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 1

- C
- CS50 IDE
- Compilando
- Funciones y argumentos
- principales, archivos de encabezado
- Instrumentos
- Comandos
- Tipos, códigos de formato,
- Operadores, limitaciones, truncamiento
- Variables, azúcar sintáctico
- Condiciones
- Expresiones booleanas, bucles
- Abstracción
- Mario
- Memoria, imprecisión y desborde

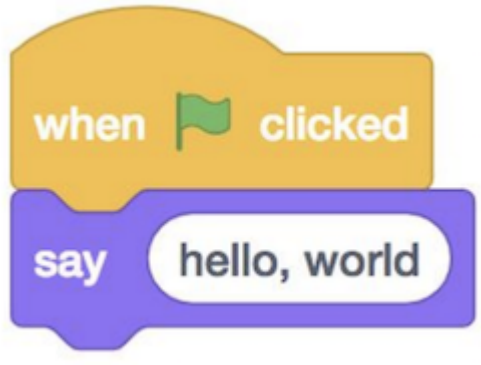
C

- Hoy aprenderemos un nuevo lenguaje, **C** : un lenguaje de programación que tiene todas las características de Scratch y más, pero quizás un poco menos amigable ya que es puramente en texto:

```c
#include <stdio.h>

int main(void)
{
    printf("hello, world");
}
```

- Aunque al principio, para tomar prestada una frase del MIT, tratar de absorber todos estos nuevos conceptos puede parecer como beber de una manguera contra incendios, tenga la seguridad de que al final del semestre estaremos capacitados y experimentados en el aprendizaje y la aplicación de estos conceptos. .
- Podemos comparar muchas de las funciones de programación en C con bloques que ya hemos visto y usado en Scratch. Los detalles de la sintaxis son mucho menos importantes que las ideas, que ya nos han presentado.
- En nuestro ejemplo, aunque las palabras son nuevas, las ideas son exactamente las mismas que los bloques "al hacer clic en la bandera verde" y "decir (hola, mundo)" en Scratch:

