Taller de Lenguaje C

Eduardo Grosclaude

2014-12-09

[2015/01/06, 10:16:21- Material en preparación, se ruega no imprimir mientras aparezca esta nota]

Índice general

Capítulo 1

Introducción al Lenguaje C

El lenguaje de programación C fue creado por **Dennis Ritchie** en 1972 en Bell Telephone Laboratories, con el objetivo de reescribir un sistema operativo, el UNIX, en un lenguaje de alto nivel, para poder adaptarlo (es decir, *portarlo*) a diferentes arquitecturas. Por este motivo sus creadores se propusieron metas de diseño especiales, tales como:

- Poder utilizar todos los recursos del hardware ("acceso al bajo nivel").
- Obtener código generado eficiente en uso de memoria y en tiempo de ejecución (programas pequeños y veloces).
- Compilador portable (implementable en cualquier arquitectura).

Actualmente existen implementaciones de C para todas las arquitecturas y sistemas operativos, y es el lenguaje más utilizado para la **programación de sistemas**. Por su gran eficiencia resulta ideal para la programación de **sistemas operativos**, **drivers de dispositivos**, **herramientas de programación**. El 95 % del sistema operativo UNIX está escrito en C, así como gran parte de los modernos sistemas y ambientes operativos, y los programas de administración o aplicación que corren sobre ellos.

1.1 Características del lenguaje

C es un lenguaje compilado. A nivel sintáctico, presenta grandes similitudes formales con Pascal, pero las diferencias entre ambos son importantes. A pesar de permitir **operaciones de bajo nivel**, tiene las **estructuras de control**, y permite la **estructuración de datos**, propias de los lenguajes procedurales de alto nivel.

Minimalidad

Un programa en C es, por lo general, más **sintético** que en otros lenguajes procedurales; la idea central que atraviesa todo el lenguaje es la minimalidad. La definición del lenguaje consta de muy pocos elementos, y tiene muy pocas **palabras reservadas**.

Como rasgo distintivo, en C no existen, rigurosamente hablando, funciones o procedimientos de uso general del programador. Por ejemplo, **no tiene funciones de entrada/salida**; la definición del lenguaje apenas alcanza a **las estructuras de control y los operadores**. La idea de definir un lenguaje sin funciones es, por un lado, hacer posible que el compilador sea **pequeño, fácil de escribir e inmediatamente portable**; y por otro, permitir que sea el usuario quien defina sus propias funciones cuando el problema de programación a resolver tenga requerimientos especiales. El usuario puede escribir sus propios procedimientos (llamados **funciones** aunque no devuelvan valores). Aunque existe la noción de **bloque** de sentencias

(sentencias encerradas entre llaves), el lenguaje se dice ***no* estructurado en bloques** porque no pueden definirse funciones dentro de otras.

Versatilidad... a un precio

El lenguaje entrega completamente el control de la máquina subyacente al programador, no realizando controles de tiempo de ejecución. Es decir, no verifica condiciones de error comunes como ***overflow* de variables**, **errores de entrada/salida**, o **consistencia de argumentos** en llamadas a funciones. Ofrece una gran libertad sintáctica al programador. No es fuertemente tipado. Cuando es necesario, se realizan **conversiones automáticas de tipo** en las asignaciones, a veces **con efectos laterales inconvenientes** si no se tiene precaución. Una función que recibe determinados parámetros formales puede ser invocada con argumentos reales de otro tipo.

Se ha dicho que estas características "liberales" posibilitan la realización de proyectos complejos con más facilidad que otros lenguajes como Pascal o Ada, más estrictos; aunque al mismo tiempo, así resulta más difícil detectar errores de programación en tiempo de compilación. Como resultado, es frecuente que el principiante, y aun el experto, cometan errores de programación que no se hacen evidentes enseguida, ocasionando problemas y costos de desarrollo. En este sentido, según los partidarios de la tipificación estricta, C no es un buen lenguaje. Gran parte del esfuerzo de desarrollo del estándar ANSI se dedicó a dotar al C de elementos para mejorar esta deficiencia.

Una característica especial del lenguaje C es que el **pasaje de argumentos a funciones** se realiza siempre **por valor**. ¿Qué ocurre cuando una función debe **modificar** datos que recibe como argumentos? La única salida es pasarle -por valor- la dirección del dato a modificar. Las consecuencias de este hecho son más fuertes de lo que parece a primera vista, ya que surge la necesidad de todo un conjunto de técnicas de **manejo de punteros** que no siempre son bien comprendidas por los programadores poco experimentados, y abre la puerta a sutiles y escurridizos errores de programación. Quizás este punto, junto con el de la ausencia de comprobaciones en tiempo de ejecución, sean los que le dan al C fama de "difícil de aprender".

Portabilidad

Los **tipos de datos** no tienen un tamaño determinado por la definición del lenguaje, sino que diferentes implementaciones pueden adoptar diferentes convenciones. Paradójicamente, esta característica obedece al objetivo de lograr la **portabilidad** de los programas en C. El programador está obligado a no hacer ninguna suposición sobre los tamaños de los **objetos de datos**, ya que lo contrario haría al software dependiente de una arquitectura determinada ("no portable").

Biblioteca Standard

Pese a no estar formalmente definidas funciones de entrada/salida en el lenguaje, se ha establecido un conjunto mínimo de funciones, llamado la **Biblioteca Standard** del lenguaje C, que todos los compiladores proveen, a veces con agregados. La filosofía de la Biblioteca Standard es la portabilidad, es decir, casi no incluye funciones que sean específicas de un sistema operativo determinado. Aquellas que sí incluye están orientadas a la programación de sistemas, y a veces no resultan suficientes para el programador de aplicaciones. No provee, por ejemplo, la capacidad de manejo de archivos indexados, ni funciones de entrada/salida interactiva por consola que sean seguras ("a prueba de usuarios"). Esta deficiencia se remedia utilizando bibliotecas de funciones "de terceras partes" (creadas por el usuario u obtenidas de otros programadores).

Las funciones de la Biblioteca Standard no tienen ningún privilegio sobre las del usuario y **sus nombres no son palabras reservadas**; el usuario puede reemplazarlas por sus propias funciones simplemente dándoles el mismo nombre.

Paradigma procedural

Por último, el C **no es un lenguaje orientado a objetos**, sino que adhiere al paradigma tradicional de **programación imperativa o procedural**. No soporta la orientación a objetos propiamente dicha, al no proporcionar herramientas fundamentales, como la herencia. Sin embargo, algunas características del lenguaje permiten que un proyecto de programación se beneficie de todas maneras con la aplicación de algunos principios de la orientación a objetos, tales como el ocultamiento de información y el encapsulamiento de responsabilidades. El lenguaje C++, orientado a objetos, **no** es una versión más avanzada del lenguaje o un compilador de C con más capacidades, sino que **se trata de un lenguaje completamente diferente**.

1.2 Evolución del lenguaje

La primera definición oficial del lenguaje fue dada en 1978 por **Brian Kernighan y Dennis Ritchie** (Fig. 1.1) en su libro **El lenguaje de programación C**. Este lenguaje fue llamado **C K&R**, por las iniciales de sus autores. En 1983 se creó el comité ANSI para el lenguaje, que en 1988 estableció el estándar ANSI C, con algunas reformas sobre el C K&R. Simultáneamente, Kernighan y Ritchie publicaron la segunda edición de su libro, describiendo la mayor parte de las características del ANSI C.

Figura 1.1: Brian Kernighan y Dennis Ritchie, los creadores del lenguaje C

Algunas nuevas características de C99 son:

- Matrices de tamaño variable
- Soporte de números complejos
- Tipos `long long int` y `unsigned long long int` de al menos 64 bits
- Familia de funciones `vscanf()`
- Comentarios al estilo de C++ prefijando las líneas con la secuencia `//`.
- Familia de funciones `snprintf()`
- Tipo boolean

1.3 El ciclo de compilación

Las herramientas esenciales de un ambiente de desarrollo, además de cualquier **editor de textos**, son el **compilador**, el **vinculador**, **linkeditor** o *linker*, y el **bibliotecario**. A estas herramientas básicas se agregan

otras, útiles para automatizar la compilación de proyectos extensos, almacenar y recuperar versiones de programas fuente, comprobar sintaxis en forma previa a la compilación, etc. Según el ambiente operativo y producto de software de que se trate, estas herramientas pueden ser comandos de línea independientes, con salidas de texto simples, o encontrarse integradas en una interfaz de usuario uniforme, en modo texto o modo gráfico.

Cuando encontramos varias de estas herramientas integradas en una sola aplicación, decimos que se trata de un **IDE** (*Integrated Development Environment*) o ambiente de desarrollo integrado. Un IDE oculta el ciclo de compilación al usuario, con la intención de simplificar el proceso de desarrollo. Sin embargo, conviene conocer qué función se cumple, y qué producto se espera, en cada fase del ciclo de compilación, para poder interpretar las diferentes situaciones de error y poder corregirlas.

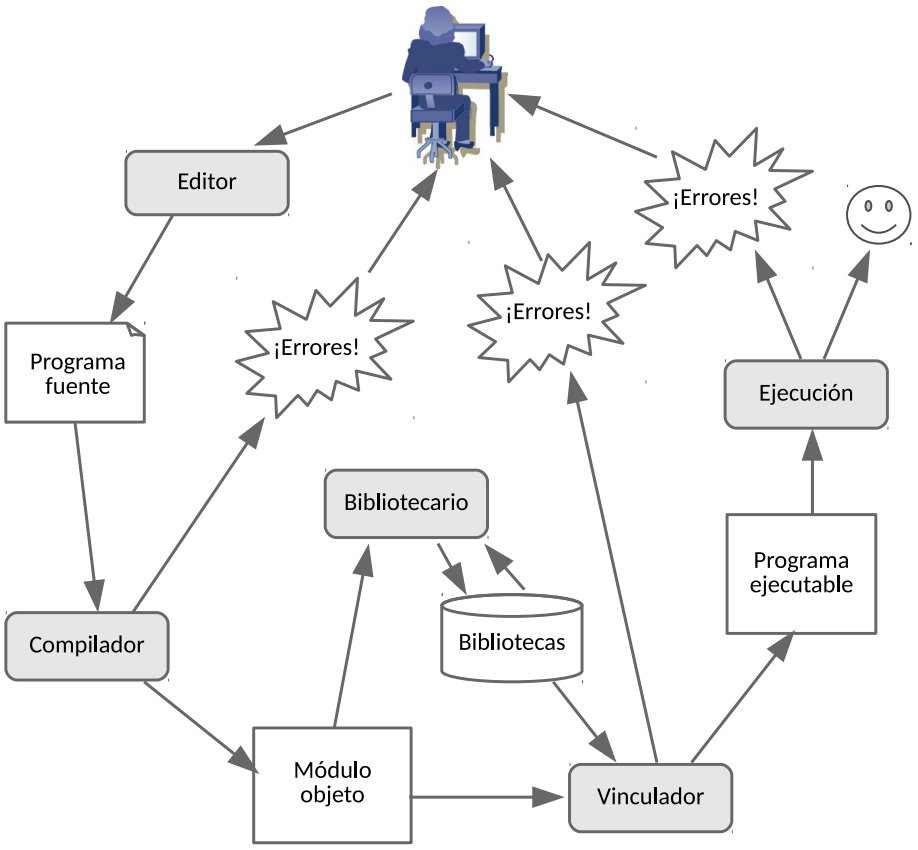

Figura 1.2: El ciclo de compilación produce un ejecutable a partir de archivos fuente.

Compilador

- El compilador acepta un archivo **fuente**, posiblemente relacionado con otros (una **unidad de traducción**), y genera con él un **módulo objeto**. Este módulo objeto contiene porciones de código ejecutable mezclado con **referencias**, aún no resueltas, a variables o funciones cuya definición no figura en los fuentes de entrada. Estas referencias quedan en forma simbólica en el módulo objeto hasta que se resuelvan en un paso posterior.

- Si ocurren errores en esta fase, se deberán a problemas de sintaxis (el código escrito por el programador no respeta la definición del lenguaje).

Vinculador, linkeditor o *linker*

- El vinculador recibe como entrada un conjunto de módulos objeto y busca **resolver**, vincular, o enlazar, las referencias simbólicas en ellos, buscando la definición de las variables o funciones faltantes en los mismos objetos o en bibliotecas. Éstas pueden ser la Biblioteca Standard, u otras provistas por el usuario. Cuando el linker encuentra la definición de un objeto buscado (es decir, de una variable o función), la copia en el archivo resultante de salida (la *resuelve*). El objetivo del linker es resolver todas las referencias pendientes para producir un programa ejecutable.

- Si ocurren errores en esta fase, se deberán a que existen variables o funciones cuya definición no ha sido dada (no se encuentran en las unidades de traducción procesadas, ni en ninguna biblioteca conocida por el linker).

Bibliotecario

- El bibliotecario es un programa administrador de módulos objeto. Su función es reunir módulos objeto en archivos llamados **bibliotecas**, y luego permitir la extracción, borrado, reemplazo y agregado de módulos. El conjunto de módulos en una biblioteca se completa con una tabla de información sobre sus contenidos para que el linker pueda encontrar rápidamente aquellos módulos donde se ha definido una variable o función, y así extraerlos durante el proceso de linkedición.

- El bibliotecario es utilizado por el usuario cuando desea mantener sus propias bibliotecas. La creación de bibliotecas propias del usuario ahorra tiempo de compilación y permite la distribución de software sin revelar la forma en que se han escrito los fuentes y protegiéndolo de modificaciones.

Una vez que el código ha sido compilado y vinculado, obtenemos un programa ejecutable. Los errores que pueden producirse en la ejecución ya no corresponden a problemas de compilación, sino que se deben a aspectos de diseño del programa que deben ser corregidos por el programador.

1.4 Un primer ejemplo

El clásico ejemplo de todas las introducciones al lenguaje C es un programa llamado "**Hello, World!**".

```
#include <stdio.h>
/* El primer programa! */
main()
{
    printf("Hola, gente!\n");
}
```

Funcionamiento del programa

- Este programa minimal comienza con una **directiva de preprocesador** que indica incluir en la unidad de traducción al archivo de cabecera o *header* **stdio.h**. Éste contiene, entre otras cosas, la declaración (o **prototipo**) de la función de salida de caracteres **printf()**, perteneciente a la Biblioteca Standard. Los prototipos se incluyen para advertir al compilador de los tipos de las funciones y de sus argumentos.

- Entre los pares de caracteres especiales **/*** y ***/**se puede insertar cualquier cantidad de líneas de comentarios.

- La función **main()** es el cuerpo principal del programa (es por donde comenzará la ejecución). Todas las funciones en C están delimitadas por un par de llaves. Terminada la ejecución de main(), terminará el programa.
- La función `printf()` imprimirá la cadena entre comillas, que es una **constante string** terminada por un carácter de **nueva línea** (la secuencia especial "`n`").

Compilación del programa

Para ver el primer ejemplo en C en funcionamiento:

1. Copiar el programa con cualquier editor de textos y guardarlo en un archivo llamado `hola.c` en el directorio de trabajo del usuario.
2. Sin cambiar de directorio, invocar al compilador ejecutando el comando `gcc hola.c -o hola`. Por defecto, el compilador **gcc** invocará al vinculador **ld** para generar el ejecutable a partir del archivo objeto intermedio generado.
3. Ejecutar el programa con el comando `./hola`.

Notar el *punto y barra* del principio al ejecutar el programa. El punto y barra le indican al **shell** que debe buscar el programa en el directorio activo.

Lo mismo, pero de otra manera:

1. Como antes, copiar el programa, o usar el mismo archivo fuente de hace un momento.
2. Sin cambiar de directorio, ejecutar el comando `make hola` en una consola o terminal.
3. Ejecutar el programa con el comando `./hola`.

La diferencia es que en el primer caso invocamos directamente al compilador `gcc`, mientras que en el segundo caso utilizamos la herramienta `make`, que nos asiste en la compilación de proyectos. En el ejemplo, le decimos al compilador que procese el archivo fuente `hola.c`, y que el ejecutable de salida (opción `-o`, de *output*) se llame `hola`.

El comando make

Cuando damos un comando como `make hola`, utilizamos el comando **make** para compilar y vincular el programa **hola.c**. El comando **make** contiene la inteligencia para:

- buscar, en el directorio activo, archivos fuente llamados **hola.***;
- determinar (a partir de la extensión) que el hallado se trata de un programa en C;
- ver que no existe en el directorio activo un programa ejecutable llamado **hola**, o que, si existe, su fecha de última modificación es anterior a la del fuente;
- razonar que, por lo tanto, es necesario compilar el fuente **hola.c** para producir el ejecutable **hola**;
- e invocar con una cantidad de opciones por defecto al compilador **gcc**, y renombrar la salida con el nombre **hola**. Éste será el ejecutable que deseamos producir.

Si se invoca al comando **make** una segunda vez, éste comprobará, en base a las fechas de modificación de los archivos fuente y ejecutable, que no es necesaria la compilación (ya que el ejecutable es posterior al fuente). Si editamos el fuente para cambiar algo en el programa, invocar nuevamente a make ahora repetirá la compilación (porque ahora el ejecutable es anterior al fuente).

1.5 Mapa de memoria de un programa

Luego de la compilación y vinculación, el programa ejecutable queda contenido en un archivo. Al ser invocado, el sistema operativo lo carga en memoria, y allí se despliega en una cantidad de secciones de diferentes tamaños y con distintas funciones.

Los sistemas operativos modernos, salvo raras excepciones, administran la memoria física usando sistemas de memoria virtual. Cada sistema de memoria virtual funciona de modo diferente. La manera como se distribuyen realmente las secciones de un programa en la memoria física depende fuertemente de la forma de administración de memoria del sistema operativo para el cual ha sido compilado y vinculado. Sin embargo, el siguiente modelo puede servir de referencia para ilustrar algunas particularidades y problemas que irán surgiendo con el estudio del lenguaje.

El programa cargado en memoria (Fig. 1.3) se dividirá en cuatro regiones: código o **texto**, **datos estáticos**, **heap** (o región de datos dinámicos), y **stack** (o pila).

El tamaño de las regiones de código y de datos estáticos está determinado al momento de compilación y es inamovible. Las otras dos regiones quedan en un bloque cuyo tamaño inicial es ajustado por el sistema operativo al momento de la carga, pero puede variar durante la ejecución. Este bloque es compartido entre ambas regiones. Una de ellas, la de datos dinámicos, o heap, crece "hacia arriba" (hacia direcciones de memoria más altas); la otra, la pila del programa, o stack, crece "hacia abajo" (en sentido opuesto).

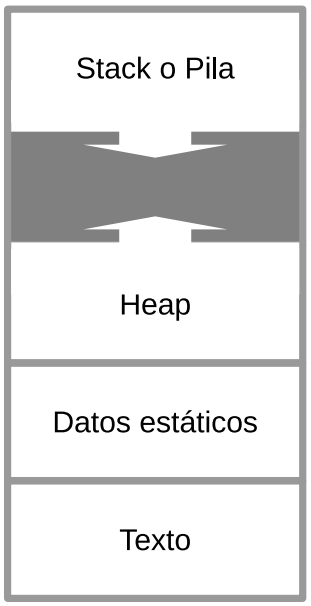

Figura 1.3: El mapa de memoria de un programa se divide conceptualmente en cuatro regiones

Texto del programa La región de texto contendrá el **código del programa**, es decir, la versión ejecutable de las **instrucciones** que escribió el programador, traducidas por el compilador al lenguaje de la máquina. En general, el programa fuente C se compondrá de funciones, que serán replicadas a nivel de máquina por subrutinas en el lenguaje del procesador subyacente. Algunas instrucciones C resultarán en última instancia en invocaciones a funciones del sistema (por ejemplo, cuando necesitamos escribir en un archivo).

Datos estáticos La región de datos estáticos es un lugar de almacenamiento para datos del programa que quedan definidos al momento de la compilación. Se trata de datos cuya vida o instanciación

no depende de la invocación de las funciones. Son las variables estáticas, definidas en el cuerpo del programa que es común a todas las funciones. A su vez, esta zona se divide en dos: la de **datos estáticos inicializados** explícitamente por el programa (zona a veces llamada **bss** por motivos históricos) y la zona de **datos estáticos sin inicializar** (a veces llamada **data**), que será llenada con ceros binarios al momento de la carga del programa.

Stack El stack, o **pila**, aloja las variables locales de las funciones a medida que esas funciones son invocadas. Cada función que declare variables locales obtendrá espacio de almacenamiento para esas variables en el stack. Al terminar la función, como el ámbito de sus variables desaparece, esas variables son desalojadas y el espacio que ocupaban vuelve a quedar disponible.

Heap Un programa C puede utilizar estructuras de datos dinámicas, como listas o árboles, que vayan creciendo al agregárseles elementos. El programa puede "pedir" memoria cada vez que necesite alojar un nuevo elemento de estas estructuras dinámicas, o para crear buffers temporarios para cualquier uso que sea necesario. El **heap** es la zona de donde el programa obtiene esos trozos de memoria, solicitada en forma dinámica al sistema operativo. El límite del heap se irá desplazando hacia las direcciones superiores.

A diferencia de otros lenguajes con administración de memoria automática, en C normalmente es necesario solicitar explícitamente cada segmento de memoria asignada dinámicamente. Es responsabilidad del programador, también, liberar esta memoria cuando no ya sea necesaria, ya que en C no existe un mecanismo de "recolección de basura", lo cual sí existe en otros lenguajes, para desalojar automáticamente objetos que ya no se utilicen.

Por otro lado, un programa que realice una cadena de invocaciones de muchas funciones, y especialmente si éstas utilizan muchas variables locales, hará crecer notablemente su stack, desplazando el tope de la pila hacia abajo. La región del stack es el lugar para la creación y destrucción de variables locales, que son aquellas que viven mientras tiene lugar la ejecución de la función a la que pertenecen. La destrucción de estas variables sí es automática, y se produce al momento de finalizar la ejecución de la función.

