# Notas de curso
# Programación Estructurada

Rodrigo Alexander Castro Campos
UAM Azcapotzalco, División de CBI
`https://racc.mx`
`https://omegaup.com/profile/rcc`

Última revisión: 22 de noviembre de 2022

## 1. Funciones y tipos de datos

El concepto que conecta a las matemáticas con la computación es el de *función*. Una función es una regla que asocia valores de entrada con un valor de salida. A continuación se muestran ejemplos de funciones usando la notación que hemos usado desde secundaria y en bachillerato:

- $f(x) = x + 5$

  Si $x = 2$ entonces $f(2) = 2 + 5 = 7$

- $g(x, y) = 2x + y$

  Si $x = 1, y = 3$ entonces $g(1, 3) = 2(1) + 3 = 5$.

- $h(x, y) = f(x) + g(y, x)$

  Si $x = 1, y = 3$ entonces $h(1, 3) = f(1) + g(3, 1) = 6 + 7 = 13$

- $\tan(x) = \frac{\text{sen}(x)}{\cos(x)}$

  Si $x = 30°$ entonces $\tan(30°) = \frac{\text{sen}(30°)}{\cos(30°)} = \frac{0.5}{0.5\sqrt{3}} = \frac{1}{\sqrt{3}}$

Los últimos dos ejemplos ilustran funciones definidas en términos de otras funciones. Es justo esta habilidad lo que nos permite especificar cálculos cada vez más complicados. La mayoría de los lenguajes de programación (y en particular, el lenguaje C que es el que veremos en el curso) trabajan definiendo funciones que cooperan unas con otras para realizar el cálculo que necesitamos.

Lo primero que haremos es introducir una notación más elaborada para definir funciones. En la nueva notación, escribiremos la función $f(x) = x + 2$ como

$$f(x) \ \left\{ \ \text{regresa } x + 2 \ \right\}$$

donde la palabra `regresa` indica que el resultado de la función es la expresión a su derecha, en este caso $x + 2$. Esta notación nos permitirá simplificar la escritura de funciones problemáticas o tediosas. Por ejemplo, la función

$$f(x, y, z) = (x + 2y + z) + (x + 2y + z)^2 + (x + 2y + z)^3 + (x + 2y + z)^4$$

la podremos escribir como

$$f(x, y, z) \ \left\{ \begin{array}{l} t = x + 2y + z \\ \text{regresa } t + t^2 + t^3 + t^4 \end{array} \right\}$$

donde primero se define $t$ como $x + 2y + z$ y después se calcula el resultado usando $t$. En este contexto, la palabra `regresa` en efecto indica que ya tenemos todos los resultados intermedios que necesitamos para

poder calcular el resultado final de la función. El poder especificar más de un paso dentro de la definición de una función permite expresar una nueva gama de cálculos matemáticos, como el valor absoluto de $x$:

$$|x| \left\{ \begin{array}{l} \text{si } x \geq 0 \text{ entonces} \\ \quad \text{regresa } x \\ \text{sino entonces} \\ \quad \text{regresa } -x \end{array} \right\}$$

Desafortunadamente volveremos a complicar aún más la notación de función, pero también por buenas razones. En la primaria, los primeros números que nos enseñaron fueron los naturales (0, 1, 2, etc.) y después nos enseñaron los enteros, que incluyen valores negativos. Después nos enseñaron los números racionales o fraccionarios y su representación con punto decimal (por ejemplo, $\frac{1}{2} = 0.5$) y por último, los números reales (por ejemplo, $\sqrt{2}$ o $\pi$). Salvo en contadas situaciones en bachillerato o universidad donde hemos tenido que manejar números complejos, prácticamente todos los cálculos que hemos realizado desde entonces han sido con números reales. Uno pensaría que no hay necesidad de volver a los naturales o a los enteros si ya sabemos manejar reales, pero estamos equivocados.

Sea $x$ algún un número real, ¿cuál es el número que le sigue a $x$? Más formalmente, ¿cuál es el *sucesor* de $x$? Por ejemplo, si $x = 3.5$ su sucesor no puede ser 4.5 porque 3.6 es mayor que 3.5 y también es más cercano, pero 3.51 es aún más cercano y 3.501 es aún más cercano. Para los números reales, el concepto del *sucesor* de un número no está definido, ¡pero para los enteros sí! Si nos concentramos únicamente en los enteros, entonces el sucesor de 3 es 4 y el sucesor de $-1$ es 0, sin que exista duda o discusión alguna. Eso quiere decir que la siguiente definición de función está mal especificada:

$$sucesor(x) \ \{ \ \text{regresa } x + 1 \ \}$$

porque es correcta para enteros, pero no para reales. Ampliaremos la notación de función para especificar para qué tipo de número sirve nuestra función (usar la función con otro tipo de número sería un error):

$$sucesor(\text{entero } x) \ \{ \ \text{regresa } x + 1 \ \}$$

Con la nueva notación de función, sucesor(5) es 6 mientras que sucesor(5.2) es un error porque 5.2 no es un entero. Ésta no es la primera vez que encontramos errores al querer usar una función en la vida real. Por ejemplo, calcular $\sqrt{-1}$ en una calculadora científica común también es un error. Cuando usemos una variable para almacenar un resultado temporal, también especificaremo su tipo:

$$s(\text{entero } x) \left\{ \begin{array}{l} \text{entero } t = 2s + sucesor(s) \\ \text{regresa } t^2 + sucesor(t) \end{array} \right\}$$

Volveremos a complicar aún más la notación de funcion, esta vez simplemente para ser más explícitos. Así como tiene sentido especificar funciones que están definidas para enteros pero no para reales, también existen funciones que toman números de un tipo y devuelven números de otro tipo. Por ejemplo:

$$r(\text{entero } x) \ \{ \ \text{regresa } sucesor(x) + \sqrt{x} \ \}$$

La función $r$ está definida únicamente para números enteros, pero el resultado de la función puede ser un real. Si evaluamos $r(2)$, ésta es igual a $sucesor(2) + \sqrt{2}$ que es aproximadamente 4.4142. Usaremos el símbolo $\mapsto$ después de los parámetros para indicar que el resultado de la función puede ser un real.

$$r(\text{entero } x) \mapsto \text{real } \{ \ \text{regresa } sucesor(x) + \sqrt{x} \ \}$$

Esta notación y este nivel de detalle son muy cercanos a lo que tendremos que especificar cuando programemos en lenguaje C. En la vida diaria, las matemáticas suelen usar la notación más simple posible que la gente pueda entender cuando se escribe en un pizarrón o en la hoja de un cuaderno, pero en computación quien realizará el cálculo es una máquina.

## 2. Algoritmos como funciones

Un algoritmo es una lista de pasos para realizar algo. La importancia de esta definición radica en que pueden existir muchas formas distintas de realizar el mismo cálculo. Anteriormente se definió una función

que calcula el valor absoluto de un número $x$ revisando si es o no negativo, para así regresar $-x$ o $x$ según sea el caso. Una forma distinta de calcular el valor absoluto de $x$ es:

$$|x| \ \left\{ \ \text{regresa } {}^{+}\sqrt{x^2} \ \right\}$$

En este curso, uno de los objetivos más importantes será encontrar algoritmos que calculen correctamente lo que nos soliciten. Excepto en el caso de cálculos extremandamente sencillos, generalmente siempre existe más de un algoritmo. Sólo en cursos posteriores interesa encontrar los mejores algoritmos.

Existen algunas funciones que son tan intuitivas, que de hecho cuesta trabajo o es imposible definirlas matemáticamente. Por ejemplo, ¿cómo definiríamos una función *trunca* que tome un real $x$ y que devuelva su parte entera? Por ejemplo, queremos que *trunca*(5.0) sea 5, que *trunca*(8.3) sea 8 y que *trunca*(11.9) sea 11. En este caso, conviene suponer que la función ya existe, aunque no la hayamos definido nosotros.

¿Cómo podríamos definir (ahora sí, nosotros) una función *redondea* que tome un real $x$ y lo redondee al entero más cercano? Por ejemplo, queremos que *redondea*(5.3) sea 5, que *redondea*(8.5) sea 9 y que *redondea*(11.9) sea 12. Podemos intentar definir esta función de la siguiente forma:

$$\textit{redondea}(\text{real } x) \mapsto \text{entero} \left\{ \begin{array}{l} \text{si } x - \textit{trunca}(x) \geq 0.5 \text{ entonces} \\ \qquad \text{regresa } \textit{trunca}(x) + 1 \\ \text{sino entonces} \\ \qquad \text{regresa } \textit{trunca}(x) \end{array} \right\}$$

La expresión $x - \textit{trunca}(x)$ sirve para obtener la parte fraccionaria de $x$. Por ejemplo, $8.7 - \textit{trunca}(8.7) = 0.7$. Aunque para la mayoría de los usos comunes esta función parezca ser correcta, ¿cuál debería ser el resultado correcto de *redondea*$(-8.7)$? La intuición nos dice que debe ser $-9$, pero desafortunadamente eso no es lo que regresa la función de arriba. Cuando calculamos la parte fraccionaria de $-8.7$ obtenemos $-0.7$ que no es mayor o igual a 0.5, por lo que la función regresará *trunca*$(x)$ que es $-8$. Es muy común diseñar una función que es correcta para algunos casos, sólo para darnos cuenta demasiado tarde que es incorrecta para otros. En este caso el problema son los números negativos. La función se puede corregir a expensas de complicarla, examinando tanto el valor absoluto de la parte fraccionaria de $x$ como el signo de $x$. Obsérvese que para redondear $-8.7$ deberemos regresar $-8 - 1 = -9$. A continuación se muestra la función resultante. ¿Se te ocurre una forma más sencilla de expresar el mismo cálculo?

$$\textit{redondea}(\text{real } x) \mapsto \text{entero} \left\{ \begin{array}{l} \text{real } t = x - \textit{trunca}(x) \\ \text{si } |t| \geq 0.5 \text{ y } x \geq 0 \text{ entonces} \\ \qquad \text{regresa } \textit{trunca}(x) + 1 \\ \text{sino si } |t| \geq 0.5 \text{ y } x < 0 \text{ entonces} \\ \qquad \text{regresa } \textit{trunca}(x) - 1 \\ \text{sino entonces} \\ \qquad \text{regresa } \textit{trunca}(x) \end{array} \right\}$$

Otro cálculo ilustrativo que trabajaremos es el siguiente: supongamos que tu profesor de química hará cuatro exámenes y que la calificación final del alumno será el promedio de sus tres mejores calificaciones, descartando la peor (en caso de varias peores, sólo se descartará una). Un alumno obtuvo calificaciones $a = 8.5, b = 6, c = 10, d = 6$. ¿Cómo podemos escribir una función que calcule su calificación final? El problema sería muy fácil si se promediaran las cuatro, ya que simplemente podríamos regresar $\frac{a+b+c+d}{4} = \frac{8.5+6+10+6}{4} = 7.625$. Sin embargo, el problema explícitamente dice que debemos ignorar la calificación más baja y sólo promediar las tres mejores. Hay varias formas de resolver el problema, una algo complicada:

$$\textit{promedio}_3(\text{real } a, \text{real } b, \text{real } c, \text{real } d) \mapsto \text{real} \left\{ \begin{array}{l} \text{si } a \leq b \text{ y } a \leq c \text{ y } a \leq d \text{ entonces} \\ \qquad \text{regresa } \frac{b+c+d}{3} \\ \text{sino si } b \leq a \text{ y } b \leq c \text{ y } b \leq d \text{ entonces} \\ \qquad \text{regresa } \frac{a+c+d}{3} \\ \text{sino si } c \leq a \text{ y } c \leq b \text{ y } c \leq d \text{ entonces} \\ \qquad \text{regresa } \frac{a+b+d}{3} \\ \text{sino entonces} \\ \qquad \text{regresa } \frac{a+b+c}{3} \end{array} \right\}$$

La función anterior detecta cuál de las cuatro calificaciónes ($a, b, c$ o $d$) es menor o igual que todas las demás y entonces regresa el promedio de las otras tres. Es importante usar $\leq$ y no $<$ para que la función sea correcta aún si hay calificaciones repetidas (por ejemplo en 5, 5, 10, 10 queremos excluir uno de los 5 a pesar de que no es estrictamente menor que el otro 5). Sin embargo, una forma más fácil de resolver el mismo problema es darnos cuenta que podemos sumar libremente $a+b+c+d$, siempre y cuando después restemos la peor calificación. Esto tendrá el efecto de cancelar dicha peor calificación por haberla sumado erróneamente. A continuación es muestra la función propuesta:

$$\textit{promedio}_3(\text{real } a, \text{real } b, \text{real } c, \text{real } d) \mapsto \text{real } \left\{ \begin{array}{l} \text{regresa } \frac{a+b+c+d-\textit{mínimo}_4(a,b,c,d)}{3} \end{array} \right\}$$

En esta nueva formulación, hay que definiir la función *mínimo*$_4$ que toma cuatro reales y regresa el menor de ellos. Para calcular esto, podemos auxiliarnos de una estrategia tipo torneo, por parejas. Primero calculamos al más pequeño de $a$ y $b$, luego calculamos al más pequeño de $c$ y $d$, y finalmente calculamos al más pequeño de entre los dos ganadores. Esto se puede hacer como se muestra a continuación:

$$\textit{mínimo}_4(\text{real } a, \text{real } b, \text{real } c, \text{real } d) \mapsto \text{real } \left\{ \begin{array}{l} \text{real } t_1 = \textit{mínimo}_2(a, b) \\ \text{real } t_2 = \textit{mínimo}_2(c, d) \\ \text{regresa } \textit{mínimo}_2(t_1, t_2) \end{array} \right\}$$

$$\textit{mínimo}_2(\text{real } a, \text{real } b) \mapsto \text{real } \left\{ \begin{array}{l} \text{si } a < b \text{ entonces} \\ \quad \text{regresa } a \\ \text{sino entonces} \\ \quad \text{regresa } b \end{array} \right\}$$

La estrategia para calcular el mínimo de dos reales es fácil. Si $a$ es menor que $b$ entonces el resultado es $a$, en otro caso regresamos $b$. Cuando $a = b$ también regresamos $b$, pero eso está bien ya que ambas variables tienen el mismo valor y entonces da lo mismo cuál regresemos. Aunque la segunda formulación de *promedio*$_3$ nos obligó a definir otras dos funciones, es más fácil darnos cuenta que cada función individual es correcta, por lo que es más fácil darnos cuenta que la formulación completa también es correcta.

## 2.1. Ejercicios

1. Escribe la siguiente función definida sobre reales usando la notación extendida.

$$f(x, y, z) = 1 + (x + yz) + (x + yz)^2 - (x + yz)^3 - (x + yz)^5$$

2. Escribe una función *máximo*$_2$ que regrese el máximo de dos reales. Compárala con la definición de la función *mínimo*$_2$ que se describe previamente en las notas.

3. Escribe una función *signo* que tome un real $x$ y regrese $-1$ si $x$ es negativo, regrese 0 si $x$ es cero y regrese $+1$ si $x$ es positivo.

4. Escribe una función $g$ que tome un real $x$ y que regrese 1 si el valor de $\sqrt{x}$ es entero y regrese 0 caso contrario. Considera que puedes ayudarte de la función *trunca*.

5. Escribe una función *mediana*$_3$ que tome tres reales $x, y, z$ y regrese su mediana. La mediana de una secuencia de números es el número que quedaría enmedio si la ordenamos. Por ejemplo, la mediana de $(8, 1, 5)$ es 5 porque si ordenamos la secuencia queda $(1, 5, 8)$ con el 5 enmedio. Considera que puedes auxiliarte de otras funciones que hayas definido previamente.

6. Dada la siguiente definición de $f$, calcula el valor de $f(1, 2)$:

$$f(\text{real } x, \text{real } y) \mapsto \text{real } \left\{ \begin{array}{l} \text{real } s = x + y^2 \\ \text{real } t = 2x + \sqrt{y} \\ \text{regresa } s + t \end{array} \right\}$$

7. Dadas las siguientes definiciones de $f$ y $g$, calcula el valor de $g(1, 2)$:

$$f(\text{real } x, \text{real } y) \mapsto \text{real } \left\{ \begin{array}{l} \text{real } t = 2x + y \\ \text{regresa } t + t^2 \end{array} \right\}$$

$$g(\text{real } x, \text{real } y) \mapsto \text{real } \left\{ \begin{array}{l} \text{regresa } f(2y, x) \end{array} \right\}$$

8. Dada la siguiente definición de $f$, calcula el valor de $f(3)$:

$$f(\text{entero } n) \mapsto \text{entero} \left\{ \begin{array}{l} \text{si } n = 0 \text{ entonces} \\ \quad \text{regresa } 0 \\ \text{sino entonces} \\ \quad \text{regresa } n + f(n-1) \end{array} \right\}$$

9. Aún no hemos hablado de cómo programar, pero a continuación se muestra la definición de una función $f$ y su implementación en el lenguaje de programación C. Identifica las similitudes y diferencias en la notación empleada. No es necesario que memorices la notación del lenguaje C en este momento, pero sí intenta convencerte que es muy similar a lo que hemos visto.

$$f(\text{real } r, \text{entero } n) \mapsto \text{real} \left\{ \begin{array}{l} \text{si } n \geq 0 \text{ entonces} \\ \quad \text{regresa } r + n \\ \text{sino si } r < 0 \text{ y } n < 0 \text{ entonces} \\ \quad \text{regresa } r - n \\ \text{sino entonces} \\ \quad \text{entero } t = 3n \\ \quad \text{regresa } \frac{r}{t+1} \end{array} \right\}$$

```
float f(float r, int n) {
   if (n >= 0) {
      return r + n;
   } else if (r < 0 && n < 0) {
      return r - n;
   } else {
      int t = 3 * n;
      return r / (t + 1);
   }
}
```

# 3. Compiladores y programas

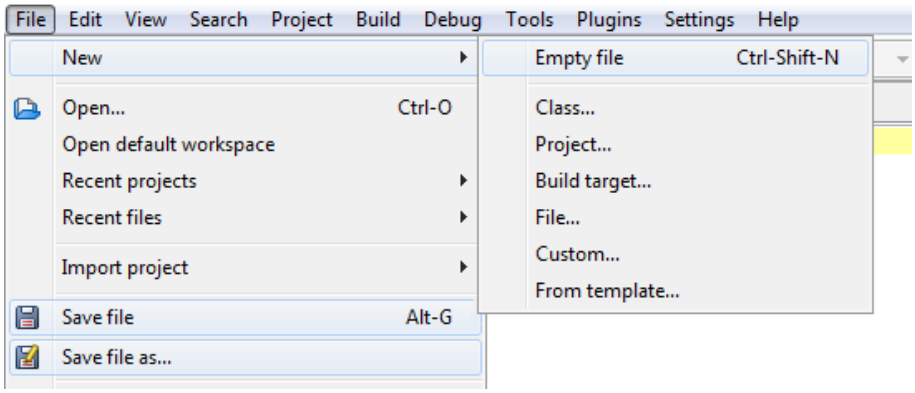
Un lenguaje de programación es aquél que nos permite especificar instrucciones que la computadora debe ejecutar. Cada computadora que sale al mercado tiene un lenguaje nativo que el fabricante diseña y la computadora sólo puede ejecutar instrucciones escritas en su lenguaje nativo. Desafortunadamente, estos lenguajes varían incluso con el modelo del procesador. Que exista tal variabilidad en los lenguajes nativos no es algo bueno para los programadores, ya que tendrían a estar aprendiendo muchos lenguajes y tendrían que traducir sus programas de un lenguaje nativo a otro cada vez que salga una nueva tecnología.

Actualmente los programadores usan lenguajes de programación que no son nativos, pero un programa llamado compilador se encarga de traducirlo al lenguaje nativo (también llamado lenguaje de máquina). A los lenguajes no nativos se les suele llamar lenguajes de alto nivel y uno de ellos es el lenguaje de programación C. Éste es el lenguaje de programación que veremos en el curso.

Los compiladores más famosos de C son GCC y Clang. En Windows lo más común es usar MinGW que es una variante de GCC, mientras que en MacOS se suele usar Clang y en Linux se suele usar GCC. Aprender a usar directamente un compilador es un poco complicado, por lo que la mayoría de las veces conviene usar otro programa llamado "entorno de desarrollo integrado" (IDE por sus siglas en inglés), el cual es un editor de texto que tiene un botón para pedirle al compilador que haga la traducción correspondiente y así poder ejecutar nuestro programa. Un IDE que está disponible para Windows, Linux y MacOS es Code::Blocks. Como Windows no trae preinstalado ningún compilador, para Windows se sugiere descargar el instalador que trae incorporado MinGW. En Linux lo normal es que GCC ya esté preinstalado. En MacOS, la recomendación de Apple es instalar Xcode que trae Clang.

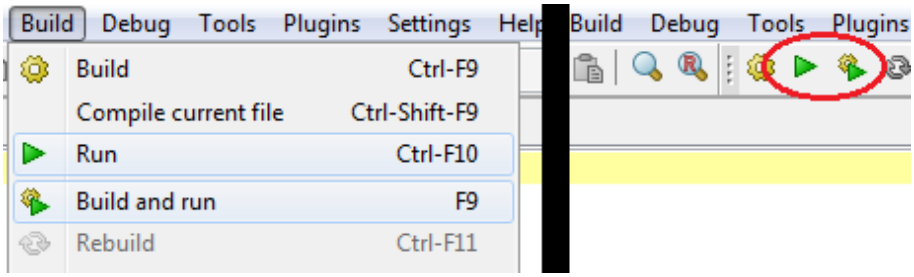
A continuación se describe el uso básico de Code::Blocks. La instalación de este entorno en Windows es sencilla. Una vez abierto el IDE, las opciones más relevantes son las siguientes:

- "`File` → `New` → `Empty file`": Crea una pestaña vacía en el editor de código.
- "`File` → `Save file as`": Permite guardar el código en un archivo. El código escrito en el lenguaje C debe guardarse en archivos con extensión *.c* de forma obligatoria.
- "`File` → `Save file`": Permite guardar las modificaciones hechas a un archivo abierto en el editor.




La ubicación de "`Empty file`", "`Save file`" y "`Save file as`".

- "`Build` → `Build and run`": Compila el archivo actual y ejecuta el programa resultante. Hay un botón con la misma funcionalidad en la barra de herramientas.
- "`Build` → `Run`": Ejecuta el programa ya compilado. Hay un botón con la misma funcionalidad en la barra de herramientas.



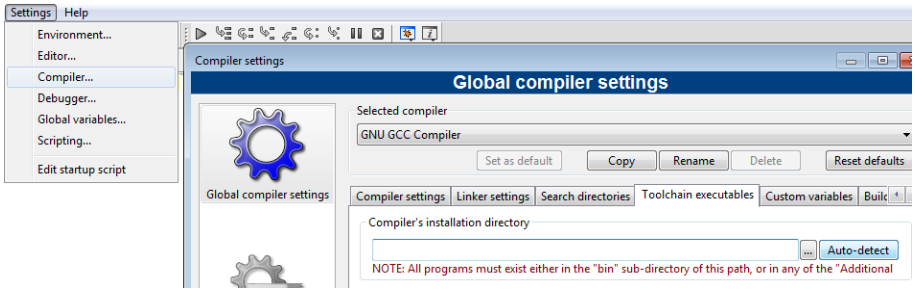

La ubicación de "`Build and run`" y "`Run`".