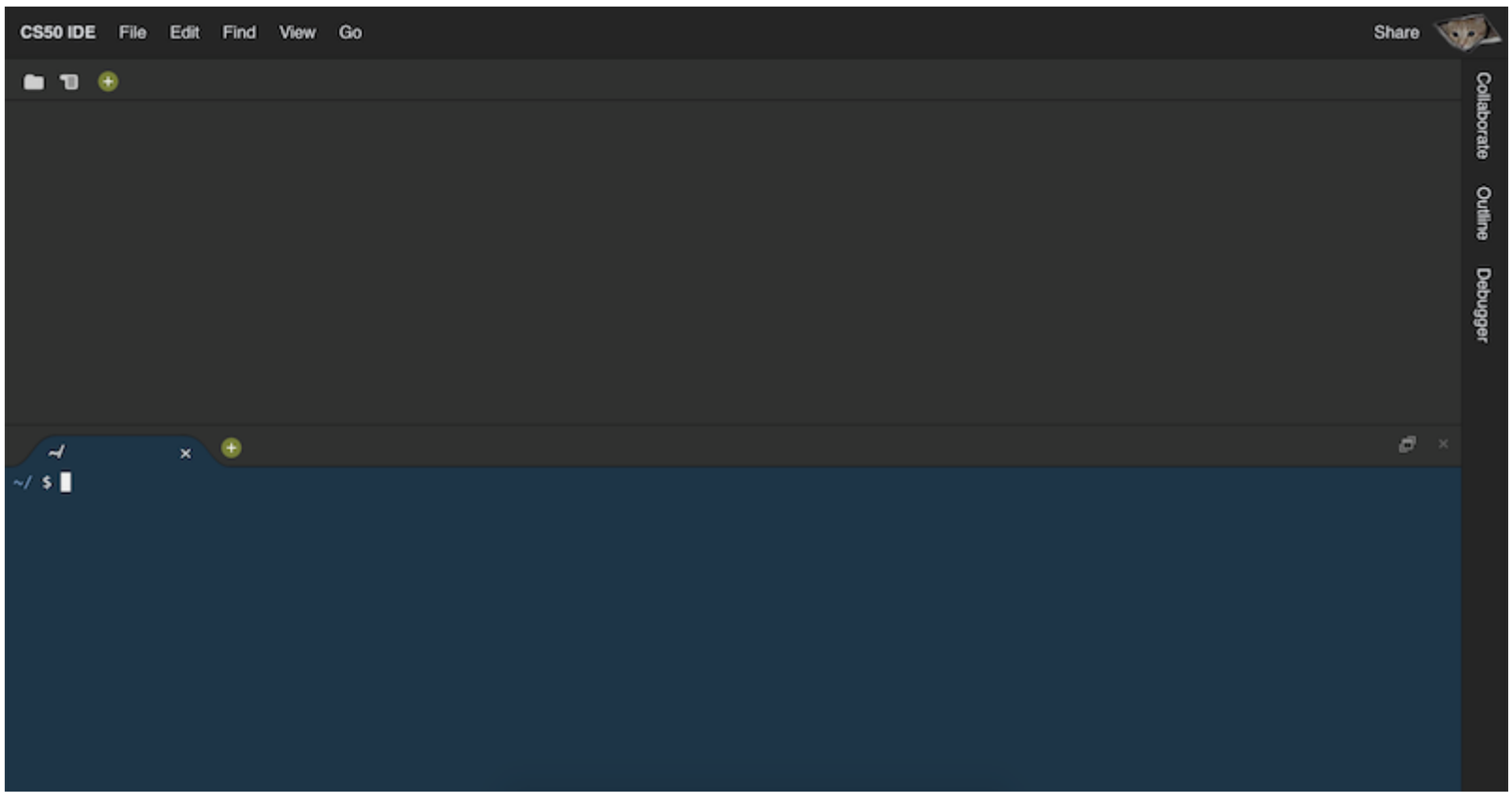


- Al escribir código, podríamos considerar las siguientes cualidades:
  - **Corrección** , o si nuestro código funciona correctamente, según lo previsto.
  - **Diseño** , o una medida subjetiva de cuán bien escrito está nuestro código, basado en cuán eficiente es y cuán elegante o lógicamente legible es, sin repeticiones innecesarias.
  - **Estilo** , o el formato estético de nuestro código, en términos de sangría consistente y otra ubicación de símbolos. Las diferencias de estilo no afectan la corrección o el significado de nuestro código, pero sí afectan su legibilidad visual.

CS50 IDE

- Para comenzar a escribir nuestro código rápidamente, usaremos una herramienta para el curso, **CS50 IDE** (https://ide.cs50.io/) , un *entorno de desarrollo integrado* que incluye programas y funciones para escribir código. CS50 IDE está construido sobre un popular IDE basado en la nube utilizado por programadores generales, pero con características educativas y personalización adicionales.

- Abriremos el IDE y, después de iniciar sesión, veremos una pantalla como esta:



  - El panel superior, en blanco, contendrá archivos de texto dentro de los cuales podemos escribir nuestro código.
  - El panel inferior, una ventana de **terminal** , nos permitirá escribir varios comandos y ejecutarlos, incluidos los programas de nuestro código anterior.
- Nuestro IDE se ejecuta en la nube y viene con un conjunto estándar de herramientas, pero sepa que también hay muchos IDE basados en escritorio, que ofrecen más personalización y control para diferentes propósitos de programación, a costa de un mayor tiempo y esfuerzo de configuración.
- En el IDE, iremos a Archivo> Nuevo archivo, y luego Archivo> Guardar para guardar nuestro archivo como `hello.c`, lo que indica que nuestro archivo será código escrito en C. Veremos que el nombre de nuestra pestaña ha cambiado a `hello.c`, y ahora pegaremos nuestro código de arriba:

```
#include <stdio.h>

int main(void)
{
    printf("hello, world");
}
```

- Para ejecutar nuestro programa, usaremos una CLI, o **una interfaz de línea de comandos** , un indicador en el que necesitamos ingresar comandos de texto. Esto contrasta con una **interfaz gráfica de usuario** , o GUI, como Scratch, donde tenemos imágenes, íconos y botones además de texto.

Compilando

- En la terminal en el panel inferior de nuestro IDE, **compilaremos** nuestro código antes de que podamos ejecutarlo. Las computadoras solo entienden binario, que también se usa para representar instrucciones como imprimir algo en la pantalla. Nuestro **código fuente** ha sido escrito en caracteres que podemos leer, pero necesita ser compilado: convertido a **código de máquina** , patrones de ceros y unos que nuestra computadora pueda entender directamente.
- Un programa llamado **compilador** tomará el código fuente como entrada y producirá código máquina como salida. En CS50 IDE, ya tenemos acceso a un compilador, a través de un comando llamado **make** . En nuestra terminal, escribiremos `make hello`, que

automáticamente encontrará nuestro `hello.c` archivo con nuestro código fuente y lo compilará en un programa llamado `hello`. Habrá algunos resultados, pero no habrá mensajes de error en amarillo o rojo, por lo que nuestro programa se compiló correctamente.