Este modelo será útil en varias ocasiones para explicar algunas cuestiones especiales del lenguaje C.

Ejemplo 1.1

Analicemos en qué lugares quedarán alojados los elementos del programa siguiente.

```
int a = 1;
int b;
int fun()
{
    int c;
    c = a + b;
}
main()
{
    fun();
}
```

- Las variables a y b corresponden a la zona de datos estáticos. La variable a está inicializada con 1, pero como b no está inicializada, recibirá un valor 0.
- La variable c aparecerá en el stack cuando se ejecute la función fun().
- Las instrucciones de máquina procedentes de la compilación del programa (funciones main() y fun()) serán almacenadas en la región de texto.

1.6 Ejercicios

1. ¿Qué nombres son adecuados para los archivos fuente C?
2. Describa las etapas del ciclo de compilación.
3. ¿Cuál sería el resultado de:
   - Editar un archivo fuente?
   - Ejecutar un archivo fuente?
   - Editar un archivo objeto?
   - Compilar un archivo objeto?
   - Editar una biblioteca?
4. ¿Qué pasaría si un programa en C **no** contuviera una función **main()**? Haga la prueba modificando **hola.c**.
5. Edite el programa `hola.c` y modifíquelo según las pautas que siguen. Interprete los errores de compilación. Identifique en qué etapa del ciclo de compilación ocurren los errores. Si resulta un programa ejecutable, observe qué hace el programa y por qué.
   - Quite los paréntesis de main().
   - Quite la llave izquierda de main().
   - Quite las comillas izquierdas.
   - Quite los caracteres "\n".
   - Agregue al final de la cadena los caracteres "\n\n\n\n".
   - Agregue al final de la cadena los caracteres "\nAdiós, mundo!\n".
   - Quite las comillas derechas.
   - Quite el signo punto y coma.
   - Quite la llave derecha de main().
   - Agregue un punto y coma en cualquier lugar del texto.
   - Agregue una coma o un dígito en cualquier lugar del texto.
   - Reemplace la palabra **main** por **program**, manteniendo los paréntesis.
   - Elimine la apertura o cierre de los comentarios.

Capítulo 2

El preprocesador

El compilador C tiene un componente auxiliar llamado **preprocesador**, que actúa en la primera etapa del proceso de compilación. Su misión es buscar, en el texto del programa fuente entregado al compilador, ciertas **directivas** que le indican realizar alguna tarea a nivel de texto. Por ejemplo, **inclusión** de otros archivos, o **sustitución** de ciertas cadenas de caracteres (**símbolos** o **macros**) por otras.

El preprocesador cumple estas directivas en forma similar a como podrían ser hechas interactivamente por el usuario, utilizando los comandos de un editor de texto ("incluir archivo" o "reemplazar texto"), pero en forma automática. Una vez cumplidas todas estas directivas, el preprocesador entrega el texto resultante al resto de las etapas de compilación, que terminarán dando por resultado un módulo objeto. Un archivo fuente, junto con todos los archivos que incluya, es llamado una **unidad de traducción**.

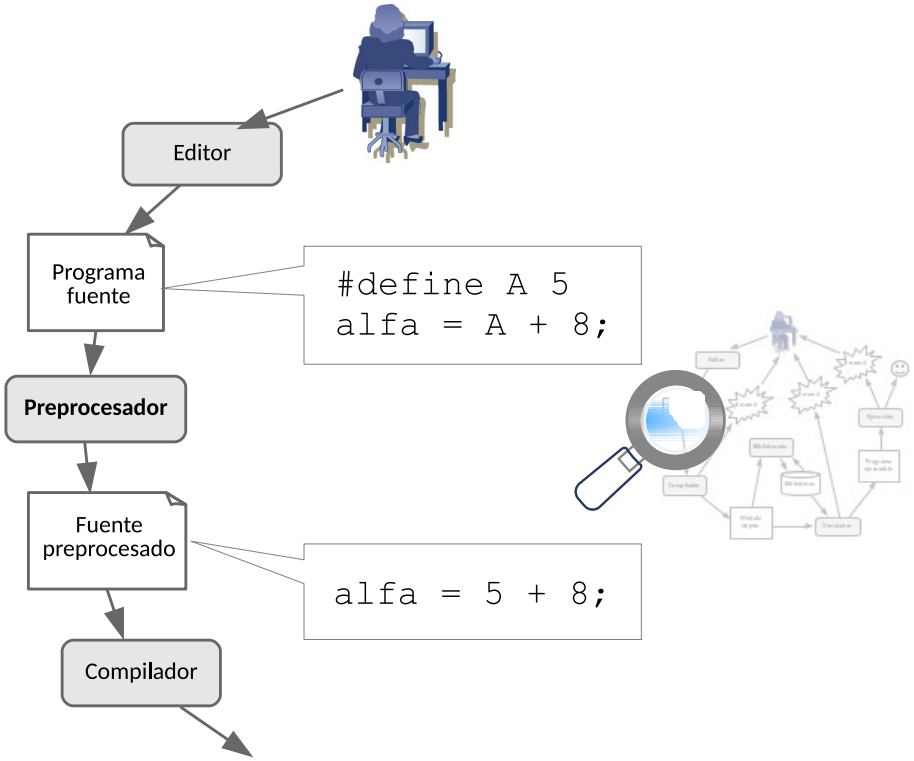

Figura 2.1: El preprocesador realiza ediciones automáticas de los fuentes.

El preprocesador sirve para eliminar redundancia y aumentar la expresividad de los programas en C,

facilitando su mantenimiento. Si una variable o función se utiliza en varios archivos fuente, es posible aislar su declaración, colocándola en un único archivo aparte que será incluido al tiempo de compilación en los demás fuentes. Esto facilita toda modificación de elementos comunes en los fuentes de un proyecto. Por otro lado, si una misma constante o expresión aparece repetidas veces en un texto, y es posible que su valor deba cambiarse más adelante, es muy conveniente definir esa constante con un símbolo y especificar su valor sólo una vez, mediante un símbolo o macro.

2.1 Directivas de preprocesador

Las directivas del preprocesador no pertenecen al lenguaje C en un sentido estricto. El preprocesador **no comprende ningún aspecto sintáctico ni semántico** de C. Las **macros** definidas en un programa C **no son variables ni funciones**, sino simplemente cadenas de texto que el preprocesador deberá sustituir por otras. Las directivas pueden aparecer en cualquier lugar del programa, pero sus efectos se ponen en vigor recién a partir del punto del programa en que aparecen, y hasta el final de la unidad de traducción. Es decir, un símbolo o macro puede utilizarse sólo después de la aparición de la directiva que la define, y no antes. Tampoco puede utilizarse en una unidad de traducción diferente, salvo que vuelva a ser definida en ella (los símbolos de preprocesador no se "propagan" entre unidades de traducción).

Símbolos y macros

Una de las funciones del preprocesador es sustituir símbolos, o cadenas de texto dadas, por otras (Fig. 2.1). La directiva `define` establece la relación entre los símbolos y su **expansión**, o cadena a sustituir. Los símbolos indicados con una directiva de definición `define` se guardan en una **tabla de símbolos** durante el preprocesamiento.

Habitualmente se llama **símbolos** a aquellas cadenas que son directamente sustituibles por una expresión, reservándose el nombre de **macros** para aquellos símbolos cuya expansión es parametrizable (es decir, llevan argumentos formales y reales como en el caso de las funciones). La cadena de expansión puede ser cualquiera, no necesariamente un elemento sintácticamente válido de C.

Ejemplo 2.1
La Fig. 2.2 muestra el programa ejemplo **hola.c** escrito usando directivas de inclusión de archivos, símbolos y macros, y las sucesivas transformaciones que hará el preprocesador.

El texto del programa una vez preprocesado quedará idéntico al del programa **hola.c** anteriormente presentado, y listo para ingresar a la etapa de compilación propiamente dicha.

Headers

Las directivas para incluir archivos suelen darse al principio de los programas, ya que en general se desea que su efecto alcance a todo el archivo fuente. Por esta razón los archivos preparados para ser incluidos se denominan *headers* o archivos de cabecera. La implementación de la Biblioteca Standard que viene con un compilador posee sus propios headers, uno por cada módulo de la biblioteca, que **declaran** funciones y variables de uso general. Estos headers contienen texto legible por humanos, y están en algún subdirectorio predeterminado (llamado `/usr/include` en UNIX, y dependiendo del compilador en otros sistemas operativos). El usuario puede escribir sus propios headers, y no necesita ubicarlos en el directorio reservado del compilador; puede almacenarlos en el directorio activo durante la compilación.

En el párrafo anterior, nótese que decimos **declarar** funciones, y no **definirlas**; la diferencia es importante y se verá en profundidad más adelante. Recordemos por el momento que **en los headers** de la Biblioteca Standard no aparecen **definiciones** -es decir, textos- de funciones, sino solamente **declaraciones o prototipos**, que sirven para anunciar al compilador detalles como los tipos y cantidad de los argumentos de las funciones.

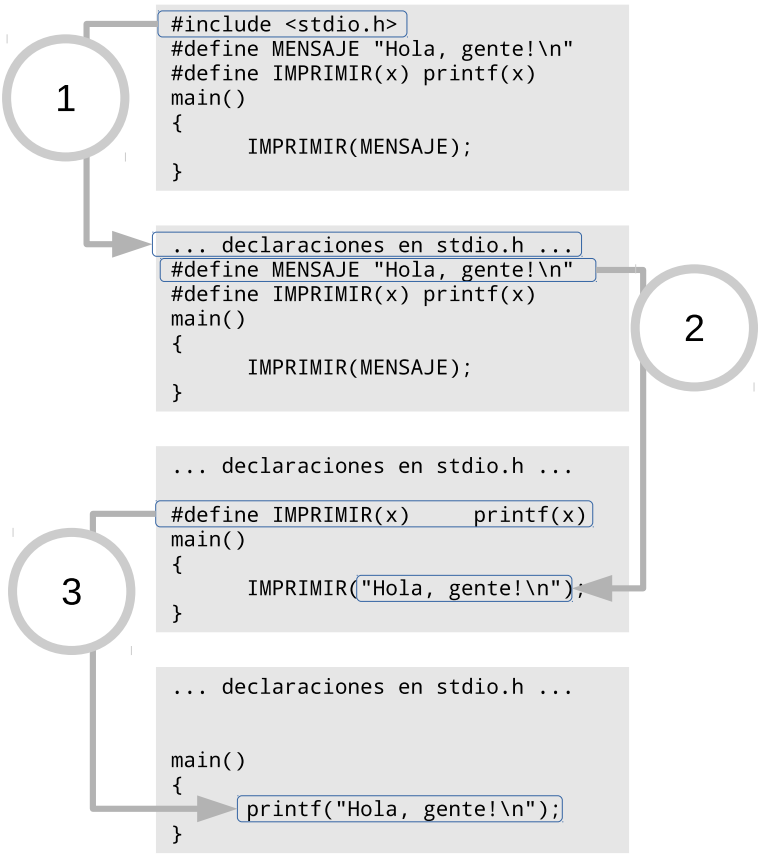

Figura 2.2: Transformaciones realizadas por directivas de preprocesador.

No se considera buena práctica de programación colocar la definición de una función de uso frecuente en un header. Esto obligaría a recompilar siempre la función cada vez que se la utilizara. Por el contrario, lo ideal sería compilarla una única vez, produciendo un módulo objeto (y posiblemente incorporándolo a una biblioteca). Esto ahorraría el tiempo correspondiente a su compilación, ocupando sólo el necesario para la vinculación.

2.2 Definición de símbolos

Si el programa dice:

```
a = 2 * 3.14159 * 20.299322;
```

Es mucho más claro poner, en su lugar:

```
#define PI 3.14159
#define RADIO 20.299322

a = 2 * PI * RADIO;
```

2.3 Definición de macros

Con las siguientes directivas:

```
#include <stdio.h>
#include "aux.h"
#define MAXITEM 100
#define DOBLE(X) 2*X
```

- Se incluye el header de biblioteca standard `stdio.h`, que contiene declaraciones necesarias para poder utilizar funciones de entrada/salida standard (hacia consola y hacia archivos).
- Se incluye un header `aux.h` escrito por el usuario. Al indicar el nombre del header entre ángulos, como en la línea anterior, especificamos que la búsqueda debe hacerse en los directorios reservados del compilador. Al indicarlo entre comillas, nos referimos al directorio actual.
- Se define un símbolo MAXITEM equivalente a la constante numérica 100.
- Se define una macro DOBLE(X) que deberá sustituirse por la cadena 2*(argumento de la llamada a la macro).

De esta manera, podemos escribir sentencias tales como:

```
a=MAXITEM;
b=DOBLE(45);
```

El texto luego de la etapa de preprocesamiento y antes de la compilación propiamente dicha será

```
a=100;
b=2*45;
```

Macros vs. funciones

Es importante comprender que, aunque sintácticamente parecido, el uso de una macro **no es una llamada a función**; los argumentos de una macro no se evalúan en tiempo de ejecución antes de la llamada, sino que **se sustituyen textualmente** en el cuerpo de la macro. Así, si ponemos

```
#define DOBLE(X) 2*X
b=DOBLE(40+5);
```

el resultado será `b=2*40+5`; y no `b=2*45`, ni `b=2*(40+5)`, que presumiblemente es lo que desea el programador.

Este problema puede solucionarse redefiniendo la macro así:

```
#define DOBLE(X) 2*(X)
b=DOBLE(40+5);
```

Ahora la expansión de la macro será la deseada. En general, es saludable rodear las apariciones de los argumentos de las macros entre paréntesis, para obligar a su evaluación al tiempo de ejecución con la precedencia debida, y evitar efectos laterales.

2.4 Compilación condicional

Una característica interesante del preprocesador es que permite la **compilación condicional** de segmentos de la unidad de traducción, en base a valores de símbolos. Una directiva condicional es aquella que comprueba si un símbolo dado ha sido definido, o si su definición coincide con cierta cadena. El texto del programa que figura entre la directiva y su `end` será considerado sólo si la comprobación resulta exitosa. Los símbolos o macros pueden ser definidos al tiempo de la compilación, sin alterar el texto del programa, permitiendo así una parametrización del programa en forma separada de su escritura.

Ejemplo 2.2
Con las directivas condicionales:

- Definimos una macro `CARTEL` que equivaldrá a invocar a una función "imprimir", pero sólo si el símbolo `DEBUG` ha sido definido. En otro caso, equivaldrá a la cadena vacía.

```
#ifdef DEBUG
#define CARTEL(x) imprimir(x)
#else
#define CARTEL(x)
#endif
```

- El segmento siguiente muestra un caso con lógica inversa pero equivalente al ejemplo anterior.

```
#ifndef DEBUG
#define CARTEL(x)
#else
#define CARTEL(x) imprimir(x)
#endif
```

- En el caso siguiente, se incluirá uno u otro header dependiendo del valor del símbolo `SISTEMA`. Tanto `DEBUG` como `SISTEMA` pueden tomar valores al momento de compilación, si se dan como argumentos para el compilador. De esta manera se puede modificar el comportamiento del programa sin necesidad de editarlo.

```
#if SISTEMA==MS_DOS
#include "header1.h"
#elif SISTEMA==UNIX
#include "header2.h"
#endif
```

2.5 Observaciones

- A veces puede resultar interesante, para depurar un programa, observar cómo queda el archivo intermedio generado por el preprocesador después de todas las sustituciones, inclusiones, etc. La mayoría de los compiladores cuentan con una opción que permite generar este archivo intermedio y detener allí la compilación, para poder estudiarlo.

- Otra opción relacionada con el preprocesador que suelen ofrecer los compiladores es aquella que permite definir, al tiempo de la compilación y sin modificar los fuentes, símbolos que se pondrán a la vista del preprocesador. Así, la estructura final de un programa puede depender de decisiones tomadas al tiempo de compilación. Esto permite, por ejemplo, aumentar la portabilidad de los programas, o generar múltiples versiones de un sistema sin diseminar el conocimiento reunido en los módulos fuente que lo componen.

- Finalmente, aunque el compilador tiene un directorio default donde buscar los archivos de inclusión, es posible agregar otros directorios para cada compilación con argumentos especiales si es necesario.

2.6 Ejercicios

1. Dé ejemplos de directivas de preprocesador:
   a. Para incluir un archivo proporcionado por el compilador.
   b. Para incluir un archivo confeccionado por el usuario.
   c. Para definir una constante numérica.
   d. Para compilar un segmento de programa bajo la condición de estar definida una constante.
   e. Idem bajo la condición de ser no definida.
   f. Idem bajo la condición de que un símbolo valga una cierta constante.
   g. Idem bajo la condición de que dos símbolos sean equivalentes.
2. Proponga un método para incluir un conjunto de archivos en un módulo fuente con una sola directiva de preprocesador.
3. ¿Cuál es el ámbito de una definición de preprocesador? Si defino un símbolo A en un fuente y lo compilo creando un módulo objeto algo.o, ¿puedo utilizar A desde otro fuente, sin declararlo, a condición de linkeditarlo con algo.o?
4. ¿Qué pasa si defino dos veces el mismo símbolo en un mismo fuente?
5. Un cierto header A es incluido por otros headers B, C y D. El fuente E necesita incluir a B, C y D. Proponga un método para poder hacerlo sin obligar al preprocesador a leer el header A más de una vez.
6. Edite el programa hello.c del ejemplo del capítulo 1 reemplazando la cadena `"Hola, mundo!\n"` por un símbolo definido a nivel de preprocesador.
7. Edite el programa hello.c incluyendo la compilación condicional de la instrucción de impresión printf() sujeta a que esté definido un símbolo de preprocesador llamado IMPRIMIR. Compile y pruebe a) sin definir el símbolo IMPRIMIR, b) definiéndolo con una directiva de preprocesador, c) definiéndolo con una opción del compilador. ¿En qué casos es necesario recompilar el programa?
8. Escriba una macro que imprima su argumento usando la función printf(). Aplíquela para reescribir hello.c de modo que funcione igual que antes.
9. ¿Cuál es el resultado de preprocesar las líneas que siguen? Es decir, ¿qué recibe exactamente el compilador luego del preprocesado?

```
#define ALFA 8
#define BETA 2*ALFA
#define PROMEDIO(x,y) (x+y)/2
a=ALFA*BETA;
b=5;
c=PROMEDIO(a,b);
```

10. ¿Qué está mal en los ejemplos que siguen?

a.
```
#define PRECIO 27.5
PRECIO=27.7;
```

b.
```
#define 3.14 PI
```

c.
```
#define doble(x) 2*x;
alfa=doble(6)+5;
```

11. Investigue la función de los símbolos predefinidos `__STDC__`, `__FILE__` y `__LINE__`.

Capítulo 3

Tipos de datos y expresiones

En general, las **expresiones** en C se construyen conectando, mediante **operadores**, diversos elementos, tales como **identificadores** de variables, **constantes** e invocaciones de **funciones**. Cada uno de estos elementos tiene un valor al tiempo de ejecución, y debe ocupar -al menos temporariamente, mientras se calcula el resultado de la expresión- un lugar en memoria. Al evaluar cada expresión, el compilador crea, para alojar cada subexpresión de las que la constituyen, **objetos de datos**, que pueden pensarse como espacio de memoria reservado temporariamente para contener valores. Al completar el cálculo de la expresión, el resultado nuevamente debe ser alojado en un objeto de datos propio. Estos espacios de memoria son de diferentes "tamaños" (cantidades de bits) de acuerdo al **tipo de dato** de la subexpresión.

Así, las expresiones y subexpresiones en C asumen siempre un **tipo de datos**: alguno de los tipos básicos del lenguaje, o uno definido por el usuario. Una expresión, según las necesidades, puede convertirse de un tipo a otro. El compilador hace esto a veces en forma **automática**. Otras veces, el programador fuerza una **conversión de tipo** para producir un determinado resultado.

3.1 Declaración de variables

Los **tipos básicos** son:

- `char` (un elemento del tamaño de un byte)
- `int` (un número entero con signo)
- `long` (un entero largo)
- `float` (un número en punto flotante)
- `double` (un número en punto flotante, doble precisión)

Cuando declaramos una variable o forzamos una conversión de tipo, utilizamos una **especificación de tipo**. Ejemplos de declaración de variables:

```
char a;
int alfa,beta;
float x1,x2;
```

Los **tipos enteros** (`char`, `int` y `long`) admiten los modificadores `signed` (con signo) y `unsigned` (sin signo). Un objeto de datos **unsigned** utiliza todos sus bits para representar magnitud; un **signed** utiliza un bit para signo, en representación complemento a dos.

El modificador `signed` sirve sobre todo para explicitar el signo de los chars. El default para un `int` es signed; el default para `char` puede ser signed o unsigned, dependiendo del compilador.

```
unsigned int edad;
signed char beta;
```

Un int puede afectarse con el modificador `short` (corto).

```
short i;
unsigned short k;
```

Cuando en una declaración aparece sólo el modificador unsigned o short, y no el tipo, **se asume int**. El tipo entero se supone el tipo básico manejable por el procesador, y es el tipo por omisión en varias otras situaciones. Por ejemplo, cuando no se especifica el **tipo del valor devuelto** por una función.

El modificador long puede aplicarse también a float y a double. Los tipos resultantes pueden tener más precisión que los no modificados.

```
long float e;
long double pi;
```

3.2 Tamaños de los objetos de datos

El lenguaje C no define el tamaño de los objetos de datos de un tipo determinado. Es decir, un entero puede ocupar 16 bits en una implementación del compilador, 32 en otra, o aun 64. Un long puede tener o no más bits que un int. Un short puede ser o no más corto que un int. Según K&R, lo único seguro es que "*un short no es mayor que un int, que a su vez no es mayor que long*".

Por supuesto, distintos tamaños en bits implican diferentes rangos de valores. Si deseamos **portar** un programa, hecho bajo una implementación del compilador, a otra, no es posible asegurar a priori el rango que tomará un tipo de datos. La fuente ideal para conocer los rangos de los diferentes tipos, en una implementación determinada, es -además del manual del compilador- el header `limits.h` de la Biblioteca Standard. Debe recordarse que cualquier suposición que hagamos sobre el rango o tamaño de un objeto de datos afecta la portabilidad de un programa en C.

