo de datos ***string* de la biblioteca de CS50**, que realmente resulta ser un tipo de dato personalizado creado con la herramienta ***structs***de **C** y finalmente incorporado en el archivo cd50.h como un tipo de dato más, parte de la base conceptual de pensar a las cadenas de texto ***(Strings)*** como: *“ir a la dirección de un primer caracter que se encuentra situado en algún lugar de la memoria del computador”.* Revise el archivo *compare.c* de ejercicios Harvard de esta misma semana para tener más claridad sobre este asunto.

Es decir, es lo mismo decir ***Strings S = “hi!”;*** a ***Char \*S = “hi!”;.*** Es exactamente lo mismo. El texto finaliza su recorrido, desde ese caracter inicial, hasta que se choca o tropieza con un *caracter nulo: “\0”.*

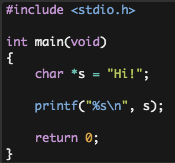
*Entonces,* ***\*S*** *apuntaría* realmente a la dirección del primer caracter (*a su valor*) de la cadena con la que ha sido definida (me refiero a la misma cadena ***“\*S”)***. En consecuencia, si quisieramos ir (e imprimir) el valor del segundo y tercer caracter, bajo este modelo *char \*,* cómo sería? Veamoslo:



Tiene sentido que se adicione una unidad a ***\*S*** debido a que, dentro del espacio de memoria, la cadena de texto ocupa por cada caracter un *byte* (y son *bytes contiguos*, caracter por caracter seguido hasta que finalice la cadena de texto en cuestión con el *caracter nulo*)*.* Entonces, para ir al *byte* del caracter número ***dos*** de la misma cadena de texto sería ***s+1****,* para ir al *byte* del caracter número ***tres*** de la misma cadena de texto sería ***s+2.*** *Esta forma de imprimir caracter por caracter, asistiendose de punteros y adiciones, se le llama:* ***“aritmética de punteros”.***

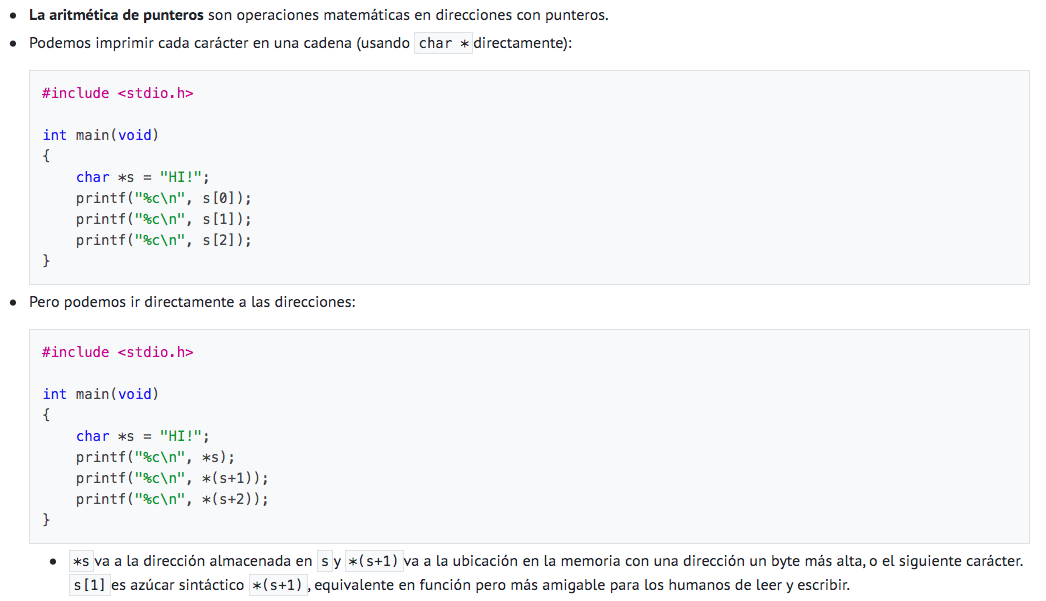
Ahora bien, si se extralimita y empieza a husmear más allá del espacio de memoria correspondiente a la cadena de texto en cuestión; es decir, tocando ya el espacio de memoria que no le corresponde -idealmente- tocar *(que no tiene nada que ver con lo que está programando)*, está expuesto a visualizar un error de tipo: ***“Segmentation*** ***fault”*** y esto tiende a significar que: *estás tocando un segmento de la memoria que no deberias tocar (porque no tiene nada que ver con lo que se está programando... se supone):* es como buscar arbitrariamente en cualquier lugar de la memoria de su computadora.

Ahora bien, sólo para probar que realmente el tipo de dato *string* de la biblioteca CS50 es exactamente lo mismo que decir *char \** en la biblioteca *estandar* de C, imprimamos literalmente toda la cadena ***S\**** (no caracter por caracter, sino todo el texto de golpe), tal que así:



El especificador de formato para llamar cadenas de texto *(strings)* en funciones es el mismo: ***%s***, posteriormente se hace un llamado a la variable en sí que conserva a la cadena de texto en su totalidad: ***S.***

***Detalles extras sobre “aritmética de punteros”.***



Ahora, revise por favor, el apartado de “Ejercicos Harvard” de esta misma semana *Memoria.* Hay verá la introducción y explicación mediante ejercicios de temas como: “malloc”, “strcpy & free”, “valgrind”, “garbage”, “more pointers”, “FILE I/O”, etc.