- Para ejecutar nuestro programa, escribiremos otro comando, `./hello` que busca en la carpeta actual `.`, un programa llamado `hello`, y lo ejecuta.
- El `$` en la terminal es un indicador de dónde está el símbolo del sistema o dónde podemos escribir más comandos.

Funciones y argumentos

- Usaremos las mismas ideas que exploramos en Scratch.
- **Las funciones** son pequeñas acciones o verbos que podemos usar en nuestro programa para hacer algo, y las entradas a las funciones se llaman **argumentos** .
    - Por ejemplo, el bloque "decir" en Scratch podría haber tomado algo como "hola, mundo" como argumento. En C, se llama a la función para imprimir algo en la pantalla `printf` (con el `f` significado de texto "formateado", que veremos pronto). Y en C, pasamos argumentos entre paréntesis, como en `printf("hello, world");`. Las comillas dobles indican que queremos imprimir las letras `hello, world` literalmente y el punto y coma al final indica el final de nuestra línea de código.
- Las funciones también pueden tener dos tipos de salidas:
    - **efectos secundarios** , como algo impreso en la pantalla,
    - y **valores de retorno** , un valor que se devuelve a nuestro programa que podemos usar o almacenar para más adelante.
        - El bloque "preguntar" en Scratch, por ejemplo, creó un bloque de "respuesta".
- Para obtener la misma funcionalidad que el bloque "ask", usaremos una **biblioteca** o un conjunto de código ya escrito. La biblioteca CS50 incluirá algunas funciones básicas y simples que podemos usar de inmediato. Por ejemplo, `get_string` le pedirá al usuario una **cadena** , o alguna secuencia de texto, y la devolverá a nuestro programa. `get_string` toma alguna entrada como solicitud para el usuario, como `What's your name?`, y tendremos que guardarla en una variable con:

```
string answer = get_string("What's your name? ");
```

    - En C, el sencillo `=` indica **asignación** , o establecer el valor de la derecha a la variable de la izquierda. Y C llamará a la `get_string` función para obtener su salida primero.
    - Y también necesitamos indicar que nuestra variable nombrada `answer` tiene un **tipo** de cadena, por lo que nuestro programa sabe interpretar los ceros y unos como texto.
    - Finalmente, debemos recordar agregar un punto y coma para finalizar nuestra línea de código.
- En Scratch, también usamos el bloque "respuesta" dentro de nuestros bloques "unirse" y "decir". En C, haremos esto:

```
printf("hello, %s", answer);
```

    - El `%s` se llama **código de formato** , lo que simplemente significa que queremos que la `printf` función sustituya una variable donde está el `%s` marcador de posición. Y la variable que queremos usar es `answer`, que le damos `printf` como otro argumento, separada del primero con una coma. (`printf("hello, answer")` literalmente se imprimirá `hello, answer` cada vez).
- De vuelta en el CS50 IDE, agregaremos lo que hemos descubierto:

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

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

    - Necesitamos decirle al compilador que incluya la biblioteca CS50, con `#include <cs50.h>`, para que podamos usar la `get_string` función.
    - También tenemos la oportunidad de usar un mejor estilo aquí, ya que podríamos nombrar nuestra `answer` variable de cualquier manera, pero un nombre más descriptivo nos ayudará a entender su propósito mejor que un nombre más corto como `a` o `x`.
- Después de guardar el archivo, necesitaremos recompilar nuestro programa con `make hello`, ya que solo hemos cambiado el código fuente pero no el código de máquina compilado. Es posible que otros lenguajes o IDE no requieran que recompilemos manualmente nuestro código después de cambiarlo, pero aquí tenemos la oportunidad de tener más control y comprensión de lo que está sucediendo bajo el capó.
- Ahora, `./hello` ejecutará nuestro programa y nos pedirá nuestro nombre según lo previsto. Podríamos notar que el siguiente mensaje se imprime inmediatamente después de la salida de nuestro programa, como en `hello, Brian~/ $`. Podemos agregar una nueva línea después de la salida de nuestro programa, por lo que el siguiente indicador está en su propia línea, con `\n`:

```
printf("hello, %s\n", answer);
```

- `\n` es un ejemplo de una **secuencia de escape** o algún texto que representa otro texto.

principales, archivos de encabezado

- El bloque "cuando se hace clic en la bandera verde" en Scratch inicia lo que consideraríamos el programa principal. En C, la primera línea para lo mismo es `int main(void)`, sobre la que aprenderemos más en las próximas semanas, seguida de una llave abierta `{` y una llave cerrada `}`, que envuelve todo lo que debería estar en nuestro programa.

```
int main(void)
{

}
```

- Aprenderemos más sobre las formas en que podemos modificar esta línea en las próximas semanas, pero por ahora simplemente usaremos esto para iniciar nuestro programa.
- **Los archivos de encabezado** que terminan en se `.h` refieren a algún otro conjunto de código, como una biblioteca, que luego podemos usar en nuestro programa. Los *incluimos* con líneas como `#include <stdio.h>`, por ejemplo, para la biblioteca *estándar de entrada / salida* , que contiene la `printf` función.