- "`Settings` → `Compiler` → `Toolchain executables` → `Auto-detect`": Esta opción se usa cuando el IDE no detecta el compilador aunque se haya instalado junto con el IDE. Después de darle clic al botón y de que el IDE haya encontrado el compilador, es importante cerrar la subventana con la opción "`OK`" de abajo, ya que lo contrario no se guardará la ubicación detectada.

Al querer compilar y ejecutar un código que tenga algún un error que impida la traducción del mismo al lenguaje de máquina, ocurre lo que se denomina un error de compilación. El diagnóstico emitido por el compilador aparecerá en la parte inferior del IDE, en las pestañas "`Build messages`" y "`Build log`".

El código fuente de un programa es la lista de instrucciones que el programador escribió. Un archivo fuente es simplemente el archivo donde guardamos el código fuente. Como programaremos en lenguaje C, guardaremos el código fuente de nuestros programas en archivos con extensión *.c* para poder usarlos. Aún no hemos hablado sobre cómo programar en C, pero adelantaremos que dependiendo de la plataforma y del compilador, una compilación exitosa puede generar varios archivos:

1. Archivo objeto: contiene la traducción realizada por el compilador pero que aún no se puede ejecutar. La extensión usual de estos archivos es *.o* en todas las plataformas. Ignoraremos estos archivos.




La ubicación de "`Auto-detect`".

2. Archivo ejecutable: contienen la traducción realizada por el compilador que ya se puede ejecutar como programa. En Windows la extensión usual de los ejecutables es *.exe*, mientras que en Linux y MacOS los archivos ejecutables suelen no tener extensión.

Si un archivo fuente se llama *programa.c* entonces en Windows una compilación exitosa producirá los archivos *programa.o* y *programa.exe* en la misma carpeta donde está el código. Se debe notar que es posible que la compilación no sea exitosa: eso significa que el compilador no entendió nuestro código y no pudo realizar la traducción al lenguaje nativo.

Existen IDE en línea que esconden mucha de la complejidad de compilar y ejecutar programas. En estos entornos no hay archivos que guardar y sólo hay un editor de texto con un botón con el que podemos compilar y ejecutar nuestro programa. Algunos de estos entornos son OnlineGDB, Rep.it y el entorno para la clase disponible en `http://callix.azc.uam.mx/rcc/ide/`.

# 4. Introducción al lenguaje de programación C

El lenguaje de programación C es un lenguaje eficiente de alto nivel diseñado por Dennis Ritchie en 1972. El lenguaje C está estandarizado ante el Instituto Nacional Estadounidense de Estándares (ANSI) y la Organización Internacional de Normalización (ISO). Fue estandarizado por primera vez en 1989 y su versión más reciente ante la ISO fue publicada en 2018 (aunque representa cambios menos con respecto a su versión de 2011). Se usa ampliamente en la implementación de sistemas operativos y embedidos.

Un programa en lenguaje C es una colección de funciones que cooperan entre sí para realizar un cálculo, justo como los ejemplos de funciones que hicimos anteriormente. Normalmente una función sólo se evalúa si alguien más la usa. Por ejemplo, dadas las siguientes definiciones:

$$f(\text{real } x, \text{real } y) \mapsto \text{real } \{ \text{ regresa } x + 2y \ \}$$

$$g(\text{real } x, \text{real } y) \mapsto \text{real } \{ \text{ regresa } y^2 - x \ \}$$

$$h(\text{real } x, \text{real } y) \mapsto \text{real } \{ \text{ regresa } f(x, y) \ \}$$

podemos evaluar $h(1, 2)$ que vale 3. Durante este cálculo, nosotros evaluamos a la función $h$ y ésta a su vez evaluó a la función $f$, pero nunca se tuvo que evaluar a la función $g$.

En todo programa en C existe una función especial que se evalúa automáticamente y a partir de la cual se hace el resto del cálculo. Esa función se llama *main* y su forma más reducida es equivalente a:

$$main(\,) \mapsto \text{entero } \{ \text{ regresa } 0 \ \}$$

Por convención, cuando *main* regresa 0 significa que el programa debe terminar sin contratiempos (si un programa se traba y muere violentamente entonces regresa otro número, pero eso no nos importará en este curso). La función *main* no necesita parámetros porque como se evalúa automáticamente, no necesita que nadie le pase valores. La forma de escribir esto en el lenguaje C es la siguiente:

```
int main( ) {
   return 0;
}
```

En el lenguaje C, el tipo que regresa una función se escribe antes del nombre de la misma. El tipo entero se especifica con la palabra `int`. Las llaves gigantes de la notación usada en la introducción del curso se expresan en C como un par de llaves que abren y cierran. Las instrucciones a ejecutar van dentro de las llaves y la mayoría de las instrucciones de una sola línea terminan con punto y coma. El programa anterior puede compilarse y ejecutarse, pero no hará nada puesto que *main* lo primero y único que hace es terminar inmediatamente regresando cero. Si un usuario abre manualmente el ejecutable producido por el compilador, sólo notará que un programa inició pero inmediatamente terminó sin poder ver qué fue lo que ocurrió (y en realidad no ocurrió nada). Si el programa se ejecuta desde un IDE, normalmente una ventana permanecerá abierta con un mensaje que indica que el programa ya terminó.

Si queremos que *main* evalúe otra función, primero debemos definirla. Afortunadamente, todo lenguaje trae predefinidas cientos o incluso miles de funciones escritas por otros programadores. Esta colección de funciones predefinidas recibe el nombre de biblioteca estándar. La biblioteca estándar de C está repartida entre distintos archivos con extensión *.h* (que viene de *header* o encabezado en inglés). Estos archivos pueden incluirse con la sentencia `#include <archivo>` y el archivo de encabezado que más usaremos es `stdio.h`, el cual define funciones para poder pedir valores al usuario e imprimir mensajes. El programa más sencillo que imprime un mensaje al usuario es:

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`int main( ) {`<br>`  printf("Hola Mundo");`<br>`  return 0;`<br>`}` | `Hola Mundo` |

La función `printf` toma un mensaje o cadena delimitado por comillas dobles y la muestra al usuario.

> A partir de este momento y salvo que se especifique lo contrario, los códigos de ejemplo se deben escribir dentro de `main` antes de `return 0;` y también se debe incluir `stdio.h` para poder ejecutar el programa.

Es posible usar `printf` varias veces, pero por omisión los mensajes se imprimirán uno tras otro:

| Código | Salida |
|---|---|
| `printf("Hola Mundo");`<br>`printf("Hola Mundo");` | `Hola MundoHola Mundo` |

La forma de corregir esto es usando el caracter de salto de línea `\n`, el cual provoca que el resto del texto se escriba en una nueva línea. Los saltos de línea se pueden usar tantas veces se desee.

| Código | Salida |
|---|---|
| `printf("Hola Mundo\n");`<br>`printf("Hola M\nundo");` | `Hola Mundo`<br>`Hola M`<br>`undo` |

Para poder escribir comillas dobles dentro de un mensaje, se debe escribir la secuencia `\"`. Para poder escribir una diagonal invertida, se debe escribir la secuencia `\\`.
Si que queremos imprimir el número 5, por supuesto que una forma es la siguiente:

| Código | Salida |
|---|---|
| `printf("5");` | `5` |

Hay otras formas más interesantes y útiles de hacer lo anterior. El mensaje o cadena que recibe `printf` como primer parámetro admite *especificadores de formato*. Estos especificadores permiten intercalar valores dentro del mensaje. En particular, el especificador `%d` permite intercalar valores enteros y `printf` tomará dichos valores de los parámetros adicionales que se usaron al evaluar la función.

| Código | Salida |
|---|---|
| `printf("Hola %d Hola %d", 5, 7);` | `Hola 5 Hola 7` |

En el lenguaje C, las variables suelen declararse con la notación *tipo nombre = valor;* donde el nombre de una variable sólo puede contener letras, dígitos y guiones bajos, pero no puede comenzar con dígitos. El ejemplo anterior lo podemos reescribir de la siguiente forma:

| Código | Salida |
|---|---|
| `int a = 5;`<br>`int b = 7;`<br>`printf("Hola %d Hola %d", a, b);` | `Hola 5 Hola 7` |

En este caso, `printf` tomará los valores a intercalar directamente de los valores de las variables. Es posible declarar varias variables en la misma línea cuando su tipo es el mismo:

| Código | Salida |
|---|---|
| `int a = 5, b = 7;`<br>`printf("Hola %d Hola %d", a, b);` | `Hola 5 Hola 7` |

No es obligatorio darle un valor inicial a una variable, pero entonces su valor es desconocido:

| Código | Salida |
|---|---|
| `int a, b;`<br>`printf("Hola %d Hola %d", a, b);` | `Hola -34213 Hola 12565724` |

Para entender por qué sucede eso, conviene imaginarnos un hotel donde los administradores jamás hacen la limpieza de los cuartos; si nosotros rentamos una habitación recientemente desocupada, lo más seguro es que nos encontremos el cochinero que los huéspedes anteriores dejaron ahí. Darle un valor inicial a una variable es el equivalente de limpiar el cuarto y dejarlo en un estado conocido.

Un programa es más útil si el usuario puede interactuar con él para darle los valores iniciales a usar durante el programa. Por ejemplo, si queremos escribir un programa que sume dos números, es deseable que el usuario pueda decidir qué números sumar. La lectura de variables se lleva a cabo con la función `scanf` que también usa el especificador de formato `%d` para indicar que queremos leer un entero. El entero leído se guardará en la variable pasada como parámetro adicional a `scanf` y se debe poner un `&` antes del nombre de la variable. Los valores leídos se denominan la entrada del programa.

| Código | Entrada | Salida |
|---|---|---|
| `int n;`<br>`scanf("%d", &n);`<br>`printf("El numero vale %d", n);` | `5` | `El numero vale 5` |

| Código | Entrada | Salida |
|---|---|---|
| `int a, b;`<br>`scanf("%d%d", &a, &b);`<br>`int c = a + b;`<br>`printf("%d", c);` | `1 2` | `3` |

La explicación de por qué se necesita `&` al usar `scanf` es un poco complicada, pero podemos dar la siguiente analogía: si pedimos una pizza a domicilio entonces tendremos que dar nuestra dirección para que nos la puedan hacer llegar. El operador `&` expone la dirección de una variable para que `scanf` pueda ubicar la variable en la que debe guardar el valor leído. Usar `&` con `printf` casi siempre es incorrecto,

porque lo que terminaremos imprimiendo es la dirección de la variable y no su valor (en México, una dirección se compone por estado, municipio, colonia, calle y número; en la computadora sólo existe la "calle memoria" y entonces la dirección de una variable está dada únicamente por un número sobre esa esa "calle"). En todo caso, el especificador correcto para imprimir una dirección es `%p` y no `%d`, aunque el número se imprimirá en base hexadecimal. Desafortunadamente, olvidar el operador `&` al usar `scanf` es un error común que no impide que el programa compile pero que suele provocar que el programa se trabe.

| Código | Salida |
|---|---|
| `int n = 5;`<br>`printf("%d  %p", n, &n);` | `5  000000004063fe4c` |

Cuando se usa `scanf` en un entorno en línea, basta escribir la entrada en el cuadro de texto disponible para ello. Cuando se usa `scanf` en una terminal interactiva como la que abre Code::Blocks con la opción "`Build and Run`", aparece un cursor parpadeando en espera de la entrada. En esa terminal (usualmente una ventana negra), basta escribir la entrada y darle *enter* para pasarle dicha entrada al programa.

Podemos escribir comentarios que serán ignorados por el compilador. El símbolo `//` inicia un comentario que continúa hasta el fin de línea, mientras que los símbolos `/*` y `*/` comentan todo un bloque.

| Código | Salida |
|---|---|
| `int n = 5;  // hola :)`<br>`/*`<br>`printf("%d\n", n);`<br>`printf("%d\n", n);`<br>`*/` | |

### 4.1. Ejercicios

El juez en línea omegaUp es una plataforma donde podemos enviar nuestro código para intentar resolver alguno de los problemas de programación disponibles. A partir de ahora, los ejercicios consistirán en problemas disponibles en esta plataforma. Puedes consultar un video sobre cómo usar omegaUp aquí.

1. Resuelve el problema `https://omegaup.com/arena/problem/Hola-Mundo-c`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Introduccion-a-OmegaUp`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Tres-Numeros-Al-Reves`.

## 5. Aritmética de enteros y reales

En C existe más de un tipo entero, aunque el más común es `int`. La diferencia entre estos tipos es el rango que manejan y su consumo de memoria:

| Tipo de dato | Número usual de bits | Rango usual de valores | Especificador de formato numérico |
|---|---|---|---|
| `char` | 8 | [-128, 127] o [0, 255] | `%hhd` |
| `signed char` | 8 | [-128, 127] | `%hhd` |
| `unsigned char` | 8 | [0, 255] | `%hhu` |
| `short` | 16 | [-32768, 32767] | `%hd` |
| `unsigned short` | 16 | [0, 65535] | `%hu` |
| `int` | 32 | [-2147483648, 2147483647] | `%d` |
| `unsigned int` | 32 | [0, 4294967295] | `%u` |
| `long` | 32 | [-2147483648, 2147483647] | `%ld` |
| `unsigned long` | 32 | [0, 4294967295] | `%lu` |
| `long long` | 64 | [-9223372036854775808, 9223372036854775807] | `%lld` |
| `unsigned long long` | 64 | [0, 18446744073709551615] | `%llu` |

Los tipos `long` suelen tener 64 bits en Linux y MacOS, por lo que se comportan como los tipos `long long` en tales plataformas. Para manipular enteros, nosotros usaremos `int` de forma casi exclusiva. En C también existen varios tipos de dato para manipular números reales:

| Tipo de dato | Número de bits | Rango de valores | Especificador de formato |
|---|---|---|---|
| `float` | 32 | [-3.40282e+038, 3.40282e+038] | `%f` |
| `double` | 64 | [-1.79769e+308, 1.79769e+308] | `%lf` |
| `long double` | 80 o 128 | [-1.18973e+4932, 1.18973e+4932] | `%Lf` |

Los operadores aritméticos disponibles son los siguientes:

| Nombre del operador | Símbolo | Resultado obtenido |
|---|---|---|
| Más | `+` | Suma de dos números |
| Menos | `-` | Resta de dos números |
| Por | `*` | Producto de dos números |
| Entre | `/` | (ambos operandos enteros) Cociente de la división entera<br>(algún operando real) División de dos números |
| Residuo o módulo | `%` | (ambos operandos enteros) Residuo de la división entera |

La división entera es la que nos enseñaron en la primaria, antes de saber manejar números con punto decimal. En este contexto, la división `5/2` da un cociente que vale 2, mientras que el residuo de la división se escribe `5%2` y vale 1. La división real (con punto decimal) se realiza si alguno de los dos operandos es un real. Como una literal numérica sin punto decimal se considera un entero y una literal numérica con punto decimal se considera un real, entonces `5.0/2` valdrá `2.5`, al igual que `5/2.0` y `5.0/2.0`.

**Código**

```
int a = 5, b = 2;
float x = 3.1416;

int s = 2 * a + 5 * (1 + 3 * b);
int c = a / b;
int m = a % b;
float t1 = a / 2;
float t2 = a / 2.0;
float t3 = x / 2;

printf("s vale %d\n", s);
printf("c vale %d\n", c);
printf("m vale %d\n", m);
printf("t1 vale %f\n", t1);
printf("t2 vale %f\n", t2);
printf("t3 vale %f\n", t3);
```

**Salida**

```
s vale 45
c vale 2
m vale 1
t1 vale 2.000000
t2 vale 2.500000
t3 vale 1.570800
```

El número de decimales en la impresión de un real se puede controlar con el especificador de formato `%.Nf` donde *N* es la cantidad de dígitos deseados. El valor impreso se obtiene mediante redondeo. De forma similar, podemos escribir un entero a *N* dígitos con el especificador de formato `%d0Nd`. El entero se imprimirá con ceros a la izquierda de ser necesario.

**Código**

```
float x = 3.976;
printf("%.2f\n", x);
int n = 7;
printf("%05d\n", n);
```

**Salida**

```
3.98
00007
```

La parte decimal de un real se trunca si se guarda el valor en un entero.

| Código | Salida |
|---|---|
| `float x = 3.5;`<br>`int n = x;`<br>`printf("%f\n%d", x, n);` | 3.500000<br>3 |

La multiplicación debe escribirse forzosamente con el operador `*`. La notación `(a)(b)` es incorrecta en C y deberá reescribirse como `a * b`. Las expresiones `(2 a)` y `2a` tampoco son correctas y deberán reescribirse como `2 * a`. Al igual que en la aritmética convencional, la multiplicación y la división tienen mayor precedencia que la suma o la resta. En caso de duda, se recomienda usar tantos paréntesis como sea necesario para asegurarse que el cálculo sea correcto.

| Código | Salida |
|---|---|
| `int a = 2, b = 5, c = 10;`<br>`printf("%d\n", a + b * c);`<br>`printf("%d\n", (a + b) * c);` | 52<br>70 |

### 5.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Programando-formulas`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Calculos-mentales-competitivos`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Evaluando-formulas-en-sucesion`.
4. Resuelve el problema `https://omegaup.com/arena/problem/Alicia-y-la-carrera-de-animales`.
5. Resuelve el problema `https://omegaup.com/arena/problem/La-banda-robatamales`.
6. Resuelve el problema `https://omegaup.com/arena/problem/La-hora-en-un-planeta-lejado`.

## 6. Variables y constantes

Una computadora se comporta de forma muy similar al de una calculadora científica común. En particular, tanto la calculadora científica como la computadora tienen variables a las que les podemos cambiar su valor en cualquier momento. Para cambiar el valor de una variable, podemos usar la notación *variable = valor*. Esto tiene el efecto de perder el viejo valor y quedarnos con el nuevo. Por ejemplo:

| Código | Salida |
|---|---|
| `int n = 5;`<br>`printf("%d\n", n);`<br>`n = 2;`<br>`printf("%d\n", n);` | 5<br>2 |

A la operación anterior se le conoce como asignación. El tipo de una variable se especifica únicamente cuando la variable se crea y es un error volver a mencionarlo. Además, el lenguaje C no permite cambiar el tipo original de una variable. Hay que recalcar que este uso del símbolo `=` es muy distinto al de su uso en matemáticas. En matemáticas, el símbolo = se usa para decretar una igualdad entre dos símbolos, mientras que en C se usa para guardar un valor en una variable, olvidando el valor viejo. Muchos matemáticos preferirían usar el símbolo ← para denotar una asignación, pero este símbolo es difícil de teclear y muchos lenguajes usan `=` para denotar una asignación a pesar del enojo de los matemáticos.

Una variable puede tomar un nuevo valor que dependa de su viejo valor. Por ejemplo:

| Código | Salida |
|---|---|
| `int n = 5;`<br>`printf("%d\n", n);`<br>`n = n + 2;`<br>`printf("%d\n", n);` | 5<br>7 |

En este caso, la expresión `n = n + 2` se interpreta como sigue: primero se calcula (mentalmente digamos) el valor de `n + 2`. Al ejecutar esa parte de la expresión `n` aún vale 5, por lo que `n + 2` vale 7. Justo ese 7 es el que ahora se guardará en `n`, reemplazando el viejo valor.

Afortunadamente, para modificar una variable en términos de su valor anterior existe una notación abreviada y más fácil de entender. A continuación se muestran ejemplos de esta notación abreviada.

| Expresión abreviada | Expresión sin abreviar | Semántica |
|---|---|---|
| `a += b;` | `a = a + b;` | Sumar `b` en `a`. |
| `a -= b;` | `a = a - b;` | Restar `b` en `a`. |
| `a *= b;` | `a = a * b;` | Multiplicar `b` en `a`. |
| `a /= b;` | `a = a / b;` | Dividir `b` en `a`. |
| `a %= b;` | `a = a % b;` | Calcular el residuo de $\frac{a}{b}$ en `a`. |

| Código | Salida |
|---|---|
| `int n = 5;`<br>`printf("%d\n", n);`<br>`n += 2;`<br>`printf("%d\n", n);`<br>`n *= 3;`<br>`printf("%d\n", n);` | `5`<br>`7`<br>`21` |

El valor de una variable se establece en el momento en el ocurre la asignación y dicho valor se vuelve independiente de lo que ocurra después con otras variables. Por ejemplo:

| Código | Salida |
|---|---|
| `int a = 5;`<br>`int b = a;  // copiamos el valor`<br>`a = 0;`<br>`printf("%d\n", b);` | `5` |

C también dispone de los operadores `++` y `--` que sirven para incrementar y decrementar una variable. Los operadores pueden escribirse como prefijos (preincremento `++a` y predecremento `--a`) o como sufijos (posincremento `a++` y posdecremento `a--`). La diferencia entre ambos estilos se puede observar al intentar usarlas dentro de una expresión más grande: la notación prefija usa el valor adquirido por la variable después de ser modificada, mientras que la notación posfija usa el valor previo a la modificación:

| Código | Salida |
|---|---|
| `int n = 0;`<br>`printf("%d\n", ++n);`<br>`printf("%d\n", n++);`<br>`printf("%d\n", n);` | `1`<br>`1`<br>`2` |

Usar la palabra `const` en la declaración de una variable provoca que el compilador prohiba modificar su valor posteriormente. Esto la vuelve una variable de sólo lectura.

| Código | Diagnóstico del compilador |
|---|---|
| `const int n = 5;`<br>`n = 7;`<br>`const int m;` | `error: assignment of read-only variable`<br>`    n = 7;`<br>`      ^` |

# 7. Uso de funciones

Del mismo modo que declaramos la función `main`, podemos definir funciones adicionales. Lo usual es que, como en matemáticas, estas funciones tomen valores y regresen un resultado. Por ejemplo:

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`float promedio(float a, float b) {`<br>`  return (a + b) / 2;`<br>`}`<br><br>`int main( ) {`<br>`  printf("%f", promedio(7, 10));`<br>`  return 0;`<br>`}` | `8.5` |

Cuando apenas estamos aprendiendo a programar, generalmente nos es difícil decidir si definir o no funciones auxiliares, ya que siempre existe la alternativa de poner toda la lógica del programa dentro de `main`. La recomendación es definir una función auxiliar cuando estemos en alguna de estas situaciones:

- Alguna parte del cálculo es muy común y general (por ejemplo, calcular el máximo de dos números).
- Cierto cálculo se realizará varias veces en distintas partes del código (entonces bastaría escribirlo una vez y evaluar la función varias veces).
- Cierto cálculo es muy complicado, pero depende únicamente de unos pocos valores de entrada (de este modo podemos aislar el cálculo del resto del programa).

