









Esto es CS50x

OpenCourseWare

Donar (<https://cs50.harvard.edu/donate>)  (<https://community.alumni.harvard.edu/give/59206872>)

David J. Malan (<https://cs.harvard.edu/malan/>)
malan@harvard.edu

 (<https://www.facebook.com/dmalan>)  (<https://github.com/dmalan>)  (<https://www.instagram.com/davidjmalan/>) 
(<https://www.linkedin.com/in/malan/>)  (<https://orcid.org/0000-0001-5338-2522>)  (<https://www.quora.com/profile/David-J-Malan>)  (<https://www.reddit.com/user/davidjmalan>)  (<https://twitter.com/davidjmalan>)

Lección 4

- [Hexadecimal](#)
- [Direcciones](#)
- [Punteros](#)
- [Instrumentos de cuerda](#)
- [Aritmética de punteros](#)
- [Comparar y copiar](#)
- [valgrind](#)
- [Valores de basura](#)
- [Intercambio](#)
- [Disposición de la memoria](#)
- [scanf](#)
- [Archivos](#)
- [Gráficos](#)

Hexadecimal

- 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.
- Resulta que, por convención, las direcciones de memoria usan el sistema de conteo **hexadecimal** , o base-16, donde hay 16 dígitos: 0-9, y AF como equivalentes a 10-15.
- Consideremos un número hexadecimal de dos dígitos:

16¹ 16⁰
0 A

- Aquí, la A en el lugar de las unidades (ya que 16 ^ 0 = 1) tiene un valor decimal de 10. Podemos seguir contando hasta 0F, que es equivalente a 15 en decimal.
- Después 0F, necesitamos llevar el uno, ya que iríamos de 09 a 10 en decimal:

16¹ 16⁰
1 0

- Aquí, 1 tiene un valor de 16 ^ 1 * 1 = 16, por lo que 10 en hexadecimal es 16 en decimal.
- Con dos dígitos, podemos tener un valor máximo de FF, o 16 ^ 1 * 15 + 16 ^ 0 * 15 = 240 + 15 = 255, que es el mismo valor máximo con 8 bits de binario. Entonces, dos dígitos en hexadecimal pueden representar convenientemente el valor de un byte en binario. (Cada dígito en hexadecimal, con 16 valores, se asigna a cuatro bits en binario).
- Por escrito, indicamos que un valor está en hexadecimal prefijándolo con 0x, como en 0x10, donde el valor es igual a 16 en decimal, en lugar de 10.
- El sistema de color RGB utiliza convencionalmente hexadecimal para describir la cantidad de cada color. Por ejemplo, 000000 en hexadecimal representa 0 para cada uno de los colores rojo, verde y azul, para un color combinado de negro. Y FF0000 sería 255, o la mayor cantidad posible de rojo. FFFFFFFF 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.
- Para la memoria de nuestra computadora, también usaremos hexadecimal para cada dirección o ubicación.

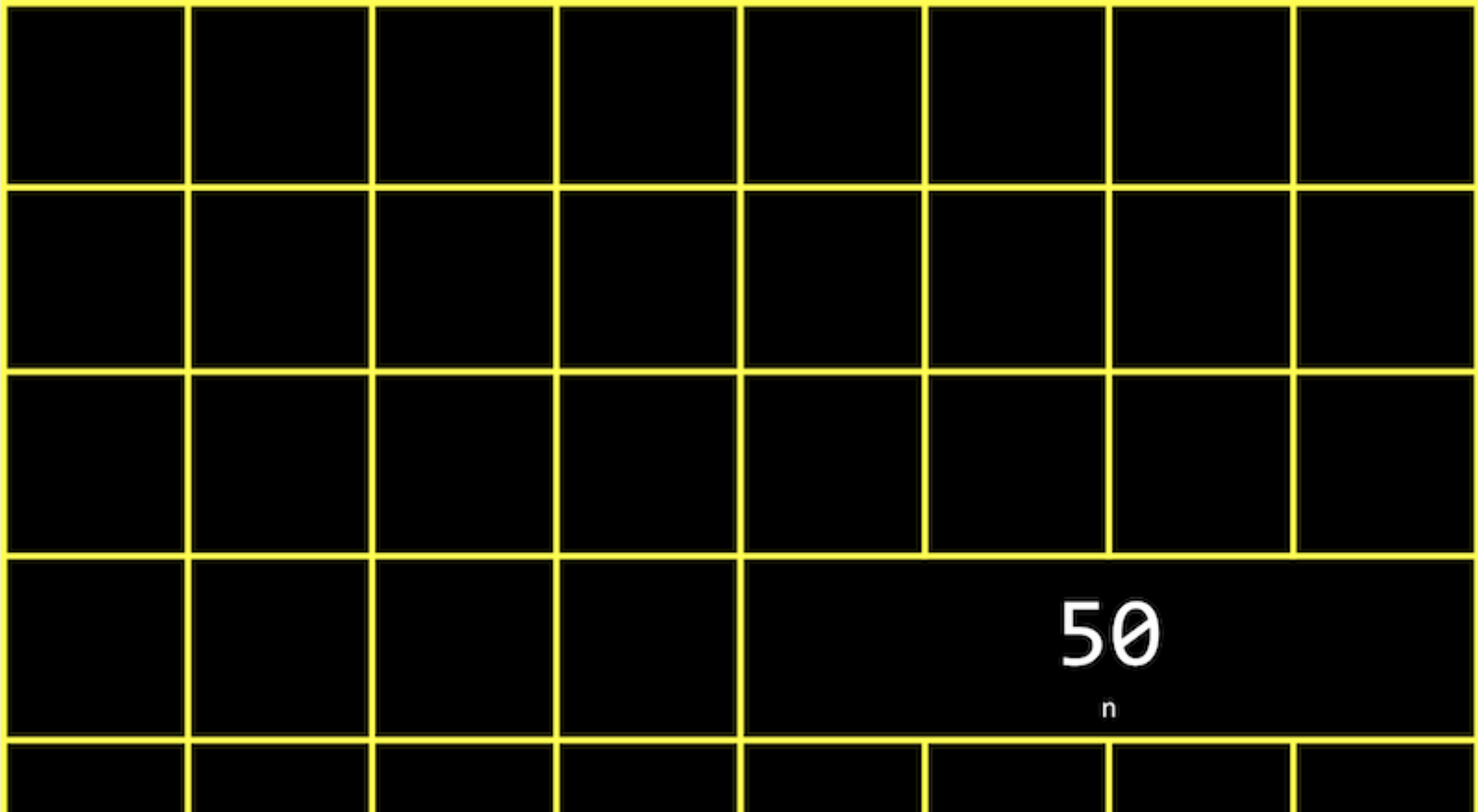
Direcciones

- Podríamos crear un valor `n` e imprimirlo:

```
#include <stdio.h>

int main(void)
{
    int n = 50;
    printf("%i\n", n);
}
```