Instrumentos

- Con toda la nueva sintaxis, es fácil para nosotros cometer errores u olvidar algo. Tenemos algunas herramientas escritas por el personal para ayudarnos.
- Es posible que olvidemos incluir una línea de código y, cuando intentemos compilar nuestro programa, veamos muchas líneas de mensajes de error que son difíciles de entender, ya que el compilador podría haber sido diseñado para una audiencia más técnica. **`help50`** es un comando que podemos ejecutar para explicar problemas en nuestro código de una manera más fácil de usar. Podemos ejecutarlo agregando `help50` al frente de un comando que estamos tratando, como `help50 make hello`, para obtener consejos que puedan ser más comprensibles.
- Resulta que, en C, las nuevas líneas y la sangría generalmente no afectan la forma en que se ejecuta nuestro código. Por ejemplo, podemos cambiar nuestra `main` función para que sea de una línea, `int main(void){printf("hello, world");}` pero es mucho más difícil de leer, por lo que consideraríamos que tiene mal estilo. Podemos ejecutar **`style50`**, como con `style50 hello.c`, con el nombre del archivo de nuestro código fuente, para ver sugerencias de nuevas líneas y sangría.
- Además, podemos agregar **comentarios** , notas en nuestro código fuente para nosotros u otros humanos que no afecten cómo se ejecuta nuestro código. Por ejemplo, podríamos agregar una línea como `// Greet user`, con dos barras `//` para indicar que la línea es un comentario, y luego escribir el propósito de nuestro código o programa para ayudarnos a recordar más adelante.
- **`check50`** comprobará la exactitud de nuestro código con algunas pruebas automatizadas. El personal escribe pruebas específicamente para algunos de los programas que escribiremos en el curso, y las instrucciones de uso `check50` se incluirán en cada conjunto de problemas o laboratorio según sea necesario. Después de ejecutar `check50`, veremos algunos resultados que nos dicen si nuestro código pasó las pruebas relevantes.
- El CS50 IDE también nos brinda el equivalente a nuestra propia computadora en la nube, en algún lugar de Internet, con nuestros propios archivos y carpetas. Si hacemos clic en el ícono de la carpeta en la parte superior izquierda, veremos un árbol de archivos, una GUI de los archivos en nuestro IDE:

  - Para abrir un archivo, podemos simplemente hacer doble clic en él. `hello.c` es el código fuente que acabamos de escribir, y `hello` tendrá muchos puntos rojos, cada uno de los cuales son caracteres no imprimibles ya que representan instrucciones binarias para nuestras computadoras.

Comandos

- Dado que el CS50 IDE es una computadora virtual en la nube, también podemos ejecutar comandos disponibles en Linux, un sistema operativo como macOS o Windows.
- En la terminal, podemos escribir `ls`, abreviatura de lista, para ver una lista de archivos y carpetas en la carpeta actual:

```
~/ $ ls
hello*  hello.c
```

  - `hello` está en verde con un asterisco para indicar que podemos ejecutarlo como un programa.
- También podemos *eliminar* archivos con `rm`, con un comando como `rm hello`. Nos pedirá una confirmación, y podemos responder con `y` o `n` para sí o no.
- Con `mv`, o *mover*, podemos cambiar el nombre de los archivos. Con `mv hello.c goodbye.c`, hemos cambiado el nombre de nuestro `hello.c` archivo para que se llame `goodbye.c`.

- Con `mkdir`, o *hacer directorio*, podemos crear carpetas o directorios. Si ejecutamos `mkdir lecture`, veremos una carpeta llamada `lecture` y podemos mover archivos a directorios con un comando como `mv hello.c lecture/`.
- Para *cambiar de directorio* en nuestra terminal, podemos usar `cd`, como con `cd lecture/`. Nuestro mensaje cambiará de `~/` a `~/lecture/`, lo que indica que estamos en el `lecture` directorio interno `~`. `~` representa nuestro directorio de inicio, o la carpeta de nivel superior predeterminada de nuestra cuenta.
- También podemos usarlo `..` como abreviatura del padre o carpeta contenedora. Dentro `~/lecture/`, podemos ejecutar `mv hello.c ..` para moverlo nuevamente `~`, ya que es la carpeta principal de `lecture/`. `cd ..`, de manera similar, cambiará el directorio de nuestra terminal al padre actual. Un solo punto`` se `.` refiere al directorio actual, como en `./hello`.
- Ahora que nuestra `lecture/` carpeta está vacía, también podemos eliminarla con `rmdir lecture/`.