Como tal, las funciones son el principal mecanismo de los lenguajes de programación actuales para construir software cada vez más complicado, sin que por ello éste se vuelva incomprensible o inmanejable. Aunque un programa tenga miles o milles de líneas de código, si una función es corta y está bien escrita entonces debe ser fácil poder razonar sobre ella.
Desafortunadamente, en C es un error usar una función que está declarada abajo de su punto de uso:

| Código | Diagnóstico del compilador |
|---|---|
| `#include <stdio.h>`<br><br>`int main( ) {`<br>`  printf("%f", promedio(7, 10));`<br>`  return 0;`<br>`}`<br><br>`float promedio(float a, float b) {`<br>`  return (a + b) / 2;`<br>`}` | `error: conflicting types for ‘promedio’`<br>` float promedio(float a, float b) {`<br>`  ^~~~~~~` |

Si nos entercamos en definir una función abajo de `main`, podemos declararla arriba y definirla abajo. La declaración sin definición de una función se hace con `;` sin especificar el cuerpo de la función:

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`float promedio(float a, float b);`<br><br>`int main( ) {`<br>`  printf("%f", promedio(7, 10));`<br>`  return 0;`<br>`}`<br><br>`float promedio(float a, float b) {`<br>`  return (a + b) / 2;`<br>`}` | `8.5` |

Salvo que esté justificado, no se recomienda declarar una función sin definirla. Hacer esto aumenta la cantidad de código escrito y nos obliga a que cualquier cambio en el nombre, parámetros o tipo de retorno de una función se tenga que hacer en dos partes del código, en lugar de sólo en una.

Como en matemáticas, los símbolos de una función son independientes de los símbolos de las demás funciones. Cuando evaluamos `g(1,3)` en el siguiente programa, las variables `x,y` de `g` valen $1, 3$ respectivamente. Cuando `g(1,3)` a su vez evalúa `f(3,2)` tenemos que las variables `x,y` de `f` valen $3, 2$.

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`float f(float x, float y) {`<br>`  return 3 * x + y;`<br>`}`<br><br>`float g(float x, float y) {`<br>`  return f(y, 2 * x);`<br>`}`<br><br>`int main( ) {`<br>`  printf("%f", g(1, 3));`<br>`  return 0;`<br>`}` | `11.000000` |

Si una función no necesita tomar valores de entrada para regresar un valor, entonces la lista de parámetros puede ir vacía (como en el caso de `main`). Aunque en matemáticas esto suele no tener sentido, en computación es usual. Por otra parte, ya hemos explicado que la sentencia `return` sirve para especificar el valor que la función debe regresar. En caso de que la función tenga más de un `return`, el primero que se ejecute tiene prioridad y además termina de inmediato la ejecución de la función:

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`int f( ) {`<br>`  return 2;`<br>`  return 7;`<br>`}`<br><br>`int main( ) {`<br>`  printf("%d", f( ));`<br>`  return 0;`<br>`}` | `2` |

En general, hay dos clases de funciones: aquéllas que calculan valores y aquéllas que realizan alguna acción. Cuando una función realiza una acción y no tiene nada útil que regresar, el tipo de retorno se especifica con la palabra `void` y no es necesario hacer un `return` dentro de la función. Por ejemplo, a continuación se muestra una función que imprime un mensaje pero que no calcula ningún valor útil:

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`void saluda( ) {`<br>`  printf("hola\n");`<br>`}`<br><br>`int main( ) {`<br>`  saluda( );`<br>`  return 0;`<br>`}` | `hola` |

Se puede usar la sentencia `return;` para terminar anticipadamente una función que regresa `void`. Una variable que esté declarada con la palabra reservada `static` retiene su valor aún después de que la función termine. La inicialización de la variable sólo ocurre la primera vez que el hilo de ejecución pasa por ahí; en caso de que no se inicialice, adquiere el valor inicial 0.

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`void f( ) {`<br>`  static int a, b = 5;`<br>`  printf("%d %d\n", a++, b++);`<br>`}`<br><br>`int main( ) {`<br>`  f( );`<br>`  f( );`<br>`  return 0;`<br>`}` | `0 5`<br>`1 6` |

La biblioteca estándar de C tiene predefinidas muchas funciones útiles. A continuación se describen sólo algunas de ellas. Para más informacion, pueden consultar `https://en.cppreference.com/w/c/header`.

| **Funciones de `math.h` o `tgmath.h`** | |
|---|---|
| **Función de biblioteca** | **Valor calculado** |
| `double fabs(double x);` | $\|x\|$. |
| `double exp(double x);` | $e^x$. |
| `double log(double x);` | $\ln(x)$. |
| `double log2(double x);` | $\log_2(x)$. |
| `double log10(double x);` | $\log_{10}(x)$. |
| `double pow(double x, double y);` | $x^y$. |
| `double sqrt(double x);` | $\sqrt{x}$. |
| `double cbrt(double x);` | $\sqrt[3]{x}$. |
| `double sin(double x);` | $\sin(x)$ donde $x$ está en radianes. |
| `double cos(double x);` | $\cos(x)$ donde $x$ está en radianes. |
| `double tan(double x);` | $\tan(x)$ donde $x$ está en radianes. |
| `double asin(double x);` | $\arcsin(x)$ en radianes. |
| `double acos(double x);` | $\arccos(x)$ en radianes. |
| `double atan(double x);` | $\arctan(x)$ en radianes. |
| `double trunc(double x);` | La parte entera de $x$. |
| `double ceil(double x);` | $\lceil x \rceil$ |
| `double floor(double x);` | $\lfloor x \rfloor$ |
| `double round(double x);` | El entero más cercano a $x$, tendiendo a 0. |
| `double fmax(double x, double y);` | El máximo de $x, y$. |
| `double fmin(double x, double y);` | El mínimo de $x, y$. |

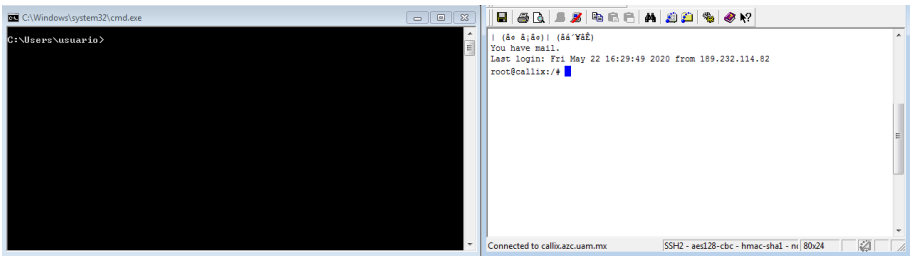
| **Funciones de `stdlib.h`** | |
|---|---|
| **Función de biblioteca** | **Acción realizada** |
| `void srand(unsigned s);` | Elige la $s$-ésima secuencia de números pseudoaleatorios de la biblioteca. |
| `int rand( );` | Regresa el siguiente número pseudoaleatorio de la secuencia seleccionada. |
| `void exit(int r);` | Termina el programa, regresando $r$ como valor de retorno de `main`. |

La mayoría de las funciones matemáticas están definidas principalmente sobre variables de tipo `double` y no de tipo `float`. De todos modos, para cada función existen versiones adicionales que trabajan sobre variables `float` y `long double` (por ejemplo, para `round` existen las versiones `roundf` y `roundl` respectivamente). Al incluir `tgmath.h` en lugar de `math.h` se puede usar el nombre principal de la función (`round`

en el ejemplo anterior) y el compilador decidirá automáticamente cuál de las tres versiones debe usar. Esto evita tener que especificar el sufijo correcto en el nombre de la función.

# 8. La línea de comandos

La línea de comandos (también llamada terminal, consola o símbolo del sistema) es un entorno basado en texto en el que es posible navegar por el sistema de archivos e invocar los programas instalados en la computadora. En Windows, la línea de comandos puede abrirse con el programa `cmd.exe`, mientras que en Linux normalmente se usa el programa `bash`. La apariencia de la línea de comandos generalmente es la de una ventana pequeña con texto y un cursor, en la cual podemos escribir comandos que deben ser interpretados y ejecutados por el sistema operativo.




La línea de comandos en Windows y Linux.

El *prompt* es un término informático que se refiere a los caracteres al inicio de una línea cuando recién se abre la terminal, justo atrás del cursor (por ejemplo, `C:\>`, `>>` `/#` o `$`). Esto indica que la línea de comandos está lista para recibir comandos por parte del usuario. Normalmente, el *prompt* también indica el denominado "directorio actual de trabajo", que es el directorio sobre el cual se buscarán, crearán o eliminarán los archivos que el usuario indique, además de ser la ruta base a partir de la cual se calculan rutas relativas de archivos o carpetas. Los comandos más importantes relacionados con este curso son:

| Comando | Acción realizada |
|---|---|
| `dir` | (Windows) Lista el contenido del directorio de trabajo. |
| `ls` | (Linux y MacOS) Lista el contenido del directorio de trabajo. |
| `type` *archivo* | (Windows) Lista el contenido del archivo. |
| `cat` *archivo* | (Linux y MacOS) Lista el contenido del archivo. |
| `cd ..` | Sube al directorio padre. |
| `cd` *carpeta* | Entra al directorio indicado. |
| `gcc` *código* `-o` *ejecutable* | Compila el código indicado y crea un ejecutable con el nombre dado. Se requiere que el compilador esté en una ubicación conocida por la línea de comandos. |
| *ejecutable* | Ejecuta el programa indicado. En Windows, el programa puede estar en el directorio de trabajo. |
| *./ejecutable* | (Linux y MacOS) Ejecuta el programa indicado que está en el directorio de trabajo. |
| *ejecutable* `<` *archivo_entrada*<br>*./ejecutable* `<` *archivo_entrada* | Ejecuta el programa indicado, tomando la entrada del archivo indicado. |
| *ejecutable* `>` *archivo_salida*<br>*./ejecutable* `>` *archivo_salida* | Ejecuta el programa indicado, enviando la salida al archivo indicado. |

Los dos últimos comandos pueden combinarse (*ejecutable* `<` *archivo_entrada* `>` *archivo_salida*) y usan un concepto denominado redirección de la entrada y la salida. La redirección es muy útil para para evitar tener que teclear la entrada del programa una y otra vez durante pruebas (podemos guardar la entrada en un archivo y redireccionarla), así como para conservar la salida del programa en un archivo.

# 9. Comparaciones y expresiones condicionales

El lenguaje C permite comparar valores y actuar en consecuencia. Para comparar valores usaremos los operadores relacionales de matemáticas, los cuales se escriben en C de una forma peculiar:

| Nombre del operador | Notación en matemáticas | Notación en C |
|---|---|---|
| Menor que | $<$ | `<` |
| Mayor que | $>$ | `>` |
| Menor o igual que | $\leq$ | `<=` |
| Mayor o igual que | $\leq$ | `>=` |
| Igual que | $=$ | `==` |
| Diferente que | $\neq$ | `!=` |

Estos operadores tienen una semántica curiosa: una comparación es muy parecida a una pregunta. Por ejemplo, escribir `a < b` es como preguntar ¿$a < b$? que puede tener como respuesta *sí* o *no*. En el lenguaje C, un *sí* se representa con el valor 1 y un *no* con el valor 0. Por ejemplo:

| Código | Salida |
|---|---|
| `int a = 2, b = 5;`<br>`int c = (a < b);`<br>`printf("%d", c);` | `1` |

Es importante hacer la distinción entre los símbolos `=` y `==` de C. El operador `=` realiza la asignación de un valor a una variable, mientras que el operador `==` pregunta si dos valores son iguales:

| Código | Salida |
|---|---|
| `int a = 2, b = 5;`<br>`int c = (a == b);`<br>`printf("%d", c);` | `0` |

Se debe tener cuidado con este par de operadores, ya que escribir `=` cuando lo que en realidad se quería escribir era `==` compila en muchos contextos pero tiene un efecto completamente distinto al deseado. Por esta razón, puede resultar difícil encontrar la fuente del error.

Aunque una comparación verdadera evalúa a 1 y una comparación falsa evalúa a 0, es posible elegir valores distintos. La expresión ternaria o expresión condicional (*condición* `?` $v_1$ `:` $v_2$) evalúa al valor $v_1$ si la condición es verdadera y evalúa al valor $v_2$ si la condición es falsa:

| Código | Salida |
|---|---|
| `int a = 2, b = 5;`<br>`int c = (a < b ? 9 : 70);`<br>`printf("%d", c);` | `9` |

En ese sentido, `(a < b)` y `(a < b ? 1 : 0)` son equivalentes. Las expresiones condicionales también sirven para elegir una de dos literales de cadena, por lo que podemos imprimir cosas distintas si combinamos esta característica con la función `printf`. Por ejemplo:

| Código | Entrada | Salida |
|---|---|---|
| `int n;`<br>`scanf("%d", &n);`<br>`printf((n % 2 == 0 ? "par" : "non"));` | `6` | `par` |

Los paréntesis alrededor de una comparación o de una expresión ternaria son opcionales, pero mejoran la legibilidad del código. Una comparación o expresión condicional que evalúa a 1 o a 0 también recibe el nombre de expresión booleana. En C los valores booleanos generalmente se guardan en un `int`, aunque tras incluir `<stdbool.h>` se define el tipo `bool` que usa menos memoria, así como las constantes `true` para verdadero y `false` para falso, las cuales valen 1 y 0 respectivamente.

### 9.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Par-o-Impar`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Fahrenheit-a-Centigrados`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Cuantos-valen-7`.
4. Resuelve el problema `https://omegaup.com/arena/problem/El-menor-de-tres-numeros`.

## 10. Operadores lógicos

Los operadores lógicos del lenguaje C permiten invertir el sentido de una comparación y combinar más de una comparación para formar una condición más grande. A continuación se listan los operadores lógicos disponibles y después se explica cada uno por separado:

| Nombre del operador | Notación en matemáticas | Notación en C |
|---|---|---|
| Negación lógica | $\neg$ | `!` |
| Conjunción lógica (conector *y*) | $\wedge$ | `&&` |
| Disyunción lógica (conector *o*) | $\vee$ | `\|\|` |

La negación lógica `!` invierte el sentido de una comparación o expresión booleana. Por ejemplo, el opuesto de preguntar si `a == b` se escribe `!(a == b)` y se lee ¿`a` *no* es igual que `b`?. Sin embargo, preguntar si una cosa *no es igual* que otra es lo mismo que preguntar si son diferentes. En ese sentido, rara vez se escribe `!(a == b)` y se prefiere escribir `a != b`. De forma similar, el opuesto de preguntar si `a < b` se escribe `!(a < b)` y se lee ¿`a` *no* es menor que `b`?, aunque una forma más simple es preguntar si `a >= b`. Los paréntesis alrededor de la comparación a negar son necesarios. Las equivalencias entonces son:

| Expresión | ¿Cómo se lee? | Expresión equivalente para números |
|---|---|---|
| `!(a < b)` | ¿`a` no es menor que `b`? | `a >= b` |
| `!(a > b)` | ¿`a` no es mayor que `b`? | `a <= b` |
| `!(a <= b)` | ¿`a` no es menor o igual que `b`? | `a > b` |
| `!(a >= b)` | ¿`a` no es mayor o igual que `b`? | `a < b` |
| `!(a == b)` | ¿`a` no es igual que `b`? | `a != b` |
| `!(a != b)` | ¿`a` no es diferente que `b`? | `a == b` |

| Código | Salida |
|---|---|
| `int a = 2, b = 5;`<br>`printf("%d\n", a < b);`<br>`printf("%d\n", !(a < b));` | `1`<br>`0` |

Como regla general, si `a < b` vale 1 entonces `!(a < b)` vale 0 y viceversa. Lo mismo aplica para el resto de los operadores de comparación en el caso de números enteros y reales.

La conjunción lógica `&&` combina dos comparaciones o expresiones booleanas, de modo que la condición compuesta vale 1 sólo si las dos expresiones individuales también valen 1. Por esta razón, este operador también recibe el nombre de conector *y*: la primera y la segunda comparación deben ser verdaderas para que la expresión conjunta sea verdadera. Por ejemplo:

| Código | Salida |
|---|---|
| `int n = 5;`<br>`printf("%d\n", n > 3);`<br>`printf("%d\n", n % 2 == 0);`<br>`printf("%d\n", n > 3 && n % 2 == 0);` | `1`<br>`0`<br>`0` |

De manera similar, la disyunción lógica `||` combina dos comparaciones o expresiones booleanas, de modo que la condición compuesta vale 1 si alguna de las dos expresiones individuales vale 1. Por esta razón, este operador también recibe el nombre de conector *o*: la primera o la segunda comparación debe ser verdadera para que la expresión conjunta sea verdadera. Por supuesto, si ambas son verdaderas entonces también lo será la expresión conjunta. Por ejemplo:

| Código | Salida |
|---|---|
| `int n = 5;`<br>`printf("%d\n", n > 3);`<br>`printf("%d\n", n % 2 == 0);`<br>`printf("%d\n", n > 3 \|\| n % 2 == 0);` | `1`<br>`0`<br>`1` |

Desafortunadamente, la notación usual en matemáticas ¿$a \leq b \leq c$? que pregunta si el valor de $b$ está entre los valores de $a$ y $c$ se debe descomponer en dos comparaciones `a <= b && b <= c`. Peor aún, la expresión `a <= b <= c` sí compila pero se interpreta como `(a <= b) <= c`, lo que compara el 1 o el 0 obtenido por `(a <= b)` contra el valor de `c`, lo que seguramente no es lo que el programador quiere.

| Código | Salida |
|---|---|
| `int a = -5, b = -3, c = -1;`<br>`printf("%d\n", a <= b <= c);  // mal`<br>`printf("%d\n", a <= b && b <= c);` | `0`<br>`1` |

A continuación se presenta una tabla que resume los valores que obtenemos al usar los operadores `!`, `&&` y `||` según los valores de las expresiones booleanas involucradas. Un par de resultados importantes que no se incluyen en la tabla son la equivalencia de `!(`$v_1$ `&&` $v_2$`)` con `(!`$v_1$ `||` `!`$v_2$`)` y la equivalencia de `!(`$v_1$ `||` $v_2$`)` con `(!`$v_1$ `&&` `!`$v_2$`)`. ¿Se te ocurre cómo inferir esas equivalencias?

| **Tabla de verdad de operadores lógicos** | | | | |
|---|---|---|---|---|
| $v_1$ | $v_2$ | $v_1$ && $v_2$ | $v_1$ \|\| $v_2$ | !$v_1$ |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

Una característica peculiar de los operadores `&&` y `||` es algo que se denomina *comportamiento en corto circuito*. Si basta evaluar el operando del lado izquierdo del conector para determinar el valor de la expresión lógica, entonces el lado derecho no se evalúa. Este comportamiento puede observarse cuando la expresión del lado derecho tiene un efecto secundario, como se muestra a continuación:

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`int f( ) {`<br>`  printf("dentro de la función f\n");`<br>`  return 1;`<br>`}`<br><br>`int main( ) {`<br>`  printf("%d\n", 0 && f( ));`<br>`  printf("%d\n", 1 \|\| f( ));`<br>`  printf("fin\n");`<br>`  return 0;`<br>`}` | `0`<br>`1`<br>`fin` |

En el ejemplo anterior, la evaluación de `f( )` no se llevará a cabo en ninguna de las dos expresiones porque en ambos casos basta examinar el lado izquierdo. En el caso del operador `&&`, se necesitaría que

ambos lados del conector valgan 1 para que la expresión valga 1, por lo que el 0 de la izquierda basta para determinar que la expresión completa valdrá 0. Algo similar ocurre en el caso del operador `||`, donde el 1 de la izquierda basta para determinar que la expresión completa valdrá 1.

### 10.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Triangulo-equilatero`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Interseccion-de-intervalos`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Cuantos-dias-tiene-febrero`.

## 11. Sentencias condicionales

Aunque las expresiones ternarias son muy útiles cuando queremos elegir entre dos expresiones, con frecuencia vamos a querer ejecutar toda una serie de instrucciones de forma condicional. Las sentencias de control `if` y `else` nos permiten hacer justo eso. La forma más común de escribir estas sentencias es:

```
if (condición 1) {

} else if (condición 2) {

} else if (condición ...) {

} else {

}
```

Las secciones `else if` y `else` son opcionales, aunque de existir una sección `else`, ésta debe aparecer al último. Se pueden escribir instrucciones arbitrarias dentro de las llaves de cada sección, aunque la única sección que se ejecutará es (de arriba para abajo) la primera cuya condición sea verdadera, o la sección del `else` si todas las condiciones fueron falsas. Después de evaluar un secuencia de sentencias `if` y `else`, la ejecución del programa continúa normalmente. Por ejemplo:

| Código | Salida |
|---|---|
| `int n = 4;`<br>`if (n > 7) {`<br>`  printf("sección 1\n");`<br>`} else if (n < 5) {`<br>`  printf("sección 2\n");`<br>`} else if (n == 4) {`<br>`  printf("sección 3\n");`<br>`}`<br><br>`printf("hola\n");`<br>`if (n < 1) {`<br>`  printf("en if\n");`<br>`} else {`<br>`  printf("en else\n");`<br>`}` | `sección 2`<br>`hola`<br>`en else` |

Como ya se dijo, cualquier tipo de instrucción puede ir dentro de una sección delimitada entre llaves. Esto incluye a la instrucción `return`. En el siguiente ejemplo, la función `minimo` tiene dos `return`, donde uno o el otro se ejecuta dependiendo de la condición:

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`int minimo(int a, int b) {`<br>`  if (a < b) {`<br>`  return a;`<br>`  } else {`<br>`  return b;`<br>`  }`<br>`}`<br><br>`int main( ) {`<br>`  printf("%d", minimo(5, 2));`<br>`}` | 2 |

En el ejemplo anterior, podríamos reemplazar el uso de `if` y `else` por una expresión ternaria con `return (a < b ? a : b);` que es más fácil de razonar. También se pueden usar sentencias `if` y `else` dentro de los bloques de otro `if` o `else`. A esto se le conoce como `if` anidado:

| Código | Salida |
|---|---|
| `int n = 5;`<br>`if (n > 2) {`<br>`  printf("en if externo\n");`<br>`  if (n == 3) {`<br>`  printf("en if anidado\n");`<br>`  } else {`<br>`  printf("en else anidado\n");`<br>`  }`<br>`} else {`<br>`  printf("en else externo\n");`<br>`}` | `en if externo`<br>`en else anidado` |

Una variable que esté declarada dentro de algún bloque oculta a otra que esté afuera con el mismo nombre. La variable interna deja de existir cuando su bloque termina. Por ejemplo:

| Código | Salida |
|---|---|
| `int n = 5;`<br>`if (n > 2) {`<br>`  int n = 8;`<br>`  printf("%d\n", n);`<br>`}`<br>`printf("%d\n", n);` | 8<br>5 |

Algo muy raro del lenguaje C es que cualquier valor distinto de 0 activa la condicion de un `if`. Se debe evitar el uso de esta característica.

| Código | Salida |
|---|---|
| `int n = -7;`<br>`if (n) {`<br>`  printf("entramos!?");`<br>`}` | `entramos!?` |

Adicionalmente, el lenguaje C proporciona una sentencia de selección basada en valores y no en condiciones. Esta sentencia, llamada `switch`, toma un valor entero y lo usa para saltar al grupo de instrucciones etiquetadas bajo el mismo valor:

| Código | Salida |
|---|---|
| `int n = 2;`<br>`switch (n) {`<br>`case 1:`<br>`  printf("uno\n");`<br>`case 2:`<br>`  printf("dos\n");`<br>`case 3:`<br>`  printf("tres\n");`<br>`}` | `dos`<br>`tres` |

Desafortunadamente, el comportamiento por omisión es saltar al grupo de código con el valor correspondiente y seguir la ejecución hacia abajo, incluso llegando a ejecutar instrucciones de otros casos. Para evitar este comportamiento "en cascada", se puede usar la sentencia `break`.

| Código | Salida |
|---|---|
| `int n = 2;`<br>`switch (n) {`<br>`case 1:`<br>`  printf("uno\n");`<br>`  break;`<br>`case 2:`<br>`  printf("dos\n");`<br>`  break;`<br>`case 3:`<br>`  printf("tres\n");`<br>`  break;`<br>`}` | `dos` |

Si el valor no está contemplado en el `switch` entonces se ejecuta el caso por defecto especificado con la palabra `default`, o ninguna instrucción si tampoco hay caso por defecto. Por ejemplo:

| Código | Salida |
|---|---|
| `int n = 5;`<br>`switch (n) {`<br>`case 1:`<br>`  printf("uno\n");`<br>`  break;`<br>`case 2:`<br>`  printf("dos\n");`<br>`  break;`<br>`default:`<br>`  printf("otro valor\n");`<br>`  break;`<br>`}` | `otro valor` |

No se sugiere usar sentencias `switch` a pesar de que, a primera vista, parezca más elegante que una serie de preguntas `if` y `else`, ya que los últimos son más generales y con frecuencia más rápidos.

