Universidad Nacional del Comahue

Facultad de Economía y Administración

Lenguaje C

Eduardo Grosclaude 2001

Contenidos

1. Introducción al Lenguaje C . . . 1
   Características del lenguaje . . . 1
   El ciclo de compilación . . . 3
      Compilador . . . 3
      Linkeditor o *linker* . . . 4
      Bibliotecario . . . 4
      El utilitario Make . . . 4
   El primer ejemplo . . . 5
   Mapa de memoria de un programa . . . 6
   Ejercicios . . . 7
2. El preprocesador . . . 9
   Directivas de preprocesador . . . 9
   Ejemplos . . . 11
   Observaciones . . . 13
   Ejercicios . . . 13
3. Tipos de datos y expresiones . . . 16
   Declaración de variables . . . 16
   Tamaños de los objetos de datos . . . 17
   Operaciones con expresiones de distintos tipos . . . 18
      Truncamiento en asignaciones . . . 18
      Promoción automática de expresiones . . . 19
      Operador *cast* . . . 19
   Reglas de promoción en expresiones . . . 20
   Observaciones . . . 20
   Una herramienta: printf() . . . 21
      Ejemplos . . . 21
   Ejercicios . . . 22
4. Constantes . . . 25
   Constantes enteras . . . 25
   Constantes long . . . 25
   Constantes unsigned . . . 25
   Constantes string . . . 25
   Constantes de carácter . . . 27
   Constantes de carácter en strings . . . 28
   Constantes de punto flotante . . . 28
   Constantes enumeradas . . . 28
      Ejemplos . . . 28
   Ejercicios . . . 29
5. Propiedades de las variables . . . 30
   Alcance de las variables . . . 30
      Ejemplo . . . 30
   Vida de las variables . . . 31
      Ejemplo . . . 31
      Ejemplo . . . 31
   Clases de almacenamiento . . . 31
      Estáticas . . . 32
      Automáticas . . . 32
      Registro . . . 32

Ejemplo . . . 32
Ejemplo . . . 33
Variables y mapa de memoria . . . 33
Ejemplo . . . 34
Liga . . . 34
Ejemplo . . . 34
Ejemplo . . . 36
Declaración y definición de variables . . . 36
Ejemplo . . . 36
Modificadores especiales . . . 37
Const . . . 37
Volatile . . . 37
Ejercicios . . . 38
6. Operadores . . . 42
Operadores aritméticos . . . 42
Ejemplo . . . 42
Ejemplo . . . 42
Ejemplos . . . 43
Operadores de relación . . . 43
Operadores lógicos . . . 43
Ejemplo . . . 43
Ejemplos . . . 44
Operadores de bits . . . 44
Ejemplos . . . 44
Ejemplos . . . 44
Operadores especiales . . . 45
Ejemplo . . . 45
Precedencia y orden de evaluación . . . 45
Resumen . . . 46
Ejercicios . . . 46
7. Estructuras de control . . . 49
Estructura alternativa . . . 49
Ejemplos . . . 49
Estructuras repetitivas . . . 49
Estructura while . . . 49
Estructura do...while . . . 50
Estructura for . . . 50
Ejemplos . . . 51
Estructura de selección . . . 52
Ejemplo . . . 52
Transferencia incondicional . . . 53
Sentencia continue . . . 53
Sentencia break . . . 53
Sentencia goto . . . 54
La sentencia return . . . 54
Observaciones . . . 54
Ejercicios . . . 54
8. Funciones . . . 57
Declaración y definición de funciones . . . 57
Ejemplo . . . 57

Prototipos de funciones . . . 58
Redeclaración de funciones . . . 58
Recursividad . . . 59
Ejercicios . . . 59
9. Arreglos y variables estructuradas . . . 61
Ejemplos . . . 61
Inicialización de arreglos . . . 61
Errores frecuentes . . . 61
1. Indexación fuera de límites . . . 61
2. Asignación de arreglos . . . 62
Arreglos multidimensionales . . . 63
Estructuras y uniones . . . 63
Ejemplos . . . 63
Uniones . . . 64
Campos de bits . . . 65
Ejercicios . . . 66
10. Apuntadores y direcciones . . . 68
Operadores especiales . . . 69
Aritmética de punteros . . . 70
Asignación entre punteros . . . 70
Suma de enteros a punteros . . . 70
Resta de punteros . . . 71
Punteros y arreglos . . . 71
Ejemplos . . . 72
Punteros y cadenas de texto . . . 72
Ejemplo . . . 73
Ejemplo . . . 74
Ejemplo . . . 74
Pasaje por referencia . . . 75
Ejemplos . . . 75
Punteros y argumentos de funciones . . . 76
Ejemplo . . . 76
Ejercicios . . . 76
10b. Temas avanzados de apuntadores y direcciones . . . 80
Observaciones . . . 80
1. Punteros sin inicializar . . . 80
Ejemplo . . . 80
2. Confundir punteros con arreglos . . . 81
3. No analizar el nivel de indirección . . . 82
Ejemplos . . . 82
Arreglos de punteros . . . 83
Ejemplo . . . 83
Estructuras referenciadas por punteros . . . 83
Ejemplo . . . 83
Estructuras de datos recursivas y punteros . . . 83
Construcción de tipos . . . 84
Ejemplo . . . 84
Asignación dinámica de memoria . . . 84
Ejemplo . . . 84
Ejemplo . . . 85