Las siguientes líneas son parte de un archivo `limits.h` para una implementación en particular:

```
/* Minimum and maximum values a 'signed short int' can hold. */
#define SHRT_MIN  (-32768)
#define SHRT_MAX  32767
/* Maximum value an 'unsigned short int' can hold. (Minimum is 0.) */
#define USHRT_MAX 65535
/* Minimum and maximum values a 'signed int' can hold. */
#define INT_MIN (-INT_MAX - 1)
#define INT_MAX 2147483647
/* Maximum value an 'unsigned int' can hold. (Minimum is 0.) */
#ifdef __STDC__
#define UINT_MAX  4294967295U
#else
#define UINT_MAX  4294967295
#endif
/* Minimum and maximum values a 'signed long int' can hold. */
#ifdef __alpha__
#define LONG_MAX  9223372036854775807L
#else
#define LONG_MAX  2147483647L
#endif
#define LONG_MIN  (-LONG_MAX - 1L)
```

```
/* Maximum value an 'unsigned long int' can hold. (Minimum is 0.) */
# ifdef __alpha__
#define ULONG_MAX 18446744073709551615UL
#else
#ifdef __STDC__
#define ULONG_MAX  4294967295UL
#else
#define ULONG_MAX 4294967295L
#endif
#endif
```

Cuando una operación sobre una variable provoca overflow, no se obtiene ninguna indicación de error. El valor sufre **truncamiento** a la cantidad de bits que pueda alojar la variable.

Así, en una implementación donde los ints son de 16 bits, si tenemos en una variable entera el máximo valor positivo:

```
int a;
a=32767; /* a=0111111111111111 binario */
a=a+1;
```

Al calcular el nuevo valor de `a`, aparece un 1 en el bit más significativo, lo cual, según la representación de los enteros, lo transforma en un negativo (el menor negativo que soporta el tipo de datos, -32768).

Si el int es sin signo:

```
unsigned a;
a=65535; /* maximo valor de unsigned int */
a=a+1;
```

el incremento de `a` provoca overflow, y el nuevo valor de `a` se trunca a 16 bits, volviendo así a 0.

Siempre se puede saber el tamaño en bits de un tipo de datos aplicando el operador `sizeof()` a una variable o a la especificación de tipo.

3.3 Operaciones con distintos tipos

En una expresión en C pueden aparecer componentes de diferentes tipos. Durante la evaluación de una expresión cuyas subexpresiones sean de **tipos diferentes**, deberá tener lugar una **conversión**, ya sea implícita o explícita, para llevar ambos operandos a un **tipo de datos común** con el que se pueda operar. La forma en que el compilador resuelve las conversiones implícitas a veces provoca algunas sorpresas.

Truncamiento en asignaciones

Para empezar, una asignación de una expresión de un tipo dado a una variable de un tipo "menor" (en el sentido del tamaño en bits de cada objeto de datos), no sólo es permitida en C, sino que la conversión se hace en forma automática y generalmente sin ningún mensaje de tiempo de compilación ni de ejecución. Por ejemplo,

```
int a;
float b;
...
a=b;
```

En esta asignación tenemos miembros de diferentes tamaños. El resultado en `a` será el truncamiento del valor entero de `b` a la cantidad de bits que permita un int. Es decir, se tomará la parte entera de `b`, y de

ese valor se copiarán en el objeto de datos de `a` tantos bits como quepan en un int, tomándose los menos significativos.

Si el valor de `b` es, por ejemplo, 20.5, `a` terminará valiendo 20, lo que es similar a aplicar una función "parte entera" implícitamente, y no demasiado incongruente. Pero si la parte entera de `b` excede el rango de un entero (por ejemplo si `b=99232.5` en una plataforma donde los enteros son de 16 bits), el resultado en `a` no tendrá lógica aparente. En el primer caso, los bits menos significativos de `b` que "caben" en `a` conservan el valor de `b`; en el segundo caso, no.

En la sentencia:

```
a=19.27 * b;
```

`a` contendrá los `sizeof(int)` bits menos significativos del resultado de evaluar la expresión de la derecha, truncada sin decimales.

Promoción automática de expresiones

Por otra parte, se tienen las reglas de promoción automática de expresiones. Enunciadas en forma aproximada (luego las daremos con más precisión), estas reglas dicen que el compilador hará estrictamente las conversiones necesarias para llevar todos los operandos al tipo del "mayor" entre ellos. El resultado de evaluar una operación aritmética será del tipo del "mayor" de sus operandos.

A veces, esto no es lo que se desea. Por ejemplo, dada la sentencia:

```
a=3/2;
```

se tiene que tanto la constante 3 como la constante 2 son vistas por el compilador como ints; el resultado de la división será también un entero (la parte entera de 3/2, o sea 1). Aun más llamativo es el hecho de que si declaramos previamente:

```
float a;
```

el resultado es casi el mismo: `a` terminará conteniendo el valor float 1.0, porque el problema de truncamiento se produce ya en la evaluación del miembro derecho de la asignación.

Operador cast

En el ejemplo anterior, ¿cómo recuperar el valor correcto, con decimales, de la división? Declarar a la variable `a` como **float** es necesario, pero no suficiente. Para que la expresión del miembro derecho sea **float** es necesario que **al menos uno** de sus operandos sea **float**. Hay dos formas de lograr esto; la primera es escribir cualquiera de las subexpresiones como constante en punto flotante (con punto decimal, o en notación exponencial):

```
a=3./2;
```

La segunda consiste en forzar explícitamente una conversión de tipo, con un importante operador llamado **cast**, de la siguiente manera.

```
a=(float)3/2;
```

El operador **cast** es la aclaración, entre paréntesis, del tipo al cual queremos convertir la expresión (en este caso, la subexpresión **3**). Da lo mismo aplicarlo a cualquiera de las constantes. Sin embargo, lo siguiente **no funcionará**:

```
a=(float)(3/2);
```

Aquí el operador **cast** se aplica a la expresión **ya evaluada como entero**, con lo que volvemos a tener un valor **1.0**, float, en `a`.

Reglas de promoción en expresiones

Son aplicadas por el compilador en el orden que se da más abajo (tomado de K&R, 2a. ed.). Ésta es una lista muy detallada de las comprobaciones y conversiones que tienen lugar. Para la mayoría de los propósitos prácticos, basta tener en cuenta la regla de **llevar ambos operandos al tipo del "mayor"** de ellos.

Entendemos por "promoción entera" el acto de llevar los objetos de tipo `char`, `enum` y **campos de bits** a **int**; o, si los bits de un int no alcanzan a representarlo, a **unsigned int**.

1. Si cualquier operando es long double, se convierte el otro a long double.
2. Si no, si cualquier operando es double, se convierte el otro a double.
3. Si no, si cualquier operando es float, se convierte el otro a float.
4. Si no, se realiza promoción entera sobre ambos operandos.
5. Si cualquiera de ellos es unsigned long int, se convierte el otro a unsigned long int.
6. Si un operando es long int y el otro es unsigned int, el efecto depende de si un long int puede representar a todos los valores de un unsigned int.
7. Si es así, el unsigned int es convertido a long int.
8. Si no, ambos se convierten a unsigned long int.
9. Si no, si cualquier operando es long int, se convierte el otro a long int.
10. Si no, si cualquier operando es unsigned int, se convierte el otro a unsigned int.
11. Si no, ambos operandos son int.

3.4 Observaciones

Nótese que **no existen** tipos `boolean` ni `string`. Más adelante veremos cómo manejar datos de estas clases.

El tipo **char**, pese a su nombre, no está restringido a la representación de caracteres. Por el contrario, un char **tiene entidad aritmética**. Almacena una cantidad **numérica** y puede intervenir en operaciones matemáticas. En determinadas circunstancias, y sin perder estas propiedades, puede ser interpretado como un carácter (el **carácter cuyo código ASCII contiene**).

En general, en C es conveniente habituarse a pensar en los datos separando la **representación** (la forma como se almacena un objeto) de la **presentación** (la forma como se visualiza). Un mismo patrón de bits almacenado en un objeto de datos puede ser visto como un número decimal, un carácter, un número hexadecimal, octal, etc. La verdadera naturaleza del dato es la representación de máquina, el patrón de bits almacenado en el objeto de datos.

3.5 Una herramienta: printf()

Con el objeto de facilitar la práctica, describimos aquí la función de Biblioteca Standard `printf()`.

- La función de salida `printf()` lleva un **número variable de argumentos**.

`%d`	entero, decimal
`%u`	entero sin signo, decimal
`%l`	long, decimal
`%c`	carácter
`%s`	cadena
`%f`	float
`%lf`	double
`%x`	entero hexadecimal

Cuadro 3.1: Especificaciones de conversión de printf().

- Su primer argumento siempre es una cadena o constante string, la **cadena de formato**, conteniendo texto que será impreso, más, opcionalmente, **especificaciones de conversión**.
- Las especificaciones de conversión comienzan con un signo "`%`". Todo otro conjunto de caracteres en la cadena de formato será impreso textualmente.
- Cada especificación de conversión determina la manera en que se imprimirán los restantes argumentos de la función.
- Deben existir tantas especificaciones de conversión como argumentos luego de la cadena de formato.
- Un mismo argumento de un tipo dado puede ser impreso o presentado de diferentes maneras según la especificación de conversión que le corresponda en la cadena de formato (de aquí la importancia de separar representación de presentación)
- Las especificaciones de conversión pueden estar afectadas por varios **modificadores** opcionales que determinan, por ejemplo, el ancho del campo sobre el cual se escribirá el argumento, la cantidad de decimales de un número, etc.
- Las principales especificaciones de conversión están dadas en el Cuadro 3.1.

Ejemplo 3.1
Este programa escribe algunos valores con dos especificaciones de formato distintas.

```
main() {
   int i,j;
   for(i=65, j=1; i<70; i++, j++)
      printf("vuelta no. %d: i=%d, i=%c\n",j,i,i);
}
```

Salida del programa:

```
vuelta no. 1: i=65, i=A
vuelta no. 2: i=66, i=B
vuelta no. 3: i=67, i=C
vuelta no. 4: i=68, i=D
vuelta no. 5: i=69, i=E
```

Ejemplo 3.2
El programa siguiente escribe el mismo valor en doble precisión pero con diferentes modificadores del campo correspondiente, para incluir una cierta cantidad de decimales o alinear la impresión.

```
main() {
    double d;
    d=3.141519/2.71728182;
    printf("d=%lf\n",d);
    printf("d=%20lf\n",d);
    printf("d=%20.10lf\n",d);
    printf("d=%20.5lf\n",d);
    printf("d=%.10lf\n",d);
}
```

Salida del programa:

```
d=1.156126
d=           1.156126
d=       1.1561255726
d=            1.15613
d=1.1561255726
```

3.6 Ejercicios

1. ¿Cuáles de entre estas declaraciones contienen errores?

   a. integer a;
   b. short i,j,k;
   c. long float (h);
   d. double long d3;
   e. unsigned float n;
   f. char 2j;
   g. int MY;
   h. float ancho, alto, long;
   i. bool i;

2. Dé declaraciones de variables con tipos de datos adecuados para almacenar:

   a. La edad de una persona.
   b. Un número de DNI.
   c. La distancia, en Km, entre dos puntos cualesquiera del globo terrestre.
   d. El precio de un artículo doméstico.
   e. El valor de la constante PI expresada con 20 decimales.

3. Prepare un programa con variables conteniendo los valores máximos de cada tipo entero, para comprobar el resultado de incrementarlas en una unidad. Imprima los valores de cada variable antes y después del incremento. Incluya **unsigneds**.

4. Lo mismo, pero dando a las variables los valores mínimos posibles, e imprimiéndolas antes y después de decrementarlas en una unidad.

5. Averigüe los tamaños de todos los tipos básicos en su sistema aplicando el operador `sizeof()`.

6. Si se asigna la expresión $(3 - 5)$ a un `unsigned short`, ¿cuál es el resultado? ¿Depende de qué formato de conversión utilicemos para imprimirlo?

7. ¿Qué hace falta corregir para que la variable `r` contenga la división exacta de `a` y `b`?

```
int a, b;
float r;
a = 5;
b = 2;
r = a / b;
```

8. ¿Qué resultado puede esperarse del siguiente fragmento de código?

```
int a, b, c, d;
a = 1;
b = 2;
c = a / b;
d = a / c;
```

9. ¿Cuál es el resultado del siguiente fragmento de código? Anticipe su respuesta en base a lo dicho en esta unidad y luego confírmela mediante un programa.

```
printf("%d\n", 20/3);
printf("%f\n", 20/3);
printf("%f\n", 20/3.);
printf("%d\n", 10%3);
printf("%d\n", 3.1416);
printf("%f\n", (double)20/3);
printf("%f\n", (int)3.1416);
printf("%d\n", (int)3.1416);
```

10. Escribir un programa que multiplique e imprima $100000 * 100000$. ¿De qué tamaño son los ints en su sistema?

11. Convertir una moneda a otra sabiendo el valor de cambio. Dar el valor a dos decimales.

12. Escriba y corra un programa que permita saber si los chars en su sistema son signed o unsigned.

13. Escriba y corra un programa que asigne el valor 255 a un char, a un unsigned char y a un signed char, y muestre los valores almacenados. Repita la experiencia con el valor `-1` y luego con `'\377'`. Explicar el resultado.

14. Copiar y compilar el siguiente programa. Explicar el resultado.

```
main() {
    double x;
    int i;
    i = 1400;
    x = i; /* conversion de int a double */
    printf("x = %10.6le\ti = %d\n",x,i);
    x = 14.999;
    i = x; /* conversion de double a int */
    printf("x = %10.6le\ti = %d\n",x,i);
    x = 1.0e+60;
    i = x;
    printf("x = %10.6le\ti = %d\n",x,i);
}
```

15. Escriba un programa que analice la variable `v` conteniendo el valor 347 y produzca la salida:

```
3 centenas
4 decenas
7 unidades
```

(y, por supuesto, salidas acordes si `v` toma otros valores).

16. Sumando los dígitos de un entero escrito en notación decimal se puede averiguar si es divisible por 3 (se constata si la suma de los dígitos lo es). ¿Esto vale para números escritos en otras bases? ¿Cómo se puede averiguar esto?

17. Indicar el resultado final de los siguientes cálculos

    a. `int a; float b = 12.2; a = b;`

    b. `int a, b; a = 9; b = 2; a /= b;`

    c. `long a, b; a = 9; b = 2; a /= b;`

    d. `float a; int b, c; b = 9; c = 2; a = b/c;`

    e. `float a; int b, c; b = 9; c = 2; a = (float)(b/c);`

    f. `float a; int b, c; b = 9; c = 2; a = (float)b/c;`

    g. `short a, b, c; b = -2; c = 3; a = b * c;`

    h. `short a, b, c; b = -2; c = 3; a = (unsigned)b * c;`

18. Aplicar operador cast donde sea necesario para obtener resultados apropiados:

    a.
    ```
    int a; long b; float c;
    a = 1; b = 2; c = a / b;
    ```

    b.
    ```
    long a;
    int b,c;
    b = 1000; c = 1000;
    a = b * c;
    ```

Capítulo 4

Constantes

4.1 Constantes numéricas

Las constantes numéricas en un programa C pueden escribirse en varias bases. Además, la forma de escribirlas puede modificar el tamaño de los **objetos de datos** donde se almacenan.

Constantes enteras

- `10, -1` son constantes **decimales**.
- `010, -012` son constantes **octales** por comenzar en `0`.
- `0x10, -0x1B, 0Xbf01` son constantes **hexadecimales** por comenzar en `0x` o `0X`.
- `'A'` es una constante de carácter. En una computadora que sigue la convención del código ASCII, equivale al decimal `65`, hexadecimal `0x41`, etc.

Constantes long

Una constante entera puede indicarse como **long** agregando una letra L mayúscula o minúscula: `0L, 431`. Si bien numéricamente son equivalentes a 0 y a 43 int, el compilador, al encontrarlas, manejará constantes de tamaño long (construirá objetos de datos sobre la cantidad de bits correspondientes a un long). Esto puede ser importante en ciertas ocasiones: por ejemplo, al invocar a funciones con argumentos formales long, usando argumentos reales que caben en un entero.

Constantes unsigned

Para hacer más claro el propósito de una constante positiva, o para forzar la promoción de una expresión, puede notársela como **unsigned**. Esto tiene que ver con las reglas de promoción expresadas en el capítulo 3. Constantes unsigned son, por ejemplo, `32u` y `298U`.

Constantes de punto flotante

Las constantes en punto flotante se caracterizan por llevar un punto decimal o un carácter 'e' (que indica que está en notación exponencial). Así `10.23, .999, 0., 1.e10, 1.e-10, 1e10` son constantes en punto flotante. La constante `6.02e23` se interpreta como el número `6.02` multiplicado por 10^{23}. La constante `-5e-1` es igual a `-1/2`.

4.2 Constantes string

El texto comprendido entre comillas dobles en un programa C es interpretado por el compilador como una **constante string**, con propiedades similares a las de un arreglo de caracteres.

El proceso de compilación, al identificarse una constante string, es como sigue:

- El compilador reserva una zona de memoria en la imagen del programa que está construyendo, **del tamaño del string más uno**.
- Se copian los caracteres entre comillas en esa zona, agregando al final **un byte conteniendo un cero** (`'\0'`).
- Se reemplaza la **referencia** a la constante string en el texto del programa por la **dirección donde quedó almacenada**.

La cadena registrada por el compilador será almacenada al momento de ejecución en la zona del programa correspondiente a **datos estáticos inicializados** (zona a veces llamada "bss"). Así, una constante string **equivale a, o se evalúa a**, una dirección de memoria: la dirección donde está almacenado su primer carácter.

El cero final

El carácter `'\0'` impuesto por el compilador al final de la secuencia de caracteres señaliza el fin de la cadena, y tiene la importante misión de funcionar como **protocolo** o **señal de terminación** para aquellas funciones de la Biblioteca Standard que manejan strings (copia de cadenas, búsqueda de caracteres, comparación de cadenas, etc.). Debido a esta representación interna, algunas veces se las ve mencionadas con el nombre de **cadenas ASCIIZ** (caracteres ASCII seguidos de cero).

Gracias a esta representación con el **cero final**, las cadenas ASCIIZ en C **no tienen una longitud máxima**. El final de la cadena simplemente ocurre donde aparezca un carácter cero.

Al construir o manipular cadenas ASCIIZ, es necesario cumplir con el protocolo de terminarlas con su **cero final** para poder utilizar las funciones que trabajan con strings. De lo contrario, esas funciones **no encontrarán el final del string**.

¿Hay diferencias entre `'\0'`, `'0'` y `"0"`? Muchas.

- La primera constante, `'\0'`, es un entero. Su valor aritmético es 0. Todos los bits del objeto de datos donde está representada son 0.
- La segunda, `'0'`, es una constante de carácter. Ocupa un objeto de datos de 8 bits de tamaño. Su valor es **48 decimal** en aquellas computadoras cuyo juego de caracteres esté basado en ASCII, pero puede ser diferente en otras.
- La tercera, `"0"`, es una constante string, y se evalúa a una dirección. Es decir, en cualquier expresión donde figure, su valor aritmético es una dirección de memoria dentro del espacio de direcciones del programa. Ocupa un objeto de datos del tamaño de una dirección (frecuentemente 16 o 32 bits), además del espacio de memoria ubicado a partir de esa dirección y ocupado por los caracteres del string. Ese espacio de memoria está ocupado por un byte igual a `'0'` (el primer y único carácter del string), que como hemos visto equivale a 48 decimal en computadoras que adoptan ASCII, y a continuación viene un byte `'\0'`, o sea, 0 (señal de fin del string).

Ejemplo 4.1
Si tenemos las declaraciones y asignaciones siguientes, donde las tres variables declaradas son char:

```
char a,b,c;
a='\0';
b='0';
c="0";
```

- La primera asignación es perfectamente válida y equivale a `a=0`.
- La segunda asignación también es correcta y equivale a `b=48` en computadores basados en ASCII.
- La tercera asignación será rechazada por el compilador, generándose un error de "asignación no portable de puntero". Los objetos a ambos lados del signo igual son de diferente naturaleza: a la izquierda tenemos algo que puede ser directamente **usado como un dato** (una constante o una variable); a la derecha, algo que, indirectamente, **referencia a un dato** (una dirección). Se dice que la variable y la constante string tienen **diferente nivel de indirección**.

4.3 Constantes de carácter

El gráfico muestra el resultado de asignar algunas constantes relacionadas con el problema anterior, suponiendo una arquitectura donde los enteros y las direcciones de memoria son de 16 bits. Las tres primeras asignaciones dejan en `a` valores aritméticos 0, 48 y 0.

Las dos últimas asignaciones dejan en `a` la dirección de una cadena almacenada en memoria. Las cadenas apuntadas son las que están representadas en el diagrama.

La primera cadena contendrá el código ASCII del carácter 0 y un cero binario señalizando el fin del string. La segunda contendrá un cero binario (expresado por la constante de carácter `'\0'`) y un cero binario fin de string.

Las constantes de carácter son una forma expresiva y portable de especificar constantes numéricas. Internamente, durante la compilación y ejecución del programa, el compilador las entiende como valores numéricos sobre ocho bits. Así, es perfectamente lícito escribir expresiones como `'A' + 1` (que equivale a `66`, o a `0x42`, o a la constante de carácter `'B'`).