## 11.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Detectando-el-orden`.
2. Resuelve el problema `https://omegaup.com/arena/problem/Escamitas`.
3. Resuelve el problema `https://omegaup.com/arena/problem/Calculos-condicionales`.
4. Resuelve el problema `https://omegaup.com/arena/problem/Calculando-con-masmenos`.
5. Resuelve el problema `https://omegaup.com/arena/problem/Una-serie-poco-interesante`.

# 12. Sentencias iterativas

El lenguaje C provee tres sentencias que nos permiten ejecutar repetidamente un bloque de instrucciones. Con ellas, podremos programar fácilmente tareas triviales como imprimir todos los números del 1 al 100, así como cálculos complicados que requieran ejecutar millones de instrucciones. La primera de estas sentencias, la sentencia `while`, es notacionalmente incluso más sencilla que un `if` y `else`:

```
while (condición) {

}
```

La sentencia `while` opera de forma muy similar a un `if`: si la condición es verdadera entonces el bloque se ejecuta y si la condición es falsa entonces no se ejecuta. La diferencia con el `if` es que la sentencia `while` vuelve a revisar la condición después de ejecutar el bloque, y si la condición sigue siendo verdadera, entonces se vuelve a ejecutar el bloque. Este proceso se repite hasta que la condición se vuelva falsa.

| Código | Salida |
|---|---|
| `int n = 1;`<br>`while (n < 5) {`<br>`  printf("%d\n", n);`<br>`  n += 1;`<br>`}` | `1`<br>`2`<br>`3`<br>`4` |

Cada ejecución del bloque de código recibe el nombre de iteración y un ciclo es el conjunto de todas las iteraciones que realizó la sentencia. Es por esto que se dice que el `while` es una sentencia iterativa.

Un ciclo infinito es aquél donde la condición nunca se vuelve falsa. Por ejemplo:

| Código | Salida |
|---|---|
| `while (1) {`<br>`  printf("hola\n");`<br>`}` | `hola`<br>`hola`<br>`...` |

Salvo que el programa se esté ejecutando en un ambiente que restrinja el tiempo límite de ejecución, tendremos que cerrar el programa manualmente en la presencia de un ciclo infinito. Si estamos ejecutando el programa desde la línea de comandos, esto puede hacerse con `Ctrl+Z`.

Dentro del bloque de código de un ciclo `while` podemos escribir instrucciones arbitrarias como `if`, `else` y `return`. En el siguiente ejemplo, la función `f` tiene un ciclo y ejecuta un `return` si el `if` dentro del ciclo se cumple. Recordemos que un `return` termina inmediatamente la función:

| Código | Salida |
|---|---|
| `#include <stdio>`<br><br>`int f( ) {`<br>`  int n = 1;`<br>`  while (n < 10) {`<br>`  if (n == 3) {`<br>`  return n;`<br>`  } else {`<br>`  n += 1;`<br>`  }`<br>`  }`<br>`}`<br><br>`int main( ) {`<br>`  printf("%d\n", f( ));`<br>`  return 0;`<br>`}` | `3` |

Dentro de un ciclo, la sentencia `break` sirve para terminarlo abruptamente. Esta sentencia suele ser muy útil cuando se vuelve muy complicado controlar un ciclo únicamente mediante su condición principal.

| Código | Salida |
|---|---|
| `int n = 1;`<br>`while (n < 1000) {`<br>`  printf("%d\n", n);`<br>`  if (n == 3) {`<br>`  break;`<br>`  }`<br>`  n += 1;`<br>`}` | 1<br>2<br>3 |

La sentencia `continue` sirve para terminar abruptamente una iteración, pero no el ciclo completo. En un ciclo `while`, ejecutar un `continue` hará que saltemos inmediatamente a la evaluación de la condición del ciclo, sin haber terminado de ejecutar el resto de la iteración. En el siguiente ejemplo, saltarnos el resto de la iteración cuando `n` vale 3 provocará que quedemos atrapados en un ciclo infinito.

| Código | Salida |
|---|---|
| `int n = 1;`<br>`while (n < 1000) {`<br>`  printf("%d\n", n);`<br>`  if (n == 3) {`<br>`  continue;`<br>`  }`<br>`  n += 1;`<br>`}` | 1<br>2<br>3<br>3<br>... |

Ocasionalmente es obvio ver que la condición de un ciclo `while` será cierta al menos la primera vez:

| Código | Salida |
|---|---|
| `int n = 1;`<br>`while (n <= 16) {  // cierto al inicio`<br>`  printf("%d\n", n);`<br>`  n *= 2;`<br>`}` | 1<br>2<br>4<br>8<br>16 |

Cuando queramos evitar tener que evaluar la condición la primera vez, podemos usar la sentencia `do`, la cual revisa la condición al final del bloque de código y no al inicio. Al estar la condición abajo, el bloque de código se ejecutará incondicionalmente al menos la primera vez. A continuación se presenta la notación del ciclo `do`. Cabe resaltar que la condición del ciclo `do` debe terminar en punto y coma.

```
do {

} while (condición);
```

El ejemplo anterior lo podemos reescribir como sigue:

| Código | Salida |
|---|---|
| `int n = 1;`<br>`do {  // preguntaremos después`<br>`  printf("%d\n", n);`<br>`  n *= 2;`<br>`} while (n <= 16);` | 1<br>2<br>4<br>8<br>16 |

El comportamiento de las sentencias `break` y `continue` dentro de un ciclo `do` es el mismo que para un ciclo `while`. El ciclo `do` tiende a usarse muy pocas veces en la práctica y no existe en algunos lenguajes. Sin embargo, suele ser muy eficiente en los casos en los que puede ser usado.
La mayoría de los ciclos consisten de tres partes bien estructuradas y fácilmente identificables:

1. Inicialización: el fragmento de código donde establecemos los valores que tendrán las variables antes de entrar al ciclo por primera vez.

2. Condición: la expresión que controla cuántas iteraciones ejecutará el ciclo.

3. Actualización: el fragmento de código que se ejecuta dentro de una iteración y donde modificamos los valores de las variables rumbo a la posible ejecución de la siguiente iteración.

Señalaremos estas tres partes en el siguiente código que imprime los números del 1 al 4:

| Código | Salida |
|---|---|
| `int i = 1;  // inicialización`<br>`while (i <= 4) {  // condición`<br>`  printf("%d\n", i);`<br>`  i += 1;  // actualización`<br>`}` | 1<br>2<br>3<br>4 |

Para este tipo de ciclos, el lenguaje C proporciona una sentencia iterativa llamada `for` que permite escribir las tres partes anteriormente mencionadas en la misma línea. La forma general de la sentencia es:

```
for (inicialización; condición; actualización) {

}
```

El ejemplo anterior se puede reescribir como sigue:

| Código | Salida |
|---|---|
| `for (int i = 1; i <= 4; i += 1) {`<br>`  printf("%d\n", i);`<br>`}` | 1<br>2<br>3<br>4 |

El ciclo `for` es por mucho el más común en C y se suele preferir `++i` o `i++` en lugar de `i += 1` en la actualización. A diferencia del ciclo `while` o del ciclo `do` donde las variables de la inicialización forzosamente deben declararse fuera del ciclo y sobreviven a éste, una variable declarada dentro de la inicialización de un `for` es local al ciclo y no perdura. Esta característica se introdujo en C 1999 y no es común encontrarla en código viejo, pero cualquier compilador moderno debe aceptarla.

| Código | Diagnóstico |
|---|---|
| `for (int i = 1; i <= 4; i += 1) {`<br>`  printf("%d\n", i);`<br>`}`<br>`printf("%d\n", i);` | `error: ‘i’ undeclared`<br>`printf("%d\n", i);`<br>`^` |

Si necesitamos que una variable perdure después del ciclo, necesitaremos declararla afuera. Sin embargo, de todos modos podemos inicializarla en el ciclo `for`:

| Código | Salida |
|---|---|
| `int i;`<br>`for (i = 1; i <= 4; i += 1) {`<br>`  //printf("%d\n", i);`<br>`}`<br>`printf("%d\n", i);` | 5 |

El ejemplo anterior también pone en evidencia que la variable `i` sí llega a superar el valor de 4, pero es debido a que la comparación `5 <= 4` fue falsa que el ciclo termina.

Las tres partes de la sección de control del ciclo `for` son opcionales. Cuando la condición se omite, entonces el ciclo es infinito (salvo que contenga un `break`):

| Código | Salida |
|---|---|
| `int i = 1`<br>`for (;; i += 1) {`<br>`  printf("%d\n", i);`<br>`}` | `1`<br>`2`<br>`3`<br>`...` |

Es posible inicializar y actualizar más de una variable usando el operador coma.

| Código | Salida |
|---|---|
| `for (int i=1, j=1; i<=3; ++i, --j) {`<br>`  printf("%d %d\n", i, j);`<br>`}` | `1 1`<br>`2 0`<br>`3 -1` |

El comportamiento de la sentencia `break` dentro de un ciclo `for` es el mismo que para los otros ciclos. La sentencia `continue` termina abruptamente la iteración actual pero sí ejecuta la actualización.

| Código | Diagnóstico |
|---|---|
| `for (int i = 1; i <= 5; ++i) {`<br>`  continue;`<br>`}`<br>`printf("terminamos");` | `terminamos` |

Es posible ejecutar un ciclo dentro de otro y a esto se le denomina ciclo anidado. Por ejemplo:

| Código | Salida |
|---|---|
| `for (int i = 1; i <= 2; ++i) {`<br>`  for (int j = 1; j <= 2; ++j) {`<br>`  printf("%d %d\n", i, j);`<br>`  }`<br>`}` | `1 1`<br>`1 2`<br>`2 1`<br>`2 2` |

En el ejemplo anterior, el ciclo externo tiene la función de elegir un valor para la variable `i`. Cada vez que esto ocurre, se ejecuta el ciclo interno que itera sobre una variable distinta llamada `j`. Ejecutar un `break` o `continue` sólo afecta al ciclo más interno.

| Código | Salida |
|---|---|
| `for (int i = 1; i <= 2; ++i) {`<br>`  printf("i vale %d\n");`<br>`  for (int j = 1; j <= 2; ++j) {`<br>`  break;`<br>`  }`<br>`}` | `i vale 1`<br>`i vale 2` |

## 12.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Un-proceso-iterativo-sencillo`
2. Resuelve el problema `https://omegaup.com/arena/problem/Suma-de-los-digitos`
3. Resuelve el problema `https://omegaup.com/arena/problem/Divide-y-sumaras`

4. Resuelve el problema `https://omegaup.com/arena/problem/Calculando-el-logaritmo-base-2`
5. Resuelve el problema `https://omegaup.com/arena/problem/El-Caracol`
6. Resuelve el problema `https://omegaup.com/arena/problem/ciclo_mientras_no_cero`
7. Resuelve el problema `https://omegaup.com/arena/problem/Pares-e-impares`
8. Resuelve el problema `https://omegaup.com/arena/problem/Divisores-positivos`
9. Resuelve el problema `https://omegaup.com/arena/problem/Sumatoria-de-sumatorias`
10. Resuelve el problema `https://omegaup.com/arena/problem/CR-Leyendo-Varios-datos`
11. Resuelve el problema `https://omegaup.com/arena/problem/Calculos-iterativos`
12. Resuelve el problema `https://omegaup.com/arena/problem/Mensaje-de-Amor`
13. Resuelve el problema `https://omegaup.com/arena/problem/El-k-esimo-numero-primo`
14. Resuelve el problema `https://omegaup.com/arena/problem/Dibujando-un-triangulo`
15. Resuelve el problema `https://omegaup.com/arena/problem/supresores_de_picos`

# 13. Apuntadores y paso por referencia

Las funciones de un programa se comunican entre sí mediante transferencia de valores: cada vez que una función evalúa otra, ésta transfiere una copia de los valores usados en la invocación y la función llamada captura estos valores en sus propias variables. Esto quiere decir que por omisión, las variables de una función (y sus nombres) son independientes de las demás funciones. Por ejemplo:

| Código | Salida |
|---|---|

```c
#include <stdio.h>

void func(int a, int b) {
   printf("a y b de func: %d, %d\n");
}

int main( ) {
   int a = 2, b = 5;
   printf("a y b de main: %d, %d\n");
   func(b, a);
   return 0;
}
```

```
a y b de main: 2, 5
a y b de func: 5, 2
```

Esta forma de comunicación mediante transferencia de valores se llama paso por valor. Una función puede modificar los valores de sus variables sin que por ello se modifiquen las variables de las demás.

| Código | Salida |
|---|---|

```c
#include <stdio.h>

void func(int n) {
   n += 1;
}

int main( ) {
   int n = 2;
   func(n);
   printf("%d", n);
   return 0;
}
```

```
2
```

Los apuntadores nos permiten escapar de esta limitación. Un apuntador es una variable que almacena la dirección de otra variable, lo que le permite visitarla para leer su valor y para sobreescribirlo. Para que una variable de tipo `T` pueda ser visitada, necesita exponer su dirección con el operador prefijo `&` y esta dirección deberá ser almacenada en un apuntador de tipo `T*`. El operador prefijo `*` sobre el apuntador permite visitar a la variable referida. Por ejemplo:

| Código | Salida |
|---|---|
| `int n = 2;`<br>`int* p = &n;`<br>`*p = 0;`<br>`printf("%d", n);` | `0` |

Si reasignamos un apuntador a una nueva dirección, ahora referiremos a otra variable:

| Código | Salida |
|---|---|
| `int n = 2, m = 2;`<br>`int* p = &n;`<br>`*p = 0;`<br>`p = &m;`<br>`*p = 0;`<br>`printf("%d %d", n, m);` | `0 0` |

Si una variable expone su dirección al invocar una función, la función llamada podrá modificar el valor de dicha variable usando el apuntador:

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`void func(int* p) {`<br>`  *p += 1;`<br>`}`<br><br>`int main( ) {`<br>`  int n = 2;`<br>`  func(&n);`<br>`  printf("%d", n);`<br>`  return 0;`<br>`}` | `3` |

En C, a la capacidad de modificar la variable de una función desde otra función se le conoce como paso por apuntador o paso por referencia. El paso por apuntador permite implementar el algoritmo de intercambio de dos valores, el cual es uno de los algoritmos fundamentales en computación.

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`void intercambia(int* a, int* b) {`<br>`  int c = *a;`<br>`  *a = *b;`<br>`  *b = c;`<br>`}`<br><br>`int main( ) {`<br>`  int a = 2, b = 5;`<br>`  intercambia(&a, &b);`<br>`  printf("%d %d", a, b);`<br>`  return 0;`<br>`}` | `5 2` |

La estrategia del algoritmo anterior es la siguiente: no podemos asignar directamente el valor de una variable en la otra porque perderíamos uno de los dos valores. La idea es hacer primero una copia de respaldo de uno de los valores y luego ejecutar las asignaciones necesarias para lograr el efecto deseado.

### 13.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Forzando-la-caja-fuerte`

2. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-numeros`

## 14. Arreglos

El lenguaje C provee una forma de declarar decenos, cientos o miles de variables del mismo tipo, en una única declaración. La colección resultante recibe el nombre de arreglo. La declaración de un arreglo debe especificar el tipo de las variables, un identificador y un tamaño entre corchetes. Por ejemplo:

```
int a[5];
```

La declaración anterior genera cinco variables enteras con nombres `a[0]`, `a[1]`, `a[2]`, `a[3]` y `a[4]`. Es decir, todas ellas son del mismo tipo, reciben el mismo prefijo que se usó en la declaración y están numeradas desde 0 hasta $n-1$, donde $n$ es el tamaño de arreglo. Esto no es configurable y el programador simplemente debe aceptar que la numeración comienza a partir de 0 y no a partir de 1. Al número usado para acceder a una variable específica se le llama posición o índice; el uso de corchetes para especificar la posición también es obligatorio. Fuera de eso, las variables de un arreglo se comportan normalmente:

| Código | Salida |
|---|---|
| `int a[3];`<br>`a[0] = 5;`<br>`a[1] = 4;`<br>`a[2] = a[0] + a[1];`<br>`printf("%d %d %d", a[0], a[1], a[2]);` | `5 4 9` |

Las variables de un arreglo se pueden leer normalmente con `scanf` y también pueden inicializarse al momento de declarar el arreglo, especificando sus valores entre llaves. Si el arreglo es más grande que la cantidad de elementos listados entre llaves, el resto de los elementos recibe el valor 0 del tipo. El tamaño de un arreglo se puede omitir cuando se inicializa y se infiere de la cantidad de elementos de la lista:

| Código | Entrada | Salida |
|---|---|---|
| `int a[4] = { 5, 2 };`<br>`scanf("%d", &arr[3]);`<br>`printf("%d %d ", a[0], a[1]);`<br>`printf("%d %d ", a[2], a[3]);`<br>`int b[] = { 1, 2, 3 };` | `8` | `5 2 0 8` |

La característica más importante de arreglos es que no es necesario escribir literalmente el número correspondiente a la posición del elemento, sino que se puede usar el valor almacenado en una variable:

| Código | Salida |
|---|---|
| `int a[3], i;`<br>`i = 0;`<br>`a[i] = 5;  // a[0] = 5;`<br>`i = 1;`<br>`a[i] = 7;  // a[1] = 7;`<br>`i = 2;`<br>`a[i] = 9;  // a[2] = 9;`<br>`printf("%d %d %d", a[0], a[1], a[2]);` | `5 7 9` |

Lo anterior nos permite leer e imprimir arreglos usando ciclos, haciendo uso de una variable auxiliar (la `i` del ejemplo anterior) que visite todas las posiciones válidas del arreglo:

| Código | Entrada | Salida |
|---|---|---|
| `int a[3];`<br>`for (int i = 0; i < 3; ++i) {`<br>`  scanf("%d", &a[i]);`<br>`}`<br>`for (int i = 3 - 1; i >= 0; --i) {`<br>`  printf("%d ", a[i]);`<br>`}` | `5 7 9` | `9 7 5` |

Lo usual al visitar los elementos de un arreglo de izquierda a derecha es poner la condición `i < n` donde `n` es el tamaño del arreglo, ya que la última posición válida es la `n - 1` y la posición `n` no existe. Por supuesto, también es válido escribir `i <= n - 1`, pero esto es más largo de escribir. Si queremos visitar los elementos de un arreglo de derecha a izquierda, debemos tener cuidado en lo mismo: la posición válida más a la derecha es la `n - 1` y no la `n`; en este caso, la variable del ciclo `for` se decrementa en lugar de incrementarse. Es un grave error intentar acceder a elementos que no existen en un arreglo.

