








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://www.quora.com/profile/David-J-Malan>) 
(<https://www.reddit.com/user/davidjmalan>)  (<https://twitter.com/davidjmalan>)

Clase 2

- [Compilando](#)
- [Depuración](#)
- [Memoria](#)
- [Matrices](#)
- [Caracteres](#)
- [Instrumentos de cuerda](#)
- [Argumentos de la línea de comandos](#)
- [Aplicaciones](#)

Compilando

- La última vez, aprendimos a escribir nuestro primer programa en C, imprimiendo "hola, mundo" en la pantalla.
- Hemos compilado con `make hello` primero, convirtiendo nuestro código fuente en código máquina antes de poder ejecutar el programa compilado con `./hello`.
- `make` en realidad es solo un programa que llama `clang`, un compilador, con opciones. Podríamos compilar nuestro archivo de código fuente `hello.c`, nosotros mismos ejecutando el comando `clang hello.c`. No parece suceder nada, lo que significa que no hubo errores. Y si lo ejecutamos `ls`, ahora vemos un `a.out` archivo en nuestro directorio. El nombre de archivo sigue siendo el predeterminado, por lo que realmente puede ejecutar un comando más específico: `clang -o hello hello.c`.
- Hemos agregado otro **argumento en la línea de comandos** o una entrada a un programa en la línea de comandos como palabras adicionales después del nombre del programa. `clang` es el nombre del programa, y `-o`, `hello` y `hello.c` son argumentos adicionales. Le decimos `clang` que lo use `hello` como nombre de archivo de *salida* y lo use `hello.c` como código fuente. Ahora, podemos ver que `hello` se crea como salida.
- Si quisiéramos usar la biblioteca de CS50, via `#include <cs50.h>`, para la `get_string` función, también tenemos que agregar una bandera `clang -o hello hello.c -lcs50`:

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

int main(void)
{
    string name = get_string("What's your name? ");
    printf("hello, %s\n", name);
}
```

- La `-l` bandera vincula el `cs50` archivo, que ya está instalado en el IDE de CS50, e incluye el código de máquina para `get_string` (entre otras funciones) que nuestro programa puede consultar y utilizar también.
- Con `make`, estos argumentos se generan para nosotros ya que el personal también se ha configurado `make` en el IDE CS50.
- La compilación del código fuente en código de máquina en realidad se compone de pasos más pequeños:
 - preprocesamiento
 - compilando
 - montaje
 - enlace

- El **preprocesamiento** generalmente implica líneas que comienzan con un `#`, como `#include`. Por ejemplo, `#include <cs50.h>` le dirá `clang` que busque ese archivo de encabezado, ya que contiene contenido que queremos incluir en nuestro programa. Luego, `clang` esencialmente reemplazará el contenido de esos archivos de encabezado en nuestro programa.
- Por ejemplo ...

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

int main(void)
{
    string name = get_string("What's your name? ");
    printf("hello, %s\n", name);
}
```

- ... se preprocesará en:

```
...
string get_string(string prompt);
int printf(string format, ...);
...

int main(void)
{
    string name = get_string("Name: ");
    printf("hello, %s\n", name);
}
```

- Esto incluye los prototipos de todas las funciones de esas bibliotecas que incluimos, para que luego podamos usarlas en nuestro código.
- La **compilación** toma nuestro código fuente, en C, y lo convierte a otro tipo de código fuente llamado **código ensamblador**, que se ve así:

```
...
main:                                # @main
    .cfi_startproc
# BB#0:
    pushq    %rbp
.Ltmp0:
    .cfi_def_cfa_offset 16
.Ltmp1:
    .cfi_offset %rbp, -16
    movq     %rsp, %rbp
.Ltmp2:
    .cfi_def_cfa_register %rbp
    subq     $16, %rsp
    xorl     %eax, %eax
    movl     %eax, %edi
    movabsq  $.L.str, %rsi
    movb     $0, %al
    callq    get_string
    movabsq  $.L.str.1, %rdi
    movq     %rax, -8(%rbp)
    movq     -8(%rbp), %rsi
    movb     $0, %al
    callq    printf
    ...
```

- Estas instrucciones son de nivel inferior y están más cerca de las instrucciones binarias que el procesador de una computadora puede entender directamente. Por lo general, operan en bytes en sí mismos, a diferencia de abstracciones como nombres de variables.
- El siguiente paso es tomar el código ensamblador y traducirlo a instrucciones en binario **ensamblándolo**. Las instrucciones en binario se denominan **código de máquina**, que la CPU de una computadora puede ejecutar directamente.
- El último paso es la **vinculación**, donde las versiones compiladas previamente de las bibliotecas que incluimos anteriormente, en `cs50.c` realidad, se combinan con el binario de nuestro programa. Así que terminamos con un archivo binario, `a.out` o `hello`, que es el código de la máquina combinada para `hello.c`, `cs50.c` y `stdio.c`. (En CS50 IDE, el código de máquina precompilado para `cs50.c` y `stdio.c` ya se ha instalado, y `clang` se ha configurado para encontrarlos y utilizarlos).
- Estos cuatro pasos han sido resumidos o simplificados por, por `make` lo que todo lo que tenemos que implementar es el código para nuestros programas.