Tipos, códigos de formato,

- Hay muchos **tipos de** datos que podemos usar para nuestras variables, que le indican a la computadora qué tipo de datos representan:
  - `bool`, una expresión booleana de `true` o `false`
  - `char`, un solo carácter ASCII como `a` o `2`
  - `double`, un valor de punto flotante con más dígitos que un `float`
  - `float`, un valor de coma flotante o un número real con un valor decimal
  - `int`, números enteros hasta cierto tamaño o número de bits
  - `long`, enteros con más bits, por lo que pueden contar más que un `int`
  - `string`, una cadena de caracteres
- Y la biblioteca CS50 tiene funciones correspondientes para obtener entradas de varios tipos:
  - `get_char`
  - `get_double`
  - `get_float`
  - `get_int`
  - `get_long`
  - `get_string`
- Porque `printf`, también, hay diferentes marcadores de posición para cada tipo:
  - `%c` para caracteres
  - `%f` para flotadores, dobles
  - `%i` para ints
  - `%li` por mucho tiempo
  - `%s` para cuerdas

Operadores, limitaciones, truncamiento

- También hay varios operadores matemáticos que podemos usar:
  - `+` para la adición
  - `-` para restar
  - `*` para multiplicar
  - `/` para la división
  - `%` por el resto

- Haremos un nuevo programa `addition.c`:

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

int main(void)
{
    int x = get_int("x: ");

    int y = get_int("y: ");

    printf("%i\n", x + y);
}
```

  - Incluiremos archivos de encabezado para las bibliotecas que sabemos que queremos usar, y luego llamaremos `get_int` para obtener enteros del usuario, almacenándolos en variables llamadas `x` y `y`.
  - Luego, `printf` imprimiremos un marcador de posición para un número entero `%i`, seguido de una nueva línea. Como queremos imprimir la suma de `x` y `y`, pasaremos `x + y` por `printf` para sustituir en la cadena.
  - Guardaremos, ejecutaremos `make addition` en la terminal y luego `./addition` veremos que nuestro programa funciona. Si escribimos algo que no es un número entero, veremos `get_int` que nos pide un número entero nuevamente. Si escribimos un número realmente grande, como `4000000000`, también `get_int` se nos volverá a indicar. Esto se debe a que, como en muchos sistemas informáticos, un `int` IDE en CS50 es de 32 bits, que solo puede contener alrededor de cuatro mil millones de valores diferentes. Y dado que los números enteros pueden ser positivos o negativos, el valor positivo más alto para an `int` solo puede ser alrededor de dos mil millones, con un valor negativo más bajo de alrededor de dos mil millones negativos, para un total de alrededor de cuatro mil millones de valores totales.
- Podemos cambiar nuestro programa para usar el `long` tipo:

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

int main(void)
{
    long x = get_long("x: ");

    long y = get_long("y: ");

    printf("%li\n", x + y);
}
```

  - Ahora podemos escribir números enteros más grandes y ver un resultado correcto como se esperaba.
- Siempre que obtengamos un error durante la compilación, es una buena idea desplazarse hacia arriba para ver el primer error y corregirlo primero, ya que a veces un error al principio del programa hará que el resto del programa también se interprete con errores. .
- Veamos otro ejemplo `truncation.c`:

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

int main(void)
{
    // Get numbers from user
    int x = get_int("x: ");
    int y = get_int("y: ");

    // Divide x by y
    float z = x / y;
    printf("%f\n", z);
}
```

  - Almacenaremos el resultado de `x` dividir entre `y` in `z`, un valor de punto flotante o un número real, y también lo imprimiremos como un flotante.
  - Pero cuando compilamos y ejecutamos nuestro programa, vemos `z` impresos como números enteros como `0.000000` o `1.000000`. Resulta que, en nuestro código, `x / y` se divide *primero* como dos enteros , por lo que el resultado devuelto por la operación de división también es un entero. El resultado se **trunca** y se pierde el valor después del punto decimal. Aunque `z` es un `float`, el valor que almacenamos en él ya es un número entero.
  - Para solucionar este problema, nos **echamos** , o convertir, nuestros números enteros a los flotadores antes de los dividimos:

```
float z = (float) x / (float) y;
```

- El resultado será un flotador como esperamos, y de hecho solo podemos lanzar uno de `x` o `y` y obtener un flotador también.

Variables, azúcar sintáctico

- En Scratch, teníamos bloques como "establecer [contador] en (0)" que establecen una **variable** en algún valor. En C, escribiríamos `int counter = 0;` para el mismo efecto.
- Podemos aumentar el valor de una variable con `counter = counter + 1;`, donde primero miramos el lado derecho, tomando el valor original de `counter`, agregando 1, y luego almacenándolo en el lado izquierdo (de nuevo `counter` en este caso).
- C también admite **azúcar sintáctico** o expresiones taquigráficas para la misma funcionalidad. En este caso, podríamos decir de manera equivalente `counter += 1;` agregar uno `counter` antes de almacenarlo nuevamente. También podríamos simplemente escribir `counter++;`, y podemos aprender esto (y otros ejemplos) mirando documentación u otras referencias en línea.

Condiciones

- Podemos traducir condiciones, o bloques "if", con:

```
if (x < y)
{
    printf("x is less than y\n");
}
```

  - Observe que en C, usamos `{` y `}` (así como sangría) para indicar cómo se deben anidar las líneas de código.

- Podemos tener condiciones "si" y "si no":

```
if (x < y)
{
    printf("x is less than y\n");
}
else
{
    printf("x is not less than y\n");
}
```

- E incluso "si no":

```
if (x < y)
{
    printf("x is less than y\n");
}
else if (x > y)
{
    printf("x is greater than y\n");
}
else if (x == y)
{
    printf("x is equal to y\n");
}
```

  - Observe que, para comparar dos valores en C, usamos `==` dos signos iguales.
  - Y, lógicamente, no necesitamos la `if (x == y)` condición final, ya que ese es el único caso restante, por lo que podemos decir `else`:

```
if (x < y)
{
    printf("x is less than y\n");
}
else if (x > y)
{
    printf("x is greater than y\n");
}
else
{
    printf("x is equal to y\n");
}
```

```
}
```

- Echemos un vistazo a otro ejemplo `conditions.c`:

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

int main(void)
{
    // Prompt user for x
    int x = get_int("x: ");

    // Prompt user for y
    int y = get_int("y: ");

    // Compare x and y
    if (x < y)
    {
        printf("x is less than y\n");
    }
    else if (x > y)
    {
        printf("x is greater than y\n");
    }
    else
    {
        printf("x is equal to y\n");
    }
}
```

  - Hemos incluido las condiciones que acabamos de ver, junto con dos llamadas o usos de `get_int` para obtener `x` y `y` del usuario.
  - Compilaremos y ejecutaremos nuestro programa para ver que realmente funciona según lo previsto.
- En `agree.c`, podemos pedirle al usuario que confirme o niegue algo:

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

int main(void)
{
    char c = get_char("Do you agree? ");

    // Check whether agreed
    if (c == 'Y' || c == 'y')
    {
        printf("Agreed.\n");
    }
    else if (c == 'N' || c == 'n')
    {
        printf("Not agreed.\n");
    }
}
```

  - Con `get_char`, podemos obtener un solo carácter, y dado que solo tenemos uno en nuestro programa, parece razonable llamarlo `c`.
  - Usamos dos barras verticales `||`, para indicar un "o" lógico, si cualquiera de las expresiones puede ser verdadera para la condición a seguir. (Dos ampersands,, `&&` indican un "y" lógico, donde ambas condiciones tendrían que ser verdaderas.) Y observe que usamos dos signos iguales `==`, para comparar dos valores, así como comillas simples `'`, para rodear nuestros valores de single caracteres.

- Si ninguna de las expresiones es verdadera, no pasará nada ya que nuestro programa no tiene un ciclo.

Expresiones booleanas, bucles

- Podemos traducir un bloque "para siempre" en Scratch con:

```
while (true)
{
    printf("hello, world\n");
}
```

  - La `while` palabra clave requiere una condición, por lo que la usamos `true` como expresión booleana para asegurarnos de que nuestro bucle se ejecute para siempre. `while` le dirá a la computadora que verifique si la expresión se evalúa `true` y luego

    ejecutará las líneas dentro de las llaves. Luego lo repetirá hasta que la expresión ya no sea verdadera. En este caso, `true` siempre será verdadero, por lo que nuestro ciclo es un **ciclo infinito** , o uno que se ejecutará para siempre.
- Podríamos hacer algo un cierto número de veces con `while` :