Desafortunadamente, el lenguaje C no permite copiar los elementos de un arreglo directamente a otro arreglo mediante una asignación, aunque sea del mismo tamaño:

| Código | Diagnóstico |
|---|---|
| `int a[3] = { 5, 5, 5 };`<br>`int b[3] = a;`<br>`printf("%d %d %d", b[0], b[1], b[2]);` | `error: invalid initializer`<br>`  int b[3] = a;`<br>`  ^` |

La copia de arreglos se suele hacer elemento a elemento con un ciclo `for`. También se puede declarar más de un arreglo en la misma línea:

| Código | Salida |
|---|---|
| `int a[3] = { 5, 5, 5 }, b[3];`<br>`for (int i = 0; i < 3; ++i) {`<br>`  b[i] = a[i];`<br>`}`<br>`printf("%d %d %d", b[0], b[1], b[2]);` | `5 5 5` |

El lenguaje C (aunque no el lenguaje C++) permite declarar arreglos de tamaño variable. Es decir, podemos declarar un arreglo cuyo tamaño esté dado por el valor guardado en otra variable. Por ejemplo:

| Código | Entrada | Salida |
|---|---|---|
| `int n;`<br>`scanf("%d", &n);`<br><br>`int a[n];`<br>`for (int i = 0; i < n; ++i) {`<br>`  scanf("%d", &a[i]);`<br>`}`<br><br>`for (int i = 0; i < n; ++i) {`<br>`  printf("%d ", a[i]);`<br>`}` | `3`<br>`1 2 3` | `1 2 3` |

Es vital que el tamaño del arreglo ya esté calculado para el momento en el que ocurre la declaración de un arreglo. De otro modo, puede ocurrir un error fatal al ejecutar el programa:

| Código | Diagnóstico |
|---|---|
| `int n, a[n];  // ¡no!`<br>`scanf("%d", &n);  // demasiado tarde` | `Error en ejecución` |

Se pueden declarar funciones que tomen arreglos como parámetros, pero desafortunadamente el lenguaje C es sumamente inconsistente en este aspecto. En primer lugar, es innecesario especificar el tamaño del arreglo en el parámetro de la funcion; en segundo lugar, aún cuando el tamaño se especifique, éste es ignorado en general; en tercer lugar, un arreglo no se pasa por valor sino que se pasa por referencia, por lo que es posible modificar su contenido desde otra función:

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`void llena(int a[], int n, int v) {`<br>`  for (int i = 0; i < n; ++i) {`<br>`  a[i] = v;`<br>`  }`<br>`}`<br><br>`int main( ) {`<br>`  int a[2];`<br>`  llena(a, 2, -1);`<br>`  printf("%d %d", a[0], a[1]);`<br>`  return 0;`<br>`}` | `-1 -1` |

Normalmente, una función que tome un arreglo también toma la cantidad de elementos que la función debe procesar. A continuación se presentan algunos algoritmos fundamentales de arreglos:

- Inversión de un arreglo: consiste en colocar los elementos de un arreglo en el orden contrario al que aparecían originalmente. Intercambiaremos los valores extremos del arreglo y avanzaremos hacia el centro, repitiendo el proceso. El ciclo termina cuando ya intercambiamos ambas mitades del arreglo.

| Código | Entrada | Salida |
|---|---|---|
| `void invierte(int a[], int n) {`<br>`  for (int i = 0; i < n / 2; ++i) {`<br>`  intercambia(&a[i], &a[n - 1 - i]);`<br>`  }`<br>`}`<br>`// leer un arreglo e invocar la función` | `4`<br>`1 2 3 4` | `4 3 2 1` |

- Búsqueda del mínimo (o máximo) de un arreglo: consiste en buscar la posición y el valor del menor (o el mayor) elemento de un arreglo. Esto lo podemos hacer memorizando la posición del elemento que es el mejor hasta ahora. Si encontramos un mejor elemento, sobreescribimos la posición.

| Código | Entrada | Salida |
|---|---|---|
| `int minimo(int a[], int n) {`<br>`  int pos = 0;`<br>`  for (int i = 1; i < n; ++i) {`<br>`  if (a[i] < a[pos]) {`<br>`  pos = i;`<br>`  }`<br>`  }`<br>`  return pos;`<br>`}`<br>`// leer un arreglo e invocar la función` | `4`<br>`7 4 9 6` | `1 4` |

- Búsqueda lineal en un arreglo: consiste en buscar la posición de un valor, usualmente de izquierda a derecha. En cuanto lo encontremos, podemos dejar de buscar. Usaremos una posición inválida (por ejemplo, -1) para denotar que el valor buscado no lo encontramos.

| Código | Entrada | Salida |
|---|---|---|
| `int buscar(int a[], int n, int v) {`<br>`  for (int i = 0; i < n; ++i) {`<br>`  if (a[i] == v) {`<br>`  return i;`<br>`  }`<br>`  }`<br>`  return -1;`<br>`}`<br>`// leer un arreglo e invocar la función` | 4<br>7 4 9 6<br>9 | 2 |

- Ordenamiento de un arreglo: en realidad, existen decenas de algoritmos para ordenar arreglos. Un algoritmo que se enseña con frecuencia en cursos introductorios es el ordenamiento por burbuja. Este algoritmo consiste en arrastrar hacia la derecha al máximo elemento del arreglo mediante intercambios; cuando éste ya está en su posición final, repetimos el proceso con el resto de los elementos.

| Código | Entrada | Salida |
|---|---|---|
| `void intercambia(int* a, int* b) {`<br>`  int c = *a;`<br>`  *a = *b;`<br>`  *b = c;`<br>`}`<br><br>`void ordena2(int* a, int* b) {`<br>`  if (*a > *b) {`<br>`  intercambia(a, b);`<br>`  }`<br>`}`<br><br>`void burbuja(int a[], int n) {`<br>`  for (int k = 0; k < n - 1; ++k) {`<br>`  for (int i = 0; i < n - 1; ++i) {`<br>`  ordena2(&a[i], &a[i + 1]);`<br>`  }`<br>`  }`<br>`}`<br>`// leer un arreglo e invocar la función` | 4<br>7 4 9 6 | 4 6 7 9 |

## 14.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Ignorando-los-primeros-elementos`
2. Resuelve el problema `https://omegaup.com/arena/problem/Ignorando-los-ultimos-elementos`
3. Resuelve el problema `https://omegaup.com/arena/problem/Reverso`
4. Resuelve el problema `https://omegaup.com/arena/problem/Conjunto-Capicua`
5. Resuelve el problema `https://omegaup.com/arena/problem/Suma-de-vectores`
6. Resuelve el problema `https://omegaup.com/arena/problem/Buscar-y-contar`
7. Resuelve el problema `https://omegaup.com/arena/problem/Ordena-Basico-1`

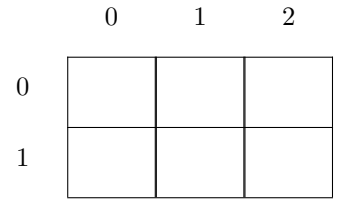
8. Resuelve el problema `https://omegaup.com/arena/problem/Modificando-un-arreglo`
9. Resuelve el problema `https://omegaup.com/arena/problem/Alicia-y-el-cachorro-saltarin`
10. Resuelve el problema `https://omegaup.com/arena/problem/Saltando-por-el-arreglo`
11. Resuelve el problema `https://omegaup.com/arena/problem/Omitiendo-el-entero-mas-grande`

# 15. Matrices

En programación, el término matriz se usa para denotar un arreglo multidimensional. La declaración de una matriz difiere con la de un arreglo simplemente en que la de una matriz especifica una secuencia de más de un tamaño. Por ejemplo, la siguiende declaración:

```
int a[2][3];
```

genera seis variables enteras con nombres `a[0][0]`, `a[0][1]`, `a[0][2]`, `a[1][0]`, `a[1][1]` y `a[1][2]`. Si $n_d$ es el tamaño de la $d$-ésima dimensión, los índices válidos en esa dimensión van de 0 a $n_d - 1$. Cuando una matriz es de dos dimensiones, suele visualizarse en términos de filas y columnas:




Visualización de una matriz de dos filas y tres columnas.

Una matriz también se puede inicializar con la notación de llaves, anidando éstas para que correspondan con la estructura de la matriz, de la primera a la última dimensión. Para visitar por filas los elementos de una matriz de dos dimensiones, geneneralmente se usan dos ciclos `for` anidados: uno selecciona la fila y el otro visita todas las columnas de esa fila. Para procesar la matriz por columnas, basta intercambiar los ciclos para seleccionar primero la columna y luego visitar todas las filas de esa columna.

**Código**

```
int a[2][3] = {
   { 1, 2, 3 },
   { 4, 5, 6 }
};

for (int i = 0; i < 2; ++i) {
   for (int j = 0; j < 3; ++j) {
      printf("%d ", a[i][j]);
   }
   printf("\n");
}

printf("\n");

for (int j = 0; j < 3; ++j) {
   for (int i = 0; i < 2; ++i) {
      printf("%d ", a[i][j]);
   }
   printf("\n");
}
```

**Salida**

```
1 2 3
4 5 6

1 4
2 5
3 6
```

Una matriz también se puede usar como parámetro de una función (y se pasa por referencia como los arreglos), pero es obligatorio especificar las dimensiones de todas las dimensiones excepto la primera, que siempre es opcional. La razón es que una matriz se almacena internamente de forma lineal como los arreglos, por lo que el lenguaje necesita conocer el tamaño de (casi) cada dimensión para poder simular el acceso multidimensional (en el caso de una matriz de dos dimensiones, necesita saber el ancho de cada fila para poder distinguir una de la otra si en realidad se almacenan de forma contigüa en memoria):

**Código**

```
void ceros(int a[][], int n, int m) {
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
         a[i][j] = 0;
      }
   }
}

int main( ) {
   int a[2][3];
   ceros(a, 2, 3);
   // imprimir la matriz
}
```

**Diagnóstico**

```
error: array has incomplete type
 void llena(int a[][], int n, int m) {
                  ^
error: type of parameter 1 is incomplete
    ceros(a, 2, 3);
          ^
```

**Código**

```
void ceros(int a[][3], int n, int m) {
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
         a[i][j] = 0;
      }
   }
}

int main( ) {
   int a[2][3];
   ceros(a, 2, 3);
   // imprimir la matriz
}
```

**Salida**

```
0 0 0
0 0 0
```

En el caso de matrices de tamaño variable, los parámetros de la función que denoten las dimensiones de la matriz pueden usarse en la declaración de la matríz parámetro:

**Código**

```
void ceros(int n, int m, int a[n][m]) {
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
         a[i][j] = 0;
      }
   }
}

int main( ) {
   int n, m;
   scanf("%d%d", &n, &m);
   int a[n][m];
   ceros(n, m, a);
   // imprimir la matriz
}
```

**Entrada**

```
2 3
```

**Salida**

```
0 0 0
0 0 0
```

### 15.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/sumar-matrices`
2. Resuelve el problema `https://omegaup.com/arena/problem/Demostrando-con-matrices`
3. Resuelve el problema `https://omegaup.com/arena/problem/Producto-de-matrices`
4. Resuelve el problema `https://omegaup.com/arena/problem/Matrices-giradas`
5. Resuelve el problema `https://omegaup.com/arena/problem/Los-cuadrados-semimagicos`
6. Resuelve el problema `https://omegaup.com/arena/problem/med`
7. Resuelve el problema `https://omegaup.com/arena/problem/svc`

## 16. Caracteres y cadenas

Como se mencionó en la sección 5 de las notas, el tipo `char` es un tipo entero que tiene una capacidad limitada. El lenguaje C sólo garantiza que este tipo pueda almacenar enteros en el rango de 0 a 127, aunque la mayoría de los compiladores lo definen en el rango de -128 a 127. Una variable de tipo `char` se puede leer e imprimir como entero con el especificador de formato `%hhd`:

| Código | Entrada | Salida |
|---|---|---|
| `char a = 5;`<br>`char b;`<br>`scanf("%hhd", &b);`<br>`printf("%hhd %hhd", a, b);` | `7` | `5 7` |

El tipo `char` consume menos memoria que un `int` y puede ser buena idea usarlo si los enteros a almacenar tienen un rango limitado y fuera importante ahorrar memoria. Por otra parte, el tipo `char` se usa principalmente para denotar símbolos y no enteros. La denotación se da de forma transparente desde el punto de vista del lenguaje y de la biblioteca. El lenguaje C requiere que el compilador elija un conjunto de caracteres, el cual establece un mapeo entre símbolos y enteros. Casi todos los compiladores eligen el el conjunto de caracteres llamado ASCII. Cuando el programador escribe un símbolo entre comillas sencillas (llamado literal de caracter), el compilador transforma dicha literal en su entero asociado.

| Código | Salida |
|---|---|
| `char c = '@';`<br>`printf("%hhd", c);` | `64` |

La literal de caracter de comilla sencilla se escribe `'\''` y la literal de cadena de diagonal invertida se escribe `'\\'`. Las funciones `scanf` y `print` también conocen el conjunto de caracteres elegido por el compilador y proveen el modificador de formato `%c`. Al leer un `char` con `%c`, instruimos a `scanf` que lea un símbolo, encuentre su entero asociado y lo guarde en la variable. Al imprimir un `char` con `%c`, instruimos a `printf` que tome el valor entero del `char`, encuentre su símbolo asociado y lo imprima.

| Código | Entrada | Salida |
|---|---|---|
| `char a;`<br>`scanf("%c", &a);`<br>`printf("%hhd\n", a);`<br><br>`char b = 64;`<br>`printf("%c\n", b);` | `#` | `35`<br>`@` |

Es importante recalcar que el entero asociado a un símbolo generalmente no tiene nada que ver con el símbolo en sí. Por ejemplo, el símbolo `'8'` que usamos para denotar el número ocho tiene asociado el entero 56 en la tabla ASCII. A continuación se muestra dicha tabla:

| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 |   | Space | 64 | 40 | 100 | @ | @ | 96 | 60 | 140 | ` | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | ! | ! | 65 | 41 | 101 | A | A | 97 | 61 | 141 | a | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | " | " | 66 | 42 | 102 | B | B | 98 | 62 | 142 | b | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | # | # | 67 | 43 | 103 | C | C | 99 | 63 | 143 | c | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | $ | $ | 68 | 44 | 104 | D | D | 100 | 64 | 144 | d | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | % | % | 69 | 45 | 105 | E | E | 101 | 65 | 145 | e | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | & | & | 70 | 46 | 106 | F | F | 102 | 66 | 146 | f | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | ' | ' | 71 | 47 | 107 | G | G | 103 | 67 | 147 | g | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | ( | ( | 72 | 48 | 110 | H | H | 104 | 68 | 150 | h | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 | ) | ) | 73 | 49 | 111 | I | I | 105 | 69 | 151 | i | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | * | * | 74 | 4A | 112 | J | J | 106 | 6A | 152 | j | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | + | + | 75 | 4B | 113 | K | K | 107 | 6B | 153 | k | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | , | , | 76 | 4C | 114 | L | L | 108 | 6C | 154 | l | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | - | - | 77 | 4D | 115 | M | M | 109 | 6D | 155 | m | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | . | . | 78 | 4E | 116 | N | N | 110 | 6E | 156 | n | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | / | / | 79 | 4F | 117 | O | O | 111 | 6F | 157 | o | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | 0 | 0 | 80 | 50 | 120 | P | P | 112 | 70 | 160 | p | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | 1 | 1 | 81 | 51 | 121 | Q | Q | 113 | 71 | 161 | q | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | 2 | 2 | 82 | 52 | 122 | R | R | 114 | 72 | 162 | r | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | 3 | 3 | 83 | 53 | 123 | S | S | 115 | 73 | 163 | s | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | 4 | 4 | 84 | 54 | 124 | T | T | 116 | 74 | 164 | t | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | 5 | 5 | 85 | 55 | 125 | U | U | 117 | 75 | 165 | u | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | 6 | 6 | 86 | 56 | 126 | V | V | 118 | 76 | 166 | v | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | 7 | 7 | 87 | 57 | 127 | W | W | 119 | 77 | 167 | w | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | 8 | 8 | 88 | 58 | 130 | X | X | 120 | 78 | 170 | x | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | 9 | 9 | 89 | 59 | 131 | Y | Y | 121 | 79 | 171 | y | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | : | : | 90 | 5A | 132 | Z | Z | 122 | 7A | 172 | z | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | ; | ; | 91 | 5B | 133 | [ | [ | 123 | 7B | 173 | { | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | < | < | 92 | 5C | 134 | \ | \ | 124 | 7C | 174 | | | \| |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | = | = | 93 | 5D | 135 | ] | ] | 125 | 7D | 175 | } | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | > | > | 94 | 5E | 136 | ^ | ^ | 126 | 7E | 176 | ~ | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | ? | ? | 95 | 5F | 137 | _ | _ | 127 | 7F | 177 | &#127; | DEL |

La tabla ASCII.

La biblioteca estándar de C provee de varias funciones de clasificación y transformación de caracteres. Estas funciones están en el archivo `ctype.h` e irónicamente toman parámetros `int` y no `char`, aunque su uso es el mismo. A continuación se listan las funciones más importantes de `ctype.h`.

| **Funciones de `ctype.h`** | |
|---|---|
| **Función de biblioteca** | **Valor calculado** |
| `int tolower(int c);` | Si `c` denota una letra, regresa el `char` que denota su versión minúscula. En otro caso, regresa `c` mismo. |
| `int toupper(int c);` | Si `c` denota una letra, regresa el `char` que denota su versión mayúscula. En otro caso, regresa `c` mismo. |
| `int isupper(int c);` | Regresa un valor distinto de 0 si `c` denota una letra mayúscula. Regresa 0 en otro caso. |
| `int islower(int c);` | Regresa un valor distinto de 0 si `c` denota una letra minúscula. Regresa 0 en otro caso. |
| `int isalpha(int c);` | Regresa un valor distinto de 0 si `c` denota una letra. Regresa 0 en otro caso. |
| `int isdigit(int c);` | Regresa un valor distinto de 0 si `c` denota un dígito. Regresa 0 en otro caso. |
| `int isspace(int c);` | Regresa un valor distinto de 0 si `c` denota un espacio. Regresa 0 en otro caso. |
| `int isalnum(int c);` | Regresa un valor distinto de 0 si `c` denota una letra o un dígito. Regresa 0 en otro caso. |

Entre los caracteres que se consideran espacios se encuentran el espacio `' '`, el retorno de carro `'\r'`, el salto de línea `'\n'` y el tabulador horizontal `'\t'`. Las funciones de clasificación pueden regresar un valor distinto de 1 y de 0 (por ejemplo, un 8) en el caso verdadero. En este caso, conviene recordar que cualquier valor distinto de 0 se considera verdadero dentro de un if.

| Código | Entrada | Salida |
|---|---|---|
| `#include <ctype.h>`<br>`#include <stdio.h>`<br><br>`void analiza(char a) {`<br>`  printf("'%c' vale %hhd\n", a, a);`<br>`  if (isalnum(a)) {`<br>`  printf("Es letra o dígito\n");`<br>`  if (isalpha(a)) {`<br>`  printf("Es sólo letra\n");`<br>`  if (islower(a)) {`<br>`  printf("Es minúscula\n");`<br>`  } else if (isupper(a)) {`<br>`  printf("Es mayúscula");`<br>`  }`<br>`  printf("%c\n", tolower(a));`<br>`  printf("%c\n", toupper(a));`<br>`  } else if (isdigit(a)) {`<br>`  printf("Es sólo dígito\n");`<br>`  }`<br>`  } else if (isspace(a)) {`<br>`  printf("Es espacio\n");`<br>`  }`<br>`  printf("\n");`<br>`}`<br><br>`int main( ) {`<br>`  char a1, a2, a3;`<br>`  scanf("%c%c%c", &a1, &a2, &a3);`<br><br>`  analiza(a1);`<br>`  analiza(a2);`<br>`  analiza(a3);`<br>`  return 0;`<br>`}` | `a` | `'a' vale 97`<br>`Es letra o dígito`<br>`Es sólo letra`<br>`Es minúscula`<br>`a`<br>`A`<br><br>`' ' vale 32`<br>`Es espacio`<br><br>`'5' vale 53`<br>`Es letra o dígito`<br>`Es sólo dígito` |

En el lenguaje C, una cadena es una secuencia de caracteres almacenada en un arreglo , la cual siempre termina en el caracter nulo. El caracter nulo tiene asociado el entero 0 en la tabla ASCII y su literal de caracter se escribe `'\0'`. Una cadena sobre un arreglo de caracteres se puede inicializar con una literal de cadena, pero siempre se debe apartar capacidad suficiente para el caracter nulo.

**Código**

```
char a[5] = { 'h', 'o', 'l', 'a', '\0' };  // bien
char b[5] = "hola";                        // equivalente, el nulo es implícito
char c[] = "hola";                         // equivalente, la capacidad es de 5
char d[4] = { 'h', 'o', 'l', 'a' };        // es un arreglo, pero no una cadena
char e[4] = "hola";                        // ¡mal!
```

Una cadena se puede leer con `scanf` usando el especificador de formato `%s` y se puede imprimir con `printf` usando el mismo especificador. En general, si esperamos que el usuario ingrese una cadena de longitud máxima `n` entonces debemos apartar un arreglo con capacidad `n + 1`, ya que `scanf` almacena también el nulo para marcar el fin de cadena. La función `printf` usará ese mismo nulo para saber en dónde termina la cadena y hasta dónde imprimir. Como un arreglo siempre se pasa por referencia, el operador `&` no se debe usar durante la lectura. El especificador `%s` lee secuencias de caracteres que no incluyan espacios, por lo que la lectura se detendrá en el primer espacio (aunque primero se ignoran los espacios que puedan existir al inicio de la entrada).

| Código | Entrada | Salida |
|---|---|---|
| `char a[6 + 1], b[6 + 1];`<br>`scanf("%s", a);`<br>`printf("%s\n", a);`<br>`scanf("%s", b);  // error en ejecución`<br>`printf("%s\n", b);` | `  bien`<br>`problemas` | `bien` |

La longitud de una cadena se define como la cantidad de caracteres almacenados antes del caracter nulo (por lo cual es diferente a la capacidad del arreglo). La longitud de una cadena se puede calcular con la función `strlen` que está declarada en el archivo `string.h` de la biblioteca estándar de C.

| Código | Entrada | Salida |
|---|---|---|
| `#include <stdio.h>`<br>`#include <string.h>`<br><br>`int main( ) {`<br>`  char a[10 + 1];`<br>`  scanf("%s", a);`<br>`  int t = strlen(a);`<br>`  printf("%d %s", t, a);`<br>`  return 0;`<br>`}` | `hola` | `4 hola` |

Para evitar que la lectura de una cadena consuma más de lo que estamos preparados para almacenar, podemos usar `%`*N*`s` donde *N* es la longitud de la cadena más grande que podemos leer. Por ejemplo:

| Código | Entrada | Salida |
|---|---|---|
| `char a[5 + 1];`<br>`scanf("%5s", a);`<br>`int t = strlen(a);`<br>`printf("%d %s", t, a);` | `holaholahola` | `5 hola` |

Como se dijo anteriormente, `%s` ignora espacios y luego consume una secuencia de caracteres que no incluye espacios. El especificador `%[^\n]` permite leer una línea, que es una secuencia de caracteres con espacios incluidos cuyo proceso de lectura se detiene en un salto de línea. Por ejemplo:

| Código | Entrada | Salida |
|---|---|---|
| `char a[10 + 1];`<br>`scanf("%[^\n]", a);`<br>`int t = strlen(a);`<br>`printf("%d %s", t, a);` | `hola hola` | `10 hola hola` |

Desafortunadamente, la lectura de dos líneas consecutivas fallará porque el salto de línea `\n` se queda obstruyendo el proceso de lectura de la segunda línea.

| Código | Entrada | Salida |
|---|---|---|
| `char a[10 + 1], b[10 + 1];`<br>`scanf("%[^\n]%[^\n]", a, b);`<br>`printf("%s\n%s\n", a, b);` | `hola hola`<br>`adios adios` | `hola hola` |

Una forma de hacer que `scanf` extraiga un caracter que nos estorbe en la entrada es especificar dicho caracter en la cadena de formato:

| Código | Entrada | Salida |
|---|---|---|
| `char a[10 + 1], b[10 + 1];`<br>`scanf("%[^\n]\n%[^\n]", a, b);`<br>`printf("%s\n%s\n", a, b);` | `hola hola`<br>`adios adios` | `hola hola`<br>`adios adios` |

Esta característica es bastante útil en general. Por ejemplo, podemos leer dos enteros que estén separados por comas y no por espacios de la siguiente forma:

| Código | Entrada | Salida |
|---|---|---|
| `int a, b;`<br>`scanf("%d,%d", &a, &b);`<br>`printf("%d %d", a, b);` | `5,7` | `5 7` |

El archivo de la biblioteca estándar `string.h` provee de varias funciones de manipulación de cadenas, además de `strlen`. A continuación se listan las funciones más importantes:

| **Funciones de `string.h`** | |
|---|---|
| **Función de biblioteca** | **Trabajo realizado** |
| `size_t strlen(char a[]);` | Calcula y regresa la longitud de la cadena `a`. El tipo `size_t` generalmente es un sinónimo de `unsigned long long`. |
| `char* strcpy(char a[], const char b[]);` | Copia la cadena `b` en el arreglo `a`, sobreescribiendo su contenido. Regresa `a`. |
| `char* strcat(char a[], const char b[]);` | Concatena la cadena `b` al final de la cadena almacenada en `a`. Regresa `a`. |
| `int strcmp(const char a[], const char b[]);` | Si `a` y `b` guardan la misma cadena, regresa `0`. En otro caso, regresa un número negativo si `a` aparecería antes que `b` en un diccionario (orden lexicográfico) y regresa un número positivo si `a` aparecería después que `b`. |

Todas las funciones anteriores usan el caracter nulo para determinar el fin de una cadena. Tanto `strcpy` como `strcat` necesitan que el programador haya apartado un arreglo de suficiente capacidad para almacenar la cadena copiada. Por ejemplo:

| Código | Salida |
|---|---|
| `char a[50 + 1] = "hola";`<br>`char b[50 + 1];`<br>`strcpy(b, a);`<br>`char c[50 + 1] = ""; // poner el nulo`<br>`strcat(c, b);`<br>`strcat(c, "gatito");`<br>`printf("%s", c);` | `holagatito` |

| Código | Salida |
|---|---|
| `printf("%d\n", strcmp("abc", "abc"));`<br>`printf("%d\n", strcmp("abc", "xyz"));`<br>`printf("%d\n", strcmp("xyz", "abc"));` | `0`<br>`-1`<br>`1` |

No apartar capacidad suficiente para las cadenas o para el caracter nulo es uno de los errores más comunes en el lenguaje C. Este error es una de las principales fuentes de vulnerabilidades en los programas en C y su explotación permite la producción de virus de computadora y otro tipo de software malicioso.