Depuración

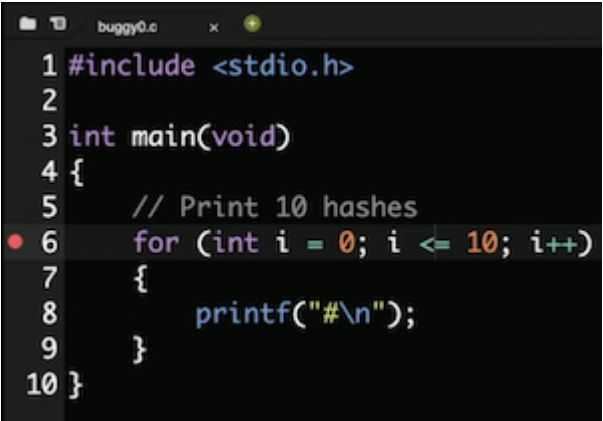
- **Los errores** son errores o problemas en los programas que hacen que se comporten de manera diferente a lo previsto. Y la depuración es el proceso de encontrar y corregir esos errores.
- La semana pasada, aprendimos sobre algunas herramientas que nos ayudan a escribir código que se compila, tiene buen estilo y es correcto:
 - `help50`
 - `style50`
 - `check50`
- Podemos usar otra “herramienta”, la `printf` función, para imprimir mensajes y variables que nos ayuden a depurar.
- Echemos un vistazo a `buggy0.c`:

```
#include <stdio.h>

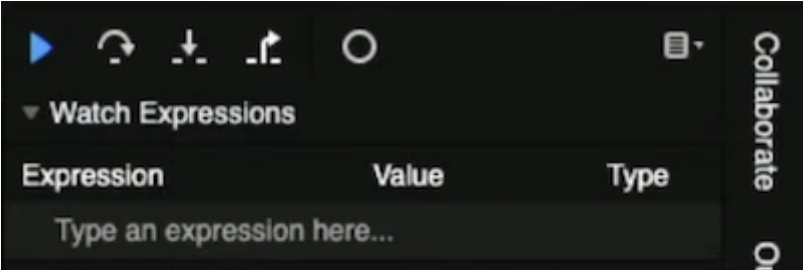
int main(void)
{
    // Print 10 hashes
    for (int i = 0; i <= 10; i++)
    {
        printf("#\n");
    }
}
```

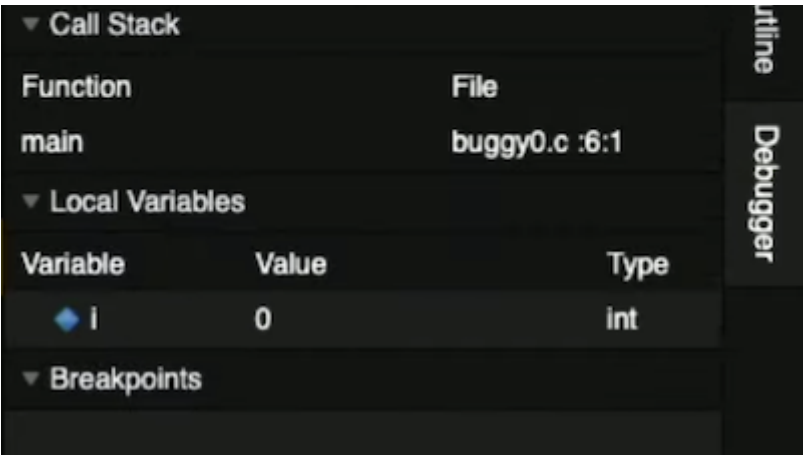
- Hmm, queremos imprimir solo 10 `#`s, pero hay 11. Si no supiéramos cuál es el problema (ya que nuestro programa se está compilando sin errores, y ahora tenemos un error lógico), podríamos agregar otro `printf` temporalmente:
- ```
#include <stdio.h>

int main(void)
{
 for (int i = 0; i <= 10; i++)
 {
 printf("i is now %i\n", i);
 printf("#\n");
 }
}
```
- Ahora, podemos ver que `i` comenzó en 0 y continuó hasta que fue 10, pero deberíamos hacer que nuestro `for` bucle se detenga una vez que esté en 10, con en `i < 10` lugar de `i <= 10`.
  - En el CS50 IDE, tenemos otra herramienta `debug50`, para ayudarnos a depurar programas. Esta es una herramienta escrita por personal que se basa en una herramienta estándar llamada `gdb`. Ambos **depuradores** son programas que ejecutarán nuestros propios programas paso a paso y nos permitirán ver variables y otra información mientras nuestro programa se está ejecutando.
  - Ejecutaremos el comando `debug50 ./buggy0` y nos dirá que recompilemos nuestro programa ya que lo cambiamos. Luego, nos dirá que agreguemos un **punto de interrupción** o indicador para una línea de código donde el depurador debería pausar nuestro programa.
    - Al usar las teclas de arriba y abajo en la terminal, podemos reutilizar comandos del pasado sin tener que volver a escribirlos.
  - Haremos clic a la izquierda de la línea 6 en nuestro código y aparecerá un círculo rojo:



- Ahora, si ejecutamos de `debug50 ./buggy0` nuevo, veremos el panel del depurador abierto a la derecha:





- Vemos que la variable que creamos `i`, está debajo de la `Local Variables` sección y vemos que hay un valor de `0`.
- Nuestro punto de interrupción ha detenido nuestro programa en la línea 6, resaltando esa línea en amarillo. Para continuar, tenemos algunos controles en el panel del depurador. El triángulo azul continuará nuestro programa hasta que alcancemos otro punto de ruptura o el final de nuestro programa. La flecha curva a su derecha, Step Over, “saltará” la línea, la ejecutará y volverá a pausar nuestro programa inmediatamente después.
- Entonces, usaremos la flecha curva para ejecutar la siguiente línea y veremos qué cambia después. Estamos en la `printf` línea, y presionando la flecha curva nuevamente, vemos una sola `#` impresa en nuestra ventana de terminal. Con otro clic de la flecha, vemos el valor de `i` cambiar a `1`. Podemos seguir haciendo clic en la flecha para ver cómo se ejecuta nuestro programa, una línea a la vez.
- Para salir del depurador, podemos presionar `control + C` para detener el programa en ejecución.
- Veamos otro ejemplo `buggy1.c`:

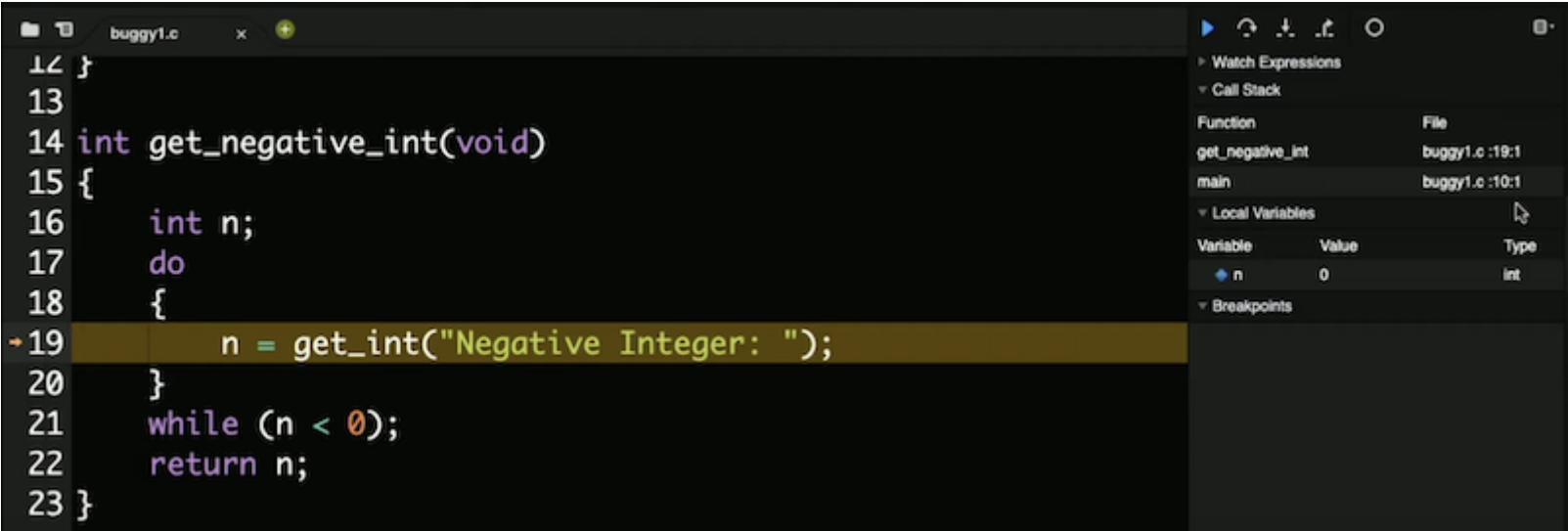
```
#include <cs50.h>
#include <stdio.h>

// Prototype
int get_negative_int(void);

int main(void)
{
 // Get negative integer from user
 int i = get_negative_int();
 printf("%i\n", i);
}

int get_negative_int(void)
{
 int n;
 do
 {
 n = get_int("Negative Integer: ");
 }
 while (n < 0);
 return n;
}
```