Algunos caracteres especiales tienen una grafía especial:

- `\b` carácter 'backspace', ASCII 8
- `\t` tabulador, ASCII 9
- `\n` fin de línea, ASCII 10 (UNIX) o secuencia 13,10 (DOS)
- `\r` retorno de carro, ASCII 13
- Una forma alternativa de escribir constantes de carácter es mediante su código ASCII: `'\033'`, `'\x1B'` Aquí representamos el carácter cuyo código ASCII es 27 decimal, en dos bases. La barra invertida (*backslash*) muestra que el contenido de las comillas simples debe ser interpretado como el código del carácter. Si el carácter siguiente al backslash es x o X, el código está en hexadecimal; si no, está en octal. Para representar el carácter backslash, sin su significado como modificador de secuencias de otros caracteres, lo escribimos dos veces seguidas.

Estas constantes de carácter pueden ser también escritas respectivamente como las constantes numéricas `033`, `27` o `0x1B`, ya que son aritméticamente equivalentes; pero con las comillas simples indicamos que el programador "ve" a estas constantes como caracteres, lo que puede agregar expresividad a un segmento de programa.

Por ejemplo, 0 es claramente una constante numérica; pero si escribimos `'\0'` (que es numéricamente equivalente), ponemos en evidencia que estamos pensando en el carácter cuyo código ASCII es 0. El carácter

`'\0'` (ASCII 0) es distinto de `'0'` (ASCII 48). La expresión de las constantes de carácter mediante backslash y algún otro contenido se llama una **secuencia de escape**.

Constantes de carácter en strings

Todas estas notaciones para las constantes de carácter pueden intervenir en la escritura de constantes string. El mecanismo de reconocimiento de constantes de caracteres dentro de strings asegura que todo el juego de caracteres de la máquina pueda ser expresado dentro de una constante string, aun cuando no sea imprimible o no pueda producirse con el teclado. Cuando el compilador se encuentre analizando una constante string asignará un significado especial al carácter **barra invertida** o backslash (\). La aparición de un backslash permite referirse a los caracteres por su código en el sistema de la máquina (por lo común, el ASCII).

4.4 Constantes enumeradas

Como una alternativa más legible y expresiva a la definición de constantes de preprocesador, se pueden definir grupos de constantes reunidas por una declaración. Una declaración de **constantes enumeradas** hace que las constantes tomen valores consecutivos de una secuencia.

Si no se especifica el primer inicializador, vale 0. Si alguno se especifica, la inicialización de los restantes continúa la secuencia. Las constantes de una enumeración no necesitan tener valores distintos, pero todos los nombres en las diferentes declaraciones `enum` de un programa deben ser diferentes.

Ejemplo 4.2
Aquí los valores de las constantes son ENE = 1, FEB = 2, MAR = 3, etc.

```
enum meses {
    ENE = 1, FEB, MAR, ABR, MAY, JUN,
    JUL,AGO, SEP, OCT, NOV, DIC
};
```

Ejemplo 4.3
Aquí los valores asumidos son respectivamente 0, 1, 2, 5, 6, y nuevamente 1 y 2.

```
enum varios { ALFA, BETA, GAMMA, DELTA = 5,IOTA, PI = 1, RHO };
```

4.5 Ejercicios

1. Indicar si las siguientes constantes están bien formadas, y en caso afirmativo indicar su tipo y dar su valor decimal.

   a. `'C'`
   b. `'0'`
   c. `'0xAB'`
   d. `70`
   e. `1A`
   f. `0xABL`
   g. `070`
   h. `'010'`
   i. `0xaB`
   j. `080`
   k. `0x10`
   l. `'0xAB'`
   m. `0XFUL`
   n. `'\030'`
   ñ. `-40L`
   o. `015L`
   p. `x41`
   q. `'B'`
   r. `'\xBB'`
   s. `'AB'`
   t. `322U`

2. Indicar qué caracteres componen las constantes string siguientes:

   a. `"ABC\bU\tZ"`
   b. `"\103B\x41"`

3. ¿Cómo se imprimirán estas constantes string?

   a. `"\0BA"`
   b. `"\\0BA"`
   c. `"BA\0CD"`

4. ¿Qué imprime esta sentencia? Pista: *nada* no es la respuesta correcta.

```
printf("0\r1\r2\r3\r4\r5\r6\r7\r8\r9\r");
```

5. Escribir una macro que devuelva el valor numérico de un carácter correspondiente a un dígito en base decimal.

6. Idem donde el carácter es un dígito hexadecimal entre A y F.

Capítulo 5

Propiedades de las variables

Las variables tienen diferentes propiedades según que sean declaradas dentro o fuera de las funciones, y según ciertos modificadores utilizados al declararlas. Entre las propiedades de las variables distinguimos:

- Alcance (desde dónde es visible una variable)
- Vida (cuándo se crea y cuándo desaparece)
- Clase de almacenamiento (dónde y cómo se aloja la información que contiene)
- Liga o *linkage* (en qué forma puede ser manipulada por el linker)

Las reglas que determinan, a partir de la declaración de una variable, cuáles serán sus propiedades, son bastante complejas. Estas reglas son tan interdependientes, que necesariamente la discusión de las propiedades de las variables será algo reiterativa.

5.1 Alcance de las variables

Una declaración puede aparecer, o bien dentro de una función, o bien fuera de todas ellas. En el primer caso, hablamos de una **variable local**; en el segundo, se trata de una variable **externa, o global**, y las diferencias entre ambas son muchas e importantes. Por supuesto, la primera consecuencia del lugar de declaración es el **alcance**, o ámbito de visibilidad de la variable: una variable local **es visible sólo desde dentro de la función** donde es declarada. Una variable externa puede ser usada **desde cualquier función de la unidad de traducción**, siendo suficiente que la declaración se encuentre antes que el uso.

Ejemplo 5.1
La variable `m` declarada al principio es externa, y puede ser vista desde `fun1()` y `fun2()`. Sin embargo, `fun1()` declara su propia variable m local, y toda operación con `m` dentro de `fun1()` se referirá a esta última. Por otro lado, la variable `n` es también externa, pero es visible sólo por `fun2()`, porque **todo uso de las variables debe estar precedido por su declaración**. Si apareciera una referencia a la variable `n` en `fun1()`, se dispararía un error de compilación.

```
int m;
int fun1()
{
    int m;
    m=1;
    ...
}
int n;
int fun2()
```

```
{
    m=1;
    ...
}
```

5.2 Vida de las variables

Una variable externa se crea al **momento de carga** del programa, y **perdura durante toda la ejecución** del mismo. Una variable local **se crea y se destruye** a cada invocación de la función donde esté declarada (excepción: las locales estáticas).

Ejemplo 5.2
Cada vez que `fun2()` asigna el resultado de `fun1()` a `j`, está utilizando el mismo objeto de datos de la misma variable `j`, porque ésta es externa; pero cada invocación de `fun1()` crea un nuevo objeto de datos para la variable `k`, el cual se destruye al terminar esta función.

```
int j;
int fun1()
{
    int k;
    ...
}
int fun2()
{
    j=fun1();
}
```

Ejemplo 5.3
La diferencia con el ejemplo anterior es que ahora `k` es declarada con el modificador **static**. Esto hace que `k` tenga las mismas propiedades de vida que una variable externa. A cada invocación de `fun1()`, ésta utiliza el mismo objeto de datos, sin modificarlo, para la variable `k`. Si lee su valor, encontrará el contenido que pueda haberle quedado de la invocación anterior. Si le asigna un valor, la invocación siguiente de `fun1()` encontrará ese valor en `k`. Este ejemplo muestra que alcance y vida no son propiedades equivalentes en C.

```
int j;
int fun1()
{
    static int k;
    ...
}
int fun2()
{
    j=fun1();
}
```

La propiedad que diferencia ambas instancias de `k` es la **clase de almacenamiento**; en el primer caso, `k` es local y automática; en el segundo, `k` es local pero estática.

5.3 Clases de almacenamiento

Dependiendo de cómo son almacenados los contenidos de las variables (es decir, en qué lugar del mapa de memoria del programa se mantienen los objetos de datos), éstas pueden tener varias clases de almacenamiento. Una variable **externa** tiene clase de almacenamiento **estática**. Una variable **local** tiene

-salvo indicación contraria- clase de almacenamiento **automática**. Una tercera clase de almacenamiento es la llamada **registro**. La clase de almacenamiento determina, como se vio recién, la vida de las variables.

Variables estáticas Las variables estáticas comienzan su vida al tiempo de carga del programa, es decir, aun antes de que se inicie la ejecución de la función main(). Existen durante todo el tiempo de ejecución del programa. Son **inicializadas con ceros binarios**, salvo que exista otra inicialización explícita. Son las variables externas y las locales declaradas static.

Variables automáticas Esta clase abarca exclusivamente las variables, declaradas localmente a una función, que no sean declaradas static. El objeto de datos de una variable automática inicia su existencia al entrar el control a la función donde está declarada, y muere al terminar la función. **No son inicializadas** implícitamente, es decir, contienen **basura** salvo que se las inicialice explícitamente.

Variables registro Una variable registro no ocupará memoria, sino que será mantenida en un registro del procesador.

Ejemplo 5.4
En el segmento de programa siguiente:

```
int m;
int fun()
{
    int j;
    register int k;
    static int l;
    ...
}
```

- La variable `m`, por ser externa, tiene clase de almacenamiento **estática**.
- Las variables `j`, `k` y `l` son locales, pero sólo `j` es **automática**.
- La variable `l` es **estática** (tiene propiedades de vida similares a las de `m`).
- Por su parte `k` es de tipo **registro**, lo que quiere decir que el compilador, siempre que resulte posible, mantendrá sus contenidos en algún registro de CPU de tamaño adecuado.

Una declaración `register` debe tomarse solamente como una *recomendación* hecha por el programador al compilador, ya que no hay garantías de que, al tiempo de ejecución, resulte posible utilizar un registro para esa variable. Más aún, el mismo programa, compilado y corrido en diferentes arquitecturas, podrá utilizar diferentes cantidades de registros para sus variables.

Una variable register tendrá un tiempo de acceso muy inferior al de una variable en memoria, porque el acceso a un registro de CPU es mucho más rápido. En general resulta interesante que las variables más frecuentemente accedidas sean las declaradas como register; típicamente, los índices de arrays, variables de control de lazos, etc. Sin embargo, la declaración register es quizás algo anacrónica, ya que los compiladores modernos ejecutan una serie de optimizaciones que frecuentemente utilizan registros para mantener las variables, aun cuando no haya indicación alguna por parte del programador.

La clase de almacenamiento automática es natural para las variables locales; ¿cuál es la idea de declarar variables locales que sean estáticas? Generalmente se desea aprovechar la capacidad de "recordar la historia" de las variables estáticas, utilizando el valor al momento de la última invocación para producir uno nuevo. Por ejemplo, una función puede contar la cantidad de veces que ha sido llamada.

Ejemplo 5.5
Aquí el ciclo while se ejecuta 50 veces. La inicialización (implícita o explícita) de una variable estática se produce una única vez, al momento de carga del programa. Por el contrario, la inicialización (explícita) de una automática se hace al crear cada instancia de la misma (al momento de la entrada del control a la función).

```
int veces()
{
    static int vez=0;
    return ++vez;
}
int fun()
{
    while(veces() <= 50) {
        ...
    }
}
```

Variables y mapa de memoria

De acuerdo a su clase de almacenamiento, las variables aparecen en diferentes regiones del mapa de memoria del programa en ejecución.

- Las variables locales (automáticas) **se disponen en la pila o stack** del programa. Debido a la forma de administración de esta zona de la memoria, existen solamente hasta la finalización de la función.
- Las variables estáticas (las externas, y las locales cuando son declaradas `static`) se alojan en la **zona de datos estáticos**. Esta zona no cambia de tamaño ni pierde sus contenidos, y queda inicializada al momento de carga del programa.

A medida que una función invoca a otras, las variables locales van apareciendo en el stack, y a medida que las funciones terminan, el stack se va desalojando en orden inverso a como aparecieron las variables. Cada función, al recibir el control, toma parte del stack, con los contenidos que hubieran quedado allí de ejecuciones previas, para alojar allí sus variables. A esto se debe que el programa las vea inicializadas con basura.

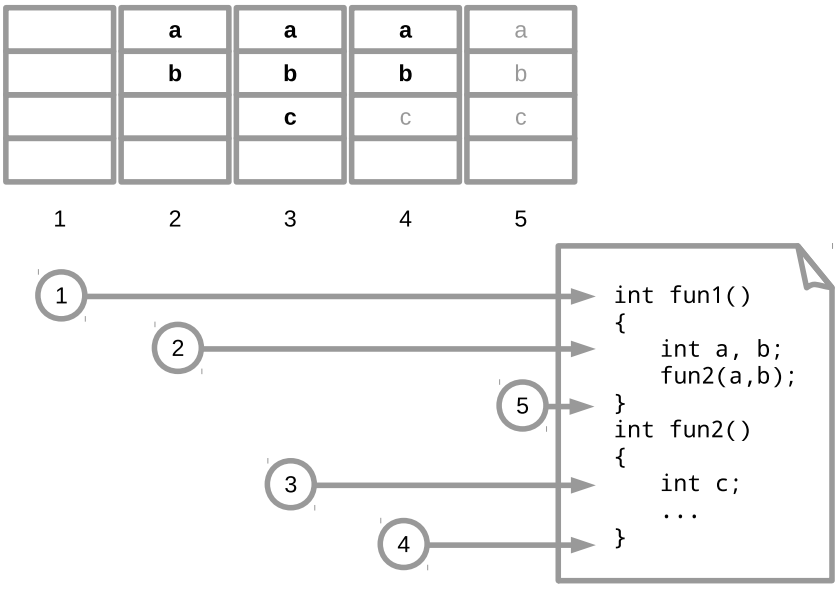

Figura 5.1: Objetos de datos en el stack.

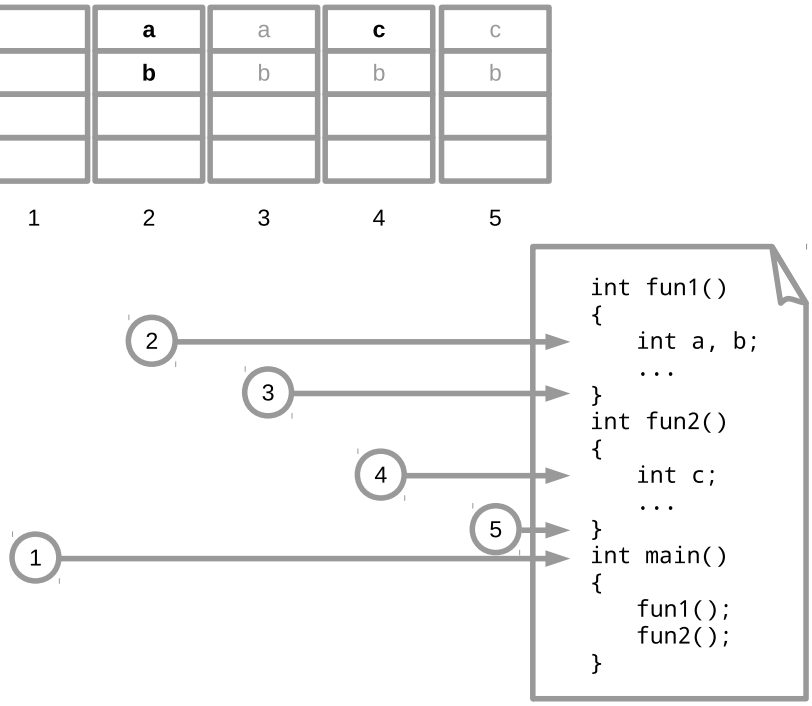

Figura 5.2: Objetos de datos en el stack.

Ejemplo 5.6
Con el código de la Fig. 5.1, el estado del stack en momentos sucesivos será:

1. Antes de entrar a `fun1()` se tiene el stack vacío.
2. Al entrar a `fun1()` se disponen sus variables locales en el stack, en orden de aparición.
3. Al entrar a `fun2()` se dispone en el stack su variable local.
4. Al salir de `fun2()` y volver a `fun1()` se desaloja la variable local de `fun2()`.
5. Al salir de `fun1()` se desmantela el stack completamente y se vuelve al estado inicial.

Ejemplo 5.7
Con el código de la Fig. 5.2, que invoca a dos funciones secuencialmente, el estado del stack en momentos sucesivos será:

1. Antes de entrar a `fun1()` se tiene el stack vacío.
2. Al entrar a `fun1()` se disponen sus variables locales en el stack, en orden de aparición.
3. Al terminar `fun1()` se desmantela el stack.
4. Al entrar en `fun2()` se dispone en el stack su variable local, cuyo objeto de datos tendrá *basura* debida al valor de a dejado por `fun1()`.
5. Al salir de `fun2()` se desmantela su stack completamente y se vuelve al estado inicial.

5.4 Liga

La **liga** es la propiedad que determina si las variables y funciones definidas en una unidad de traducción serán o no visibles por el linker. Una vez que un conjunto de unidades de traducción pasa exitosamente la compilación, tenemos un conjunto de módulos objeto. Cada módulo objeto puede contener, en forma simbólica, pendiente de resolución, **referencias** a variables o funciones definidas en otros módulos. La propiedad de las variables y funciones que permite que el linker encuentre la **definición** de un objeto para

aparearlo con su **referencia** es la **liga externa**. Tienen liga externa por defecto las **variables externas y las funciones**, de modo que todas éstas pueden ser referenciadas desde otras unidades de traducción.

El concepto de liga externa es importante cuando el proyecto de desarrollo abarca varias unidades de traducción que deben dar lugar a un ejecutable. Aprovechando la propiedad de liga externa de las funciones, se puede ubicar cada definición de función, o un conjunto de ellas, en un archivo separado. Esto suele facilitar el mantenimiento y aportar claridad a la estructura de un proyecto de desarrollo.

La excepción a la regla de liga externa se produce cuando las **variables externas o funciones** son declaradas con el modificador **static**. Este modificador cambia el tipo de los objetos a liga interna. Un objeto que normalmente sería de liga externa, declarado como static, pasa a ser visible únicamente dentro de la unidad de traducción donde ha sido declarado.

Esta particularidad permite realizar, en cierta medida, ocultamiento de información. Si una unidad de traducción utiliza variables externas o funciones de su uso privado, que no deben hacerse visibles desde afuera, puede declarárselas static, con lo cual se harán inaccesibles a toda otra unidad de traducción. El caso típico se presenta cuando se desea hacer opacas las funciones que implementan un tipo de datos abstracto, haciéndolas de liga interna mientras que las funciones públicas (las de interfaz) se dejan con liga externa.

Finalmente, las variables **locales**, al ser visibles únicamente dentro de su función, se dice que **no tienen liga** (el linker nunca llega a operar con ellas).

Ejemplo 5.8
En el ejemplo dado en el Cuadro 5.1, `fun1()`, `fun2()` y `fun3()` están definidas en unas unidades de traducción distintas de la de `main()`. El fuente **alfa.c** es capaz de dar origen a un programa ejecutable (porque contiene el punto de entrada al programa), pero solamente si al momento de linkedición se logra que el linker resuelva las referencias pendientes a `fun1()` y a `fun2()` (que no están definidas en **alfa.c**). Por motivos similares, las referencias en **gamma** necesitan de las definiciones en **beta** al momento de linkedición.

En la práctica logramos esto de varias maneras.

1. O bien, con:

   ```
   gcc alfa.c beta.c gamma.c -o alfa
   ```

   que significa "compilar separadamente los tres fuentes, linkeditarlos juntos y al ejecutable resultado renombrarlo como **alfa**";

2. o bien con:

   ```
   gcc -c alfa.c
   gcc -c beta.c
   gcc -c gamma.c
   gcc alfa.o beta.o gamma.o -o alfa
   ```

   que es la misma tarea pero distribuida en etapas separadas;

3. o bien preparando un archivo **makefile** indicando este modo de construcción e invocar a **make**.

Ejemplo 5.9
El ejemplo del Cuadro 5.2 es casi idéntico al anterior, salvo que la función `fun3()` ahora está declarada `static`, y por este motivo no podrá ser vista por el linker para resolver la referencia pendiente de `fun2()` en lambda.c. La función `fun3()` tiene liga interna. Las tres unidades de traducción jamás podrán satisfacer la compilación.

5.5 Declaraciones y definiciones

- Una **declaración** consiste en la **mención** de un objeto (variable o función) antes de su uso.
- Una **definición** consiste en una sentencia de **creación** de dicha variable o función, que a partir de la ejecución de esa sentencia comienza a ser una entidad viva del programa.

alfa.c	beta.c	gamma.c

`main()`
`{`
`  fun1();`
`  fun2();`
`}`
	`int fun1()`
	`{`
	`  ...`
	`}`
		`int fun2()`
		`{`
		`  ...`
		`  fun3();`
		`  ....`
		`}`
		`int fun3()`
		`{`
		`  ...`
		`}`

Cuadro 5.1: Liga de las variables

iota.c	kappa.c	lambda.c

`main()`
`{`
`  fun1();`
`  fun2();`
`}`
	`int fun1()`
	`{`
	`  ...`
	`}`
	`static int fun3()`
	`{`
	`  ...`
	`}`
		`int fun2()`
		`{`
		`  ...`
		`  fun3();`
		`  ...`
		`}`

Cuadro 5.2: Liga de las variables

Normalmente una **declaración** de variable (de la forma `especificacion_de_tipo identificador`) funciona también como **definición** de la variable. Es decir, no sólo queda advertido el compilador de cuál es el **tipo** del objeto que se va a utilizar, sino que también se crea el espacio de memoria (el **objeto de datos**) que va a alojar la información asociada.

Ejemplo 5.10
Son declaraciones que además funcionan como definiciones:

```
int a;
float b, c;
int fun()
{
    ...
}
```

Cuando la declaración de una variable cualquiera aparece precedida del modificador `extern`, ésta indica el nombre y tipo asociado, pero no habilita al compilador para crear el objeto de datos; se trata de una variable cuya **definición** puede ser encontrada **más adelante, o aun en otra unidad de traducción**. La declaración `extern` tan sólo enuncia el tipo y nombre de la variable para que el compilador los tenga en cuenta.