- En la memoria de nuestra computadora, ahora hay 4 bytes en algún lugar que tienen el valor binario de 50, etiquetados `n`:



- Resulta que, con los miles de millones de bytes en la memoria, esos bytes para la variable `n` comienzan en alguna ubicación, que podría parecerse a algo así `0x12345678`.
- En C, podemos ver la dirección con el `&` operador, lo que significa "obtener la dirección de esta variable":

```
#include <stdio.h>

int main(void)
{
    int n = 50;
    printf("%p\n", &n);
}
```

- `%p` es el código de formato de una dirección.
- En el IDE CS50, vemos una dirección como `0x7ffd80792f7c`. El valor de la dirección en sí mismo no es útil, ya que es solo una ubicación en la memoria en la que se almacena la variable; en cambio, la idea importante es que podamos *usar* esta dirección más adelante.
- El `*` operador, o el operador de desreferencia, nos permite "ir a" la ubicación a la que apunta un puntero.
- Por ejemplo, podemos imprimir `*&n`, en la que "ir a" la dirección de `n`, y eso va a imprimir el valor de `n`, `50` ya que ese es el valor en la dirección de `n`:

```
#include <stdio.h>

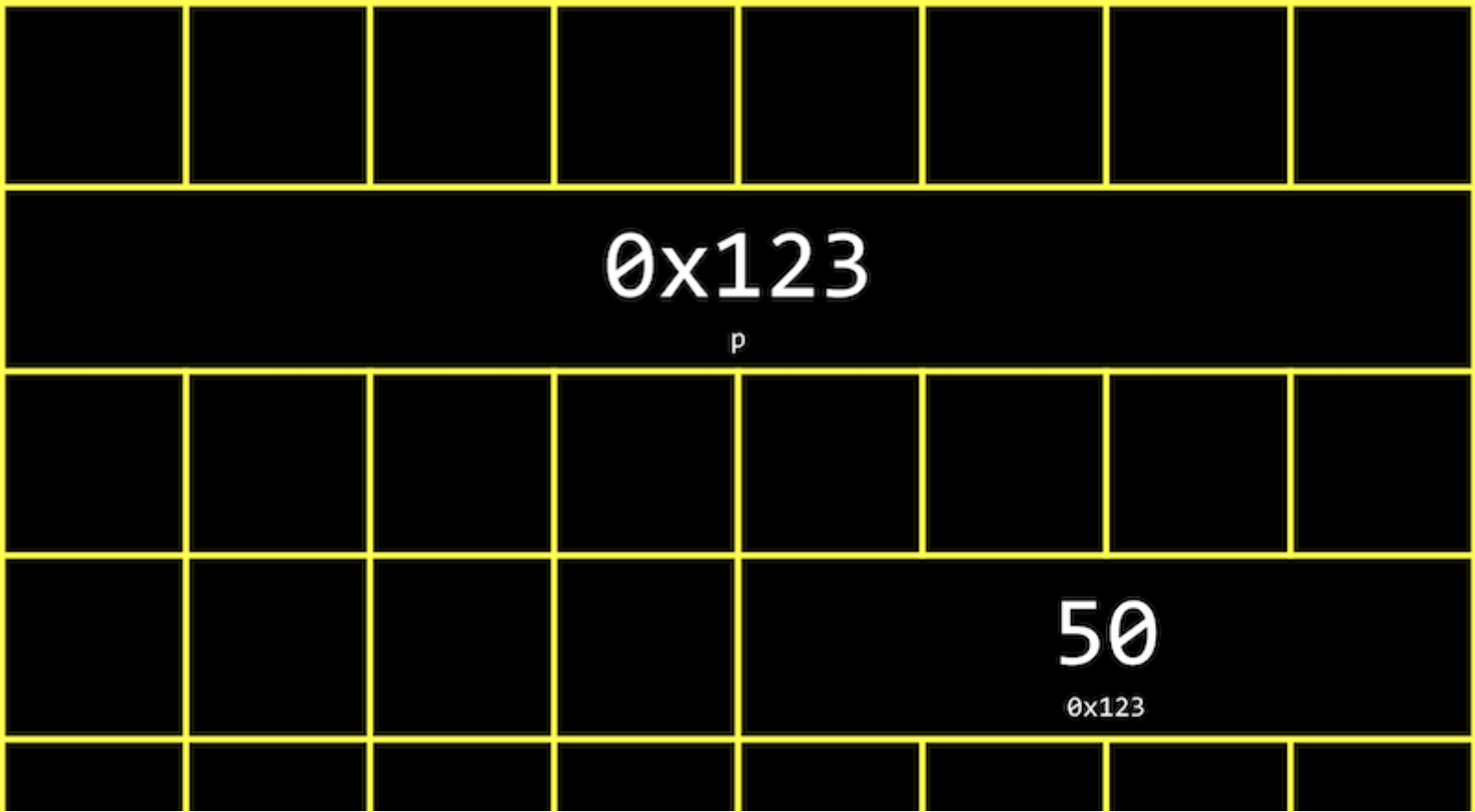
int main(void)
{
    int n = 50;
    printf("%i\n", *&n);
}
```

- Una variable que almacena una dirección se llama **puntero** , que podemos considerar como un valor que "apunta" a una ubicación en la memoria. En C, los punteros pueden referirse a tipos específicos de valores.
- Podemos usar el `*` operador (de una manera lamentablemente confusa) para declarar una variable que queremos que sea un puntero:

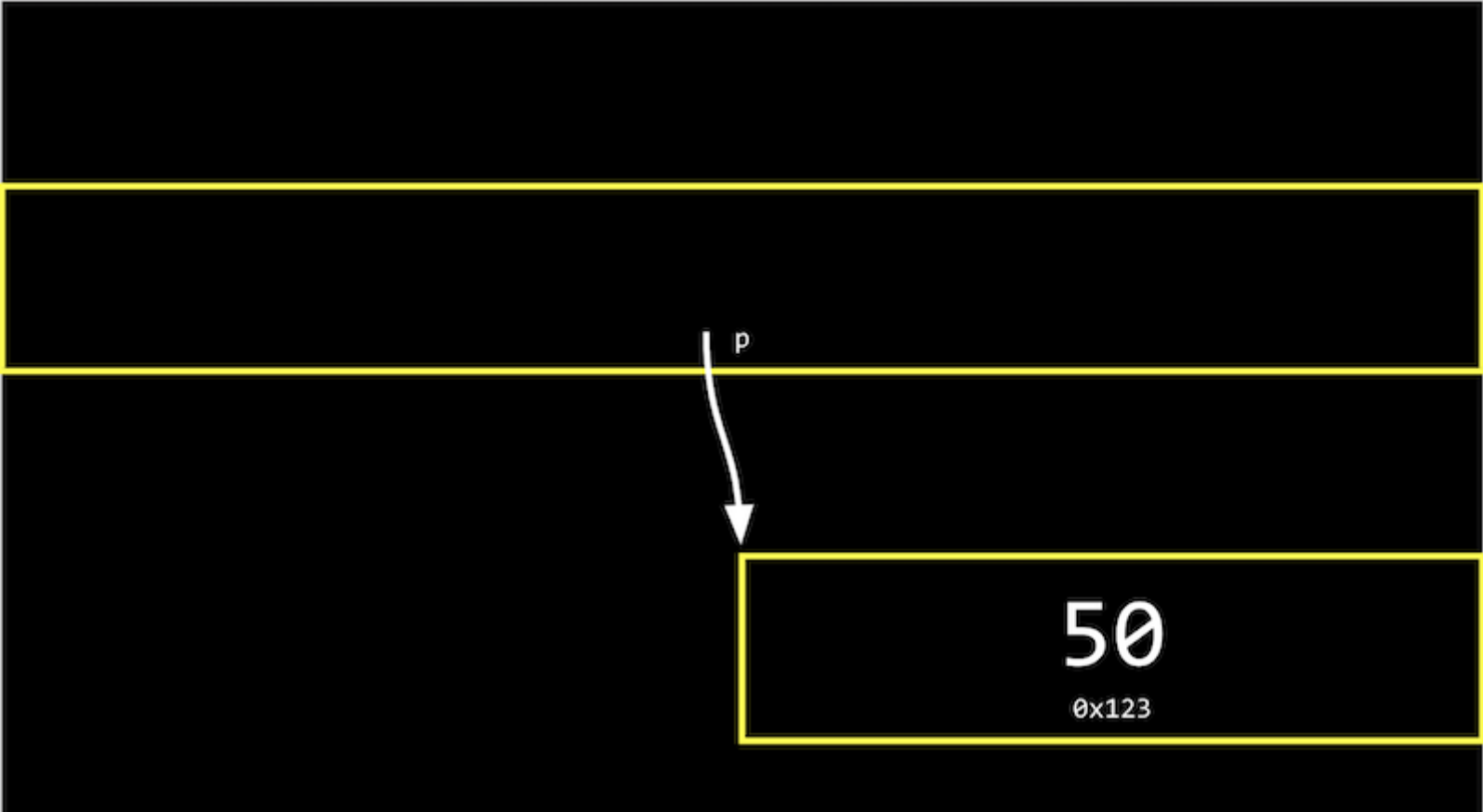
```
#include <stdio.h>

int main(void)
{
    int n = 50;
    int *p = &n;
    printf("%p\n", p);
}
```

- Aquí, usamos `int *p` para declarar una variable, `p` que tiene el tipo de `*`, un puntero, a un valor de tipo `int`, un entero. Luego, podemos imprimir su valor (una dirección, algo así como `0x12345678`), o imprimir el *valor en su ubicación* con `printf("%i\n", *p);`.
- En la memoria de nuestra computadora, las variables se verán así:



- Dado que `p` es una variable en sí misma, está en algún lugar de la memoria y el valor almacenado allí es la dirección de `n`.
- Los sistemas informáticos modernos son de "64 bits", lo que significa que utilizan 64 bits para direccionar la memoria, por lo que un puntero será en realidad de 8 bytes, el doble que un número entero de 4 bytes.
- Podemos abstraer el valor real de las direcciones, ya que serán diferentes a medida que declaramos variables en nuestros programas y no serán muy útiles, y simplemente pensamos `p` en "señalar" algún valor:



- En el mundo real, podríamos tener un buzón con la etiqueta "p", entre muchos buzones con direcciones. Dentro de nuestro buzón, podemos poner un valor como `0x123` que es la dirección de algún otro buzón `n` con la dirección `0x123`.

podemos poner un valor como `0x123`, que es la dirección de algún otro buzón `n`, con la dirección `0x123`.

Instrumentos de cuerda

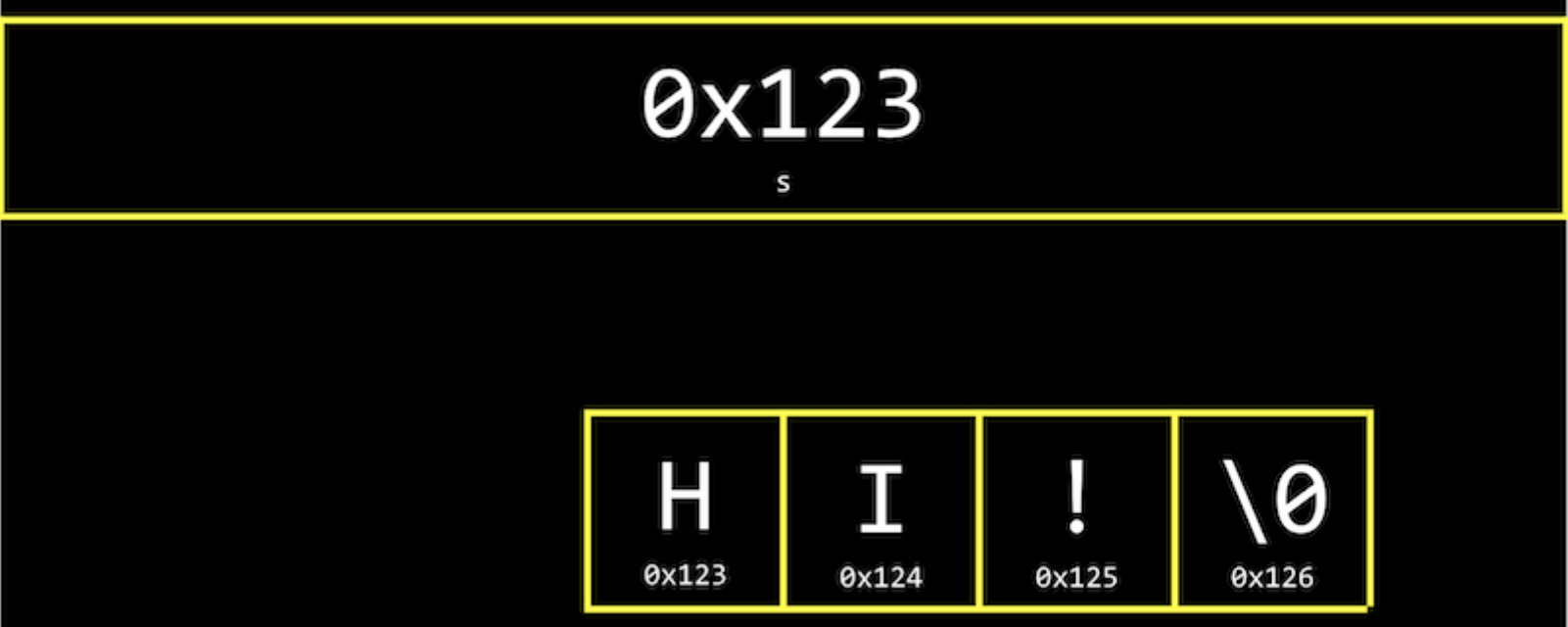
- Una variable declarada con `string s = "HI!";` se almacenará un carácter a la vez en la memoria. Y podemos acceder a cada personaje con `s[0]`, `s[1]`, `s[2]`, y `s[3]`:



- Pero resulta que cada carácter, dado que está almacenado en la memoria, *también* tiene una dirección única, y en `s` realidad es solo un puntero con la dirección del primer carácter:



- Y la variable `s` almacena la dirección del primer carácter de la cadena. El valor `\0` es el único indicador del final de la cadena:



- Dado que el resto de los caracteres están en una matriz, uno tras otro, podemos comenzar en la dirección `s` y continuar leyendo un carácter a la vez de la memoria hasta llegar a `\0`.
- Imprimamos una cadena:

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string s = "HI!";
    printf("%s\n", s);
}
```

- Podemos ver el valor almacenado en `s` con `printf("%p\n", s);`, y vemos algo como, `0x4006a4` ya que estamos imprimiendo la dirección en la memoria del primer carácter de la cadena.
- Si añadimos otra línea, `printf("%p\n", &s[1]);` nos vemos de hecho la siguiente dirección en la memoria: `0x4006a5`.
- Resulta que `string s` es solo un puntero, una dirección a algún personaje en la memoria.
- De hecho, la biblioteca CS50 define un tipo que no existe en C `string`, como `char *`, con `typedef char *string;`. El tipo personalizado `string` se define simplemente como `char *` con `typedef`. Entonces `string s = "HI!"` es lo mismo que `char *s = "HI!"`. Y podemos usar cadenas en C exactamente de la misma manera sin la biblioteca CS50, usando `char *`.

Aritmética de punteros

- La aritmética de punteros** son operaciones matemáticas en direcciones con punteros.
- Podemos imprimir cada carácter en una cadena (usando `char *` directamente):

```
#include <stdio.h>

int main(void)
{
```

```

char *s = "HI!";
printf("%c\n", s[0]);
printf("%c\n", s[1]);
printf("%c\n", s[2]);
}

```

- Pero podemos ir directamente a las direcciones:

```

#include <stdio.h>

int main(void)
{
    char *s = "HI!";
    printf("%c\n", *s);
    printf("%c\n", *(s+1));
    printf("%c\n", *(s+2));
}

```

- `*s` va a la dirección almacenada en `s` y `*(s+1)` va a la ubicación en la memoria con una dirección un byte más alta, o el siguiente carácter. `s[1]` es azúcar sintáctico `*(s+1)`, equivalente en función pero más amigable para los humanos de leer y escribir.
- Incluso podemos intentar ir a direcciones en la memoria que no deberíamos, como con `*(s+10000)`, y cuando ejecutamos nuestro programa, obtendremos una **falla de segmentación** o fallaremos como resultado de que nuestro programa toque la memoria en un segmento que no debería. tengo.

Comparar y copiar

- Intentemos comparar dos enteros del usuario:

```

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    int i = get_int("i: ");
    int j = get_int("j: ");

    if (i == j)
    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}

```

- Compilamos y ejecutamos nuestro programa, y funciona como era de esperar, con los mismos valores de los dos enteros que nos dan "Igual" y valores diferentes "Diferente".
- Cuando intentamos comparar dos cadenas, vemos que las mismas entradas hacen que nuestro programa imprima "Diferente":

```

#include <cs50.h>
#include <stdio.h>

int main(void)
{
    char *s = get_string("s: ");
    char *t = get_string("t: ");