```
int i = 0;
while (i < 50)
{
    printf("hello, world\n");
    i++;
}
```

  - Creamos una variable, `i` y la establecemos en 0. Luego, mientras `i` es menor que 50, ejecutamos algunas líneas de código, incluida una en la que agregamos 1 `i` cada vez. De esta manera, nuestro ciclo terminará eventualmente cuando `i` alcance un valor de 50.
  - En este caso, estamos usando la variable `i` como un contador, pero como no tiene ningún propósito adicional, podemos simplemente nombrarla `i` .
- Aunque *podríamos* hacer lo siguiente y comenzar a contar desde 1, por convención deberíamos comenzar en 0:

```
int i = 1;
while (i <= 50)
{
    printf("hello, world\n");
    i++;
}
```

- Otra solución correcta, pero posiblemente menos bien diseñada, podría comenzar en 50 y contar hacia atrás:

```
int i = 50;
while (i > 0)
{
    printf("hello, world\n");
    i--;
}
```

  - En este caso, la lógica de nuestro ciclo es más difícil de razonar sin tener ningún propósito adicional, e incluso podría confundir a los lectores.
- Finalmente, más comúnmente podemos usar la `for` palabra clave:

```
for (int i = 0; i < 50; i++)
{
    printf("hello, world\n");
}
```

  - Nuevamente, primero creamos una variable nombrada `i` y la establecemos en 0. Luego, verificamos que `i < 50` cada vez que lleguemos a la parte superior del ciclo, antes de ejecutar cualquiera de los códigos internos. Si esa expresión es verdadera, ejecutamos el código interno. Finalmente, después de ejecutar el código interno, usamos `i++` para agregar uno `i` y el ciclo se repite.
  - El `for` ciclo es más elegante que un `while` ciclo en este caso, ya que todo lo relacionado con el ciclo está en la misma línea, y solo el código que realmente queremos ejecutar varias veces está dentro del ciclo.
- Observe que para muchas de estas líneas de código, como `if` condiciones y `for` bucles, no ponemos un punto y coma al final. Así es como se diseñó el lenguaje de C, hace muchos años, y una regla general es que solo las líneas para acciones o verbos tienen punto y

coma al final.

Abstracción

- Podemos escribir un programa que imprima `meow` tres veces:

```
#include <stdio.h>

int main(void)
{
    printf("meow\n");
    printf("meow\n");
    printf("meow\n");
}
```

- Podríamos usar un `for` bucle, por lo que no tenemos que copiar y pegar tantas líneas:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        printf("meow\n");
    }
}
```

- Podemos mover la `printf` línea a su propia función, como nuestra propia pieza de rompecabezas:

```
#include <stdio.h>

void meow(void) {
  printf("meow\n");
}

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        meow();
    }
}
```

  - Definimos una función `meow`, por encima de nuestra `main` función.
- Pero convencionalmente, nuestra `main` función debería ser la primera función en nuestro programa, por lo que necesitamos algunas líneas más:

```
#include <stdio.h>

void meow(void);

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        meow();
    }
}

void meow(void)
{
    printf("meow\n");
}
```

  - Resulta que `meow` primero debemos declarar nuestra función con un **prototipo** , antes de usarlo `main`, y realmente definirlo después. El compilador lee nuestro código fuente de arriba a abajo, por lo que necesita saber que `meow` existirá más adelante en el archivo.
- Incluso podemos cambiar nuestra `meow` función para tomar algunas entradas `n`, y `n` tiempos de maullido :

```c
#include <stdio.h>

void meow(int n);

int main(void)
{
    meow(3);
}

void meow(int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("meow\n");
    }
}
```

- El `void` antes de la `meow` función significa que no devuelve un valor, y del mismo modo, `main` no podemos hacer nada con el resultado de `meow`, por lo que simplemente lo llamamos.

- La abstracción aquí conduce a un mejor diseño, ya que ahora tenemos la flexibilidad de reutilizar nuestra `meow` función en múltiples lugares en el futuro.
- Veamos otro ejemplo de abstracción `get_positive_int.c`:

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

int get_positive_int(void);