Ejemplo 5.11
Las declaraciones siguientes no crean objetos de datos y por lo tanto **no son** definiciones.

```
extern int a;
extern float b, c;
int fun();
```

Una variable externa es visible desde todas las funciones de la unidad de traducción, y además puede ser utilizada desde otras. Esto se debe a la propiedad de liga externa de las variables externas: son visibles al linker como candidatos para resolver referencias pendientes.

El requisito para poder utilizar una variable definida en otra unidad de traducción es declararla con el modificador **extern** en aquella unidad de traducción donde se va a utilizar.

Ejemplo 5.12
En el Cuadro 5.3, el texto `delta.c` es una unidad de traducción que declara dos variables externas y dos funciones, pero hace **opacas** a la variable `n` y a la función `fun2()` con el modificador **static**.

- La función `fun1()` puede utilizar a todas ellas por estar dentro de la misma unidad de traducción, pero `fun3()`, que está en otra, sólo puede referenciar a `m` y a `fun1()`, que son de liga externa. Para ello debe declarar a `m` como `extern`, o de lo contrario no superará la compilación (*"todo uso debe ser precedido por una declaración"*).
- Si, además, `eta.c` declarara una variable `extern int n`, con la intención de referirse a la variable `n` definida en `delta.c`, la referencia no podría ser resuelta a causa de la condición de liga interna de `n`.
- Los usos de funciones (como `fun1()` en `eta.c`) pueden aparecer sin declaración previa, pero en este caso el compilador asumirá tipos de datos default para los argumentos y para el tipo del valor devuelto por la función (**int** en todos los casos).

5.6 Modificadores especiales

Const El modificador `const` indica que una variable no será modificada. Una variable `const` solamente puede inicializarse al momento de carga del programa (y debería hacerse así, ya que no hay otra manera de asignarle un valor).

```
const int a=12;   /* se declara un entero constante, con inicializacion */
```

delta.c	eta.c

`int m;`
`static int n;`
`int fun1()`
`{`
`  n=fun2();`
`  ...`
`}`
`static int fun2()`
`{`
`  ...`
`}`
	`extern int m;`
	`int fun3()`
	`{`
	`  m=fun1();`
	`}`

Cuadro 5.3: Liga de las variables

Sin optimizar	Optimizado	Optimización inhibida

`while(!fin) {`
`  a = beta;`
`  b = fun(a);`
`}`
	`a = beta;`
	`while(!fin)`
	`  b = fun(a);`
		`volatile int beta;`
		`while(!fin) {`
		`  a = beta;`
		`  b = fun(a);`
		`}`

Cuadro 5.4: Optimización de ciclos

```
a++;            /* el compilador no aprobara esta sentencia */
```

El modificador `const` también permite expresar, en el prototipo de una función, que un argumento no podrá ser modificado por la función, aun cuando sea pasado por referencia.

Volatile Los compiladores modernos aplican una cantidad de pasos de optimización cuando ven instrucciones aparentemente redundantes o sin efectos, porque su desplazamiento o eliminación puede implicar ventajas en tiempo de ejecución o espacio de almacenamiento. Esto es especialmente así si las instrucciones sospechosas se encuentran dentro de ciclos. El modificador volatile sirve para advertir al compilador de que una variable será modificada asincrónicamente con la ejecución del programa (por ejemplo, por efecto de una rutina de atención de interrupciones) y por lo tanto el optimizador no puede inferir correctamente su utilidad dentro del programa. Esto evitará que el compilador aplique la lógica de optimización a las instrucciones que involucran a esta variable.

Ejemplo 5.13
El ciclo **`while`** del Cuadro 5.4 podría ser reescrito por un optimizador, extrayendo del ciclo la asignación **`a=beta`** en el entendimiento de que **`beta`** no cambiará en ninguno de los pasos del ciclo.

Sin embargo, si esperamos que la variable beta cambie por acción de algún agente externo a la rutina en cuestión, con la declaración previa **`volatile int beta`** el compilador se abstendrá de optimizar las líneas donde intervenga **`beta`**.

5.7 Ejercicios

1. Copie, compile y ejecute el siguiente programa. Posteriormente agregue un modificador static sobre la variable a y repita la experiencia.

```
int fun()
{
    int a;
    a = a + 1;
    return a;
}
main()
{
    printf("%d\n", fun());
    printf("%d\n", fun());
}
```

2. ¿Qué imprime este programa?

```
int alfa;
int fun()
{
    int alfa;
    alfa = 1;
    return alfa;
}
main()
{
    alfa = 2;
    printf("%d\n",fun());
    printf("%d\n",alfa);
}
```

3. ¿Qué imprime este programa?

```
int alfa;
int fun(int alfa)
{
    alfa = 1;
    return alfa;
}
main()
{
    alfa = 2;
    printf("%d\n",fun(alfa));
    printf("%d\n",alfa);
}
```

4. Copie y compile, juntas, las unidades de traducción que se indican abajo. ¿Qué hace falta para que la compilación sea exitosa?

fuente1.c

```
int a;
int fun1(int x)
{
    return 2 * x;
}
```

fuente2.c

```
main()
{
    a = 1;
    printf("d\n", fun1(a));
}
```

5. ¿Qué ocurre si un fuente intenta modificar una variable externa, declarada en otra unidad de traducción como `const`? Prepare, compile y ejecute un ejemplo.

6. ¿Qué resultado puede esperarse de la compilación de estos fuentes?

header.h

```
#include <stdio.h>
#define VALOR 6
```

fuente1.c

```
#include "header.h"
main()
{
    static int c;
    printf("%d\n",fun(c));
}
```

fuente2.c

```
#include "header.h"
int fun(int x)
{
    return VALOR * x;
}
```

7. Denotemos esquemáticamente que un módulo objeto `prueba.o` contiene un objeto de datos `x` y una función `fun()`, ambos de liga externa, de esta manera:

prueba.o
x fun()

Si se tiene un conjunto de archivos y unidades de traducción que se compilarán para formar los respectivos módulos objeto, ¿cómo se aplicaría la notación anterior al conjunto de módulos objeto resultantes? Hacer el diagrama para los casos que aparecen en el Cuadro 5.5. ¿Hay colisión de nombres? ¿Hay referencias que el linker no pueda resolver? Cada grupo de fuentes, ¿puede producir un ejecutable?

8. Un conjunto de programas debe modelar eventos relativos a un aeropuerto. Se necesita preparar una implementación de las estructuras de datos y funciones del aeropuerto, para ser usada por los demás programas. Especifique las variables y funciones (en pseudocódigo) que podrán satisfacer los siguientes requerimientos. Preste atención a las declaraciones extern y static.

   - El aeropuerto tendrá cinco pistas.
   - Se mantendrá un contador de la cantidad total de aviones en el aeropuerto y uno de la cantidad total de aviones en el aire.
   - Para cada pista se mantendrá la cantidad de aviones esperando permiso para despegar de ella y la cantidad de aviones esperando permiso para aterrizar en ella.
   - Habrá una función para modelar el aterrizaje y otra para modelar el despegue por una pista dada (decrementando o incrementando convenientemente la cantidad de aviones en una pista dada, en tierra y en el aire).

- Habrá una función para consultar, y otra para establecer, la cantidad de aviones esperando aterrizar o despegar por cada pista.
- Habrá una función para consultar la cantidad de aviones en tierra y otra para consultar la cantidad de aviones en el aire.
- No deberá ser posible que un programa modifique el estado de las estructuras de datos sino a través de las funciones dichas.

9. ¿Cuáles pueden ser los "agentes externos" al programa que sean capaces de cambiar el valor de una variable `volatile`?

	hdr1.h	fuente1.c	fuente2.c	fuente3.c

a)	`#define UNO 1`
	`#define DOS 2`
	`extern int a;`
		`#include "hdr1.h"`
		`main()`
		`{`
		`  int b;`
		`  b = fun1(a);`
		`}`
			`#include "hdr1.h"`
			`int fun1(int x)`
			`{`
			`  return`
			`  x+fun2(x);`
			`}`
			`static int`
			`  fun2(int x)`
			`{`
			`  return x + DOS;`
			`}`
				`#include`
				`  "hdr1.h"`
				`int a;`

b)	`extern int c;`
	`extern int`
	`  fun1(int p),`
	`  fun2(int p);`
		`#include "hdr1.h"`
		`int fun1(int x)`
		`{`
		`  return`
		`  fun2(x)+1;`
		`}`
			`int a, b, c=1;`
			`int fun2(int x)`
			`{`
			`  return x-1;`
			`}`
				`main()`
				`{`
				`  int d;`
				`  d = fun1(3);`
				`}`

c)		`int fun1(int x)`
		`{`
		`  return x+1;`
		`}`
		`int fun2(int x)`
		`{`
		`  return x+2;`
		`}`
			`int fun3(int x)`
			`{`
			`  return x+3;`
			`}`
			`static int`
			`  fun2(int x)`
			`{`
			`  return x+4;`
			`}`
				`main()`
				`{`
				`  int a;`
				`  a = fun1(a);`
				`}`

d)		`int fun1(int x)`
		`{`
		`  extern int b;`
		`  x = b-fun2(x);`
		`}`
			`static int`
			`  a = 1;`
			`static int`
			`  b = 1;`
			`int fun2(int x)`
			`{`
			`  return x-a;`
			`}`
				`int b;`
				`main()`
				`{`
				`  b = 2;`
				`  printf("%d",`
				`  fun1(3));`
				`}`

Cuadro 5.5: Conjuntos de fuentes y liga

Capítulo 6

Operadores

6.1 Operadores aritméticos

El C tiene un conjunto de operadores muy rico, incluyendo algunos operadores que es difícil encontrar en otros lenguajes de programación. Comenzamos por los operadores aritméticos.

- $+$, $-$, $*$, $/$ (operaciones aritméticas usuales)
- $++$, $--$ (incremento y decremento)
- $\%$ (operador módulo)
- No existe operador de **exponenciación** en C. En cambio, está implementada una función `pow()` en la Biblioteca Standard.
- No existe operador de **división entera**, opuesto a la división entre números reales, como en Pascal. Sin embargo, la división entre enteros da un entero: el resultado se trunca debido a las reglas de evaluación de expresiones.

Ejemplo 6.1
Aquí `a` y `b` reciben respectivamente el cociente y el resto de dividir un VALOR por 256. Imprimiéndolos podemos ver cómo una CPU, en la cual un `unsigned short int` tuviera un tamaño de 16 bits, almacenaría ese VALOR sobre dos bytes.

```
unsigned char a,b;
a=VALOR / 256;
b=VALOR % 256;
```

Ejemplo 6.2
La siguiente división de `j` por `k` es entera. La variable `c` valdrá 1.

```
float c;
int j,k;
j=3;
k=2;
c=j/k;
```

Ejemplo 6.3
Los operadores de incremento y decremento ($++$ y $--$) equivalen a las sentencias del tipo `a = a+1` o `b = b-1`. Suman o restan una unidad a su argumento, que debe ser un tipo entero. Se comportan diferentemente según que se apliquen en forma prefija o sufija.

Sentencias	a	b
a=5; b=a++;	6	5
a=5; b=++a;	6	6
a=3; b=a--;	2	3
a=3; b=--a;	2	2

- Aplicados como prefijos, el valor devuelto por la expresión es el valor incrementado o decrementado.
- Aplicados como sufijos, el incremento o decremento se realiza como efecto lateral, pero el valor devuelto por la expresión es el anterior al incremento o decremento.

Abreviaturas

Existe una forma de abreviar la notación en las expresiones del tipo `a = a*b` y `a = a+b`. Podemos escribir, respectivamente:

```
a *= b;
a += b;
```

Esto se aplica a todos los operadores aritméticos y de bits.

6.2 Operadores de relación

- `==` (igualdad)
- `<, >, <=, >=`
- `!=` (distinto de)

  Es un error muy común sustituir el operador de relación `==` por el de asignación `=`. Este error lógico no provocará error de compilación ni de ejecución, sino que será interpretado como una asignación, que en C puede ser hecha en el mismo contexto que una comparación. La sentencia:

  ```
  if(a = 38)
      ...
  ```

  es sintácticamente válida, pero en lugar de **comparar** `a` con 38, **asigna** el valor 38 a la variable `a`.

  La expresión `a=38` es legal en C como expresión lógica. Su valor de verdad es **siempre** `TRUE`, independientemente del valor que tuviera `a` anteriormente. Si el programador desea obtener el valor lógico de la comparación de `a` con 38, debe utilizar la expresión `if(a == 38)`.

6.3 Operadores lógicos

```
!, &&, || (not, and, or)
```

Ejemplo 6.4
La siguiente es una expresión lógica ("`a` igual a `b` o no `c` menor que 2").

```
a==b || !(c < 2)
```

Al no existir tipo booleano en C, los valores lógicos se equiparan a los aritméticos. Una expresión formada con un operador `||`, `&&` o `!` da 1 si es V, 0 si es F. Además, toda expresión cuyo valor aritmético sea diferente de 0 es "verdadera". Toda expresión que dé 0 es "falsa".

Ejemplo 6.5

- `a - b`, que es una expresión aritmética, es también una expresión lógica. Será F cuando a sea igual a b.
- `a` como expresión lógica será V sólo cuando a sea distinto de 0 .
- `a = 8` es una expresión de asignación. Su valor aritmético es el valor asignado, por lo cual, como expresión lógica, es V, ya que 8 es distinto de 0.

Constantes lógicas

Generalmente los compiladores definen dos símbolos o macros, `FALSE` y `TRUE`, en el header `stdlib.h`, y a veces también en otros. Su definición suele ser la siguiente:

```
#define FALSE 0
#define TRUE  !FALSE
```

Es conveniente utilizar estos símbolos para agregar expresividad a los programas.

6.4 Operadores de bits

- `~` (negación de bits)
- `&`, `|` (**and**, **or** de bits)
- `^` (**or** de bits exclusivo)
- `>>`, `<<` (desplazamiento de bits a derecha y a izquierda)

  El desplazamiento de bits es **con pérdida**, en el sentido de que:

  - Un desplazamiento a la izquierda en un bit equivale a una multiplicación por 2; un desplazamiento a la derecha en un bit equivale a una división por 2.
  - Si el bit menos significativo es 1 (si el número es impar), al desplazar a la derecha ese bit se pierde (la división no es exacta).
  - Si el bit más significativo es 1 (si el número es igual o mayor que la mitad de su rango posible), al desplazar a la izquierda ese bit se pierde (la multiplicación da overflow).

Ejemplo 6.6
El operador de negación es unario y provoca el complemento a uno de su operando. Los operadores and, or y or exclusivo son binarios.

```
unsigned char a;
a = ~255;           /* a <-- 0   */
a = 0xf0 ^ 255;     /* a <-- 0x0f */
a = 0xf0 & 0x0f;    /* a <-- 0x00 */
a = 0xf0 | 0x0f;    /* a <-- 0xff */
```

Ejemplo 6.7
Los operadores >> y << desplazan los bits de un objeto de tipo `char`, `int` o `long`, una cantidad dada de posiciones.

```
unsigned char a,b,c;
a = 1 << 4;
b = 2 >> 1;
c <<= 2;
```

- En el primer caso, el 1 se desplaza 4 bits a la izquierda; `a` vale finalmente 16.
- En el segundo caso, el bit 1 de la constante 2, a uno, se desplaza un lugar a la derecha; `b` vale 1.
- En el tercero, `c` se desplaza dos bits a la izquierda.

6.5 Operadores especiales

Distinguimos como especiales los operadores de **asignación** e **inicialización**, que son operaciones diferentes aunque desafortunadamente son representadas por el mismo signo, y el operador **ternario**.

- = (operador de asignación)
- = (operador de inicialización)
- ?: (operador ternario)

Una inicialización es la operación que permite asignar un valor inicial a una variable en el momento de su creación:

```
int a=1;
```

El operador ternario comprueba el valor lógico de una expresión, y, según este valor, se evalúa a una u otra de las restantes expresiones. Supongamos tener la expresión siguiente.

```
a = (expresion_1) ? expresion_2 : expresion_3;
```

Entonces, si `expresion_1` es V, el ternario se evaluará a `expresion_2`, y ese valor será asignado a la variable `a`. Si `expresion_1` es F, `a` quedará con el valor `expresion_3`.

Ejemplo 6.8
La expresión

```
c = b + (a < 0) ? -a : a;
```

asigna a `c` el valor de `b` más el valor absoluto de `a`.

6.6 Precedencia y orden de evaluación

En una expresión compleja, formada por varias subexpresiones conectadas por operadores, es peligroso hacer depender el resultado del orden de evaluación de las subexpresiones. Si los operadores tienen la misma precedencia, la evaluación se hace de izquierda a derecha, pero en caso contrario el orden de evaluación no queda definido.

Por ejemplo, en la expresión `w*x/++y + z/y` se puede contar con que primeramente se ejecutará `w*x` y sólo entonces `w*x/++y`, porque los operadores de multiplicación y división son de la misma precedencia; pero no se puede asegurar que `w*x/++y` sea evaluado antes o después de `z/y`, lo que hace que el resultado de esta expresión sea **indefinido** en C.

La solución es **secuencializar** la ejecución, dividiendo las expresiones en sentencias:

```
a = w * x / ++y;
b = a + z / y;
```

Precedencia	Operador	

1	`()`
	`[]`
	`-> .`
		Llamada a función
		Indexación de arreglo
		Acceso a miembros de estructuras

2	`! ~`
	`++ --`
	`*`
	`&`
	`()`
	`sizeof`
	`+` unario
	`-` unario
		Negación y complemento a uno
		Incremento y decremento
		Indirección
		Dirección de
		Conversión forzada de tipo (*cast*)
		Tamaño de variables o tipos

3	`* / %`	Multiplicación, división y módulo
4	`+ -`	Suma y resta
5	`<< >>`	Desplazamiento de bits
6	`< <= >= >`	Operadores de relación
7	`== !=`	Operadores lógicos de igualdad
8	`&`	AND de bits
9	`^`	XOR de bits
10	`|`	OR de bits
11	`&&`	AND lógico
12	`||`	OR lógico
13	`?:` `=`	Operador ternario Asignación

14	`+= -= *= /= %=`
	`&= ^= |=`
	`<<= >>=`
		Aritméticos y lógicos abreviados

15	`,`	Operador coma

Cuadro 6.1: Relaciones de precedencia entre operadores

6.7 Resumen

El Cuadro 6.1 ilustra las relaciones de precedencia entre los diferentes operadores. La tabla está ordenada con los operadores de mayor precedencia a la cabeza. Los que se encuentran en el mismo renglón tienen la misma precedencia.

6.8 Ejercicios

1. ¿Qué valor lógico tienen las expresiones siguientes?

   a. `TRUE && FALSE`

   b. `TRUE || FALSE`

   c. `0 && 1`

   d. `0 || 1`

   e. `(c > 2)? (b < 5): (2 != a)`

   f. `(b == c)? 2 : FALSE;`

   g. `c == a;`

h. `C = A;`

i. `0 || TRUE`

j. `TRUE || 2-(1+1)`

k. `TRUE && !FALSE`

l. `!(TRUE && !FALSE)`

m. `x == y > 2`

2. Escriba una macro `IDEM(x,y)` que devuelva el valor lógico `TRUE` si `x` e `y` son iguales, y `FALSE` en caso contrario. Escriba `NIDEM(x,y)` que devuelva `TRUE` si las expresiones `x` e `y` son diferentes y `FALSE` si son iguales.

3. Escriba una macro `PAR(x)` que diga si un entero es par. Muestre una versión usando el operador `%`, una usando el operador `>>`, una usando el operador `&` y una usando el operador `|`.

4. Escriba macros `MIN(x,y)` y `MAX(x,y)` que devuelvan el menor y el mayor elemento entre `x` e `y`. Usando las anteriores, escriba macros `MIN3(x,y,z)` y `MAX3(x,y,z)` que devuelvan el menor y el mayor elemento entre tres expresiones.

5. ¿Cuál es el significado aritmético de la expresión `1<<x` para diferentes valores de `x` $= 0, 1, 2...$?

6. Utilice el resultado anterior para escribir una macro `DOSALA(x)` que calcule 2 elevado a la x-ésima potencia.

7. ¿A qué otra expresión es igual `a<b || a<c && c<d`?

   a. `a<b || (a<c && c<d)`

   b. `(a<b || a<c)&& c<d`

8. Reescribir utilizando abreviaturas:

   a. `a = a + 1;`

   b. `b = b * 2;`

   c. `b = b - 1;`

   d. `c = c - 2;`

   e. `d = d % 2;`

   f. `e = e & 0x0F;`

   g. `a = a + 1;`

   h. `b = b + a;`

   i. `a = a - 1;`

   j. `c = c * a;`

9. ¿Qué escribirá este programa?

```
main()
{
   int a = 1;
   int b;

   b = a || 12;
```

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

Capítulo 7

Estructuras de control

Las estructuras de control de C no presentan, en conjunto, grandes diferencias con las del resto de los lenguajes estructurados. En los esquemas siguientes, donde figura una sentencia puede reemplazarse por varias sentencias encerradas entre llaves (un **bloque**).

7.1 Estructura alternativa

Como en la casi totalidad de los lenguajes de programación, la estructura alternativa permite ejecutar un bloque de una o más sentencias cuando la expresión condicional es V, y opcionalmente otro bloque cuando dicha expresión es F. Formas típicas de la estructura alternativa son las siguientes.

```
if(expresion)
    sentencia;

if(expresion) {
    sentencias
    ...
}

if(expresion)
    sentencia;
else
    sentencia;
```

Ejemplo 7.1
El formato de C es libre, y las llaves sólo son necesarias cuando las sentencias de ejecución condicional son más de una.

```
if(a == 8) c++;
```

Las condiciones lógicas se construyen en base a operadores de relación y lógicos.

```
if(c >= 2 || func(b) < 0)
```

La cláusula `else` indica el bloque que se ejecutará en el caso negativo de la condición.

```
if(d)
    c++;
else
    c += 2;
```

En las estructuras anidadas, la cláusula `else` se aparea con el `if` más interno, salvo que las llaves expresen otra cosa.