## 16.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Consonantes-y-vocales`
2. Resuelve el problema `https://omegaup.com/arena/problem/El-caballo-de-John-Carter`
3. Resuelve el problema `https://omegaup.com/arena/problem/Invirtiendo-palabras`
4. Resuelve el problema `https://omegaup.com/arena/problem/Maquina-descompuesta`
5. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-las-letras-de-la-linea`

# 17. Definición de tipos y estructuras

El lenguaje C permite definir sinónimos de tipos de datos preexistentes, así como definir nuevos tipos de datos. La palabra reservada `typedef` nos permite realizar justo lo primero:

**Código**

```
typedef int entero;             // el tipo int ahora también se llama entero
entero a = 0;                   // int a = 0;
entero arr[3] = { 1, 2, 3 };    // int arr[3] = { 1, 2, 3 };
int b = a;                      // bien
typedef entero entero2;         // sinónimo de un sinónimo, válido pero no común
entero2 c = 0;                  // int c = 0;
```

Por su parte, la palabra reservada `struct` nos permite definir un nuevo tipo de dato, el cual es un tipo de dato compuesto (también llamado estructura). Por ejemplo, la definición:

```
struct fecha {
   int dia, mes, anyo;
};
```

permite declarar variables de tipo `fecha` que tienen a su vez subvariables `.dia`, `.mes` y `.anyo`. Por ejemplo:

| Código | Entrada | Salida |
|---|---|---|
| `#include <stdio.h>`<br><br>`struct fecha {`<br>`  int dia, mes, anyo;`<br>`};`<br><br>`int main( ) {`<br>`  struct fecha f;`<br>`  f.dia = 1;`<br>`  f.mes = 5;`<br>`  scanf("%d", &f.anyo);`<br>`  printf("%d %d %d", f.dia, f.mes,`<br>`  f.anyo);`<br>`  return 0;`<br>`}` | 2010 | 1<br>5<br>2010 |

Desafortunadamente, es necesario escribir la palabra `struct` cada vez que se declare una variable usando el nombre original de la estructura. Irónicamente, declarar un sinónimo de la estructura nos permite escapar de esta obligación:

**Código**

```
struct fecha_impl {
   int dia, mes, anyo;
};
typedef struct fecha_impl fecha;   // el tipo struct fecha_impl ahora
                                   // también se llama fecha
int main( ) {
   fecha f;                        // ¡ok!
   return 0;
}
```

El lenguaje C permite colapsar ambas declaraciones de la siguiente forma:

**Código**

```
typedef struct {
   int dia, mes, anyo;
} fecha;                    // el struct anónimo también se llama fecha

int main( ) {
   fecha f;
   return 0;
}
```

Una estructura se puede inicializar con llaves y los valores especificados se asignan a los miembros de la estructura en el orden en el que fueron declarados. Usando la notación de llaves, los miembros que no reciban un valor se inicializan con 0 (en contraste, no inicializar la estructura de ninguna forma dejaría indefinidos los valores de las subvariables). El operador = permite copiar una estructura .

| Código | Salida |
|---|---|
| `fecha f1 = { 1, 2 };`<br>`fecha f2 = f1;`<br>`fecha f3;`<br>`printf("%d %d %d\n", f1.dia, f1.mes,`<br>`f1.anyo);`<br>`printf("%d %d %d\n", f2.dia, f2.mes,`<br>`f2.anyo);`<br>`printf("%d %d %d\n", f3.dia, f3.mes,`<br>`f3.anyo);` | `1 2 0`<br>`1 2 0`<br>`-563 2356646 -17` |

Una función puede tomar estructuras como parámetros y también puede regresar estructuras. Las estructuras se pueden pasar tanto por valor como por referencia:

| Código | Salida |
|---|---|
| `#include <stdio.h>`<br><br>`typedef struct {`<br>`int dia, mes, anyo;`<br>`} fecha;`<br><br>`fecha f(fecha f1, fecha* p2) {`<br>`f1.anyo += 1;`<br>`(*p2).anyo -= 1;`<br>`return f1;`<br>`}`<br><br>`int main( ) {`<br>`fecha f1 = { 1, 1, 2000 };`<br>`fecha f2 = { 2, 2, 2000 };`<br>`fecha f3 = f(f1, &f2);`<br>`printf("%d %d %d\n", f1.dia, f1.mes,`<br>`f1.anyo);`<br>`printf("%d %d %d\n", f2.dia, f2.mes,`<br>`f2.anyo);`<br>`printf("%d %d %d\n", f3.dia, f3.mes,`<br>`f3.anyo);`<br>`return 0;`<br>`}` | `1 1 2000`<br>`2 2 1999`<br>`1 1 2001` |

Si p es un apuntador a una estructura, entonces `(*p).a` también se puede escribir como `p->a`. Ambas notaciones son equivalentes, pero la segunda fue introducida porque tener que escribir los paréntesis de la primera le resultaba fastidioso al autor del lenguaje.

### 17.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/El-lado-mas-corto`
2. Resuelve el problema `https://omegaup.com/arena/problem/Consultando-Registros`

## 18. Flujos de entrada/salida y archivos

En todos los programas hechos hasta el momento, hemos supuesto que la entrada es correcta y que conocemos el tamaño de la misma. Sin embargo, esto rara vez es cierto en la vida real. El valor de retorno de `scanf` nos permite conocer cuántos de los especificadores de formato de lectura tuvieron éxito.

| Código | Entrada | Salida |
|---|---|---|
| `int a, b, x, y;`<br>`int r1 = scanf("%d%d", &a, &b);`<br>`int r2 = scanf("%d%d", &x, &z);`<br>`printf("(%d): %d %d\n", r1, a, b);`<br>`printf("(%d): %d %d\n", r2, x, y);` | `5 5`<br>`5 gatito` | `(2): 5 5`<br>`(1): 5 22092` |

La función `scanf` se detiene al primer error que encuentre. Por otra parte, cuando `scanf` falla porque no hay qué leer, ésta regresa un valor que casi siempre vale -1 pero que formalmente se llama `EOF`.

| Código | Entrada | Salida |
|---|---|---|
| `int a, b;`<br>`int r = scanf("%d%d", &a, &b);`<br>`printf("%d", r);` | `gatito 5` | `0` |

| Código | Entrada | Salida |
|---|---|---|
| `int a, b;`<br>`int r = scanf("%d%d", &a, &b);`<br>`printf("%d", r);` | | `-1` |

La característica anterior nos permite procesar una cantidad desconocida de datos, preguntando en un ciclo si `scanf` pudo leer los datos que le pedimos:

| Código | Entrada | Salida |
|---|---|---|
| `int n;`<br>`while (scanf("%d", &n) == 1) {`<br>`  printf("Leímos %d\n", n);`<br>`}` | `1`<br>`2`<br>`3` | `Leímos 1`<br>`Leímos 2`<br>`Leímos 3` |

La función `scanf` lee datos desde lo que se denomina la *entrada estándar* mientras que la función `printf` imprime en lo que se denomina la *salida estándar*. En C, la forma de referirnos explícitamente a la entrada y la salida estándares es haciendo uso de las variables `stdin` y `stdout`, respectivamente. Ambas están disponibles en `stdio.h`.

Las variables `stdin` y `stdout` son de tipo `FILE*` y `stdio.h` provee de funciones que toman variables de este tipo como parámetros. Las funciones `fscanf` y `fprintf` son generalizaciones de `scanf` y `printf` que permiten volver explícito (en su primer parámetro) el origen y el destino de los datos.

| Código | Entrada | Salida |
|---|---|---|
| `int n;`<br>`fscanf(stdin, "%d", &n);`<br>`fprintf(stdout, "%d", n);` | `5` | `5` |

Por supuesto, es un error intentar leer desde `stdout` o intentar imprimir en `stdin`. Adicionalmente, `stdio.h` provee la función `fopen` que permite crear un `FILE*` que esté vinculado a un archivo de entrada o de salida. El primer parámetro de `fopen` es una cadena con el nombre del archivo a usar y el segundo parámetro es una cadena que indica el modo de apertura (por ejemplo, `"r"` para abrir un archivo en modo lectura y `"w"` para abrir un archivo en modo escritura).

| Código | entrada.txt | salida.txt |
|---|---|---|
| `FILE* ent = fopen("entrada.txt", "r");`<br>`FILE* sal = fopen("salida.txt", "w");`<br>`int n;`<br>`fscanf(ent, "%d", &n);`<br>`fprintf(sal, "%d", n);` | `5` | `5` |

Un archivo de escritura abierto en modo `"w"` se crea automáticamente si éste no existe o se limpia si ya existía, de modo que `"w"` no se debe usar si se desea conservar el contenido previo del archivo. Por otra parte, intentar abrir un archivo inexistente en modo lectura `"r"` es un error y `fopen` regresará el valor especial `NULL` para indicar que ocurrió tal error.

**Código**

```
FILE* ent = fopen("inexistente.txt", "r");
if (ent == NULL) {
   printf("No pudimos abrir el archivo");
} else {
   //... usar el archivo
}
```

Finalmente, la función `fclose` se usa para cerrar un archivo, lo cual indica que ya no haremos uso de él durante el resto del programa. Cerrar un archivo es innecesario si el programa está por terminar, ya que el sistema operativo se encarga de cerrarlo de todos modos. Por otra parte, se recomienda cerrar un archivo si el programa todavía se mantendrá en ejecución durante una cantidad considerable de tiempo, pues el sistema podría limitar qué y cuántos archivos que se pueden abrir al mismo tiempo y sería contraproducente mantener abierto un archivo que ya no usamos. El contenido final de un archivo de salida podría reflejarse hasta que el programador o el sistema operativo cierra el archivo.

**Código**

```
FILE* sal = fopen("salida.txt", "w");
//... usar el archivo
fclose(sal);
```

## 18.1. Ejercicios

1. Resuelve el problema `https://omegaup.com/arena/problem/Anyos-con-dos-y-cuatro-digitos`

# A. Soluciones de ejercicios

En general, siempre hay más de una posible respuesta correcta para cada pregunta. Razonen cuidadosamente las respuestas que propongo para que decidan si las suyas son equivalentes.

## Soluciones de 2.1

1. Escribe la siguiente función definida sobre reales usando la notación extendida.

$$f(x, y, z) = 1 + (x + yz) + (x + yz)^2 - (x + yz)^3 - (x + yz)^5$$

**Solución:**

$$f(\text{real } x, \text{real } y, \text{real } z) \mapsto \text{real } \left\{ \begin{array}{l} \text{real } t = x + yz \\ \text{regresa } 1 + t + t^2 - t^3 - t^5 \end{array} \right\}$$

2. Escribe una función *máximo*$_2$ que regrese el máximo de dos reales. Compárala con la definición de la función *mínimo*$_2$ que se describe previamente en las notas.

**Solución:** En comparación con *mínimo*$_2$, basta cambiar el signo relacional.

$$\textit{máximo}_2(\text{real } a, \text{real } b) \mapsto \text{real } \left\{ \begin{array}{l} \text{si } a > b \text{ entonces} \\ \quad \text{regresa } a \\ \text{sino entonces} \\ \quad \text{regresa } b \end{array} \right\}$$

3. Escribe una función *signo* que tome un real $x$ y regrese -1 si $x$ es negativo, regrese 0 si $x$ es cero y regrese +1 si $x$ es positivo.

**Solución:**

$$\textit{signo}(\text{real } x) \mapsto \text{entero } \left\{ \begin{array}{l} \text{si } x < 0 \text{ entonces} \\ \quad \text{regresa } -1 \\ \text{sino si } x > 0 \text{ entonces} \\ \quad \text{regresa } +1 \\ \text{sino entonces} \\ \quad \text{regresa } 0 \end{array} \right\}$$

4. Escribe una función $g$ que tome un real $x$ y que regrese 1 si el valor de $\sqrt{x}$ es entero y regrese 0 caso contrario. Considera que puedes ayudarte de la función *trunca*.

**Solución:**

$$g(\text{real } x) \mapsto \text{entero } \left\{ \begin{array}{l} \text{si } \textit{trunca}(\sqrt{x}) = \sqrt{x} \text{ entonces} \\ \quad \text{regresa } 1 \\ \text{sino entonces} \\ \quad \text{regresa } 0 \end{array} \right\}$$

5. Escribe una función *mediana*$_3$ que tome tres reales $x, y, z$ y regrese su mediana. La mediana de una secuencia de números es el número que quedaría enmedio si la ordenamos. Por ejemplo, la mediana de (8,1,5) es 5 porque si ordenamos la secuencia queda (1, 5, 8) con el 5 enmedio. Considera que puedes auxiliarte de otras funciones que hayas definido previamente.

**Solución:**

$$\textit{mediana}(\text{real } x, \text{real } y, \text{real } z) \mapsto \text{real} \left\{ \begin{array}{l} \text{si } \textit{mínimo}_2(y,z) \leq x \leq \textit{máximo}_2(y,z) \text{ entonces} \\ \quad \text{regresa } x \\ \text{sino si } \textit{mínimo}_2(x,z) \leq y \leq \textit{máximo}_2(x,z) \text{ entonces} \\ \quad \text{regresa } y \\ \text{sino entonces} \\ \quad \text{regresa } z \end{array} \right\}$$

6. Dada la siguiente definición de $f$, calcula el valor de $f(1,2)$:

$$f(\text{real } x, \text{real } y) \mapsto \text{real} \left\{ \begin{array}{l} \text{real } s = x + y^2 \\ \text{real } t = 2x + \sqrt{y} \\ \text{regresa } s + t \end{array} \right\}$$

**Solución:** La evaluación de la función es directa. Tenemos $s = 1 + 2^2$ y $t = 2(1) + \sqrt{2}$, por lo que $f(1,2) = (1 + 2^2) + (2(1) + \sqrt{2})$ que es aproximadamente 8.4142.

7. Dadas las siguientes definiciones de $f$ y $g$, calcula el valor de $g(1,2)$:

$$f(\text{real } x, \text{real } y) \mapsto \text{real} \left\{ \begin{array}{l} \text{real } t = 2x + y \\ \text{regresa } t + t^2 \end{array} \right\}$$

$$g(\text{real } x, \text{real } y) \mapsto \text{real} \left\{ \begin{array}{l} \text{regresa } f(2y, x) \end{array} \right\}$$

**Solución:** La evaluación de la función es un poco más complicada. Tenemos $g(x,y) = f(2y,x)$ por lo que $g(1,2) = f(2(2),1) = f(4,1)$. Al evaluar $f(4,1)$ tenemos $t = 2(4) + 1 = 9$, por lo que $f(4,1) = g(1,2) = 9 + 9^2 = 90$.

8. Dada la siguiente definición de $f$, calcula el valor de $f(3)$:

$$f(\text{entero } n) \mapsto \text{entero} \left\{ \begin{array}{l} \text{si } n = 0 \text{ entonces} \\ \quad \text{regresa } 0 \\ \text{sino entonces} \\ \quad \text{regresa } n + f(n-1) \end{array} \right\}$$

**Solución:** Podemos evaluar $f(3)$ por etapas:

- $f(3)$ vale $3 + f(2)$
- $f(2)$ vale $2 + f(1)$
- $f(1)$ vale $1 + f(0)$
- $f(0)$ vale 0.

Con esto, podemos calcular los valores finales de cada función de abajo hacia arriba:

- $f(0)$ vale 0.
- $f(1)$ vale $1 + f(0) = 1 + 0 = 1$
- $f(2)$ vale $2 + f(1) = 2 + 1 = 3$
- $f(3)$ vale $3 + f(2) = 3 + 3 = 6$

por lo que $f(3)$ vale 3.

9. Aún no hemos hablado de cómo programar, pero a continuación se muestra la definición de una función $f$ y su implementación en el lenguaje de programación C. Identifica las similitudes y diferencias en la notación empleada. No es necesario que memorices la notación del lenguaje C en este momento, pero sí intenta convencerte que es muy similar a lo que hemos visto.

$$f(\text{real } r, \text{entero } n) \mapsto \text{real} \left\{ \begin{array}{l} \text{si } n \geq 0 \text{ entonces} \\ \quad \text{regresa } r + n \\ \text{sino si } r < 0 \text{ y } n < 0 \text{ entonces} \\ \quad \text{regresa } r - n \\ \text{sino entonces} \\ \quad \text{entero } t = 3n \\ \quad \text{regresa } \frac{r}{t+1} \end{array} \right\}$$

```
float f(float r, int n) {
   if (n >= 0) {
      return r + n;
   } else if (r < 0 && n < 0) {
      return r - n;
   } else {
      int t = 3 * n;
      return r / (t + 1);
   }
}
```

**Solución:** A continuación se realiza una comparación entre ambas notaciones:

- Existe la siguiente equivalencia notacional:

| Notación usada | Lenguaje C |
|---:|---|
| entero | `int` |
| real | `float` |
| regresa | `return` |
| $\geq$ | `>=` |
| y | `&&` |
| si ... entonces | `if (...)` |
| sino si ... entonces | `else if (...)` |
| sino entonces | `else` |

- El tipo que regresa la función se escribe antes del nombre de la función (sin $\mapsto$).
- Las sentencias que ocupan una línea terminan en punto y coma.
- Los bloques condicionales van entre llaves en lugar de sangría.
- La multiplicación se indica explícitamente con `*`.
- Las fracciones deben expresarse de forma lineal con `/` y paréntesis donde sea necesario.

## Soluciones de 4.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Hola-Mundo-c`.

**Solución:** Para resolver este problema, basta enviar el código de ejemplo de las notas.

```
#include <stdio.h>

int main( ) {
```

```
    printf("Hola Mundo");
    return 0;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Introduccion-a-OmegaUp`.

**Solución:** Para resolver este problema, se necesita saber cómo se declaran variables, cómo se suma y cómo se usan las funciones `scanf` y `printf`.

```
#include <stdio.h>

int main( ) {
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d", a + b);
    return 0;
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Tres-Numeros-Al-Reves`.

**Solución:** Para resolver este problema, se necesita entender que las funciones `scanf` y `printf` aplican los especificadores de formato sobre sus parámetros en el orden en el que éstos aparecen.

```
#include <stdio.h>

int main( ) {
    int a, b, c;
    scanf("%d%d%d", &a, &b, &c);
    printf("%d\n%d\n%d", c, b, a);
    return 0;
}
```

## Soluciones de 5.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Programando-formulas`.

**Solución:** Para resolver este problema, se debe saber cómo manipular números reales y también se debe tener cuidado en emplear la notación correcta de multiplicación. Además, se deben usar paréntesis en los lugares en los que más convenga.

```
#include <stdio.h>

int main( ) {
    float x, y, z;
    scanf("%f%f%f", &x, &y, &z);

    float num = x + x * (y + z * z);
    float den = (x + 3.1416) * (y + 3.1416);
    printf("%f", num / den);
    return 0;
```

```
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Calculos-mentales-competitivos`.

**Solución:** Para resolver este problema, conviene declarar una nueva variable para cada cálculo intermedio. Se debe tener cuidado en relacionar tales variables con los valores del problema.

```
#include <stdio.h>

int main( ) {
   float r1;
   scanf("%f", &r1);

   float r2 = r1 + 5;
   float r3 = r2 * r2;
   float r4 = r3 / (r1 / 3);
   float r5 = r4 * r4 * r4;

   printf("%f %f %f %f %f\n", r1, r2, r3, r4, r5);
   return 0;
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Evaluando-formulas-en-sucesion`.

**Solución:** Para resolver este problema, se debe saber cómo manipular números reales, se debe tener cuidado en emplear la notación correcta de multiplicación y también se debe cuidar el uso de paréntesis. Conviene usar variables auxiliares para facilitar la escritura de las expresiones.

```
#include <stdio.h>

int main( ) {
   float x;
   scanf("%f", &x);

   float y = (x + 5) / (2 * (x + 1));
   float z = (y * y + x * (x - 2 * y)) / (x * y);
   printf("%f\n", z);
   return 0;
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/Alicia-y-la-carrera-de-animales`.

**Solución:** Para resolver este problema, hay que poder plantear el problema matemático. Si el ratón tarda $r$ segundos en dar una vuelta, entonces recorre $\frac{1}{r}$ vueltas por segundo. Lo mismo ocurre con el pato: él recorre $\frac{1}{p}$ vueltas por segundo. El ratón le saca $\frac{1}{r} - \frac{1}{p}$ vueltas de ventaja por segundo al pato, por lo que le sacará una vuelta completa de ventaja en $\frac{1}{\frac{1}{r} - \frac{1}{p}}$ segundos. Si los datos se leyeron como enteros, se deben usar literales reales para que la división sea real.

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

```
int main( ) {
   int r, p;
   scanf("%d%d", &r, &p);
   printf("%f", 1.0 / (1.0 / r - 1.0 / p));
   return 0;
}
```

5. Resuelve el problema `https://omegaup.com/arena/problem/La-banda-robatamales`.

**Solución:** Para resolver este problema, lo que más conviene es calcular por partes los datos de interés del problema. El jefe apartará la mitad de los tamales (esto es `t/2`) pero también se queda con el que sobre si el número de tamales no es par (esto es `t%2` porque el residuo de una división entre 2 vale 1 si el dividendo es impar). Si la variable `jefe` es la cantidad de tamales que el jefe acapara inicialmente, entonces los tamales que se repartirán entre el resto de la banda son `t - jefe`. El número de subordinados de la banda es `b-1`. Los tamales que sobren de esta repartición también se los quedará el jefe.

---

```
#include <stdio.h>

int main( ) {
   int t, b;
   scanf("%d%d", &t, &b);

   int jefe = t / 2 + t % 2;
   int repartir = t - jefe;
   int sobran = repartir % (b - 1);

   printf("%d\n", jefe + sobran);
   return 0;
}
```

6. Resuelve el problema `https://omegaup.com/arena/problem/La-hora-en-un-planeta-lejado`.

**Solución:** Para resolver este problema, primero hay que calcular la cantidad de días completos que representan los segundos dados. Los segundos que no alcanzaron a contabilizar días completos tal vez sí contabilicen horas completas. Los segundos que no alcanzaron a contabilizar horas completas tal vez sí contabilicen minutos completos. Finalmente, Los segundos que no alcanzaron a contabilizar minutos completos son los segundos que sobraron. Conviene descomponer el cálculo en cada uno de estos pasos, usando tantas variables auxiliares como convenga.

---

```
#include <stdio.h>

int main( ) {
   int t1;
   scanf("%d", &t1);

   int d = t1 / (50 * 70 * 12);
   int t2 = t1 % (50 * 70 * 12);
   int h = t2 / (50 * 70);
   int t3 = t2 % (50 * 70);
   int m = t3 / (50);
   int t4 = t3 % (50);
```

```
    int s = t4;

    printf("%d %d %d %d\n", d, h, m, s);
    return 0;
}
```

## Soluciones de 9.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Par-o-Impar`.

**Solución:** Para resolver este problema, es suficiente entender cómo realizar comparaciones y cómo opera la expresión ternaria, de modo que podamos seleccionar el mensaje correcto a imprimir.