    if (s == t)
    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}

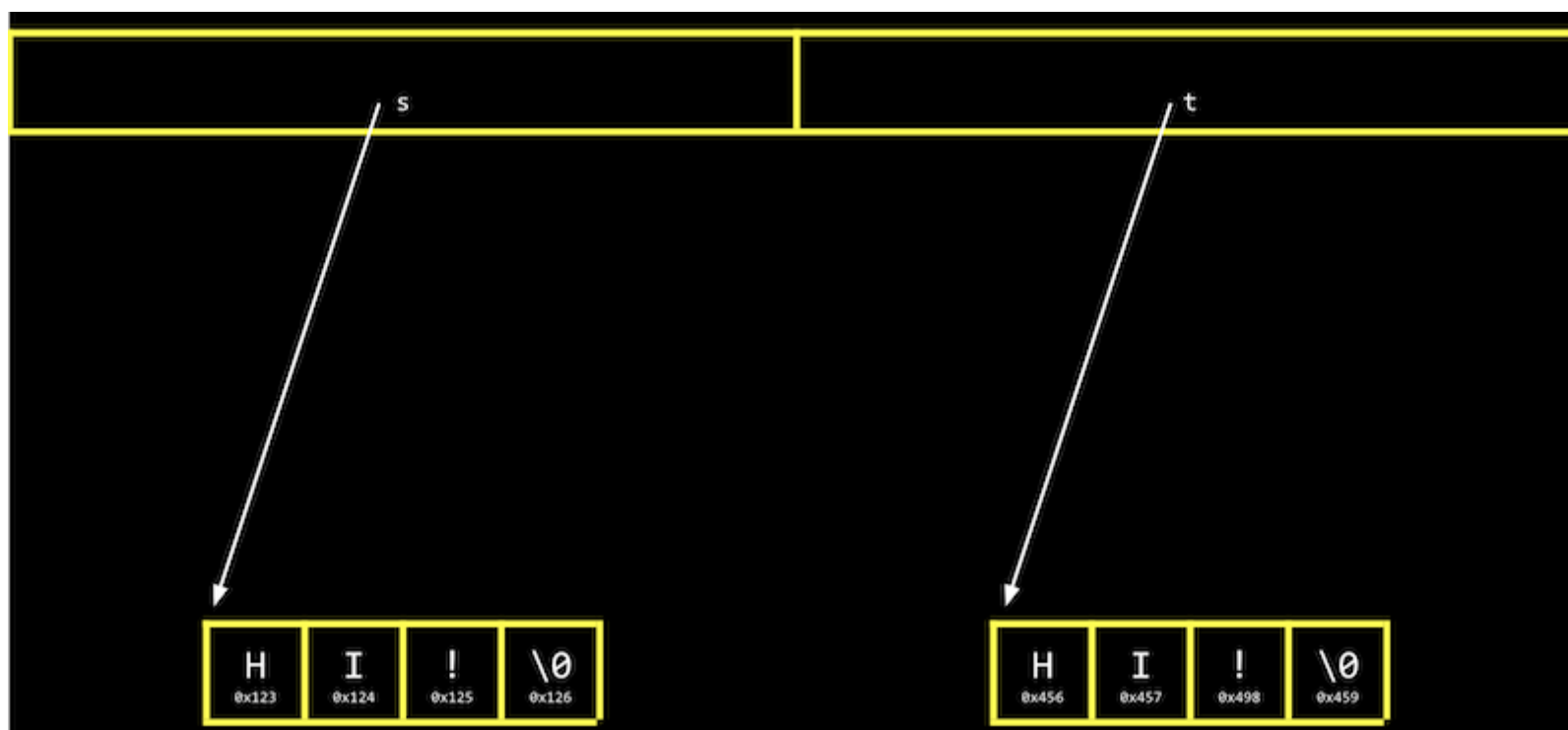
```

```

    {
        printf("Same\n");
    }
    else
    {
        printf("Different\n");
    }
}

```

- Incluso cuando nuestras entradas son las mismas, vemos impreso "Diferente".
- Cada "cadena" es un puntero, `char *` a una ubicación diferente en la memoria, donde se almacena el primer carácter de cada cadena. Entonces, incluso si los caracteres en la cadena son los mismos, esto siempre imprimirá "Diferente".
- Por ejemplo, nuestra primera cadena podría estar en la dirección 0x123, la segunda podría estar en 0x456 y `s` tendrá el valor de `0x123`, apuntando a esa ubicación, y `t` tendrá el valor de `0x456`, apuntando a otra ubicación:



- Y `get_string`, todo este tiempo, ha estado devolviendo solo un `char *`, o un puntero al primer carácter de una cadena del usuario. Como llamamos `get_string` dos veces, obtuvimos dos sugerencias diferentes.
- Intentemos copiar una cadena:

```

#include <cs50.h>
#include <ctype.h>
#include <stdio.h>

int main(void)
{
    char *s = get_string("s: ");

    char *t = s;

    t[0] = toupper(t[0]);

    printf("s: %s\n", s);
    printf("t: %s\n", t);
}

```

- Obtenemos una cadena `s` y copiamos el valor de `s` en `t`. Luego, escribimos en mayúscula la primera letra `t`.
- Pero cuando ejecutamos nuestro programa, vemos que ambos `s` y `t` ahora están en mayúsculas.
- Dado que establecemos `s` y `t` al mismo valor, o la misma dirección, ambos apuntan al mismo carácter, ¡así que escribimos el mismo carácter en mayúscula en la memoria!
- Para hacer una copia de una cadena, tenemos que trabajar un poco más y copiar cada carácter en `s` otro lugar de la memoria:

```

#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *s = get_string("s: ");

```

```

char *t = malloc(strlen(s) + 1);

for (int i = 0, n = strlen(s); i < n + 1; i++)
{
    t[i] = s[i];
}

t[0] = toupper(t[0]);

printf("s: %s\n", s);
printf("t: %s\n", t);
}

```

- Creamos una nueva variable `t` del tipo `char *`, con `char *t`. Ahora, queremos apuntar a una nueva porción de memoria que sea lo suficientemente grande como para almacenar la copia de la cadena. Con `malloc`, *asignamos una* cierta cantidad de bytes en la memoria (que aún no se usan para almacenar otros valores) y pasamos la cantidad de bytes que nos gustaría marcar para su uso. Ya conocemos la longitud de `s`, y agregamos 1 a eso para el carácter nulo de terminación. Entonces, nuestra última línea de código es `char *t = malloc(strlen(s) + 1);`.
- Luego, copiamos cada carácter, uno a la vez, con un `for` bucle. Usamos `i < n + 1`, ya que queremos llegar *hasta* `n`, la longitud de la cadena, para asegurarnos de copiar el carácter final en la cadena. En el bucle, configuramos `t[i] = s[i]`, copiando los personajes. Si bien podríamos usar `*(t+i) = *(s+i)` para el mismo efecto, podría decirse que es menos legible.
- Ahora, podemos poner en mayúscula solo la primera letra de `t`.
- Podemos agregar alguna verificación de errores a nuestro programa:

```

#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    char *s = get_string("s: ");

    char *t = malloc(strlen(s) + 1);
    if (t == NULL)
    {
        return 1;
    }

    for (int i = 0, n = strlen(s); i < n + 1; i++)
    {
        t[i] = s[i];
    }

    if (strlen(t) > 0)
    {
        t[0] = toupper(t[0]);
    }

    printf("s: %s\n", s);
    printf("t: %s\n", t);

    free(t);
}

```

- Si nuestra computadora se queda sin memoria, `malloc` devolverá `NULL` el puntero nulo o un valor especial que indica que no hay una dirección a la que apuntar. Así que deberíamos comprobar ese caso y salir si `t` es así `NULL`.
- También podríamos comprobar que `t` tiene una longitud, antes de intentar poner en mayúscula el primer carácter.
- Finalmente, deberíamos **liberar** la memoria que asignamos anteriormente, lo que la marca como utilizable nuevamente por algún otro programa. Llamamos a la `free` función y pasamos el puntero `t`, ya que hemos terminado con ese trozo de memoria. (`get_string` también, llamadas `malloc` para asignar memoria para cadenas y llamadas `free` justo antes de `main` que regrese la función).
- De hecho, también podemos usar la `strcpy` función, de la biblioteca de cadenas de C, con en `strcpy(t, s);` lugar de nuestro bucle, para copiar la cadena `s` en `t`.

valgrind

- `valgrind` es una herramienta de línea de comandos que podemos usar para ejecutar nuestro programa y ver si tiene **pérdidas de memoria**, o memoria que hayamos asignado sin liberar, lo que eventualmente podría hacer que nuestra computadora se quede sin memoria.
- Construyamos una cadena pero asignemos menos de lo que necesitamos en `memory.c`:

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

int main(void)
{
    char *s = malloc(3);
    s[0] = 'H';
    s[1] = 'I';
    s[2] = '!';
    s[3] = '\0';
    printf("%s\n", s);
}
```

- Tampoco liberamos la memoria que hemos asignado.
- Ejecutaremos `valgrind ./memory` después de compilar y veremos muchos resultados, pero podemos ejecutar `help50 valgrind ./memory` para ayudar a explicar algunos de esos mensajes. Para este programa, vemos fragmentos como "Escritura no válida de tamaño 1", "Lectura no válida de tamaño 1" y, finalmente, "3 bytes en 1 bloque definitivamente se pierden", con números de línea cerca. De hecho, estamos escribiendo en la memoria, `s[3]` que no es parte de lo que asignamos originalmente `s`. Y cuando imprimimos `s`, también estamos leyendo hasta el final `s[3]`. Y finalmente, `s` no se libera al final de nuestro programa.
- Podemos asegurarnos de asignar la cantidad correcta de bytes y la memoria libre al final:

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

int main(void)
{
    char *s = malloc(4);
    s[0] = 'H';
    s[1] = 'I';
    s[2] = '!';
    s[3] = '\0';
    printf("%s\n", s);
    free(s);
}
```

- Ahora, `valgrind` no muestra ningún mensaje de advertencia.

Valores de basura

- Echemos un vistazo a lo siguiente:

```
int main(void)
{
    int *x;
    int *y;

    x = malloc(sizeof(int));