- Hemos implementado otra función, `get_negative_int` para obtener un número entero negativo del usuario. Necesitamos recordar el prototipo antes de nuestra `main` función, y luego compilamos nuestro código.
- Pero cuando ejecutamos nuestro programa, sigue pidiéndonos un número entero negativo, incluso después de que proporcionamos uno. Estableceremos un punto de interrupción en la línea 10 `int i = get_negative_int();`, ya que es la primera línea de código interesante. Ejecutaremos `debug50 ./buggy1` y veremos en la sección Pila de llamadas del panel de depuración que estamos en la `main` función. (La "pila de llamadas" se refiere a todas las funciones que se han llamado en nuestro programa en ese momento, y aún no se han devuelto. Hasta ahora, solo `main` se ha llamado a la función).
- Haremos clic en la flecha que apunta hacia abajo, Step Into, y el depurador nos lleva a la función llamada en esa línea `get_negative_int`,. Vemos la pila de llamadas actualizada con el nombre de la función y la variable `n` con un valor de `0`:



- Podemos volver a hacer clic en la flecha de paso a paso y ver cómo `n` se actualiza `-1`, que es lo que ingresamos:

```

12 }
13
14 int get_negative_int(void)
15 {
16 int n;
17 do
18 {
19 n = get_int("Negative Integer: ");
20 }
21 while (n < 0);
22 return n;
23 }

```

```

~/ $ debug50 ./buggy1
Negative Integer: -1

```

- Hacemos clic en Step Over nuevamente y vemos que nuestro programa regresa dentro del ciclo. Nuestro `while` bucle todavía se está ejecutando, por lo que la condición que comprueba debe ser `true` aún. Y vemos que `n < 0` es cierto incluso si ingresamos un número entero negativo, por lo que deberíamos corregir nuestro error cambiándolo a `n >= 0`.
- ¡Podemos ahorrar mucho tiempo en el futuro invirtiendo un poco ahora para aprender a usarlo `debug50`!
- También podemos usar `ddb`, abreviatura de "depurador de pato", una [técnica real](https://en.wikipedia.org/wiki/Rubber_duck_debugging) ([https://en.wikipedia.org/wiki/Rubber\\_duck\\_debugging](https://en.wikipedia.org/wiki/Rubber_duck_debugging)) en la que explicamos lo que estamos tratando de hacer con un pato de goma, y muchas veces nos daremos cuenta de nuestro propio error en la lógica o la implementación mientras lo explicamos.

## Memoria

- En C, tenemos diferentes tipos de variables que podemos usar para almacenar datos, y cada una de ellas ocupa una cantidad fija de espacio. Los diferentes sistemas informáticos en realidad varían en la cantidad de espacio que se usa realmente para cada tipo, pero trabajaremos con las cantidades aquí, como se usa en el IDE de CS50:
  - `bool` 1 byte
  - `char` 1 byte
  - `double` 8 bytes
  - `float` 4 bytes
  - `int` 4 bytes
  - `long` 8 bytes
  - `string` ? bytes
  - ...
- Dentro de nuestras computadoras, tenemos chips llamados RAM, memoria de acceso **aleatorio**, que almacena datos para uso a corto plazo, como el código de un programa mientras se está ejecutando o un archivo mientras está abierto. Podríamos guardar un programa o archivo en nuestro disco duro (o SSD, unidad de estado sólido) para un almacenamiento a largo plazo, pero usamos RAM porque es mucho más rápido. Sin embargo, la RAM es volátil o requiere energía para mantener los datos almacenados.
- Podemos pensar en los bytes almacenados en la RAM como si estuvieran en una cuadrícula:







- En realidad, hay millones o miles de millones de bytes por chip.
- Cada byte tendrá una ubicación en el chip, como el primer byte, el segundo byte y así sucesivamente.
- En C, cuando creamos una variable de tipo `char`, que tendrá un tamaño de un byte, se almacenará físicamente en una de esas cajas en la RAM. Un número entero, con 4 bytes, ocupará cuatro de esos cuadros.