```
#include <stdio.h>

int main( ) {
    int n;
    scanf("%d", &n);
    printf((n % 2 == 0 ? "Par" : "Impar"));
    return 0;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Fahrenheit-a-Centigrados`.

**Solución:** Para resolver este problema se necesitan realizar algunas operaciones aritméticas simples y una comparación. Conviene aprovechar el hecho de que una comparación verdadera evalúa a 1 y una falsa evalúa a 0.

```
#include <stdio.h>

int main( ) {
    int f;
    scanf("%d", &f);
    int c = 5 * (f - 32) / 9;
    printf("%d %d\n", c, c > 36);
    return 0;
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Cuantos-valen-7`.

**Solución:** Para resolver este problema se puede aprovechar el hecho de que una comparación verdadera evalúa a 1 y una falsa evalúa a 0, de modo que usaremos estos valores en una suma.

```
#include <stdio.h>

int main( ) {
    int a, b;
    scanf("%d%d", &a, &b);
    printf("%d", (a == 7) + (b == 7));
    return 0;
```

```
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/El-menor-de-tres-numeros`.

**Solución:** Aunque este problema se puede resolver sin funciones, conviene concentrar el cálculo del mínimo de dos o tres enteros en sus funciones respectivas. El menor de dos enteros se puede calcular con una expresión ternaria y el menor de tres enteros se puede calcular usando repetidamente la función del mínimo de dos enteros.

```
#include <stdio.h>

int minimo2(int a, int b) {
   return (a < b ? a : b);
}

int minimo3(int a, int b, int c) {
   return minimo2(a, minimo2(b, c));
}

int main( ) {
   int a, b, c;
   scanf("%d%d%d", &a, &b, &c);
   printf("%d", minimo3(a, b, c));
   return 0;
}
```

## Soluciones de 10.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Triangulo-equilatero`.

**Solución:** Para intentar formar un triángulo equilátero, debemos elegir tres de los cuatro palitos disponibles. Eso quiere decir que hay cuatro formas distintas de elegir los tres palitos, dependiendo de qué palito excluimos de la selección. Para cada selección, podemos comparar si los tres palitos elegidos son iguales usando el operador `==` y la conectiva `&&`. Como no sabemos de antemano qué selección cumplió la condición de poder formar un triángulo equilátero (si es que alguna lo hizo), revisaremos todas e imprimiremos el resultado de la conectiva `||`, que es 1 si alguna fue verdadera.

```
#include <stdio.h>

int main( ) {
   int a, b, c, d;
   scanf("%d%d%d%d", &a, &b, &c, &d);

   int p1 = (b == c && c == d);
   int p2 = (a == c && c == d);
   int p3 = (a == b && b == d);
   int p4 = (a == b && b == c);
   printf("%d", p1 || p2 || p3 || p4);
   return 0;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Interseccion-de-intervalos`.

**Solución:** En este problema hay muchas situaciones que se pueden dar, pero lo más fácil es detectar si los intervalos no se intersecan. Cuando no hay intersección es porque un intervalo está a la izquierda del otro (un intervalo está a la izquierda si termina antes de que el otro comience). Para este caso lo único que falta por determinar es cuál de los intervalos de la entrada es el que está a la izquierda. Dado que debemos imprimir 1 cuando los intervalos sí se intersecan, invertiremos el sentido de la condición con el operador `!`.

```
#include <stdio.h>

int main( ) {
   int ini1, ult1, ini2, ult2;
   scanf("%d%d%d%d", &ini1, &ult1, &ini2, &ult2);
   printf("%d", !(ult1 < ini2 || ult2 < ini1));
   return 0;
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Cuantos-dias-tiene-febrero`.

**Solución:** Un año es bisiesto si es múltiplo de 4, excepto si es múltiplo de 100 porque entonces también se requiere que sea múltiplo de 400. Esto se puede implementar con comparaciones y conectivas, aunque se requieren paréntesis y ambos tipos de conectivas `&&` y `||`. Como debemos responder para cuatro años, lo mejor es implementar una función.

```
#include <stdio.h>

int es_bisiesto(int a) {
   return (a % 4 == 0 && (a % 100 != 0 || a % 400 == 0));
}

int main( ) {
   int a1, a2, a3, a4;
   scanf("%d%d%d%d", &a1, &a2, &a3, &a4);

   printf("%d ", 28 + es_bisiesto(a1));
   printf("%d ", 28 + es_bisiesto(a2));
   printf("%d ", 28 + es_bisiesto(a3));
   printf("%d ", 28 + es_bisiesto(a4));
   return 0;
}
```

## Soluciones de 11.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Detectando-el-orden`.

**Solución:** Para resolver este problema, es necesario entender que algunas comparaciones se escriben de una forma muy peculiar en C. En particular, la expresión lógica $a \leq b \leq c$ se debe escribir como `a <= b && b <= c`. También se necesita comprender el funcionamiento de las sentencias `if` y `else`, recordando que el orden en el que se verifican las condiciones importa ya se evaluará sólo el bloque de la primera condición verdadera (o el bloque del `else` si todas fueron falsas).

```
#include <stdio.h>

int main( ) {
   int a, b, c;
   scanf("%d%d%d", &a, &b, &c);

   if (a == b && b == c) {
      printf("I");
   } else if (a <= b && b <= c) {
      printf("C");
   } else if (a >= b && b >= c) {
      printf("D");
   } else {
      printf("X");
   }

   return 0;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Escamitas`.

**Solución:** En este problema conviene hacer una secuencia de `if` y `else` para escoger el mensaje que se imprimirá con respecto a la temperatura, y luego un `if` y `else` independiente de lo anterior para imprimir el mensaje de si Escamitas muere.

---

```
#include <stdio.h>

int main( ) {
   int t;
   scanf("%d", &t);

   if (t < 21) {
      printf("fria!\n");
   } else if (t > 29) {
      printf("caliente!\n");
   } else {
      printf("tibia\n");
   }

   if (t < 15 || t > 34) {
      printf("RIP escamitas :(");
   }

   return 0;
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Calculos-condicionales`.

**Solución:** Para resolver este problema basta implementarlo cuidadosamente. Cada regla se evalúa en sucesión y es independiente de las otras, aunque cada regla requiere de su propia secuencia de `if` y `else`. La variable que lleva la cuenta de cuántas modificaciones se han realizado debe incrementarse cada vez que alguna regla modifique `n`. Lo más fácil es escribir la instrucción que

incrementa `m` en todos los lugares donde se modifique `n`.

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);
   int m = 0;

   if (n % 2 == 0) {
      n /= 2;
      m += 1;
   } else {
      n += 1;
      m += 1;
   }

   if (n >= 100) {
      n /= 100;
      m += 1;
   } else if (n >= 10) {
      n /= 10;
      m += 1;
   }

   if (n % 3 == 0) {
      n -= 1;
      m += 1;
   }

   printf("%d %d", n, m);
   return 0;
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/Calculando-con-masmenos`.

**Solución:** La expresión está indefinida cuando `b` vale 0 (porque tendríamos que hacer una división entre 0). Si la expresión está bien definida entonces tiene un único valor cuando `c` vale 0. Conviene hacer las pruebas en este orden, ya que el caso general se da cuando no ocurren ninguno de las dos casos anteriores.

```
#include <stdio.h>

int main( ) {
   float a, b, c;
   scanf("%f%f%f", &a, &b, &c);

   if (b == 0) {
      printf("indefinido");
   } else if (c == 0) {
      printf("%f", a / b);
   } else {
      printf("%f %f", a / b + c, a / b - c);
```

```
    }
    return 0;
}
```

5. Resuelve el problema `https://omegaup.com/arena/problem/Una-serie-poco-interesante`.

**Solución:** Este problema parece imposible de resolver si no nos damos cuenta que al saltar tres posiciones a la izquierda o a la derecha caemos en el mismo número en el que estamos actualmente. De forma similar, saltar seis posiciones se puede ver como saltar tres posiciones dos veces, por lo que volveremos a caer en el mismo número. Por consiguiente, reduciremos la cantidad de casos a revisar usando el residuo que se obtiene tras dividir la longitud del salto entre 3.

---

```
#include <stdio.h>

int main( ) {
   int x, y ;
   scanf("%d%d", &x, &y);
   y %= 3;

   if (x == 1){
      if (y == 0) {
         printf("1 1");
      } else if (y == 1) {
         printf("3 2");
      } else if (y == 2) {
         printf("2 3");
      }
   } else if (x == 2) {
      if (y == 0) {
         printf("2 2");
      } else if (y == 1) {
         printf("1 3");
      } else if (y == 2) {
         printf("3 1");
      }
   } else if (x == 3) {
      if (y == 0) {
         printf("3 3");
      } else if (y == 1) {
         printf("2 1");
      } else if (y == 2) {
         printf("1 2");
      }
   }
   return 0;
}
```

Curiosamente, este problema también se puede resolver con puros cálculos aritméticos. ¿Puedes entender lo que hace el siguiente programa y por qué es correcto?

```
#include <stdio.h>

int main( ) {
   int n, p;
```

```
   scanf("%d%d", &n, &p);
   printf("%d ", 1 + ((n - 1) - (p % 3) + 3) % 3);
   printf("%d ", 1 + ((n - 1) + (p % 3)) % 3);
   return 0;
}
```

## Soluciones de 12.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Un-proceso-iterativo-sencillo`

**Solución:** Para resolver este problema, basta con implementar el algoritmo descrito. Se recomienda emplear un ciclo `while`, el cual a su vez tiene anidado un `if` con su `else`.

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   while (n < 100) {
      if (n % 2 == 0) {
         n += 3;
      } else {
         n *= 2;
      }
   }

   printf("%d", n);
   return 0;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Suma-de-los-digitos`

**Solución:** Para resolver este problema, se deben observar dos cosas: podemos obtener el dígito de las unidades de un entero `n` con la expresión `n% 10` y podemos recorrer los dígitos de `n` un lugar a la derecha con `n /= 10;`. La estrategia entonces será sumar las unidades de `n`, recorrer los dígitos y repetir hasta que `n` se quede sin dígitos significativos.

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int s = 0;
   while (n > 0) {
      s += n % 10;
      n /= 10;
   }

   printf("%d", s);
```

```
   return 0;
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Divide-y-sumaras`

**Solución:** Para resolver este problema, debemos observar dos cosas: el denominador eventualmente se hará más grande que el numerador y, en ese momento, el piso de la división se volverá cero. Esto vuelve irrelevante el hecho de que la sumatoria es infinita.

---

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int d = 1, s = 0;
   while (d <= n) {
      s += n / d;
      d *= 2;
   }

   printf("%d", s);
   return 0;
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/Calculando-el-logaritmo-base-2`

**Solución:** El problema nos pide calcular el logaritmo base 2 de una potencia de 2, por lo que basta determinar en cuántas veces necesitamos duplicar una variable que comience en 1 para alcanzar el valor dado en la entrada.

---

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int v = 1, r = 0;
   while (v < n) {
      v *= 2;
      r += 1;
   }

   printf("%d", r);
   return 0;
}
```

5. Resuelve el problema `https://omegaup.com/arena/problem/El-Caracol`

**Solución:** Este problema se puede resolver con una simulación. La dificultad del problema es que la condición de paro está enmedio de lo que ocurre en un día: el caracol primero intenta subir y puede ser salga del agujero antes de tener que dormir; si no lo logra entonces resbala y vuelve a comenzar el siguiente día. Esto se puede modelar con un ciclo infinito y una sentencia `break` para romper el ciclo.

```
#include <stdio.h>

int main( ) {
   int h, s, r;
   scanf("%d%d%d", &h, &s, &r);
   int c = 0, d = 0;

   for (;;) {
      d += 1;
      c += s;
      if (c >= h) {
         break;
      }
      c -= r;
   }

   printf("%d", d);
   return 0;
}
```

6. Resuelve el problema `https://omegaup.com/arena/problem/ciclo_mientras_no_cero`

**Solución:** En este problema se deben leer números de forma repetitiva. La dificultad es que la condición de paro sólo se puede evaluar después de haber leído el primer número. Un ciclo `do` se presta para implementar este problema.

```
#include <stdio.h>

int main( ) {
   int n, s = 0;
   do {
      scanf("%d", &n);
      s += n;
   } while (n != 0);

   printf("%d", s);
   return 0;
}
```

7. Resuelve el problema `https://omegaup.com/arena/problem/Pares-e-impares`

**Solución:** En este problema, la cantidad de iteraciones que debe realizar el ciclo está dado por el valor de `n`, por lo que un ciclo `for` se presta para implementar el programa. Para cada número leído, sólo hay que revisar su paridad y actualizar el contador de pares o nones según corresponda.

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int pares = 0, nones = 0;
   for (int i = 1; i <= n; ++i) {
      int actual;
      scanf("%d", &actual);
      if (actual % 2 == 0) {
         pares += 1;
      } else {
         nones += 1;
      }
   }

   printf("%d %d", pares, nones);
   return 0;
}
```

8. Resuelve el problema `https://omegaup.com/arena/problem/Divisores-positivos`

**Solución:** Los divisores positivos de un entero positivo `n` se pueden calcular revisando cuáles divisiones entre 1, 2, ..., `n` son exactas. Para éstas, hay que incrementar un contador.

---

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int d = 0;
   for (int i = 1; i <= n; ++i) {
      if (n % i == 0) {
         d += 1;
      }
   }

   printf("%d\n", d);
   return 0;
}
```

9. Resuelve el problema `https://omegaup.com/arena/problem/Sumatoria-de-sumatorias`

**Solución:** Un ciclo `for` es adecuado para implementar la suma de los números en un rango. La dificultad del problema radica en que hay tres sumatorias, donde los rangos de la tercera están en función de las otras dos. Lo ideal es implementar una función que sume un rango de números y evaluar las tres sumatorias usándola.

---

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

```
int sumatoria(int a, int b) {
   int res = 0;
   for (int i = a; i <= b; ++i) {
      res += i;
   }
   return res;
}

int main( ) {
   int x, y;
   scanf("%d%d", &x, &y);
   printf("%d\n", sumatoria(sumatoria(1, x), sumatoria(1, y)));
   return 0;
}
```

10. Resuelve el problema `https://omegaup.com/arena/problem/CR-Leyendo-Varios-datos`

**Solución:** Este problema sería muy fácil si sólo tuvieramos que leer dos enteros e imprimir su suma. Sin embargo, lo único que nos piden es hacer esto `n` veces. Leeremos el valor de `n` desde la entrada y haremos un ciclo que realice `n` veces la lectura de una pareja y la impresión de su suma.

---

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   for (int i = 1; i <= n; ++i) {
      int a, b;
      scanf("%d%d", &a, &b);
      printf("%d\n", a + b);
   }

   return 0;
}
```

11. Resuelve el problema `https://omegaup.com/arena/problem/Calculos-iterativos`

**Solución:** Los dos primeros pases de este problema puede implementarse con ciclos `for`, teniendo cuidado de usar la actualización adecuada para la variable controladora del ciclo. El último paso puede implementarse con un ciclo `while`, recordando que un entero es múltiplo de otro si el residuo de su división es cero.

---

```
#include <stdio.h>

int main( ) {
   int a, b, c;
   scanf("%d%d%d", &a, &b, &c);

   int n = 0;
   for (int i = 1; i <= a; i += 2) {
      n += i;
```

```
    }
    for (int i = 1; i <= b; i *= 2) {
        n -= i;
    }
    while (n % c == 0) {
        n /= c;
    }

    printf("%d", n);
    return 0;
}
```

12. Resuelve el problema `https://omegaup.com/arena/problem/Mensaje-de-Amor`

**Solución:** Para resolver este problema, leeremos un entero `n` y repetiremos `n` veces lo siguiente: leer un entero `k` para imprimir `k` corazones en un ciclo, seguidos de un salto de línea.

---

```
#include <stdio.h>

int main( ) {
    int n;
    scanf("%d", &n);

    for (int i = 1; i <= n; ++i) {
        int k;
        scanf("%d", &k);

        for (int i = 1; i <= k; ++i) {
            printf("<3");
        }
        printf("\n");
    }

    return 0;
}
```

13. Resuelve el problema `https://omegaup.com/arena/problem/El-k-esimo-numero-primo`

**Solución:** Para resolver este problema, lo adecuado es implementar una función que decida si un entero `n` es primo o no. Esto puede hacerse probando si algún número de 2 a `n - 1` es un divisor exacto (también hay que recordar que, por definición, el 1 no es primo). Si encontramos algún divisor, entonces `n` no es primo; sólo si `n` pasa todas las verificaciones entonces es primo. En `main` debemos llevar la cuenta de cuántos primos hemos encontrado hasta el momento, y si un entero acaba de ser el $k$-ésimo primo encontrado, lo imprimimos y terminamos.

---

```
#include <stdio.h>

int es_primo(int n) {
    if (n <= 1) {
        return 0;
    }
```

```
   for (int i = 2; i < n; ++i) {
      if (n % i == 0) {
         return 0;
      }
   }

   return 1;
}

int main( ) {
   int k;
   scanf("%d", &k);

   int primos = 0;
   for (int i = 1; ; ++i) {
      primos += es_primo(i);
      if (primos == k) {
         printf("%d", i);
         break;
      }
   }

   return 0;
}
```

14. Resuelve el problema `https://omegaup.com/arena/problem/Dibujando-un-triangulo`

**Solución:** La primera fila del triángulo tiene un arroba, la segunda tiene tres arrobas, así hasta llegar a la base del triángulo que tiene `n` arrobas. Se puede considerar que cada fila del dibujo tiene `n` símbolos, sólo que los símbolos que no son arrobas son espacios (y la mitad aparece a la izquierda de las arrobas). Para cada fila hay que calcular la cantidad de espacios a la izquierda de las arrobas, imprimiras con un ciclo, luego imprimir las arrobas con otro ciclo y terminar con un salto de línea.

---

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   for (int a = 1; a <= n; a += 2) {
      int izq = (n - a) / 2;
      for (int i = 1; i <= izq; ++i) {
         printf(" ");
      }
      for (int i = 1; i <= a; ++i) {
         printf("@");
      }
      printf("\n");
   }
}
```

15. Resuelve el problema `https://omegaup.com/arena/problem/supresores_de_picos`

**Solución:** En el problema nos dan la cantidad de entradas que tienen cada uno de `k` supresores de picos. Al conectarlos en serie, cada supresor de atrás ocupa una entrada del de adelante; el único supresor que no tiene a nadie atrás es el último. Podemos sumar la cantidad de entradas de los `k` supresores y luego simplemente restar las entradas ocupadas. El programa se complica ligeramente porque hay que resolver el problema anterior `n` veces para distintas colecciones de supresores.

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   for (int i = 0; i < n; ++i) {
      int k;
      scanf("%d", &k);

      int total = 0;
      for (int j = 0; j < k; ++j) {
         int actual;
         scanf("%d", &actual);
         total += actual;
      }

      printf("%d\n", total - k + 1);
   }

   return 0;
}
```

## Soluciones de 13.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Forzando-la-caja-fuerte`

**Solución:** Este problema se presta para implementar una función `rota_izq` que realice la rotación izquierda de cinco enteros mediante apuntadores. La rotación requiere hacer un respaldo del valor del primer elemento, reasignar los elementos entre sí y asignar el valor respaldado al último elemento. Una función `rota_der` puede implementarse en términos de la anterior, observando que una rotación a la izquierda del reverso de los elementos es en realidad una rotación a la derecha. La última observación es que rotar cinco veces hacia un lado produce el mismo resultado que no rotar, por lo que basta hacer esto la cantidad de veces que indique el residuo de la división.

```
#include <stdio.h>

void rota_izq(int* pa, int* pb, int* pc, int* pd, int* pe) {
   int t = *pa;
   *pa = *pb;
   *pb = *pc;
   *pc = *pd;
   *pd = *pe;
   *pe = t;
}
```

```
void rota_der(int* pa, int* pb, int* pc, int* pd, int* pe) {
   rota_izq(pe, pd, pc, pb, pa);
}

int main( ) {
   int ri, rd;
   scanf("%d%d", &ri, &rd);

   int a = 1, b = 2, c = 3, d = 4, e = 5;
   for (int i = 1; i <= ri % 5; ++i) {
       rota_izq(&a, &b, &c, &d, &e);
   }
   for (int i = 1; i <= rd % 5; ++i) {
       rota_der(&a, &b, &c, &d, &e);
   }

   printf("%d %d %d %d %d", a, b, c, d, e);
   return 0;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-numeros`

**Solución:** Para ordenar dos enteros, podemos intercambiarlos si el primer entero es mayor que el segundo. Para ordenar cuatro enteros, podemos ejecutar los siguientes ordenamientos por parejas para simular un torneo con desempate: el primer entero contra el segundo, el tercer entero contra el cuarto, el menor del primer ordenamiento contra el menor del segundo ordenamiento, el mayor del primer ordenamiento contra el mayor del segundo ordenamiento, y finalmente, el mayor de los menores contra el menor de los mayores.

---

```
#include <stdio.h>

void intercambia(int* a, int* b) {
   int c = *a;
   *a = *b;
   *b = c;
}

void ordena2(int* a, int* b) {
   if (*a > *b) {
      intercambia(a, b);
   }
}

int main( ) {
   int a, b, c, d;
   scanf("%d%d%d%d", &a, &b, &c, &d);
   ordena2(&a, &b);
   ordena2(&c, &d);
   ordena2(&a, &c);
   ordena2(&b, &d);
   ordena2(&b, &c);
   printf("%d %d %d %d", a, b, c, d);
```

```
    return 0;
}
```

## Soluciones de 14.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Ignorando-los-primeros-elementos`

**Solución:** Este problema se puede resolver simplemente leyendo el arreglo y comenzando el ciclo `for` que lo imprime a partir de `k`.