    *x = 42;
    *y = 13;
```



```

    y = x;

    *y = 13;
}

```

- Declaramos dos punteros a números enteros `x` y `y`, pero no les asignamos valores. Usamos `malloc` para asignar suficiente memoria para un número entero con `sizeof(int)` y almacenarlo en `x`. `*x = 42` va a los `x` puntos de dirección y establece esa ubicación en la memoria en el valor 42.
- Con `*y = 13`, estamos tratando de poner el valor 13 en los `y` puntos de dirección. Pero como nunca le asignamos `y` un valor, tiene un **valor basura**, o cualquier valor desconocido que estuviera en la memoria, de cualquier programa que se estuviera ejecutando en nuestra computadora antes. Entonces, cuando intentamos ir al valor de basura `y` como una dirección, vamos a una dirección desconocida, lo que probablemente cause una falla de segmentación o una falla de segmentación.
- Vemos [Pointer Fun with Binky \(https://www.youtube.com/watch?v=3uLKjb973HU\)](https://www.youtube.com/watch?v=3uLKjb973HU), un video animado que demuestra los conceptos del código anterior.
- Podemos imprimir valores basura, declarando una matriz pero sin establecer ninguno de sus valores:

```

#include <stdio.h>

int main(void)
{
    int scores[3];
    for (int i = 0; i < 3; i++)
    {
        printf("%i\n", scores[i]);
    }
}

```

- Cuando compilamos y ejecutamos este programa, vemos varios valores impresos.

Intercambio

- Intentemos intercambiar los valores de dos números enteros.

```

#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i, y is %i\n", x, y);
    swap(x, y);
    printf("x is %i, y is %i\n", x, y);
}

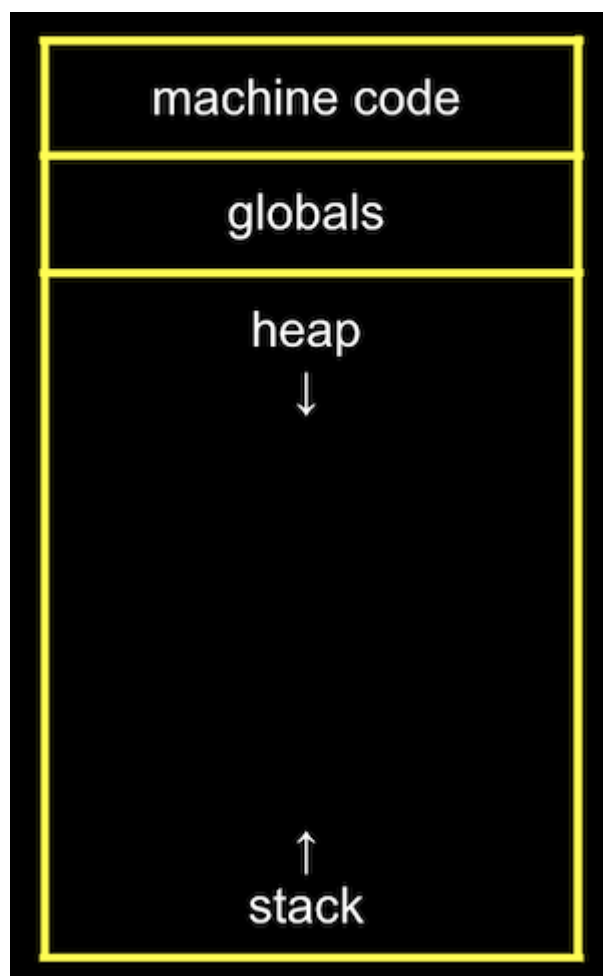
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

```

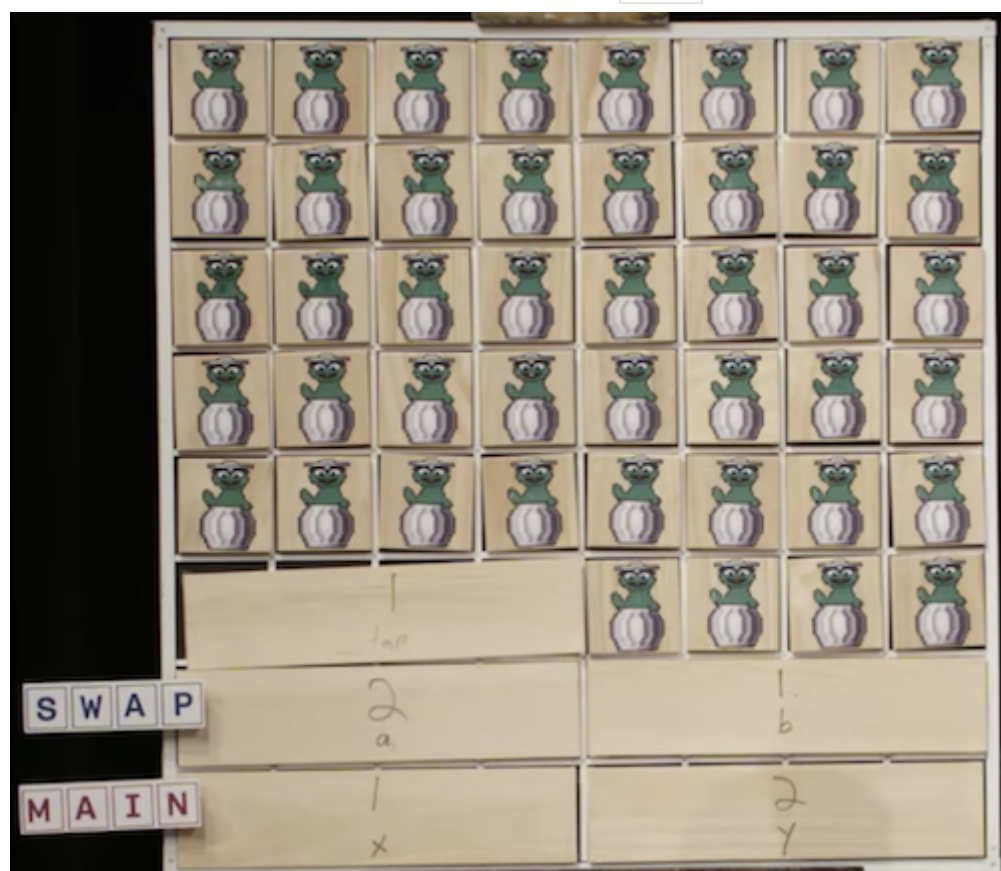
- En el mundo real, si tuviéramos un líquido rojo en un vaso y un líquido azul en otro, y quisiéramos intercambiarlos, necesitaríamos un tercer vaso para contener temporalmente uno de los líquidos, quizás el vaso rojo. Luego podemos verter el líquido azul en el primer vaso, y finalmente el líquido rojo del vaso temporal en el segundo.
- En nuestra `swap` función, también tenemos una tercera variable para usar como espacio de almacenamiento temporal. Ponemos `a` en `tmp`, y luego establecemos `a` el valor de `b`, y finalmente `b` se puede cambiar al valor original de `a`, ahora en `tmp`.
- Pero, si intentamos usar esa función en un programa, no vemos ningún cambio. Resulta que la `swap` función obtiene sus propias variables, `a` y `b` cuando se pasan, son copias de `x` y `y`, por lo que cambiar esos valores no cambia `x` y `y` en la `main` función.

Disposición de la memoria

- Dentro de la memoria de nuestra computadora, los diferentes tipos de datos que necesitan ser almacenados para nuestro programa están organizados en diferentes secciones:



- La sección de **código de máquina** es el código binario de nuestro programa compilado. Cuando ejecutamos nuestro programa, ese código se carga en la "parte superior" de la memoria.
- Justo debajo, o en la siguiente parte de la memoria, están **las variables globales** que declaramos en nuestro programa.
- La sección del **montón** es un área vacía desde donde `malloc` puede obtener memoria libre para que la use nuestro programa. Como llamamos `malloc`, comenzamos a asignar memoria de arriba hacia abajo.
- La sección de **pila** es utilizada por funciones en nuestro programa como se llaman, y crece hacia arriba. Por ejemplo, nuestra `main` función está en la parte inferior de la pila y tiene las variables locales `x` y `y`. La `swap` función, cuando se llama, tiene su propia área de memoria que está encima de `main`'s, con las variables locales `a`, `b` y `tmp`:



- Una vez que la función `swap` regresa, la memoria que estaba usando se libera para la siguiente llamada a la función. `x` y `y` son

- Una vez que la función `swap` regresa, la memoria que estaba usando se libera para la siguiente llamada a la función. `x` y `y` son argumentos, por lo que se copian como `a` y `b` para `swap`, por lo que no volvemos a ver nuestros cambios en `main`.
- Al pasar la dirección de `x` y `y`, nuestra `swap` función puede funcionar:

```
#include <stdio.h>

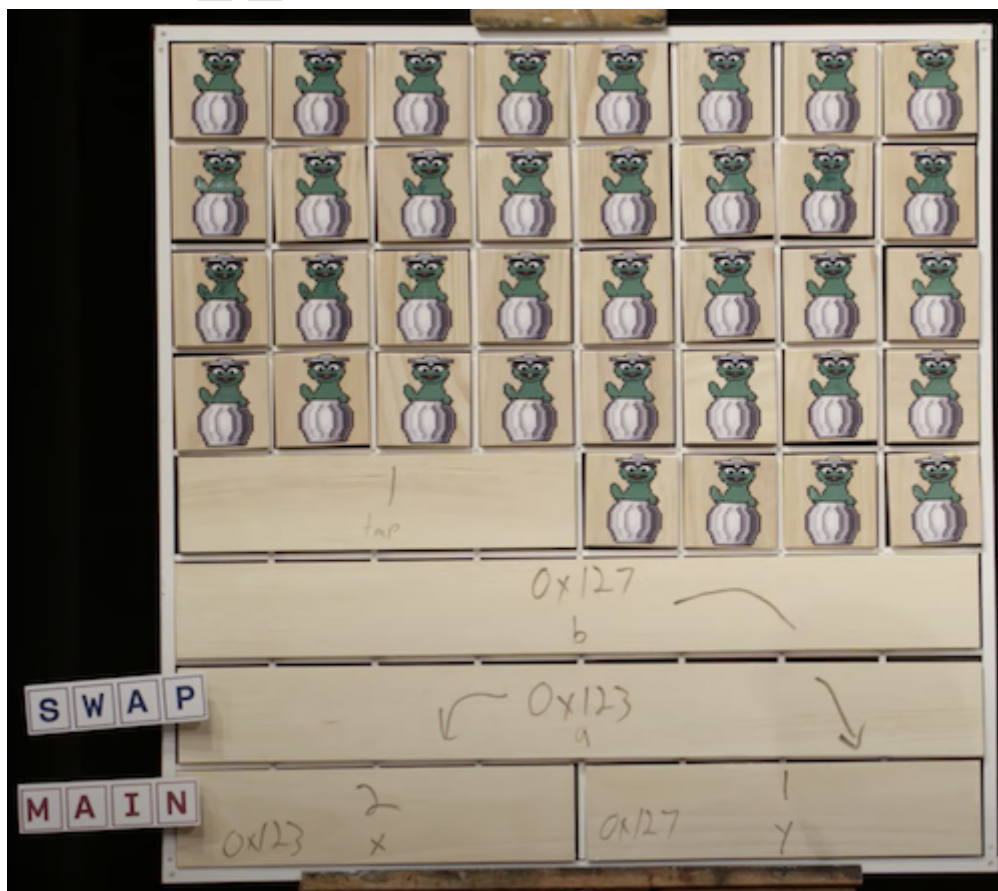
void swap(int *a, int *b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x is %i, y is %i\n", x, y);
    swap(&x, &y);
    printf("x is %i, y is %i\n", x, y);
}

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

- Las direcciones de `x` y `y` se pasan de `main` a `swap` con `&x` y `&y`, y usamos la `int *a` sintaxis para declarar que nuestra `swap` función toma punteros. Guardamos el valor de `x` a `tmp` siguiendo el puntero `a`, y luego tomamos el valor de `y` siguiendo el puntero `b`, y lo almacenamos en la ubicación a la `a` que apunta (`x`). Finalmente, almacenamos el valor de `tmp` en la ubicación apuntada por `b` (`y`), y hemos terminado:



- Si pedimos `malloc` demasiada memoria, tendremos un **desbordamiento del montón**, ya que terminamos pasando nuestro montón. O, si llamamos a demasiadas funciones sin regresar de ellas, tendremos un **desbordamiento de pila**, donde nuestra pila también tiene demasiada memoria asignada.
- Implementemos el dibujo de la pirámide de Mario, llamando a una función:

```

#include <cs50.h>
#include <stdio.h>

void draw(int h);

int main(void)
{
    int height = get_int("Height: ");
    draw(height);
}

void draw(int h)
{
    for (int i = 1; i <= h; i++)
    {
        for (int j = 1; j <= i; j++)
        {
            printf("#");
        }
        printf("\n");
    }
}

```

- Podemos cambiar `draw` para ser recursivos:

```

void draw(int h)
{
    draw(h - 1);

    for (int i = 0; i < h; i++)
    {
        printf("#");
    }
    printf("\n");
}

```

- Cuando intentamos compilar esto con `make`, vemos una advertencia de que la `draw` función se llamará a sí misma de forma recursiva sin detenerse. Así que lo usaremos `clang` sin las comprobaciones adicionales, y cuando ejecutamos este programa, obtenemos una falla de segmentación de inmediato. `draw` se llama a sí mismo una y otra vez, y nos quedamos sin memoria en la pila.
- Al agregar un caso base, la `draw` función dejará de llamarse a sí misma en algún momento:

```

void draw(int h)
{
    if (h == 0)
    {
        return;
    }

    draw(h - 1);

    for (int i = 0; i < h; i++)
    {