## Matrices

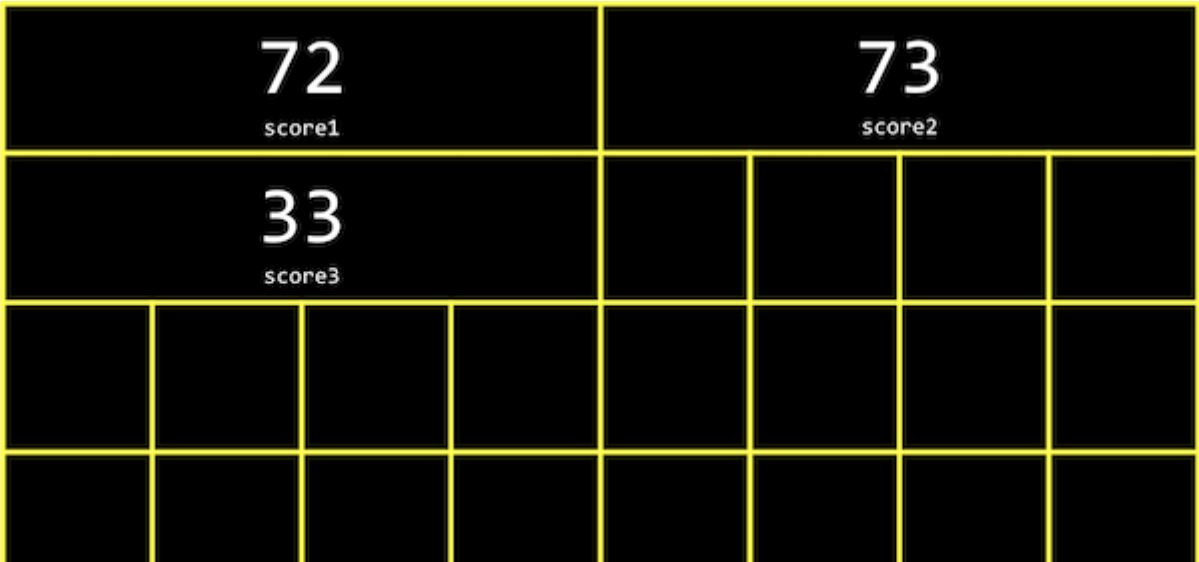
- Digamos que queremos tomar el promedio de tres variables:

```
#include <stdio.h>

int main(void)
{
 int score1 = 72;
 int score2 = 73;
 int score3 = 33;

 printf("Average: %f\n", (score1 + score2 + score3) / 3.0);
}
```

- Dividimos por no `3`, pero `3.0` el resultado también es un flotador.
- Podemos compilar y ejecutar nuestro programa y ver un promedio impreso.
- Mientras nuestro programa se ejecuta, las tres `int` variables se almacenan en la memoria:



- Cada uno `int` ocupa cuatro casillas, que representan cuatro bytes, y cada byte a su vez está compuesto por ocho bits, ceros y unos almacenados por componentes eléctricos.
- Resulta que, en la memoria, podemos almacenar variables una tras otra, una tras otra, y acceder a ellas más fácilmente con bucles. En C, una lista de valores almacenados uno tras otro de forma contigua se denomina **matriz**.
- Para nuestro programa anterior, podemos usar `int scores[3];` para declarar una matriz de tres enteros en su lugar.
- Y podemos asignar y usar variables en una matriz con `scores[0] = 72`. Con los corchetes, estamos indexando o yendo a la posición "0" en la matriz. Las matrices están indexadas a cero, lo que significa que el primer valor tiene un índice 0 y el segundo valor tiene un índice 1, y así sucesivamente.
- Actualicemos nuestro programa para usar una matriz:

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

int main(void)
{
 int scores[3];
 scores[0] = get_int("Score: ");
 scores[1] = get_int("Score: ");
 scores[2] = get_int("Score: ");

 // Print average
```

```
// Print average
printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0);
}
```

- Ahora, le pedimos al usuario tres valores e imprimimos el promedio como antes, pero usando los valores almacenados en la matriz.
- Dado que podemos configurar y acceder a elementos en una matriz en función de su posición, y esa posición *también* puede ser el valor de alguna variable, podemos usar un bucle:

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

int main(void)
{
 int scores[3];
 for (int i = 0; i < 3; i++)
 {
 scores[i] = get_int("Score: ");
 }

 // Print average
 printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0);
}
```