En el cuadro siguiente, las llaves del Listado 1 muestran cómo están asociadas las estructuras. En el Listado 2, se han quitado, con lo cual la semántica cambia, pero se mantiene la misma indentación, lo que puede dar lugar a confusión. El Listado 2 en realidad es equivalente al Listado 3, que utiliza llaves aunque en forma redundante, y además tiene la indentación correcta.

Listado 1

```
if(c >= 2) {
    if(d)
        c++;
} else
    c += 2;
```

Listado 2

```
if(c >= 2)
    if(d)
        c++;
else
    c += 2;
```

Listado 3

```
if(c >= 2) {
    if(d)
        c++;
    else
        c += 2;
}
```

En general, el estilo de programación debe sugerir la estructura, usando indentación; pero es, por supuesto, la ubicación de las llaves, y no la indentación, la que determina la estructura.

7.2 Estructuras repetitivas

El lenguaje C dispone de las estructuras adecuadas para procesar conjuntos de **0 o más** datos (estructura `while`) y **1 o más** datos (estructura `do-while`).

Ambas estructuras ejecutan su cuerpo de sentencias mientras la expresión resulte verdadera, pero en un lazo `while`, la comprobación de la expresión se hace **al principio de cada ciclo**. En cambio, en el lazo `do-while`, se hace **al final**.

Estructura while

```
while(expresion)
    sentencia;
```

Estructura do-while

```
do {
    sentencias;
} while(expresion);
```

Estructura for

La iteración es un caso particular de lazo `while` donde necesitamos que un bloque de sentencias se repita una cantidad previamente conocida de veces. Estos casos implican la inicialización de variables de control, el incremento o decremento de las mismas, y la comprobación por valor límite. Estas tareas administrativas se pueden hacer más cómoda y expresivamente con un lazo `for`.

El esquema es:

```
for(inicializacion; condicion_mientras; incremento)
    sentencia;
```

Donde:

- **inicialización** es una o más sentencias, separadas por comas, que se ejecutan una única vez al entrar al lazo.

- **condición_mientras** es una expresión lógica, que se comprueba al principio de cada iteración. Mientras resulte verdadera, se continúa ejecutando el cuerpo.
- **incremento** es una o más sentencias, separadas por comas, que se realizan al final de cada ejecución del cuerpo de la iteración.

La estructura for es equivalente al siguiente lazo while:

```
inicializacion;
while(condicion_mientras) {
    sentencia;
    incremento;
}
```

Aunque el uso más común de las **sentencias de incremento** es hacer avanzar o retroceder un contador de la cantidad de iteraciones, nada impide que se utilice esa sección para cualquier otro fin.

Cualesquiera de las secciones inicialización, condición_mientras o incremento pueden estar vacías. En particular, la sentencia:

```
for( ; ; )
```

es un lazo infinito.

Ejemplo 7.2

- El siguiente lazo acumula los números 1 a 10 sobre la variable `a`:

  ```
  for(i=1; i<=10; i++)
      a += i;
  ```

- Si se quiere asegurar que la variable `a` tiene un valor inicial cero, se puede escribir:

  ```
  for(i=1, a=0; i<=10; i++)
      a += i;
  ```

- Aprovechando la propiedad del corto circuito en las expresiones lógicas, se puede introducir el cuerpo del lazo `for` en la comprobación (aunque no es recomendable si complica la lectura):

  ```
  for(i=1, a=0; i<=10 && a+=i; i++);
  ```

  Nótese que el cuerpo de este último for es la sentencia nula. A propósito: es un error muy común utilizar un signo ";"de más, así:

  ```
  for(i=1; i<=10; i++);
      a += i;
  ```

  Esta estructura llevará la variable `i` desde 1 hasta 10 sin ejecutar ningún otro trabajo (lo que se repite es la sentencia nula) y después incrementará `a`, una sola vez, en el valor de la última iteración de `i`.

- Una clásica estructura de control para un proceso consumidor de objetos:

  ```
  a=leercaracter();
  while( a != '\033' ) {
      procesar(a);
      a = leercaracter();
  }
  ```

- La propiedad de que toda asignación en C tiene un valor como expresión (el valor asignado) permite reescribir la estructura de control anterior como:

```
while( (a=leercaracter()) != '\033' )
       procesar(a);
```

- Las expresiones conectadas por los operadores lógicos se evalúan de izquierda a derecha, y la evaluación se detiene apenas alcanza a determinarse el valor de verdad de la expresión (propiedad "del corto circuito"). Así, si suponemos que procesar() siempre devuelve un valor distinto de cero, la sentencia siguiente equivale a los lazos anteriores.

```
while((a=leercaracter()) != '\033' && procesar(a));
```

- Otra versión, utilizando la estructura do-while, podría ser:

```
do {
    if((a=leercaracter()) != '\033')
       procesar(a);
} while(a != '\033');
```

- Si utilizamos `for`, que es esencialmente un *while*:

```
for( ; (a=leercaracter()) != '\033'; ) procesar(a);
```

  Aquí dejamos vacías las secciones de inicialización y de incremento.

- También, pero algo menos claro:

```
for( ; (a=leercaracter()) != '\033'; procesar(a) );
```

7.3 Estructura de selección

La estructura de selección (*switch*) es una estructura alternativa múltiple. Dadas varias alternativas, la estructura de selección desvía el control al segmento de programa correspondiente. La sintaxis de la estructura `switch` es como sigue:

```
switch(expresion_entera) {
    case expresion_constante1:
        sentencias;
        break;
    case expresion_constante2:
        sentencias;
        break;
    default:
        sentencias;
}
```

Al entrar al switch, se comprueba el valor de la expresión entera, y si coincide con alguna de las constantes propuestas en los rótulos `case`, se deriva el control directamente allí. La sección `default` no es obligatoria. Sirve para derivar allí todos los casos que no se contemplen explícitamente. En las expresiones_constantes no se permite la aparición de variables ni funciones. Un ejemplo válido con expresiones_constantes sería:

```
#define ARRIBA 10
#define ABAJO  8

switch(valor(tecla)) {
    case 127+ARRIBA:
        arriba();
        break;
    case 127+ABAJO:
```

```
        abajo();
        break;
}
```

La sentencia `break` es necesaria aquí porque, al contrario que en Pascal, el control no se detiene al llegar al siguiente rótulo.

Ejemplo 7.3
Esta estructura recibe como entrada las variables `m` y `a` (mes y año) y da como salida `d` (la cantidad de días del mes).

```
switch(m) {
    case 2:
        d=28 + bisiesto(a) ? 1 : 0;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        d= 30;
    break; default:
        d= 31;
    }
```

Si `m` vale 4, 6, 9, u 11, la variable `d` recibe el valor 30. Al no haber un `break` intermedio, el control cae hasta la asignación `d = 30`.

La estructura switch tiene algunas limitaciones con respecto a sus análogos en otros lenguajes. A saber, no se puede comparar la expresión de selección con expresiones no constantes, ni utilizar rangos (el concepto de rango no está definido en C).

7.4 Transferencia incondicional

Hay varias sentencias de transferencia incondicional de control en C. Algunas tienen aplicación exclusivamente como modificadoras del control dentro de estructuras, como break y continue.

Sentencia continue

Utilizada dentro de un lazo while, do-while o for, hace que el control salte directamente a la comprobación de la condición de iteración. Así:

```
for(i=0; i<100; i++) {
    if(no_procesar(i))
        continue;
    procesar(i);
}
```

En este lazo, si la función `no_procesar()` devuelve valor distinto de cero, no se ejecuta el resto del lazo (la función procesar() y otras, si las hubiera, hasta la llave final del lazo). Se comprueba la validez de la expresión i<100, y si corresponde se inicia una nueva iteración.

Sentencia break

La sentencia break, por el contrario, hace que el control abandone definitivamente el lazo:

```
while(expresion) {
    if(ya_no_procesar())
```

```
        break;
    procesar();
}
seguir();
```

Cuando la función `ya_no_procesar()` dé distinto de cero, el control saltará a la función `seguir()`, terminando la ejecución de la estructura repetitiva.

Sentencia goto

Un rótulo es un nombre, seguido del carácter ":", que se asocia a un segmento de un programa. La sentencia `goto` transfiere el control a la instrucción siguiente a un rótulo. Aunque no promueve la programación estructurada, y se sabe que su abuso es perjudicial, `goto` es útil para resolver algunas situaciones. Por ejemplo: anidamiento de lazos con salida forzada.

```
for(i=0; i<10; i++) {
    for(j=0; j<50; j++) {
        if(ya_no_procesar(i,j))
            goto final;
        procesar(i,j);
    }
}
final: imprimir(i,j);
```

Aquí se podría implementar una estrategia estructurada, en base a `break`, pero el control quedaría retenido en el lazo exterior y se requeriría más lógica para resolver este problema. Se complicaría la legibilidad del programa innecesariamente.

Los rótulos a los que puede dirigirse un `goto` tienen un espacio de nombres propio. Es decir, no hay peligro de conflicto entre un rótulo y una variable del mismo nombre. Además, el ámbito de un rótulo es local a la función (una sentencia `goto` sólo puede acceder a los rótulos dentro del texto de la función donde aparece).

Sentencia return

Permite devolver un valor a la función llamadora. Implica una transferencia de control incondicional hasta el punto de llamada de la función que se esté ejecutando.

7.5 Observaciones

Hay errores de programación típicos, relacionados con estructuras de control en C, que vale la pena enumerar:

- Terminar el encabezado de las estructuras de control con un punto y coma extra
- Olvidar la sentencia break separando casos de un switch
- Confundir el significado de un lazo do-while tomando la condición de mientras como si fuera una condición de hasta (por analogía con repeat de Pascal).

7.6 Ejercicios

1. Reescribir estas sentencias usando while en vez de for:

a.
```
for(i=0; i<=10; i++)
    a = i;
```

b.
```
for( ; j<100; j+=2) {
    a = j;
    b = j * 2;
}
```

c.
```
for( ; ; )
    a++;
```

2. Si la función `quedanDatos()` devuelve el valor lógico que sugiere su nombre, ¿cuál es la estructura preferible?

a.
```
while(quedanDatos()) {
    procesar();
}
```

b.
```
do {
    procesar();
} while(quedanDatos());
```

3. ¿Cuál es el error de programación en estos ejemplos?

a.
```
for(i = 0; i < 10; i++);
    a = i - 50L;
```

b.
```
while(i < 100) {
    procesar(i);
    a = a + i;
}
```

4. ¿Cuál es el valor de x a la salida de los lazos siguientes?

a.
```
for(x = 0; x<100; x++);
```

b.
```
for(x = 32; x<55; x += 3);
```

c.
```
for(x = 10;x>0; x--);
```

5. ¿Cuántas X escriben estas líneas?

```
for (x = 0; x < 10; x++)
   for (y = 5; y > 0; y--)
      escribir("X");
```

6. Escribir sentencias que impriman la tabla de multiplicar para un entero dado.

7. Imprimir la tabla de los diez primeros números primos (sólo divisibles por sí mismos y por la unidad).
8. Escribir las sentencias para calcular el factorial de un entero.

Capítulo 8

Funciones

Una unidad de traducción en C contiene un conjunto de funciones. Si entre ellas existe una con el nombre especial **main**, entonces esa unidad de traducción puede dar origen a un programa ejecutable, y el comienzo de la función main será el punto de entrada al programa.

8.1 Declaración y definición de funciones

Los tipos de datos de los parámetros recibidos y del resultado que devuelve la función quedan especificados en su cabecera. El valor devuelto se expresa mediante **return**:

Ejemplo 8.1
Una función que recibe un int y un long, y devuelve un int.

```
int fun1(int alfa, long beta)
{
    ...
}
```

Una función que recibe dos doubles y devuelve un double.

```
double sumar(double x, double y)
{
    ...
    return x+y;
}
```

El caso especial de una función que no desea devolver ningún valor se especifica con el modificador **void**, y en tal caso un return, si lo hay, no debe tener argumento. Los paréntesis son necesarios aunque la función no lleve parámetros, y en ese caso es recomendable indicarlo con un parámetro void:

Ejemplo 8.2
Una función que recibe un int, y no devuelve ningún valor.

```
void procesar(int k)
{
    ...
}
```

Una función que devuelve un int, y no recibe argumentos.

```
int hora(void)
{
```

```
    ...
}
```

Una función puede ser declarada de un tipo cualquiera y sin embargo no contar con una instrucción return. En ese caso su valor de retorno queda indeterminado. Además, la función que llama a otra puede utilizar o ignorar el valor devuelto, a voluntad, sin provocar errores.

Ejemplo 8.3
En el caso siguiente se recoge basura en la variable `a`, ya que `fun2` no devuelve ningún valor pese a ser declarada como de tipo entero.

```
int fun2(int x)
{
    ...
    return;
}

    ...
    a=fun2(1);
```

El cuerpo de la función, y en general cualquier cuerpo de instrucciones entre llaves, es considerado un bloque. Las **variables locales** son aquellas declaradas dentro del cuerpo de una función, y su declaración debe aparecer antes de cualquier sentencia ejecutable. Es legal ubicar la declaración de variables como la primera sección dentro de cualquier bloque, aun cuando ya se hayan incluido sentencias ejecutables. Sin embargo, no es legal declarar funciones dentro de funciones.

Ejemplo 8.4
La variable `v` declarada dentro del bloque interno opaca a la declarada al principio de la función.

```
int fun3()
{
    int j,k,v;

     for(i=0; i<10; i++) {
        double v;
        ...
}
```

8.2 Prototipos de funciones

En general, como ocurre con las variables, el uso de una función debe estar precedido por su declaración. Sin embargo, el compilador trata el caso de las funciones con un poco más de flexibilidad. Un uso de variable sin declaración es ilegal, mientras que un uso de función sin definición obliga al compilador a suponer ciertos hechos, pero permite proseguir la compilación.

La suposición que hará el compilador, en la primera instancia en que se utilice una función y en ausencia de una definición previa, es que el resultado y los parámetros de la función son del tipo más "simple" que pueda representarlos. Esto vale tanto para las funciones escritas por el usuario como para las mismas funciones de la biblioteca standard.

Así, si se intenta calcular e^5:

```
main()
{
    double a;
```

```
    a=exp(5);
}
```

Nada permite al compilador suponer que la función exp() debe devolver algo distinto de un entero (el hecho de que se esté asignando su valor a un double no es informativo, dada la conversión automática de expresiones que hace el C). Además, el argumento 5 puede tomarse a primera vista como int, pudiendo ser que en la definición real de la función se haya especificado como double, o alguna otra elección de tipo.

En cualquier caso, esto es problemático, porque la comunicación de parámetros entre funciones, normalmente, se hace mediante el stack del programa, donde los objetos se almacenan como sucesiones de bytes. La función llamada intentará recuperar del stack los bytes necesarios para "armar" los objetos que necesita, mientras que la función que llama le ha dejado en el mismo stack menos información de la esperada. El programa compilará correctamente pero los datos pasados a y desde la función serán truncamientos de los valores deseados.

8.3 Redeclaración de funciones

Otro problema, relacionado con el anterior, es el que ocurre si permitimos que el compilador construya esa declaración provisoria y luego, en la misma unidad de traducción, damos la definición de la función, y ésta no concuerda con la imaginada por el compilador. La compilación abortará con error de "redeclaración de función". La forma de advertir al compilador de los tipos correctos antes del uso de la función es, o bien, definirla (proporcionando su fuente), o incluir su prototipo:

```
double exp(double x); /* prototipo de exp() */
main()
{
    double a;
    a=exp(5);
}
```

En el caso particular de las funciones de biblioteca standard, cada grupo de funciones cuenta con su header conteniendo estas declaraciones, que podemos utilizar para ahorrarnos tipeo. Para las matemáticas, utilizamos el header math.h:

```
#include <math.h>
main()
{
    double a;
    a=exp(5);
}
```

Un problema más serio que el de la redeclaración de funciones es cuando una función es compilada en una unidad de traducción separada A y luego se la utiliza, desde una función en otra unidad de traducción B, pero con una declaración incorrecta, ya sea porque se ha suministrado un prototipo erróneo o porque no se ha suministrado ningún prototipo explícito. y el implícito, que puede inferir el compilador, no es el correcto. En este caso la compilación y la linkedición tendrán lugar sin errores, pero la conducta al momento de ejecución depende de la diferencia entre ambos prototipos, el real y el inferido.

8.4 Recursividad

Las funciones en C pueden ser recursivas, es decir, pueden invocarse a sí mismas directa o indirectamente. Siguiendo el principio de que las estructuras de programación deben replicar la estructura de los

datos, la recursividad de funciones es una manera ideal de tratar las estructuras de datos recursivas, como árboles, listas, etc.

```
int factorial(int x)
{
    if(x==0)
        return 1;
    return x * factorial(x-1);
}
```

8.5 Ejercicios

1. Escribir una función que reciba tres argumentos enteros y devuelva un entero, su suma.
2. Escribir una función que reciba dos argumentos enteros y devuelva un long, su producto.
3. Escribir una función que reciba dos argumentos enteros a y b, y utilice a las dos anteriores para calcular:

   ```
   (a * b + b * 5 + 2) * (a + b + 1)
   ```

4. Escribir un programa que utilice la función anterior para realizar el cálculo con a=7 y b=3.
5. ¿Qué está mal en estos ejemplos?

   a.
   ```
   int f1(int x, int y);
   {
       int z;
       z = x - y;
       return z;
   }
   ```

   b.
   ```
   void f2(int k)
   {
       return k + 3;
   }
   ```

   c.
   ```
   int f3(long k)
   {
       return (k < 0) ? -1 : 1;
   }
   printf("%d\n",f3(8));
   ```

6. Escribir una función que reciba dos argumentos, uno de tipo int y el otro de tipo char. La función debe repetir la impresión del char tantas veces como lo diga el otro argumento. Escribir un programa para probar la función.

Capítulo 9

Variables estructuradas

Como casi todos los lenguajes, el C provee varias maneras de estructurar, o combinar, variables simples en alguna forma de agregación. Las variables estructuradas permiten manejar un conjunto de datos como una sola entidad.

9.1 Arreglos

La declaración

```
tipo nombre[cant_elementos];
```

define un bloque llamado `nombre` de `cant_elementos` **objetos consecutivos** de tipo `tipo`, lo que habitualmente recibe el nombre de **vector, array o arreglo**. Sus elementos se acceden **indexando** el bloque con expresiones enteras entre corchetes.

Ejemplo 9.1
Declaraciones y uso de arreglos.

```
int dias[12];
```

```
dias[0] = 31;
enero = dias[0];
febrero = dias[1];
a = dias[6 * b - 1];
```

```
double saldo[10];
for(i=0; i<10; i++)
    saldo[i] = entradas[i] - salidas[i];
```

En C, los arreglos se indexan a partir de 0.

Inicialización de arreglos

Al ser declarados, los arreglos pueden recibir una **inicialización**, que es una lista de valores del tipo correspondiente, indicados entre llaves. Esta inicialización puede ser completa o incompleta. Si se omite la dimensión del arreglo, el compilador la infiere por la cantidad de valores de inicialización.

Ejemplo 9.2
Inicializaciones completas e incompletas.

```
/* inicializacion completa */
int dias[12] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

```
/* inicializacion incompleta */
double saldo[10] = { 150.40, 170.20 };
```

```
long altura[] = { 3600, 3400, 3200, 6950 }; /* se infiere "long altura[4]" */
```

Errores frecuentes con arreglos

Los siguientes son errores de programación muy comunes al tratar con arreglos, y lamentablemente el lenguaje C no provee ayuda para prevenirlos:

- **Indexación fuera de límites**

  La dimensión dada en la declaración del arreglo dice cuántos elementos tiene. Esto no quiere decir que exista un elemento del arreglo con ese índice (porque se numeran a partir de 0).

  Ejemplo 9.3
  La última instrucción equivale a acceder a un elemento fuera de los límites del arreglo. Este programa, aunque erróneo, **compilará correctamente**.

  ```
  int tabla[5];

  tabla [5] = 1; /* error! el ultimo elemento es tabla[4] */
  ```

- **Asignación de arreglos**

  Es frecuente confundir las operaciones de inicialización y de asignación. La inicialización sólo es válida en el momento de la declaración: **no es legal asignar un arreglo**. La asignación debe forzosamente hacerse elemento por elemento.

  Ejemplo 9.4
  Notar la diferencia sintáctica entre inicialización y asignación.