```

```

        printf("#");
    }
    printf("\n");
}

```

- Pero si ingresamos un valor lo suficientemente grande para la altura, como `20000000000`, todavía nos quedaremos `draw` sin memoria, ya que estamos llamando demasiadas veces sin regresar.
- Un **desbordamiento de búfer** ocurre cuando pasamos el final de un búfer, una parte de la memoria que hemos asignado como una matriz y accedemos a la memoria que no deberíamos estar.

scanf

- Podemos aplicar `get_int` a nosotros mismos con una función de biblioteca C, `scanf`:

```

#include <stdio.h>

int main(void)
{
    int x;
    printf("x: ");
    scanf("%i", &x);
    printf("x: %i\n", x);
}

```

- `scanf` toma un formato, `%i` por lo que la entrada se "escanea" para ese formato. También pasamos la dirección en la memoria donde queremos que vaya esa entrada. Pero `scanf` no tiene mucha verificación de errores, por lo que es posible que no obtengamos un número entero.
- Podemos intentar obtener una cadena de la misma manera:

```

#include <stdio.h>

int main(void)
{
    char *s;
    printf("s: ");
    scanf("%s", s);
    printf("s: %s\n", s);
}

```

- Pero en realidad no hemos asignado memoria para `s`, por lo que necesitamos llamar `malloc` para asignar memoria para caracteres para nuestra cadena. También podríamos utilizar `char s[4];` para declarar una matriz de cuatro caracteres. Luego, `s` se tratará como un puntero al primer carácter en `scanf` y `printf`.
- Ahora, si el usuario escribe una cadena de longitud 3 o menos, nuestro programa funcionará de forma segura. Pero si el usuario escribe una cadena más larga, `scanf` podría estar intentando escribir más allá del final de nuestra matriz en una memoria desconocida, provocando que nuestro programa se bloquee.
- `get_string` de la biblioteca CS50 asigna continuamente más memoria a medida que `scanf` lee más caracteres, por lo que no tiene ese problema.

Archivos

- Con la capacidad de usar punteros, también podemos abrir archivos, como una guía telefónica digital:

```

#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    FILE *file = fopen("phonebook.csv", "a");
    if (file == NULL)
    {
        return 1;
    }

    char *name = get_string("Name: ");
}

```

```
char *name = get_string("Name: ");
char *number = get_string("Number: ");

fprintf(file, "%s,%s\n", name, number);

fclose(file);
}
```

- `fopen` es una nueva función que podemos usar para abrir un archivo. Devolverá un puntero a un nuevo tipo `FILE`, desde el que podemos leer y escribir. El primer argumento es el nombre del archivo, y el segundo argumento es el modo en el que queremos abrir el archivo (`r` para leer, `w` escribir y `a` agregar o agregar).
- Agregaremos una marca para salir si no pudimos abrir el archivo por alguna razón.
- Después de obtener algunas cadenas, podemos usar `fprintf` para imprimir en un archivo.
- Finalmente, cerramos el archivo con `fclose`.

- Ahora podemos crear nuestros propios archivos CSV, un archivo de valores separados por comas (como una mini hoja de cálculo), mediante programación.

Gráficos

- Podemos leer en binario y mapearlos a píxeles y colores, para mostrar imágenes y videos. Sin embargo, con un número finito de bits en un archivo de imagen, solo podemos acercarnos hasta cierto punto antes de comenzar a ver píxeles individuales.
 - Sin embargo, con la inteligencia artificial y el aprendizaje automático, podemos usar algoritmos que pueden generar detalles adicionales que no existían antes, adivinando en base a otros datos.
- Veamos un programa que abre un archivo y nos dice si es un archivo JPEG, un archivo de imagen en un formato particular:

```
#include <stdint.h>
#include <stdio.h>

typedef uint8_t BYTE;

int main(int argc, char *argv[])
{
    // Check usage
    if (argc != 2)
    {
        return 1;
    }

    // Open file
    FILE *file = fopen(argv[1], "r");
    if (!file)
    {
        return 1;
    }

    // Read first three bytes
    BYTE bytes[3];
    fread(bytes, sizeof(BYTE), 3, file);

    // Check first three bytes
    if (bytes[0] == 0xff && bytes[1] == 0xd8 && bytes[2] == 0xff)
    {
        printf("Maybe\n");
    }
    else
    {
        printf("No\n");
    }

    // Close file
    fclose(file);
}
```

- Primero, definimos a `BYTE` como 8 bits, por lo que podemos referirnos a un byte como un tipo más fácilmente en C.
- Luego, intentamos abrir un archivo (comprobando que efectivamente obtenemos un archivo no NULL) y leemos los primeros tres bytes del archivo con `fread` en un búfer llamado `bytes`.

bytes del archivo con `fread`, en un buffer llamado `bytes`.

- Podemos comparar los primeros tres bytes (en hexadecimal) con los tres bytes necesarios para comenzar un archivo JPEG. Si son iguales, es probable que nuestro archivo sea un archivo JPEG (aunque otros tipos de archivos pueden comenzar con esos bytes). Pero si no son iguales, sabemos que definitivamente no es un archivo JPEG.
- Incluso podemos copiar archivos nosotros mismos, un byte a la vez ahora:

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

typedef uint8_t BYTE;

int main(int argc, char *argv[])
{
    // Ensure proper usage
    if (argc != 3)
    {
        fprintf(stderr, "Usage: copy SOURCE DESTINATION\n");
        return 1;
    }

    // open input file
    FILE *source = fopen(argv[1], "r");
    if (source == NULL)
    {
        printf("Could not open %s.\n", argv[1]);
        return 1;
    }

    // Open output file
    FILE *destination = fopen(argv[2], "w");
    if (destination == NULL)
    {
        fclose(source);
        printf("Could not create %s.\n", argv[2]);
        return 1;
    }

    // Copy source to destination, one BYTE at a time
    BYTE buffer;
    while (fread(&buffer, sizeof(BYTE), 1, source))
    {
        fwrite(&buffer, sizeof(BYTE), 1, destination);
    }

    // Close files
    fclose(source);
    fclose(destination);
    return 0;
}
```

- Usamos `argv` para obtener argumentos, usándolos como nombres de archivo para abrir archivos para leer y uno para escribir.
- Luego, leemos un byte del `source` archivo en un búfer y escribimos ese byte en el `destination` archivo. Podemos usar un `while` bucle para llamar `fread`, que se detendrá una vez que no queden más bytes para leer.
- Podemos usar estas habilidades para leer y escribir archivos, recuperar imágenes de un archivo y agregar filtros a las imágenes cambiando los bytes que contienen, ¡en el conjunto de problemas de esta semana!