- Ahora, en lugar de codificar o especificar manualmente cada elemento tres veces, usamos un `for` ciclo y `i` como índice de cada elemento en la matriz.
- Y repetimos el valor 3, que representa la longitud de nuestra matriz, en dos lugares diferentes. Entonces podemos usar una **constante** o variable con un valor fijo en nuestro programa:

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

const int TOTAL = 3;

int main(void)
{
 int scores[TOTAL];
 for (int i = 0; i < TOTAL; i++)
 {
 scores[i] = get_int("Score: ");
 }

 printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / TOTAL);
}
```

- Podemos usar la `const` palabra clave para decirle al compilador que `TOTAL` nuestro programa nunca debe cambiar el valor de . Y por convención, colocaremos nuestra declaración de la variable fuera de la `main` función y escribiremos su nombre en mayúscula, lo cual no es necesario para el compilador pero muestra a otros humanos que esta variable es una constante y hace que sea fácil de ver desde el principio.
- Pero ahora nuestro promedio será incorrecto o estará roto si no tenemos exactamente tres valores.
- Agreguemos una función para calcular el promedio:

```
float average(int length, int array[])
{
 int sum = 0;
 for (int i = 0; i < length; i++)
 {
 sum += array[i];
 }
 return sum / (float) length;
}
```

- Pasaremos la longitud y una matriz de `int`s (que podría ser de cualquier tamaño) y usaremos otro bucle dentro de nuestra función auxiliar para sumar los valores en una `sum` variable. Solíamos `(float)` lanzar `length` en un flotador, por lo que el resultado que obtenemos al dividir los dos también es un flotador.
- Ahora, en nuestra `main` función, podemos llamar a nuestra nueva `average` función con `printf("Average: %f\n", average(TOTAL, scores));`. Tenga en cuenta que los nombres de las variables en `main` no necesitan coincidir con lo que los `average` llama, ya que solo se pasan los *valores*.
- Necesitamos pasar la longitud de la matriz a la `average` función, para que sepa cuántos valores hay.

## Caracteres

- Podemos imprimir un solo carácter con un programa simple:

```
#include <stdio.h>

int main(void)
{
 char c = '#';

 printf("%c\n", c);
}
```

- Cuando ejecutamos este programa, nos `#` imprimimos en la terminal.
- Veamos qué pasa si cambiamos nuestro programa para imprimir `c` como un número entero:

```
#include <stdio.h>

int main(void)
{
 char c = '#';

 printf("%i\n", (int) c);
}
```

- Cuando ejecutamos este programa, se `35` imprime. Resulta que de `35` hecho es el código ASCII de un símbolo `#`.
- De hecho, no es necesario enviar `c` a un `int` explícitamente; el compilador puede hacer eso por nosotros en este caso.
- A `char` es un solo byte, por lo que podemos imaginarlo como almacenado en un cuadro en la cuadrícula de memoria de arriba.

## Instrumentos de cuerda

- Podemos imprimir una cadena, o algún texto, creando una variable para cada carácter e imprimiéndolos:

```
#include <stdio.h>

int main(void)
{
 char c1 = 'H';
 char c2 = 'I';
 char c3 = '!';

 printf("%c%c%c\n", c1, c2, c3);
}
```

- Aquí, veremos `HI!` impreso.
- Ahora imprimamos los valores enteros de cada carácter:

```
#include <stdio.h>

int main(void)
{
 char c1 = 'H';
 char c2 = 'I';
 char c3 = '!';
```



```
printf("%i %i %i\n", c1, c2, c3);
}
```

- Veremos 72 73 33 impreso y nos daremos cuenta de que estos caracteres se almacenan en la memoria de esta manera:



- Las cadenas son en realidad solo matrices de caracteres y no están definidas en C sino por la biblioteca CS50. Si tuviéramos una matriz llamada s, cada personaje se puede acceder con s[0], s[1] y así sucesivamente.
- Y resulta que una cadena termina con un carácter especial '\0', o un byte con todos los bits puestos a 0. Este carácter se llama **carácter nulo** o NUL. Entonces, en realidad, necesitamos cuatro bytes para almacenar nuestra cadena con tres caracteres:



- Podemos usar una cadena como una matriz en nuestro programa e imprimir los códigos ASCII, o valores enteros, de cada carácter en la cadena:

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

int main(void)
{
 string s = "HI!";
 printf("%i %i %i %i\n", s[0], s[1], s[2], s[3]);
}
```