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

// Prompt user for positive integer
int get_positive_int(void)
{
    int n;
    do
    {
        n = get_int("Positive Integer: ");
    }
    while (n < 1);
    return n;
}
```

- Tenemos nuestra propia función que llama `get_int` repetidamente hasta que tenemos un número entero que *no es* menor que 1. Con un ciclo do-while, nuestro programa hará algo primero, luego verificará alguna condición y repetirá mientras la condición sea verdadera. Un ciclo while, por otro lado, verificará la condición primero.
- Necesitamos declarar nuestro entero `n` fuera del ciclo do-while, ya que necesitamos usarlo después de que finalice el ciclo. El **alcance** de una variable en C se refiere al contexto, o líneas de código, dentro del cual existe. En muchos casos, serán las llaves que rodean la variable.

- Observe que la función `get_positive_int` ahora comienza con `int`, lo que indica que tiene un valor de retorno de tipo `int` y `main`, de hecho, lo almacenamos `i` después de llamar `get_positive_int()`. En `get_positive_int`, tenemos una nueva palabra clave, `return` para devolver el valor `n` al lugar donde se llamó a la función.

Mario

- Podríamos querer un programa que imprima parte de una pantalla de un videojuego como Super Mario Bros. En `mario.c`, podemos imprimir cuatro signos de interrogación, simulando bloques:

```
#include <stdio.h>

int main(void)
{
    printf("????\n");
}
```

- Con un bucle, podemos imprimir una serie de signos de interrogación, siguiéndolos con una sola línea nueva después del bucle:

```
#include <stdio.h>

int main(void)
{
    for (int i = 0; i < 4; i++)
    {
        printf("?");
    }
    printf("\n");
}
```

- Podemos obtener un número entero positivo del usuario e imprimir ese número de signos de interrogación usando `n` para nuestro ciclo:

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

int main(void)
{
    // Get positive integer from user
    int n;
    do
    {
        n = get_int("Width: ");
    }
    while (n < 1);

    // Print out that many question marks
    for (int i = 0; i < n; i++)
    {
        printf("?");
    }
    printf("\n");
}
```

- Y podemos imprimir un conjunto bidimensional de bloques con bucles anidados, uno dentro del otro:

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

int main(void)
{
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
```

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

- Tenemos dos bucles anidados, donde el bucle externo usa `i` para hacer todo dentro 3 veces, y el bucle interno usa `j`, una variable diferente, para hacer algo 3 veces para cada una de *esas* veces. En otras palabras, el bucle externo imprime 3 "filas" o líneas, terminando cada una de ellas con una nueva línea, y el bucle interno imprime 3 "columnas" o `#` caracteres, *sin* una nueva línea.

Memoria, imprecisión y desborde

- Nuestra computadora tiene memoria, en chips de hardware llamados RAM, memoria de acceso aleatorio. Nuestros programas usan esa RAM para almacenar datos mientras se ejecutan, pero esa memoria es finita.
- Con `imprecision.c`, podemos ver qué pasa cuando usamos flotadores:

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

int main(void)
{
    float x = get_float("x: ");
    float y = get_float("y: ");

    printf("%.50f\n", x / y);
}
```

  - Con `%.50f`, podemos especificar el número de posiciones decimales que se muestran.
  - Hmm, ahora tenemos ...

```
x: 1
y: 10
0.10000000149011611938476562500000000000000000000000
```

  - Resulta que esto se llama **imprecisión de punto flotante** , donde no tenemos suficientes bits para almacenar todos los valores posibles. Con un número finito de bits para a `float`, no podemos representar todos los números reales posibles (de los cuales hay un número *infinito*), por lo que la computadora tiene que almacenar el valor más cercano que pueda. Y esto puede llevar a problemas en los que incluso las pequeñas diferencias en el valor se suman, a menos que el programador utilice alguna otra forma de representar los valores decimales con la precisión necesaria.
- La semana pasada, cuando tuvimos tres bits y es necesario contar más allá de siete (o `111`), se añadió otro poco para conseguir ocho, `1000`. Pero si solo tuviéramos tres bits disponibles, no tendríamos lugar para el extra `1`. Desaparecería y estaríamos de vuelta en `000`. Este problema se llama **desbordamiento de enteros** , donde un entero solo puede ser tan grande antes de quedarse sin bits.
- El problema Y2K surgió porque muchos programas almacenaban el año calendario con solo dos dígitos, como 98 para 1998 y 99 para 1999. Pero cuando se acercó el año 2000, los programas tenían que almacenar solo 00, lo que generó confusión entre los años 1900 y 2000. .
- En 2038, también nos quedaremos sin bits para rastrear el tiempo, ya que hace muchos años algunos humanos decidieron usar 32 bits como el número estándar de bits para contar el número de segundos desde el 1 de enero de 1970. Pero con 32 bits que representan solo números positivos, solo podemos contar hasta unos cuatro mil millones, y en 2038 alcanzaremos ese límite a menos que actualicemos el software en todos nuestros sistemas informáticos.