```
#include <stdio.h>

int main( ) {
    int n;
    scanf("%d", &n);

    int arr[n];
    for (int i = 0; i < n; ++i) {
        scanf("%d", &arr[i]);
    }

    int k;
    scanf("%d", &k);

    for (int i = k; i < n; ++i) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Ignorando-los-ultimos-elementos`

**Solución:** Este problema se puede resolver simplemente leyendo el arreglo y ajustando el ciclo `for` para que simule que el tamaño del arreglo es `n - k`.

```
#include <stdio>

int main( ) {
    int n;
    scanf("%d", &n);

    int arr[n];
    for (int i = 0; i < n; ++i) {
        scanf("%d", &arr[i]);
    }

    int k;
    scanf("%d", &k);

    for (int i = 0; i < n - k; ++i) {
```

```
      printf("%d ", arr[i]);
   }
   return 0;
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Reverso`

**Solución:** Este problema se puede resolver simplemente leyendo el arreglo y ajustando el ciclo `for` para que visite todas las posiciones válidas del arreglo de $n-1$ a 0, en ese orden.

---

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int arr[n];
   for (int i = 0; i < n; ++i) {
      scanf("%d", &arr[i]);
   }

   for (int i = n - 1; i >= 0; --i) {
      printf("%d ", arr[i]);
   }
   return 0;
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/Conjunto-Capicua`

**Solución:** En este problema, conviene ir verificando si el arreglo es capicua de los extremos al centro. Si cualquier verificación falla entonces el arreglo ya no es capicua, independientemente de lo que pudiera ocurrir en las verificaciones faltantes. Tendremos una variable que nos ayude a determinar si en algún momento alguna verificación falló.

---

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int arr[n];
   for (int i = 0; i < n; ++i) {
      scanf("%d", &arr[i]);
   }

   int res = 1;
   for (int i = 0, j = n - 1; i < j; ++i, --j) {
      if (arr[i] != arr[j]) {
         res = 0;
         break;
      }
   }
```

```
   printf((res == 1 ? "SI" : "NO"));
   return 0;
}
```

5. Resuelve el problema `https://omegaup.com/arena/problem/Suma-de-vectores`

**Solución:** En este problema, basta con leer los dos arreglos y luego imprimir la suma de los elementos que están en la misma posición en ambos arreglos. ¿Se te ocurre cómo resolver el problema empleando sólo un arreglo?

---

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int v1[n], v2[n];
   for (int i = 0; i < n; ++i) {
      scanf("%d", &v1[i]);
   }
   for (int i = 0; i < n; ++i) {
      scanf("%d", &v2[i]);
   }
   for (int i = 0; i < n; ++i) {
      printf("%d ", v1[i] + v2[i]);
   }

   return 0;
}
```

6. Resuelve el problema `https://omegaup.com/arena/problem/Buscar-y-contar`

**Solución:** Para resolver este problema, podemos leer primero al arreglo y luego leer el valor a buscar. Ya con este último, visitaremos los elementos del arreglo y contaremos cada coincidencia.

---

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int arr[n];
   for (int i = 0; i < n; ++i) {
      scanf("%d", &arr[i]);
   }

   int b;
   scanf("%d", &b);

   int res = 0;
   for (int i = 0; i < n; ++i) {
```

```
      res += (arr[i] == b);
   }

   printf("%d", res);
   return 0;
}
```

7. Resuelve el problema `https://omegaup.com/arena/problem/Ordena-Basico-1`

**Solución:** Para resolver este problema, basta con leer el arreglo, emplear el ordenamiento de burbuja descrito en las notas y luego imprimir el arreglo.

---

```
#include <stdio.h>

void intercambia(int* a, int* b) {
   int c = *a;
   *a = *b;
   *b = c;
}

void ordena2(int* a, int* b) {
   if (*a > *b) {
      intercambia(a, b);
   }
}

int main( ) {
   int n;
   scanf("%d", &n);

   int arr[n];
   for (int i = 0; i < n; ++i) {
      scanf("%d", &arr[i]);
   }

   for (int k = 0; k < n - 1; ++k) {
      for (int i = 0; i < n - 1; ++i) {
         ordena2(&arr[i], &arr[i + 1]);
      }
   }

   for (int i = 0; i < n; ++i) {
      printf("%d ", arr[i]);
   }
   return 0;
}
```

8. Resuelve el problema `https://omegaup.com/arena/problem/Modificando-un-arreglo`

**Solución:** Para resolver este problema, debe quedar clara la distinción entre una posición del arreglo y el valor almacenado en dicha posición. En este problema debemos leer posiciones y usarlas para acceder y modificar los valores de un arreglo.

```c
#include <stdio.h>

int main( ) {
   int n, m;
   scanf("%d%d", &n, &m);

   int arr[n] = { };  // versiones nuevas de gcc aceptan esto
                      // si no les compila, inicialicen el arreglo
                      // manualmente con ceros
   for (int i = 0; i < m; ++i) {
      int pos;
      scanf("%d", &pos);
      arr[pos] += 1;
   }

   for (int i = 0; i < n; ++i) {
      printf("%d ", arr[i]);
   }
   return 0;
}
```

9. Resuelve el problema `https://omegaup.com/arena/problem/Alicia-y-el-cachorro-saltarin`

**Solución:** En este problema, tanto Alicia como el cachorro están parados en una recta numérica. Basta con ir acumulando los saltos del cachorro y contar cuántas veces coincide su ubicación sobre la recta con la de Alicia. La única razón por la que necesitamos un arreglo es porque nos dan la ubicación de Alicia sobre la recta después de darnos los saltos del cachorro.

```c
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int arr[n];
   for (int i = 0; i < n; ++i) {
      scanf("%d", &arr[i]);
   }

   int a;
   scanf("%d", &a);

   int c = 0, res = 0;
   for (int i = 0; i < n; ++i) {
      c += arr[i];
      res += (c == a);
   }

   printf("%d", res);
   return 0;
}
```

10. Resuelve el problema `https://omegaup.com/arena/problem/Saltando-por-el-arreglo`

**Solución:** Para resolver este problema, debe quedar clara la distinción entre una posición del arreglo y el valor almacenado en dicha posición. En este problema, nuestra posición actual se verá modificada por el valor almacenado en el arreglo en dicha posición.

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int arr[n];
   for (int i = 0; i < n; ++i) {
      scanf("%d", &arr[i]);
   }

   int pos = 0, res = 0;
   while (pos != n - 1) {
      pos += arr[pos];
      res += 1;
   }

   printf("%d\n", res);
   return 0;
}
```

11. Resuelve el problema `https://omegaup.com/arena/problem/Omitiendo-el-entero-mas-grande`

**Solución:** En este problema, es inevitable tener que recorrer varias veces el arreglo. Podemos calcular el máximo del arreglo conforme lo leemos (aprovechando el hecho de que todos los enteros son positivos). Una vez calculado, tendremos que dar otra pasada sobre el mismo para contar cuántos números son distintos a dicho máximo. Finalmente, debemos reimprimir el arreglo ignorando todos los valores que sean igual al máximo.

```
#include <stdio.h>

int max(int a, int b) {
   return (a > b ? a : b);
}

int main( ) {
   int n;
   scanf("%d", &n);

   int arr[n], m = -1;
   for (int i = 0; i < n; ++i) {
      scanf("%d", &arr[i]);
      m = max(m, arr[i]);
   }

   int veces = 0;
```

```
   for (int i = 0; i < n; ++i) {
      veces += (arr[i] != m);
   }

   printf("%d\n", veces);
   for (int i = 0; i < n; ++i) {
      if (arr[i] != m) {
         printf("%d ", arr[i]);
      }
   }
}
```

## Soluciones de 15.1

1. Resuelve el problema `https://omegaup.com/arena/problem/sumar-matrices`

**Solución:** Para resolver este problema, basta leer las dos matrices y luego imprimir la suma de elementos correspondientes (elementos de ambas matrices que estén en la misma posición). Un ciclo `for` externo elige la fila de la matriz y un `for` anidado visita todas las celdas de esa fila.

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int mat1[n][n], mat2[n][n];
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         scanf("%d", &mat1[i][j]);
      }
   }
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         scanf("%d", &mat2[i][j]);
         printf("%d ", mat1[i][j] + mat2[i][j]);
      }
      printf("\n");
   }

   return 0;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Demostrando-con-matrices`

**Solución:** Una matriz `a` es simétrica si cada `a[i][j]` es igual a `a[i][j]`. Esto es cierto porque la diagonal de la matriz corresponde con los elementos `a[i][i]` y una matriz simétrica es aquélla cuya parte triangular inferior es un reflejo de su parte triangular superior. Comenzaremos creyendo que la matriz es simétrica y cambiaremos de opinión al primer error que encontremos. Se tiene que hacer esto para cada matriz dada.

```
#include <stdio.h>

int main( ) {
   int m;
   scanf("%d", &m);

   for (int i = 0; i < m; ++i) {
      int n;
      scanf("%d", &n);

      int mat[n][n];
      for (int i = 0; i < n; ++i) {
         for (int j = 0; j < n; ++j) {
            scanf("%d", &mat[i][j]);
         }
      }

      int res = 1;
      for (int i = 0; i < n; ++i) {
         for (int j = 0; j < n; ++j) {
            if (mat[i][j] != mat[j][i]) {
               res = 0;
            }
         }
      }
      printf((res == 1 ? "Simétrica\n" : "No Simétrica\n"));
   }

   return 0;
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Producto-de-matrices`

**Solución:** Una matriz `m1` de dimensiones $a \times b$ se puede multiplicar por una matriz `m2` de dimensiones $b \times c$ y el resultado es una matriz de dimensiones $a \times c$. Conviene concentrarnos en calcular cada elemento `r[i][j]` del resultado, lo cual requiere seleccionar la fila `i` de `a` y la columna `j` de `b` donde ambas selecciones tienen `b` elementos. Los elementos correspondientes de ambas selecciones se multiplican entre sí y luego se suman sus productos.

---

```
#include <stdio.h>

int main( ) {
   int a, b, c;
   scanf("%d%d%d", &a, &b, &c);

   int mat1[a][b], mat2[b][c];
   for (int i = 0; i < a; ++i) {
      for (int j = 0; j < b; ++j) {
         scanf("%d", &mat1[i][j]);
      }
   }
   for (int i = 0; i < b; ++i) {
      for (int j = 0; j < c; ++j) {
```

```
            scanf("%d", &mat2[i][j]);
        }
    }

    for (int i = 0; i < a; ++i) {
        for (int j = 0; j < c; ++j) {
            int t = 0;
            for (int k = 0; k < b; ++k) {
                t += mat1[i][k] * mat2[k][j];
            }
            printf("%d ", t);
        }
        printf("\n");
    }
    return 0;
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/Matrices-giradas`

**Solución:** Para resolver este problema, basta observar que la matriz se tiene que imprimir por columnas de izquierda a derecha, y de abajo para arriba en cuanto a filas. Cabe señalar que no es necesario almacenar el resultado en otra matriz, porque lo único que el juez en línea evalúa es lo que el programa imprime.

---

```
#include <stdio.h>

int main( ) {
    int n;
    scanf("%d", &n);

    int mat[n][n];
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            scanf("%d", &mat[i][j]);
        }
    }

    for (int j = 0; j < n; ++j) {
        for (int i = n - 1; i >= 0; --i) {
            printf("%d ", mat[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

5. Resuelve el problema `https://omegaup.com/arena/problem/Los-cuadrados-semimagicos`

**Solución:** Comenzaremos creyendo que la matriz es semimágina y cambiaremos de opinión al primer error que encontremos. Tras leer la matriz, podemos sumar la diagonal y la contradiagonal para ver si sus sumas coinciden. La diagonal tiene sus elementos en `a[i][i]` y la contradiagonal en `a[i][n - 1 - i]` (la contradiagonal comienza en la última columna, pero conforme bajemos

nos movemos cada vez más a la izquierda). La suma de ambas diagonales deben coincidir entre ellas y con la suma de cada fila y columna. Podemos usar la fila `i` y la columna `i` al mismo tiempo, usando otra variable que visite todas las celdas de cada una.

```
#include <stdio.h>

int main( ) {
   int n;
   scanf("%d", &n);

   int mat[n][n];
   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
         scanf("%d", &mat[i][j]);
      }
   }

   int diag1 = 0, diag2 = 0;
   for (int i = 0; i < n; ++i) {
      diag1 += mat[i][i];
      diag2 += mat[i][n - 1 - i];
   }

   int res = (diag1 == diag2);
   for (int i = 0; i < n; ++i) {
      int sf = 0, sc = 0;
      for (int j = 0; j < n; ++j) {
         sf += mat[i][j];
         sc += mat[j][i];
      }
      if (sf != diag1 || sc != diag1) {
         res = 0;
      }
   }

   printf("%d", res);
   return 0;
}
```

6. Resuelve el problema `https://omegaup.com/arena/problem/med`

**Solución:** Una matriz de `n` × `m`. tiene exactamente `n + m - 1` contradiagonales. Una estrategia muy sencilla de implementar consiste en calcular, para cada contradiagonal, las coordenadas de su celda sobre la fila 0 (aunque esta celda se salga de la matriz) y después luego visitar la contradiagonal completa, pero sólo escribiendo sobre celdas de la matriz que sí sean válidas.

```
#include <stdio.h>

int main( ) {
   int n, m;
   scanf("%d%d", &n, &m);

   int mat[n][m], numero = 1;
```

```
   for (int d = 0; d < n + m - 1; ++d) {
      for (int i = 0, j = d; i < n && j >= 0; ++i, --j) {
         if (j < m) {
            mat[i][j] = numero++;
         }
      }
   }

   for (int i = 0; i < n; ++i) {
      for (int j = 0; j < m; ++j) {
         printf("%d ", mat[i][j]);
      }
      printf("\n");
   }
}
```

7. Resuelve el problema `https://omegaup.com/arena/problem/svc`

**Solución:** Este problema conviene apartar una matriz de tres dimensiones, donde cada elemento de la matriz denote una unidad cúbica del espacio tridimensional del problema. Inicializaremos la matriz con 0 y luego sobreescribiremos con 1 cada unidad cúbica que ocupen los cubos. El resultado corresponde con la cantidad de 1 que quedan en la matriz después de procesar todos los cubos. La primera esquina cada cubo no necesariamente es la más cercana al origen, por lo que debemos preprocesar sus coordenadas para facilitar la implementación de los ciclos `for` que visitan el espacio ocupado. Un inconveniente más es que las esquinas de los cubos pueden tener coordenadas negativas; la idea se reubicar todos los cubos al cuadrante no negativio del espacio para poder acceder a los elementos de la matriz sin problemas. Otro posible problema es que la matriz que denota el espacio tridimensional es bastante grande y puede haber problemas al ejecutar el programa en Windows. Uno de los efectos de la palabra reservada `static` es que las limitaciones de memoria en la pila del proceso ya no aplican para ella.

---

```
#include <stdio.h>

int min(int a, int b) {
   return (a < b ? a : b);
}

int max(int a, int b) {
   return (a > b ? a : b);
}

int main( ) {
   int n;
   scanf("%d", &n);

   static int espacio[200 + 1][200 + 1][200 + 1] = { };
   for (int i = 0; i < n; ++i) {
      int x1, y1, z1, x2, y2, z2;
      scanf("%d%d%d%d%d%d", &x1, &y1, &z1, &x2, &y2, &z2);
      for (int dx = min(x1, x2); dx < max(x1, x2); ++dx) {
         for (int dy = min(y1, y2); dy < max(y1, y2); ++dy) {
            for (int dz = min(z1, z2); dz < max(z1, z2); ++dz) {
               espacio[dx + 100][dy + 100][dz + 100] = 1;
```

```
            }
         }
      }
   }

   int res = 0;
   for (int i = 0; i <= 200; ++i) {
      for (int j = 0; j <= 200; ++j) {
         for (int k = 0; k <= 200; ++k) {
            res += espacio[i][j][k];
         }
      }
   }
   printf("%d", res);

   return 0;
}
```

## Soluciones de 16.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Consonantes-y-vocales`

**Solución:** Para resolver este problema, basta leer una cadena y comparar cada uno de sus caracteres con las literales de las vocales mayúsculas. Hay que recordar apartar un arreglo de 80 + 1 caracteres (por el nulo) y que la longitud de la cadena leída se puede calcular con `strlen`.

---

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

int main( ) {
   char a[80 + 1];
   scanf("%s", a);

   int t = strlen(a), c = 0, v = 0;
   for (int i = 0; i < t; ++i) {
      if (a[i] == 'A' || a[i] == 'E' || a[i] == 'I' ||
                         a[i] == 'O' || a[i] == 'U') {
         v += 1;
      } else {
         c += 1;
      }
   }

   printf("%d %d", c, v);
   return 0;
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/El-caballo-de-John-Carter`

**Solución:** Para resolver este problema, basta leer una cadena y comparar cada uno de sus caracteres con las literales del terreno, sumando en un contador el tiempo de recorrido que corresponda.

---

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

int main( ) {
   int p, s, b;
   scanf("%d%d%d", &p, &s, &b);

   char arr[1000 + 1];
   scanf("%s", arr);
   int t = strlen(arr);

   int res = 0;
   for (int i = 0; i < t; ++i) {
      if (arr[i] == '-') {
         res += p;
      } else if (arr[i] == '/') {
         res += s;
      } else if (arr[i] == '\\') {
         res += b;
      }
   }

   printf("%d", res);
   return 0;
}
```

3. Resuelve el problema `https://omegaup.com/arena/problem/Invirtiendo-palabras`

**Solución:** Una vez leída la cadena y calculada su longitud, tendremos una variable cuyo rol será recordar la posición de inicio de la palabra actual. Cuando encontremos una coma, invertiremos toda la subcadena desde tal posición hasta antes de la coma y la nueva posición de inicio será la de adelante de la coma. Debemos recordar hacer la inversión de la última palabra, la cual ya no termina en coma..

---

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

void intercambia(char* a, char* b) {
   char c = *a;
   *a = *b;
   *b = c;
}

void invierte(char arr[], int ini, int fin) {
   for (int ult = fin - 1; ini < ult; ++ini, --ult) {
      intercambia(&arr[ini], &arr[ult]);
   }
}

int main( ) {
   char arr[1000 + 1];
   scanf("%s", arr);
```

```
   int t = strlen(arr), ini = 0;
   for (int i = 0; i < t; ++i) {
      if (arr[i] == ',') {
         invierte(arr, ini, i);
         ini = i + 1;
      }
   }
   invierte(arr, ini, t);

   printf("%s", arr);
   return 0;
}
```

4. Resuelve el problema `https://omegaup.com/arena/problem/Maquina-descompuesta`

**Solución:** En este problema debemos leer una línea y debemos tener una variable que indique el tipo de transformación que haremos en el siguiente caracter a procesar (por ejemplo, una variable que valga 1 si el caracter debemos volverlo mayúscula y 0 si debemos volverlo minúscula). Cada vez que encontremos un punto, debemos reactivar el modo mayúscula y cada vez que convirtamos una letra a mayúscula, debemos desactivarlo. La conversión a mayúsculas o minúsculas se puede hacer con las funciones `toupper` y `tolower` de `ctype.h`.

---

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

int main( ) {
   char arr[80 + 1];
   scanf("%[^\n]", arr);
   int t = strlen(arr);
   int mayus = 1;

   for (int i = 0; i < t; ++i) {
      if (arr[i] == '.') {
         mayus = 1;
      } else if (isalpha(arr[i])) {
         if (mayus == 1) {
            arr[i] = toupper(arr[i]);
            mayus = 0;
         } else {
            arr[i] = tolower(arr[i]);
         }
      }
   }

   printf("%s", arr);
   return 0;
}
```

5. Resuelve el problema `https://omegaup.com/arena/problem/Ordenando-las-letras-de-la-linea`

**Solución:** Existen varias formas de resolver este problema, pero una de ellas es la siguiente. Lo primero que haremos es contar cuántas veces apararece cada caracter en la cadena. Esto se puede hacer recordando que cada caracter a final de cuentas es un entero que podemos usar como índice sobre otro arreglo que guarde las cuentas. Afortunadamente, las letras en la tabla ASCII aparecen ordenadas. Recorreremos la cadena nuevamente y cada vez que encontremos una letra, la sobreescribiremos con la menor letra que sabemos que sí aparecía según la tabla de cuentas (decrementaremos la cuenta de esa letra para denotar que ya la usamos una vez).

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

int main( ) {
   char s[100 + 1];
   scanf("%[^\n]", s);
   int t = strlen(s);

   int tabla[128] = { };
   for (int i = 0; i < t; ++i) {
      tabla[s[i]] += 1;
   }

   char letra = 'a';
   for (int i = 0; i < t; ++i) {
      if (isalpha(s[i])) {
         while (tabla[letra] == 0) {
            ++letra;
         }
         tabla[letra] -= 1;
         s[i] = letra;
      }
   }

   printf("%s", s);
   return 0;
}
```

## Soluciones de 17.1

1. Resuelve el problema `https://omegaup.com/arena/problem/El-lado-mas-corto`

**Solución:** Aunque este problema se puede resolver sin estructuras, la definición de un `struct punto` permite reducir el número de variables involucradas en la distintas expresionas usadas en el código, más allá de la lectura de la entrada.

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

typedef struct {
   float x, y;
} punto;
```

```
float distancia(punto p1, punto p2) {
   float dx = p1.x - p2.x;
   float dy = p1.y - p2.y;
   return sqrt(dx * dx + dy * dy);
}

float min(float a, float b) {
   return (a < b ? a : b);
}

int main( ) {
   punto p1, p2, p3, p4;
   scanf("%f%f%f%f%f%f%f%f", &p1.x, &p1.y, &p2.x, &p2.y,
                             &p3.x, &p3.y, &p4.x, &p4.y);
   printf("%f\n", min(
      min(distancia(p1, p2), distancia(p2, p3)),
      min(distancia(p3, p4), distancia(p4, p1))
   ));
}
```

2. Resuelve el problema `https://omegaup.com/arena/problem/Consultando-Registros`

**Solución:** Aunque este problema se puede resolver sin estructuras, la definición de un `struct usuario` permite modelar el problema en términos de un ordenamiento de estructuras mediante intercambios (también de estructuras), en donde simplemente comparamos un miembro específico de la estructura (en este caso, la edad).

---

```
#include <stdio.h>

typedef struct {
   char id[8 + 1];
   float promedio;
   int edad;
} usuario;

void intercambia(usuario* a, usuario* b) {
   usuario c = *a;
   *a = *b;
   *b = c;
}

void ordena2(usuario* a, usuario* b) {
   if (a->edad > b->edad) {
      intercambia(a, b);
   }
}

int main( ) {
   int n;
   scanf("%d", &n);

   usuario arr[n];
   for (int i = 0; i < n; ++i) {
```

```
      scanf("%s%f%d", &arr[i].id, &arr[i].promedio, &arr[i].edad);
   }
   for (int k = 0; k < n - 1; ++k) {
      for (int i = 0; i < n - 1; ++i) {
         ordena2(&arr[i], &arr[i + 1]);
      }
   }

   int m;
   scanf("%d", &m);
   if (m < 0 || m >= n) {
      printf("ERROR");
   } else {
      printf("%s %.2f %d", arr[m].id, arr[m].promedio, arr[m].edad);
   }

   return 0;
}
```

## Soluciones de 18.1

1. Resuelve el problema `https://omegaup.com/arena/problem/Anyos-con-dos-y-cuatro-digitos`

**Solución:** Para resolver este problema, lo más fácil es usar `scanf` para extraer los enteros y las diagonales que aparecen en cada fecha, verificando el valor de retorno para saber si se tuvo éxito.

---

```
#include <stdio.h>

int main( ) {
   int d, m, a;
   while (scanf("%d/%d/%d", &d, &m, &a) == 3) {
      printf("%d/%d/%d\n", d, m, a + (a >= 74 ? 1900 : 2000));
   }
   return 0;
}
```