- Y como podríamos haber esperado, lo vemos 72 73 33 0 impreso.
- De hecho, podríamos intentar acceder s[4] y ver impreso algún símbolo inesperado. Con C, nuestro código tiene la capacidad de acceder o cambiar la memoria que de otra manera no debería, lo cual es poderoso y peligroso.
- Podemos usar un bucle para imprimir todos los caracteres de una cadena:

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

int main(void)
{
 string s = get_string("Input: ");
 printf("Output: ");
 for (int i = 0; s[i] != '\0'; i++)
 {
 printf("%c", s[i]);
 }
 printf("\n");
}
```

- Podemos cambiar la condición de nuestro bucle para que continúe independientemente de lo que i sea, pero solo cuando s[i] != '\0', o cuando el carácter en la posición actual en s no es el carácter nulo.
- Podemos usar una función que viene con la string biblioteca de C strlen, para obtener la longitud de la cadena para nuestro ciclo:

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

int main(void)
{
 string s = get_string("Input: ");
 printf("Output: ");
 for (int i = 0; i < strlen(s); i++)
 {
 printf("%c", s[i]);
 }
 printf("\n");
}
```

```

 printf("%c", s[i]);
 }
 printf("\n");
}

```

- Tenemos la oportunidad de mejorar el diseño de nuestro programa. Nuestro bucle fue un poco ineficiente, ya que verificamos la longitud de la cadena, después de que se imprime cada carácter, en nuestra condición. Pero dado que la longitud de la cadena no cambia, podemos verificar la longitud de la cadena una vez:

```

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

int main(void)
{
 string s = get_string("Input: ");
 printf("Output:\n");
 for (int i = 0, n = strlen(s); i < n; i++)
 {
 printf("%c\n", s[i]);
 }
}

```

- Ahora, en el inicio de nuestro bucle, inicializamos la vez una `i` y `n` variables, y recordamos la duración de nuestra cadena en `n`. Luego, podemos verificar los valores sin tener que llamar `strlen` para calcular la longitud de la cadena cada vez.
- Y necesitamos usar un poco más de memoria para almacenar `n`, pero esto nos ahorra algo de tiempo sin tener que verificar la longitud de la cadena cada vez.
- Podríamos declarar una matriz de dos cadenas:

```

string words[2];
words[0] = "HI!";
words[1] = "BYE!";

```

- Y en la memoria, la matriz de cadenas se puede almacenar y acceder con:

|             |             |             |             |             |             |             |             |
|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
| H           | I           | !           | \0          | B           | Y           | E           | !           |
| words[0][0] | words[0][1] | words[0][2] | words[0][3] | words[1][0] | words[1][1] | words[1][2] | words[1][3] |
| \0          |             |             |             |             |             |             |             |
| words[1][4] |             |             |             |             |             |             |             |
|             |             |             |             |             |             |             |             |

- `words[0]` se refiere al primer elemento, o valor, de la `words` matriz, que es una cadena, por lo que se `words[0][0]` refiere al primer elemento de esa cadena, que es un carácter.
- Entonces, una matriz de cadenas es solo una matriz de matrices de caracteres.
- Ahora podemos combinar lo que hemos visto para escribir un programa que pueda usar letras mayúsculas:

```

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

int main(void)
{
 string s = get_string("Before: ");
 printf("After: ");
 for (int i = 0, n = strlen(s); i < n; i++)
 {
 if (s[i] >= 'a' && s[i] <= 'z')
 {
 printf("%c", s[i] - 32);
 }
 }
}

```

```

 }
 else
 {
 printf("%c", s[i]);
 }
}
printf("\n");
}

```

- Primero, obtenemos una cadena `s` del usuario. Luego, para cada carácter de la cadena, si está en minúsculas (lo que significa que tiene un valor entre el de `a` y `z`), lo convertimos a mayúsculas. De lo contrario, simplemente lo imprimimos.
- Podemos convertir una letra minúscula a su equivalente en mayúscula restando la diferencia entre sus valores ASCII. (Sabemos que las letras minúsculas tienen un valor ASCII más alto que las letras mayúsculas, y la diferencia es la misma entre las mismas letras, por lo que podemos restar para obtener una letra mayúscula de una letra minúscula).
- Resulta que hay otra biblioteca `ctype.h`, que podemos usar:

```

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

int main(void)
{
 string s = get_string("Before: ");
 printf("After: ");
 for (int i = 0, n = strlen(s); i < n; i++)
 {
 if (islower(s[i]))
 {
 printf("%c", toupper(s[i]));
 }
 else
 {
 printf("%c", s[i]);
 }
 }
 printf("\n");
}