  ```
  /* Inicializacion */
  int tabla[5] = { 1, 3, 2, 3, 4 };  /* correcto */

  /* Asignacion */
  int tabla[5];
  tabla[] = { 1, 3, 2, 3, 4 };     /* incorrecto */
  ```

  La última sentencia no es compilable. Debe reemplazarse por:

  ```
  tabla[0] = 1;
  tabla[1] = 3; ...etc
  ```

Al momento de ejecución, la conducta dependerá de cuestiones estructurales del sistema operativo. En variantes modernas de UNIX puede resultar un error de **violación de segmentación**, o **falla de segmentación**, lo que indica que el proceso ha **rebasado los límites de su espacio asignado** y el sistema de protección del sistema operativo ha terminado el proceso.
Sin embargo, en algunos otros casos, el error puede pasar inadvertido al momento de ejecución, porque la dirección del acceso, a pesar de superar los límites del arreglo, no cae fuera del espacio de memoria del proceso. En este caso la ejecución proseguirá, pero **se habrá leído o escrito basura** en alguna zona impredecible del espacio de memoria del proceso.
Éste y otros casos de accesos no controlados a la memoria pueden tener consecuencias, también impredecibles, más adelante en la ejecución. Como la causa y la manifestación del error están separadas, esta clase de errores resulta muy difícil de diagnosticar.
El diagrama ilustra dos casos de acceso indebido a un elemento inexistente de un arreglo. Suponemos tener una declaración tal como int tabla[5], y una instrucción errónea que hace referencia al elemento tabla[5]. En el primer caso, la variable estructurada tiene algún otro contenido contiguo dentro del espacio del proceso, y el acceso lee o escribe basura. En el segundo caso, el acceso cae fuera del espacio del proceso, y según la reacción del sistema operativo, puede ocurrir lo mismo que en el caso anterior, o bien el proceso puede ser terminado por la fuerza.

9.2 Arreglos multidimensionales

En C se pueden simular matrices y arreglos de más dimensiones creando **arreglos cuyos elementos son arreglos**. La declaración:

```
int matriz[3][4];
```

expresa un arreglo de tres posiciones cuyos elementos son arreglos de cuatro posiciones. Una declaración con inicialización podría escribirse así:

```
int matriz[3][4] = {
    {1, 2, 5, 7},
    {3, 0, 0, 1},
    {2, 8, 5, 4}};
```

y correspondería a una matriz de tres filas por cuatro columnas. La primera dimensión de un arreglo multidimensional puede ser inferida:

```
int matriz[][4] = {
    {1, 2, 5, 7},
    {3, 0, 0, 1},
    {2, 8, 5, 4}};
```

El recorrido de toda una matriz implica necesariamente un lazo doble, a dos variables:

```
for(i=0; i<3; i++)
    for(j=0; j<4; j++)
      a[i][j] = i + j;
```

9.3 Estructuras y uniones

Estructuras

Las **estructuras** de C permiten agrupar una cantidad de variables simples, de tipos eventualmente diferentes. Las estructuras y uniones aportan la ventaja de que es posible manipular este conjunto de variables como un todo.

Es posible inicializar estructuras, asignar conjuntos de constantes a las estructuras, asignar estructuras entre sí, pasarlas como argumentos reales a funciones, y devolverlas como valor de salida de funciones. En particular, ésta viene a ser la única manera de que una función devuelva más de un dato.

Ejemplo 9.5
Las declaraciones siguientes no definen variables, con espacio de almacenamiento, sino que simplemente enuncian un nuevo tipo que puede usarse en nuevas declaraciones de variables.

```
struct persona {
    long DNI;
    char nombre[40];
    int edad;
};

struct cliente {
    int num_cliente;
    struct persona p;
    double saldo;
};
```

El nombre o *tag* dado a la estructura es el nombre del nuevo tipo. En las instrucciones siguientes se utilizan los tags definidos anteriormente y se acceden a los diferentes miembros de las estructuras.

```
struct cliente c1, c2;

c1.num_cliente = 1001;
c1.p.DNI = 14233326; /* acceso anidado */
c1.p.edad=40;

c2 = c1;             /* copia de estructuras */
struct persona p1 = {17698735, "Juan Perez", 30};
c2.p = p1;
```

Una declaración con inicialización completa:

```
struct cliente c3 = {
    1002,
    {17698735, "Juan Perez", 30},
    150.25 };
```

Ejemplo 9.6
Es legal declarar una variable `struct` junto con la enunciación de su tipo, con o sin el tag asociado y con o sin inicialización.

```
struct complejo { double real, imag; } c;
struct { double real, imag; } c;
struct complejo {
   double real, imag;
} c = { 20.5, -7.3 };
```

Ejemplo 9.7
Una función que recibe y devuelve estructuras:

```
struct punto {
    int x, y;
};

struct punto puntomedio(struct punto p1, struct punto p2)
{
    struct punto z;
    z.x = (p1.x + p2.x) / 2;
    z.y = (p1.y + p2.y) / 2;
    return z;
}
```

Uniones

En una estructura, el compilador calcula la dirección de inicio de cada uno de los miembros dentro de la estructura sumando los tamaños de los elementos de datos. Una **unión** es un caso especial de estructura donde todos los miembros "nacen" en el mismo lugar de origen de la estructura.

```
union intchar {
    int i;
    char a[sizeof(int)];
};
```

Este ejemplo de unión contiene dos miembros: un entero y un arreglo de tantos caracteres como bytes contiene un `int` en la arquitectura destino. Ambos miembros, por la propiedad fundamental de los `unions`, quedan "superpuestos" en memoria. El resultado es que podemos asignar un campo por un nombre y acceder por el otro.

Ejemplo 9.8
En este caso particular, podemos conocer qué valores recibe cada byte de los que forman un `int`.

```
union intchar k;
k.i = 30541;
b = k.a[2];
```

Campos de bits

Se pueden definir estructuras donde los miembros son agrupaciones de bits. Esta construcción es especialmente útil en programación de sistemas donde se necesita la máxima compactación de las estructuras de datos. Cada miembro de un **campo de bits** es un `unsigned int` que lleva explícitamente un "ancho" indicando la cantidad de bits que contiene.

```
struct disp {
    unsigned int encendido : 1;
    unsigned int
        online : 1,
        estado : 4;
};
```

Ejemplo 9.9
Imaginemos, en base a la declaración anterior, un dispositivo mapeado en memoria con el cual comunicarnos en base a un protocolo, también imaginario. Implementamos con un campo de bits un registro de control `encendido`, que permite

encenderlo o apagarlo, consultar su disponibilidad (**online** o no), y hacer una lectura de un valor de **estado** de cuatro bits (que entonces puede tomar valores entre 0 y 15). Todo el registro de comunicación cabe en un byte.

Podríamos encender nuestro dispositivo imaginario, esperar a que se ponga online, tomar el promedio de diez lecturas de estado y apagarlo, con las instrucciones siguientes. Como se ve, no hay diferencia sintáctica de acceso con las estructuras.

```
struct disp {
    unsigned int encendido : 1;
    unsigned int
        online : 1,
        estado : 4;
};
```

```
struct disp d;
d.encendido = 1;
while(!d.online);
for(p=0, i=0; i<10; i++)
    p += d.estado;
p /= 10;
d.encendido = 0;
```

9.4 Ejercicios

1. Escribir una declaración con inicialización de un arreglo de diez elementos `double`, todos inicialmente iguales a $2{,}25$.
2. Escribir las sentencias para copiar un arreglo de cinco `longs` en otro.
3. Escribir las sentencias para obtener el producto escalar de dos vectores.
4. Escribir una función que devuelva la posición donde se halla el menor elemento de un arreglo de `floats`.
5. Dado un vector de diez elementos, escribir todos los promedios de cuatro elementos consecutivos.
6. Declarar una estructura punto conteniendo coordenadas x e y de tipo `double`. Dar ejemplos de inicialización y de asignación.
7. Declarar una estructura **segmento** conteniendo dos estructuras **punto**. Dar ejemplos de inicialización y de asignación. Dar una función que calcule su longitud. Dar una función que reciba un segmento S y un punto p, y devuelva `TRUE` si $p \in S$.
8. ¿Cuál es el error en estas sentencias de inicialización?

   a.
   ```
   struct alfa {
       int a, b;
   };
   alfa = { 10, 25 };
   ```

   b.
   ```
   struct alfa {
       int a, b;
   };
   alfa d = { 10, 25 };
   ```

c.
```
union dato {
    char dato_a[4];
    long dato_n;
} xdato = { "ABC", 1000 };
```

Capítulo 10

Apuntadores y Direcciones

El tema de esta unidad es el más complejo del lenguaje C y por este motivo se han separado los contenidos en dos partes (llamadas 10 y 10b).

La memoria del computador está organizada como un vector o arreglo unidimensional. Los índices en este arreglo son las direcciones de memoria. Este arreglo puede accederse indexando a cada byte individualmente, y en particular a cada estructura de datos del programa, mediante su dirección de comienzo.

Para manipular direcciones se utilizan en C variables especiales llamadas apuntadores o punteros, que son aquellas capaces de contener direcciones. En la declaración de un apuntador se especifica el tipo de los objetos de datos cuya dirección contendrá.

La notación:

```
char *p;
```

es la declaración de una variable puntero a carácter. El contenido de la variable `p` puede ser, en principio, cualquiera dentro del rango de direcciones de la máquina subyacente al programa. Una vez habiendo recibido un valor, se dice que la variable `p` apunta a algún objeto en memoria.

Esquemáticamente representamos la situación de una variable que contiene una dirección (y por lo tanto "apunta a esa dirección") según el diagrama siguiente (mas_información). La posición 1 de la memoria aloja un puntero que actualmente apunta a la posición 5.

El tema de apuntadores (o punteros) y direcciones es crucial en la programación en C, y parece ser el origen más frecuente de errores. Programas con mala lógica de acceso a memoria pueden ser declarados válidos por el compilador: su compilación puede ser exitosa y sin embargo ser completamente erróneos en ejecución. Esta es una de las críticas más frecuentes al lenguaje C, aunque en rigor de verdad, el problema no es del lenguaje, sino del programador con una mala comprensión de las cuestiones del lenguaje relacionadas con memoria.

Es fundamental, para no cometer estos errores, comprender en profundidad los conceptos de direcciones y punteros, así como la sintaxis de las declaraciones de punteros, para asegurarnos de que escribimos lo que se pretende lograr.

10.1 Operadores especiales

Para manipular punteros se hacen necesarios dos operadores especiales:

- El operador **dirección** devuelve **la dirección de un objeto**. La construcción siguiente:

  ```
  p = &a;
  ```

puede leerse: "asignar a p la dirección de a".

- El operador de **indirección**, o de **dereferenciación**, surte el efecto contrario: accede al **objeto apuntado por** una dirección. La construcción

```
a = *p;
```

puede leerse como "a es igual a lo apuntado por p".

Para obtener el efecto lógicamente esperado, en las expresiones anteriores `p` deberá ser un **puntero**, capaz de recibir y entregar una dirección.

En general, si el contenido de la variable `p` es igual a la dirección de `a`, es decir, `&a`, la expresión de dereferenciación `*p` puede aparecer en cualquier contexto en el que apareciera `a`. En particular, es legal asignar indirectamente a través de un apuntador.

Ejemplo 10.1
Las instrucciones

```
int a, *p;
p = &a;
*p = 1;
```

equivalen a

```
a = 1;
```

10.2 Aritmética de punteros

Son operaciones legales asignar punteros entre sí, sumar algebraicamente un entero a un puntero y restar dos punteros. Las consecuencias de cada operación se esquematizan en las figuras siguientes.

Asignación entre punteros

Luego de asignar un puntero a otro, ambos apuntan al mismo objeto. Cualquier modificación al objeto apuntado por uno se refleja al accederlo mediante el otro puntero.

Suma de enteros a punteros

La suma algebraica de una dirección más un entero es nuevamente una dirección. El sentido de la operación es desplazar el punto de llegada del apuntador (hacia arriba o hacia abajo en memoria) en tantas unidades como diga el entero, con la particularidad de que el resultado final es dependiente del tamaño del objeto apuntado. Esto es en general lo que desea el programador al aplicar un incremento a un apuntador.

Es decir que sumar (o restar) una unidad a un puntero, lo incrementa (decrementa) en tantos bytes como mida el objeto al cual apunta. Por ejemplo, para punteros a carácter, la instrucción p++ incrementa el valor del puntero en uno,que es el sizeof() de los chars; pero si p es un puntero a long, en una arquitectura donde los longs miden cuatro bytes, p++ incrementa el valor de p en cuatro (y p queda apuntando "un long más allá en la memoria"). El cálculo realizado al tiempo de ejecución para la instrucción p+i, donde p es un puntero a tipo e i es un entero, es siempre p+i*sizeof(tipo).

Resta de punteros

El sentido de una resta de punteros (o, equivalentemente, de una diferencia de direcciones) es obtener el tamaño del área de memoria comprendida entre los objetos apuntados por ambos punteros. La resta tendrá sentido únicamente si se hace entre variables que apuntan a objetos del mismo tipo.

Nuevamente se aplica la lógica del punto anterior: el resultado obtenido quedará expresado en unidades del tamaño del objeto apuntado. Es decir, si una diferencia entre punteros a long da 3, debe entenderse el resultado como equivalente a 3 longs, y por lo tanto a 3*sizeof(long) bytes.

10.3 Punteros y arreglos

Una consecuencia de que sea posible sumar enteros a punteros es que se puede simular el recorrido de un arreglo mediante el incremento sucesivo de un puntero. La operación de acceder a un elemento del arreglo es equivalente a obtener el objeto apuntado por el puntero. Las sentencias:

```
int *p;
int a[10];
p = &a[0];
```

Habilitan al programador para acceder a cada elemento del arreglo a mediante aritmética sobre el puntero p. Como el nombre de un arreglo se evalúa a su dirección inicial, la última sentencia también puede escribirse simplemente así:

```
p = a;
```

Ejemplo 10.2
Supongamos dadas las definiciones siguientes:

```
int alfa[] = { 2, 4, 6, 7, 4, 2, 3, 1 };
int *p, *q;
int b;
```

Con estas definiciones, veamos algunas manipulaciones de arreglos y punteros.

```
p = alfa;       /* el nombre de un arreglo
                   equivale a su direccion */
```

```
*p = 3;         /* equivalente a alfa[0] = 3 */
*(p+2) = 4;     /* equivalente a alfa[2] = 4 */
b = *p;         /* equiv. a b = alfa[0]    */
*(p+3) = *(p+6); /* sobreescribe el 7 con el 3 */
```

```
q = alfa + 2;   /* apunta al tercer elemento */
```

```
printf("%d\n",*q);            /* imprime 4 */
printf ("%d\n",q - p);        /* imprime 2 */
```

```
p += q;         /* ERROR - la suma de punteros
                           no esta definida */
```

Los dos segmentos de código siguientes hacen exactamente la misma tarea pero de maneras diferentes.

10.4 Punteros y cadenas de texto

Un caso típico del uso de punteros ocurre cuando se necesita trabajar con **cadenas de texto**, a veces llamadas **strings**, o **constantes string**.

Como se vio en la sección 4.2, las constantes string son todas aquellas secuencias de caracteres (eventualmente la secuencia vacía) en el texto del programa, que aparecen entre comillas. La representación interna de las constantes string durante la compilación, con el **cero final**, y con la referencia a la constante reemplazada por su dirección, se trasladará al programa compilado y aparecerá en la memoria, en la zona de datos estáticos, al momento de ejecución del programa.

Como la constante string es reemplazada por la dirección de su primer carácter, la asignación o inicialización de una constante string a un puntero es legal, y almacena en el puntero dicha dirección. En la inicialización de punteros, las constantes string son un caso especialmente frecuente.

Ejemplo 10.3
Un puntero a carácter inicializado con una constante string.

```
char *s = "Lenguaje C";
```

Ejemplo 10.4
La inicialización de `s` y la asignación de `t` cargan a ambas variables con las direcciones del primer carácter, o direcciones iniciales, de las cadenas respectivas.

```
char *s = "Esta es una cadena";
char *t;
t = "Esta es otra cadena";
```

Representamos en el diagrama el carácter 0 final (que no es imprimible) con el símbolo ¤. La expresión en C de este carácter es simplemente 0 (un entero) o `'\0'` (una constante carácter cuyo código ASCII es cero).

La función de biblioteca standard `printf()` permite imprimir una cadena con el especificador de conversión `%s`.

Ejemplo 10.5
Las líneas siguientes;

```
char *s = "Cadena de prueba";
char *t;
t = s + 7;
printf("%s\n", s);
printf("%s\n", t);
```

O bien, equivalentemente:

```
char *s = "Cadena de prueba";
printf("%s\n", s);
printf("%s\n", s + 7);
```

imprimen:

```
Cadena de prueba
de prueba
```

Ejemplo 10.6
Una función que recorre una cadena ASCIIZ buscando un carácter y devuelve la primera dirección donde se lo halló, o bien el puntero nulo (NULL).

```
char *donde(char *p, char c)
{
    for( ; *p != 0; p++)
        if(*p == c)
            return p;
    return NULL;
}

main()
{
    char *cadena = "Buscando exactamente esto";
    char *s;
    s = donde(cadena, 'e');
    if(s != NULL)
        printf("%s\n", s);
}
```

El ejemplo de uso imprime:

```
exactamente esto
```

10.5 Pasaje por referencia

En C, donde todo pasaje de parámetros a funciones se realiza **por valor**, los punteros brindan una manera de entregar a las funciones **referencias a objetos**.

El pasaje por referencia permite que una función pueda **modificar** un objeto que es local a otra función. Un pasaje por referencia de un objeto implica entregar a la función **la dirección del objeto**.

Ejemplo 10.7
Modificación de un objeto externo a una función. La función `f2()` debe poner a cero una variable entera que es externa a ella, por lo cual el argumento formal `h` debe ser la dirección de un entero.

```
void f2(int *h)
{
    *h = 0;
}

int f1()
{
    int j,k;
    int *p;

    p = &j;
    f2(p);     /* le pasamos una direccion */
    f2(&k);    /* y tambien aqui           */
}
```

Ejemplo 10.8
Uso incorrecto de argumentos pasados por valor.

```
void swap(int x, int y) /* incorrecta */
{
    int temp;
    temp = x;
    x = y;
    y= temp;
}
```

La función `swap()`, que podría ser usada por un algoritmo de ordenamiento para intercambiar los valores de dos variables, está **incorrectamente escrita**, ya que los valores que intercambia son los de sus argumentos, que vienen a estar al nivel de variables locales. El uso de la función swap() no tendrá efecto en el exterior de la misma. La versión correcta debe escribirse con pasaje por referencia:

```
void swap(int *x, int *y) /* correcta */
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

La invocación de `swap()` debe hacerse con las direcciones de los objetos a intercambiar:

```
int a, b;
swap(&a, &b);
```

10.6 Punteros y argumentos de funciones

En las funciones que reciben direcciones, los argumentos formales pueden tener cualquiera de dos notaciones: como punteros, o como arreglos. No importa qué objeto sea exactamente el argumento real (arreglo o puntero): en ambos casos, la función únicamente **recibe una dirección** y no sabe, ni le importa, cuál es la naturaleza real del objeto exterior a ella.

Ejemplo 10.9
La función que busca un carácter en una cadena, vista más arriba, puede escribirse correctamente así, cambiando el tipo del argumento formal. El uso es exactamente el mismo que antes, sin cambios en la función que llama.

```
char *donde(char p[], char c)
{
    int i;
    for(i=0 ; p[i] != 0; i++)
        if(p[i] == c)
            return p+i;
    return NULL;
}
```

Nótese que dentro del cuerpo de la función podemos seguir utilizando la notación de punteros si lo deseamos, aun con la declaración del argumento formal como arreglo.

```
char *donde(char p[], char c) /* el argumento p es un arreglo */
{
    for( ; *p != 0; p++)    /* pero se le puede aplicar el */
        if(*p == c)         /* operador de dereferenciacion */
            return p;
    return NULL;
}
```

Del mismo modo, si quisiéramos, podríamos representar los argumentos como punteros y manipular los datos con indexación. Todo esto se debe, por un lado, a que las notaciones `*p` y `p[]`, para argumentos formales, expresan únicamente que el argumento es una dirección; y por otro lado, a la equivalencia entre las formas de acceso mediante apuntadores y mediante índices de arreglos.

¡Esto no quiere decir que punteros y arreglos sean lo mismo! Véanse las observaciones en la próxima unidad.

10.7 Ejercicios

1. Dado el programa siguiente, ¿a dónde apunta `k1`?

```
main()
{
    int k;
    int *k1;
}
```

2. Dado el programa siguiente, ¿a dónde apunta `m1`?

```
int *m1;
main()
{
    ...
}
```

3. ¿Cuánto espacio de almacenamiento ocupa un arreglo de diez enteros? ¿Cuánto espacio de almacenamiento ocupa un puntero a entero?

4. Declarar variables `long` llamadas a, b y c, y punteros a `long` llamados p, q y r; y dar las sentencias en C para realizar las operaciones siguientes. Para cada caso, esquematizar el estado final de la memoria.

   a. Cargar p con la dirección de a. Si ahora escribimos `*p = 1000`, ¿qué ocurre?
   b. Cargar r con el contenido de p. Si ahora escribimos `*r = 1000`, ¿qué ocurre?
   c. Cargar q con la dirección de b, y usar q para almacenar una constante `4L` en el espacio de b.
   d. Cargar en c la suma de a y b, pero sin escribir la expresión $a + b$.
   e. Almacenar en c la suma de a y b pero haciendo todos los accesos a las variables en forma indirecta.

5. Compilar y ejecutar:

   a.
```
main()
{
    char *a = "Ejemplo";
    printf("%s\n",a);
}
```

   b.
```
main()
{
    char *a;
    printf("%s\n",a);
}
```

c.
```
main()
{
    char *a = "Ejemplo";
    char *p;
    p = a;
    printf("%s\n", p);
}
```

6. ¿Qué imprimirán estas sentencias?

   a. `char *s = "ABCDEFG";`
   b. `printf("%s\n", s);`
   c. `printf("%s\n", s + 0);`
   d. `printf("%s\n", s + 1);`
   e. `printf("%s\n", s + 6);`
   f. `printf("%s\n", s + 7);`
   g. `printf("%s\n", s + 8);`

7. ¿Son correctas estas sentencias? Bosqueje un diagrama del estado final de la memoria para aquellas que lo sean.

   a. `char *a = "Uno";`
   b. `char *a, b; a = "Uno"; b = "Dos";`
   c. `char *a,*b ; a = "Uno"; b = a;`
   d. `char *a,*b ; a = "Uno"; b = *a;`
   e. `char *a,*b ; a = "Uno"; *b = a;`
   f. `char *a = "Dos"; *a = 'T';`
   g. `char *a = "Dos"; a = "T";`
   h. `char *a = "Dos"; *(a + 1)= 'i'; *(a + 2)= 'a';`
   i. `char *a, *b ; b = a;`

8. Escribir funciones para:

   a. Calcular la longitud de una cadena.
   b. Dado un carácter determinado y una cadena, devolver la primera posición de la cadena en la que se lo encuentre, o bien -1 si no se halla.
   c. Buscar una subcadena en otra, devolviendo un puntero a la posición donde se la halle.

9. Escribir una función para reemplazar en una cadena todas las ocurrencias de un carácter dado por otro, suponiendo:

   a. Que no interesa conservar la cadena original, sino que se reemplazarán los caracteres sobre la misma cadena.
   b. Que se pretende obtener una segunda copia, modificada, de la cadena original, sin destruirla.