```

- Ahora, nuestro código es más legible y probablemente correcto, ya que otros han escrito y probado estas funciones por nosotros.
- Podemos simplificar aún más y simplemente pasar cada carácter a `toupper`, ya que no cambia los caracteres que no sean minúsculas:

```

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

int main(void)
{
 string s = get_string("Before: ");
 printf("After: ");
 for (int i = 0, n = strlen(s); i < n; i++)
 {
 printf("%c", toupper(s[i]));
 }
 printf("\n");
}

```

- Podemos usar las **páginas** (<https://man.cs50.io/>) del **manual** (<https://man.cs50.io/>) de CS50 para encontrar y aprender sobre las funciones comunes de la biblioteca. Al buscar en las páginas de manual, vemos `toupper()` una función, entre otras, de una biblioteca llamada `ctype`, que podemos usar.

## Argumentos de la línea de comandos

- Los programas propios también pueden aceptar argumentos de línea de comandos o palabras agregadas después del nombre de nuestro programa en el comando mismo.
- En `argv.c`, cambiamos el `main` aspecto de nuestra función:

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

int main(int argc, string argv[])
{
 if (argc == 2)
 {
 printf("hello, %s\n", argv[1]);
 }
 else
 {
 printf("hello, world\n");
 }
}
```

- `argc` y `argv` son dos variables que nuestra `main` función ahora obtendrá automáticamente cuando nuestro programa se ejecute desde la línea de comandos. `argc` es el *número de argumentos*, o el número de argumentos, y `argv`, *vector de argumento* (o lista de argumentos), una matriz de cadenas.
- El primer argumento, `argv[0]` es el nombre de nuestro programa (la primera palabra escrita, como `./hello`). En este ejemplo, verificamos si tenemos dos argumentos e imprimimos el segundo si es así.
- Por ejemplo, si ejecutamos `./argv David`, se `hello, David` imprimirá, ya que escribimos `David` como la segunda palabra en nuestro comando.
- También podemos imprimir cada carácter individualmente:

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

int main(int argc, string argv[])
{
 if (argc == 2)
 {
 for (int i = 0, n = strlen(argv[1]); i < n; i++)
 {
 printf("%c\n", argv[1][i]);
 }
 }
}
```

- Usaremos `argv[1][i]` para acceder a cada carácter en el primer argumento de nuestro programa.
- Resulta que nuestra `main` función también devuelve un valor entero. Por defecto, nuestra `main` función regresa `0` para indicar que nada salió mal, pero podemos escribir un programa para devolver un valor diferente:

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

int main(int argc, string argv[])
{
 if (argc != 2)
 {
 printf("missing command-line argument\n");
 return 1;
 }
 printf("hello, %s\n", argv[1]);
 return 0;
}
```

- El valor de retorno de `main` en nuestro programa se llama **código de salida**, generalmente se usa para indicar códigos de error. (Escribiremos `return 0` explícitamente al final de nuestro programa aquí, aunque técnicamente no es necesario).
- A medida que escribimos programas más complejos, códigos de error como este pueden ayudarnos a determinar qué salió mal, incluso si no es visible o significativo para el usuario.

## Aplicaciones

- Ahora que sabemos cómo trabajar con cadenas en nuestros programas, así como con el código escrito por otros en bibliotecas, podemos

analizar párrafos de texto para determinar su nivel de legibilidad, basándonos en factores como qué tan largas y complicadas son las palabras y oraciones.

- **La criptografía** es el arte de codificar u ocultar información. Si quisiéramos enviar un mensaje a alguien, es posible que queramos **encriptar** , o codificar de alguna manera ese mensaje para que sea difícil de leer para otros. El mensaje original, o entrada a nuestro algoritmo, se llama **texto plano** y el mensaje cifrado, o salida, se llama **texto cifrado** . Y el algoritmo que realiza la codificación se llama **cifrado** . Un cifrado generalmente requiere otra entrada además del texto sin formato. Una **clave** , como un número, es otra entrada que se mantiene en secreto.
- Por ejemplo, si quisiéramos enviar un mensaje como `I L O V E Y O U` , podemos convertirlo primero a ASCII: `73 76 79 86 69 89 79 85` . Entonces, podemos cifrarlo con una clave de poco `1` y un algoritmo sencillo, en el que sólo tiene que añadir la clave para cada valor: `74 77 80 87 70 90 80 86` . Entonces, el texto cifrado después de convertir los valores a ASCII sería `J M P W F Z P V` . Para descifrar esto, alguien tendría que saber cuál es la clave `1` y restarla de cada carácter.
- ¡Aplicaremos estos conceptos en nuestras secciones y conjunto de problemas!