10. Escribir funciones para:

a. Rellenar una cadena con un carácter dado, hasta que se encuentre el 0 final, o hasta alcanzar n iteraciones.

b. Pasar una cadena a mayúsculas o minúsculas.

11. Reescriba dos de las funciones escritas en 8 y dos de las escritas en 10 usando la notación opuesta (cambiando punteros por arreglos).

10.8 Errores más frecuentes

Punteros sin inicializar

El utilizar un puntero sin inicializar parece estar entre las primeras causas de errores en C.

Ejemplo 10.10
Si bien sintácticamente correctas, las líneas del ejemplo presentan un problema muy común en C. Declaran un puntero a entero, `p`, y almacenan un valor entero en la dirección apuntada por el mismo.

```
/* Incorrecto */
   int *p;
   *p = 8;
```

Como `p` es un puntero, su contenido será interpretado como una dirección de memoria. El problema es que el contenido de `p` es **impredecible**:

- Si la variable `p` es local, su clase de almacenamiento es **auto** y por lo tanto contiene **basura**, salvo inicialización explícita.
- Si `p` es externa, **será inicializada a 0**.

En el primer caso, esa dirección es aleatoria. En el segundo caso, será `cero`, que en la mayoría de los sistemas operativos conocidos es una dirección **inaccesible** para los procesos no privilegiados.

En cualquier caso, el programa compilará pero se encontrará con problemas de ejecución. Lo que falta es hacer que `p` apunte a alguna zona válida.

Direcciones válidas, que pueden ser manipuladas mediante punteros, son las de los objetos conocidos por el programa: **variables, estructuras, arreglos, funciones, bloques de memoria asignados dinámicamente**, son todos objetos cuya dirección ha sido **obtenida legítimamente**, ya sea al momento de carga o al momento de ejecución.

Si un puntero no está explícitamente inicializado a alguna dirección válida dentro del espacio del programa, se estará escribiendo en **alguna dirección potencialmente prohibida**. En el mejor de los casos, el programa intentará escribir en el espacio de memoria de otro proceso y el sistema operativo **lo terminará**. En casos más sutiles, el programa continuará funcionando pero luego de **corromper algún área de memoria impredecible** dentro del espacio del proceso.
Este problema es por lo general difícil de detectar, porque el efecto puede no mostrar ninguna relación con el origen del problema. El error puede estar en una instrucción que se ejecuta pero corrompe la memoria, y sin embargo manifestarse recién cuando se accede a esa zona de memoria corrupta. Para este momento, el control del programa puede estar en un punto arbitrariamente lejano de la instrucción que causó el problema.

Ejemplo 10.11
La solución al problema de los punteros sin inicializar es asegurarse, **siempre**, que los punteros apuntan a **lugares válidos del programa**, asignándoles **direcciones de objetos conocidos**.

```
/* Correcto */
    int a;
    int *p;
    p = &a;
    *p = 8;
```

Confundir punteros con arreglos

Es imprescindible comprender rigurosamente la diferencia entre arreglos y punteros. Aunque son intercambiables en algunos contextos, suponer que son lo mismo lleva a graves errores. Es frecuente confundirlos, y esta confusión es explicable a partir de algunos hechos que se enumeran a continuación. No hay que dejarse engañar por ellos.

1. **Ambos se evalúan a direcciones.**

   El nombre de un arreglo equivale a una dirección, y usar un puntero equivale a usar la dirección que contiene. Es decir, tienen usos similares.

   a.
   ```
   char formato[] = "%d %d\n";
   printf(formato, 5, -1);
   ```

   b.
   ```
   char *formato = "%d %d\n";
   printf(formato, 5, -1);
   ```

   Aquí usamos como equivalentes a un arreglo y a un puntero, porque de cualquiera de las dos maneras estamos expresando una dirección en la invocación a `printf()`.

2. **Como argumentos formales, son equivalentes.**

   En una función, un argumento formal que sea una dirección puede ser declarado como puntero o como arreglo, intercambiablemente. Convirtamos los ejemplos de más arriba a funciones:

   a.
   ```
   int fun(char *s, int x, int y)
   {
         printf(s, x, y);
   }
   ```

   b.
   ```
   int fun(char s[], int x, int y)
   {
          printf(s, x, y);
   }
   ```

   Ambas formas son válidas, porque lo único que estamos expresando es que un argumento es una dirección; y, como se ha dicho, tanto punteros como nombres de arreglos los representan. Además, cualquiera de las funciones escritas puede usarse en cualquiera de las dos maneras siguientes, pasando punteros o arreglos en la llamada:

   a.
   ```
   char formato[] = "%d %d\n";
     fun(formato, 5, -1);
   ```

   b.
   ```
   char *formato = "%d %d\n";
     fun(formato, 5, -1);
   ```

3. **Comparten operadores.**

   Se pueden aplicar los mismos operadores de acceso a ambos; a saber, se puede dereferenciar un arreglo (igual que un puntero) para acceder a un elemento y se puede indexar un puntero (como un arreglo) para acceder a una posición dentro del espacio al que apunta.

a.
```
char cadena[] = "abcdefghijkl";
   char c;
   c = *(cadena + 4); /* c = 'e' */
```

b.
```
char *cadena = "abcdefghijkl";
   char c;
   c = cadena[4];   /* c = 'e' */
```

En muchas formas, entonces, punteros y arreglos pueden ser intercambiados porque son dos maneras de acceder a direcciones de memoria, pero **un arreglo no es un puntero**, y **ninguno de ellos es una dirección** (aunque las representan), porque:

- Un arreglo tiene memoria asignada para todos sus elementos (desde la carga del programa, para arreglos globales o `static`, y desde la entrada a la función donde se lo declara, para los arreglos locales).
- Un puntero, en cambio, solamente contiene una dirección, que puede ser o no válida en el sentido de apuntar o no a un objeto existente en el espacio direccionable por el programa. La validez de la dirección contenida en un puntero es responsabilidad del programador.

No analizar el nivel de indirección

Una variable de un tipo básico cualquiera contiene un dato que puede ser directamente utilizado en una expresión para hacer cálculos. En cambio, un puntero que apunte a esa variable contiene su dirección; es una **referencia** al dato, y necesita ser **dereferenciado** para acceder al dato. La variable y su puntero tienen **diferente nivel de indirección**.

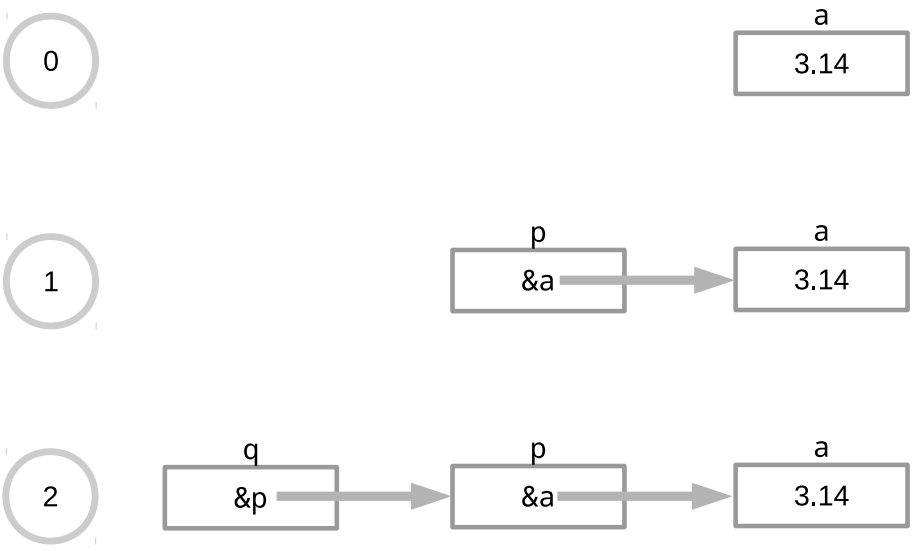

Figura 10.1: Diferentes niveles de indirección.

Un `char`, un `int`, un `long`, un `double`, una estructura de un tipo definido por el usuario, tienen un mismo nivel de indirección, que podríamos llamar "el nivel 0". Un puntero, una dirección, un nombre de arreglo, tienen un "nivel de indirección 1". La Fig. 10.1 esquematiza situaciones donde los niveles de indirección son **0, 1, y 2**.

Aplicar el operador `&` (dirección de) a algo aumenta su nivel de indirección. Aplicar el operador `*` (dereferenciación) lo decrementa.

Cuando escribimos una expresión, o hacemos una asignación, o proveemos argumentos reales para una función, etc., necesitamos que los niveles de indirección de los elementos que componen la expresión sean consistentes, es decir, que el resultado final de cada subexpresión tenga el nivel de indirección necesario.
El concepto de consistencia de niveles de indirección es análogo al que usamos al escribir una ecuación con magnitudes físicas, donde ambos miembros de la ecuación deben tener el mismo sentido físico. Por ejemplo, si $V = velocidad$, $E = espacio$, $T = tiempo$, no tiene sentido en Física escribir la igualdad $V = E * T$, simplemente porque si **multiplicamos** metros por segundo no obtenemos m/s, que son las unidades de V. Del mismo modo podemos verificar la consistencia de las expresiones en C preguntándonos qué nivel de indirección debe tener cada subexpresión.

Ejemplo 10.12
Este tipo de análisis es sumamente útil para prevenir errores de programación. Conviene utilizarlo para dar una segunda mirada crítica a las expresiones que escribimos.

```
char *s = "cadena";
char *t;
char u;
t = s + 2; /* CORRECTO */
u = s;     /* INCORRECTO */
u = *s;    /* CORRECTO */
t = &u;    /* CORRECTO */
```

La asignación `t = s + 2` es correcta porque la suma de una dirección más un entero está definida y devuelve una dirección; con lo cual la expresión mantiene el mismo nivel de indirección en ambos miembros (**puntero = dirección**).

En cambio, la asignación `u = s` intenta asignar una dirección (la dirección contenida en `s`) a un char. No se respeta el mismo nivel de indirección en ambos miembros de la asignación (**puntero = char**), de modo que ésta es incorrecta y será rechazada por el compilador.

En las dos últimas asignaciones se usan los operadores de dirección y de indirección para hacer consistentes los niveles de indirección de ambos miembros:

```
u = *s;
char = lo apuntado por(puntero a char);
char = char;
```

```
t = &u;
puntero a char = direccion de(char);
direccion de char = direccion de char;
```

10.9 Arreglos de punteros

Una construcción especialmente útil es la de **arreglos de punteros a carácter**. Esta construcción permite expresar una lista de rótulos y navegar por ellos con la indexación natural de los arreglos.

Ejemplo 10.13
Aquí el tipo de los elementos del arreglo mes es puntero a carácter. Cada elemento se inicializa en la declaración a una constante string.

```
char *mes[] = { "Ene","Feb","Mar","Abr","May","Jun",
```

```
                "Jul","Ago","Sep","Oct","Nov","Dic" };

printf("Mes: %s\n",mes[6]);
```

10.10 Estructuras referenciadas por punteros

En el caso particular de **estructuras o uniones referenciadas por punteros**, la notación para acceder a sus miembros cambia ligeramente, reemplazando el operador "punto" por "->".

Ejemplo 10.14
Accedemos a los miembros de la estructura `persona` en forma directa:

```
struct persona p;
p.DNI = 14233326;
p.edad = 40;
```

En forma indirecta:

```
struct persona p, *pp;
pp = &p;
pp->DNI = 14233326;
pp->edad = 40;
```

10.11 Estructuras de datos recursivas

Las estructuras de datos recursivas se expresan efectivamente creando miembros que sean punteros a estructuras del mismo tipo.

```
struct itemlista {
    double dato;
    struct itemlista *proximoitem;
}
```

```
struct nodoarbol {
    int valor;
    struct nodoarbol *hijoizquierdo;
    struct nodoarbol *hermanoderecho;
}
```

En cambio, no es legal la composición de estructuras dentro de sí mismas:

```
struct itemlista { /* INCORRECTO */
    double dato;
    struct itemlista proximoitem;
}
```

10.12 Construcción de tipos

Aunque la construcción de **tipos definidos por el usuario** no es una característica directamente ligada a los punteros o a las variables estructuradas, es un momento apropiado para introducirla. El lenguaje C

admite la generación de nuevos nombres para tipos (en particular, los tipos estructurados) mediante la primitiva `typedef`.

Ejemplo 10.15
Las declaraciones del ejemplo anterior se podrían reescribir más claramente de la forma que sigue.

```
typedef struct nodoarbol {
    int valor;
    struct nodoarbol *hijoizquierdo;
    struct nodoarbol *hermanoderecho;
}nodo;
typedef struct nodoarbol *nodop;
```

Entonces, el tipo de un argumento de una función podría quedar expresado sintéticamente como `nodop`:

```
nodop crearnodo(nodop padre);
```

La construcción con `typedef` no es indispensable, pero aporta claridad al estilo de programación, y, bien manejada, promueve la portabilidad.

10.13 Asignación dinámica de memoria

Se ha visto la necesidad de que los punteros apunten a direcciones válidas. ¿Qué hacer cuando la lógica del programa pide la creación de estructuras de datos en forma dinámica? Los punteros son muy convenientes para manejarlas, pero se debe asegurar que apunten a zonas de memoria legítimamente obtenidas por el programa.

En C se tiene como **herramientas básicas de gestión dinámica de memoria** a las funciones `malloc()` y `free()`. Con `malloc()` pedimos una cantidad de bytes contiguos que serán tomados del **heap**. La función `malloc()` devuelve la dirección del bloque de memoria asignado. Esta dirección debe reservarse en un puntero para uso futuro y para liberarla con `free()`.

Ejemplo 10.16
En lugar de hacer que `p` apunte a un objeto existente al momento de compilación, solicitamos tanta memoria como sea necesaria para alojar un entero y ponemos a `p` apuntando allí. Luego podemos hacer la asignación. Luego del uso se debe liberar la zona apuntada por `p`.

```
/* Correcto */
int *p;
p = malloc(sizeof(int));
*p = 8;
free(p);
```

Para ser completamente correcto, el programa del ejemplo debería verificar que `malloc()` no devuelva **NULL** por ser imposible satisfacer el requerimiento de memoria. El símbolo `NULL` corresponde a la dirección 0, o, equivalentemente, al **puntero nulo**, y nunca es una dirección utilizable.

Ejemplo 10.17
La propiedad de poder aplicar indexación a los punteros hace que, virtualmente, el C sea capaz de proporcionarnos **arreglos dimensionables en tiempo de ejecución**. En efecto:

```
double *tabla;
tabla = malloc(k);
tabla[50] = 15.25;
```

Estas líneas son virtualmente equivalentes a un arreglo de k elementos `double`, donde k, por supuesto, puede ser calculado en tiempo de ejecución.

Una variante de `malloc()` es la función `calloc()`, que solicita una cantidad dada de elementos de memoria, de un tamaño también dado, y además garantiza que todo el bloque de memoria concedido esté **inicializado con ceros binarios**.

```
float *lista;
int i;
lista = calloc(k, sizeof(float));
for(i=0; i<k; i++)
    lista[i] = fun(i);
```

10.14 Punteros a funciones

Así como se pueden tomar las direcciones de los elementos de datos, es posible manipular las **direcciones iniciales de los segmentos de código** representados por las funciones de un programa C, mediante punteros a funciones. Esta característica es **sumamente poderosa**.

La declaración de un puntero a función tiene una sintaxis algo complicada: debe indicar **el tipo devuelto** por la función y los **tipos de los argumentos**. El nombre del puntero, con el signo de "puntero" incluido, debe estar **entre paréntesis** en la declaración.

Ejemplo 10.18
Definición de un puntero llamado `p`, a una función que recibe dos enteros y devuelve un entero:

```
int (*p)(int x, int y);
```

o también:

```
int (*p)(int, int);
```

Los paréntesis alrededor de `*p` son importantes: sin ellos, se define una **función que devuelve un puntero a entero**, que no es lo que se pretende.

Asignación del puntero `p`:

```
int fun(int x, int y)
{
    ...
}
p = fun;
```

Uso del puntero `p` para invocar a la función `fun` cuya dirección tiene asignada:

```
a = (*p)(k1, 20 - k2);
```

10.15 Aplicación de punteros a funciones

La Biblioteca Standard contiene una función, `qsort()`, que realiza el ordenamiento de una tabla de datos mediante el método de **Quicksort**. Para que pueda ser completamente flexible (para poder ordenar datos de cualquier naturaleza y según cualquier criterio), la función acepta a su vez una **función provista por el usuario**, que determina el **orden de dos elementos**.

Es responsabilidad del usuario, entonces, definir cuándo un elemento es mayor que el otro, a través de esta función de comparación. La función de ordenamiento recibe un puntero a la función de comparación y la invoca repetidamente.

La función de comparación sólo debe aceptar `p1` y `p2`, dos punteros a un par de datos, y seguir el protocolo siguiente:

Si	Devolver
`*p1 < *p2`	un número menor que cero
`*p1 == *p2`	cero
`*p1 < *p2`	un número mayor que cero

Ejemplo 10.19
La declaración de argumentos formales como `void *` expresa que esos argumentos pueden tratarse de direcciones de objetos de cualquier tipo.

```
#include <stdlib.h>

struct p {
    ...
    char nombre[40];
    double salario;
    ...
} lista[100];

int cmpSalario(const void *p1, const void *p2)
{
    return p1->salario - p2->salario;
}

int cmpNombre(const void *p1, const void *p2)
{
    return strncmp(p1->nombre, p2->nombre, 40);
}
```

Con estas definiciones, la tabla **lista** se puede ordenar por uno u otro campo de la estructura con las sentencias:

```
qsort(lista, 100, sizeof(struct p),cmpSalario);
```

```
qsort(lista, 100, sizeof(struct p),cmpNombre);
```

10.16 Punteros a punteros

La indirección mediante punteros puede ser **doble, triple**, etc. Los punteros dobles tienen aplicación en el manejo de conjuntos de strings o matrices bidimensionales.

Como en el caso de punteros a funciones, esto brinda una gran potencia, pero a costa de complicar enormemente la notación y la programación, por lo que se recomienda no abordar el tema en un curso introductorio y, sólo una vez dominadas las técnicas y conceptos básicos de punteros, referirse a una fuente como el libro de Kernighan y Ritchie, 2ª edición.

Ejemplo 10.20
Un ejemplo de uso de punteros dobles.

- En estas sentencias, el puntero `q` recibe la dirección de `a`; pero `p` recibe la dirección de `q`. Luego, `p` contiene "la dirección de la dirección" de `a`.
- Similarmente, la expresión `*p` significa "lo apuntado por `p`"; pero entonces `**p` significa "lo apuntado por lo apuntado por p".

```
int **p;   /* un puntero doble */
int *q;
int a;
```

```
q = &a;    /* q recibe la direccion de a */
p = &q;    /* p recibe la direccion de q */

**p = 8;   /* la variable a recibe un valor 8 */
```

10.17 Una herramienta: gets()

Para facilitar la práctica damos la descripción de otra función de Biblioteca Standard.

La función `gets()` pide al usuario una cadena terminada por `ENTER`. Recibe como argumento un espacio de memoria (expresado por una dirección) donde copiará los caracteres tipeados. El fin de la cadena recibirá automáticamente un **cero final** para hacerla compatible con las funciones de tratamiento de strings de la Biblioteca Standard.

La función `gets()` debe recibir la dirección de un área de memoria legal para el programa y donde no haya riesgo de sobreescribir contenidos. Es importante comprender la teoría dada en este capítulo para evitar el **uso de un puntero no inicializado**, una situación de error frecuente cuando se utilizan `gets()` y funciones similares.

Ejemplo 10.21
Creamos un área de memoria utilizable por gets() simplemente definiendo un arreglo de caracteres del tamaño deseado.

```
main()
{
    char arreglo[100];
    gets(arreglo);
}
```

El puntero del siguiente ejemplo no está inicializado. Por ser una variable local, contiene **basura**, y por lo tanto apunta a un lugar impredecible.

```
main()
{
    char *s;
    gets(s); /* Mal!!! */
}
```

Podemos corregir el ejemplo anterior proveyendo espacio legítimo a donde apunte `s`, reservándolo estáticamente mediante la declaración del arreglo.

```
main()
{
    char arreglo[100];
    char *s;
    s = arreglo;
    gets(s);       /* Ahora si */
}
```

En el tercer ejemplo usamos asignación dinámica y liberación de memoria. En un sentido estricto, en este caso particular no es necesaria la liberación, porque la terminación del programa devuelve todas las estructuras creadas dinámicamente; pero es útil habituarse a la disciplina de aparear cada invocación de `malloc()` con el correspondiente `free()`.

```
main()
{
    char *s;
    s = malloc(100); /* Ahora si */
    gets(s);
```

```
    printf("ingresado: %s\n",s);
    free(s);
}
```

10.18 Ejercicios

1. ¿Qué objetos se declaran en las sentencias siguientes?
   a. double (*nu)(int kappa);
   b. int (*xi)(int *rho);
   c. long phi();
   d. int *chi;
   e. int pi[3];
   f. long *beta[3];
   g. int *(gamma[3]);
   h. int (*delta)[3];
   i. void (*eta[5])(int *rho);
   j. int *mu(long delta);
2. Construir una función que reciba un arreglo de punteros a string A y un string B, y busque a B en el array A, devolviendo su índice en el array, o bien -1 si no se halla.
3. Construir una función iterativa que imprima una cadena en forma inversa. Muestre una versión y una recursiva.
4. Construir un programa que lea una sucesión de palabras y las busque en un pequeño diccionario. Al finalizar debe imprimir la cuenta de ocurrencias de cada palabra en el diccionario.
5. Construir un programa que lea una sucesión de palabras y las almacene en un arreglo de punteros a carácter.
6. Ordenar lexicográficamente el arreglo de punteros del ejercicio 4.