Punteros a funciones . . . 85
Ejemplos . . . 85
Aplicación . . . 86
Ejemplo . . . 86
Punteros a punteros . . . 87
Ejemplo . . . 87
Una herramienta: gets() . . . 87
Ejemplos . . . 87
Ejercicios . . . 88
11. Entrada/salida . . . 90
Funciones de E/S standard . . . 90
E/S standard de caracteres . . . 91
E/S standard de líneas . . . 91
Ejemplo . . . 91
E/S standard con formato . . . 91
Ejemplo . . . 92
E/S standard sobre strings . . . 92
Ejemplo . . . 92
E/S sobre archivos . . . 93
Funciones ANSI C de E/S sobre archivos . . . 94
Modos de acceso . . . 95
Funciones ANSI C de caracteres sobre archivos . . . 95
Funciones ANSI C de líneas sobre archivos . . . 95
Funciones ANSI C con formato sobre archivos . . . 96
Ejemplo . . . 96
Funciones ANSI C de acceso directo . . . 96
Ejemplo . . . 96
Sincronización de E/S . . . 97
Resumen de funciones ANSI C de E/S . . . 97
Funciones POSIX de E/S sobre archivos . . . 97
Ejemplo . . . 98
Ejemplo . . . 99
Ejemplo . . . 99
Ejercicios . . . 100
12. Comunicación con el ambiente . . . 101
Redirección y piping . . . 101
Variables de ambiente . . . 103
Ejemplo . . . 103
Argumentos de ejecución . . . 103
Ejemplo . . . 103
Salida del programa . . . 104
Ejemplo . . . 104
Opciones . . . 104
Ejemplo . . . 104
Ejercicios . . . 106
13. Biblioteca Standard . . . 107
Funciones de strings (<stdio.h>) . . . 108
Listas de argumentos variables (<stdarg.h>) . . . 108
Ejemplo . . . 109
Funciones de tratamiento de errores (<errno.h> y <assert.h>) . . . 109

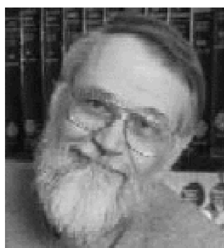
Ejemplos 109
Funciones de fecha y hora (<time.h>) 110
Ejemplo 110
Funciones matemáticas (<math.h>) 110
Funciones utilitarias (<stdlib.h>) 111
Clasificación de caracteres (<ctype.h>) 112
Ejercicios 112

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 (*portarlo*) a diferentes arquitecturas. Por este motivo sus creadores se propusieron metas de diseño especiales, tales como:

- Posibilidad de acceder al "bajo nivel" (poder utilizar todos los recursos del hardware).
- Código generado eficiente en memoria y tiempo (programas pequeños y veloces)
- Compilador portable (implementable en todas las arquitecturas)

La primera definición oficial del lenguaje fue dada en 1978 por **Brian Kernighan y Dennis Ritchie** en su libro "El lenguaje de programación C". Este lenguaje fue llamado "C K&R". En 1983 se creó el comité ANSI que en 1988 estableció el standard 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.



Brian Kernighan

Dennis Ritchie

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 la gran mayoría de los modernos sistemas y ambientes operativos y programas de aplicación que corren sobre ellos.

Pese a la gran cantidad de tiempo que lleva en actividad el lenguaje, todavía es objeto de reformulaciones. En 1999 los organismos de estandarización ISO y ANSI, conjuntamente, han declarado adoptada la norma llamada "C99" o "C2k". A partir de este momento los compiladores han empezado a incorporar, paulatinamente, todas las características de la norma; este es un proceso que llevará una cantidad de tiempo difícil de predecir. Es posible esperar la publicación de un standard C04 en el año 2004, ya que en teoría el ISO revisa sus normas cada cinco años.

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 como Pascal.

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

Sin embargo, 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. Las que 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).

El usuario puede escribir sus propios procedimientos (llamados **funciones** aunque no devuelvan valores). Aunque existe la noción de bloque de sentencias, el lenguaje se dice *no* estructurado en bloques porque no pueden definirse funciones dentro de otras. 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.

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. 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. Permite 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. En este sentido, según los partidarios de la tipificación estricta, C no es un buen lenguaje. Gran parte del esfuerzo de desarrollo del standard ANSI se dedicó a dotar al C de elementos para mejorar esta deficiencia.

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.

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 le son pasados 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 chequeos en tiempo de ejecución, sean los que le dan al C fama de "difícil de aprender".

Por último, el C no es un lenguaje orientado a objetos, sino que adhiere al paradigma tradicional de programación 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.

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

El ciclo de compilación

Las herramientas esenciales de un ambiente de desarrollo, además de cualquier editor de textos, son el **compilador**, el **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, chequear sintaxis en forma previa a la compilación, etc. Según el ambiente operativo y producto de software de que se trate, estas herramientas pueden encontrarse integradas en una interfaz de usuario uniforme, en modo texto o gráfico, o ser comandos de línea independientes, con salidas de texto simples.

Compilador

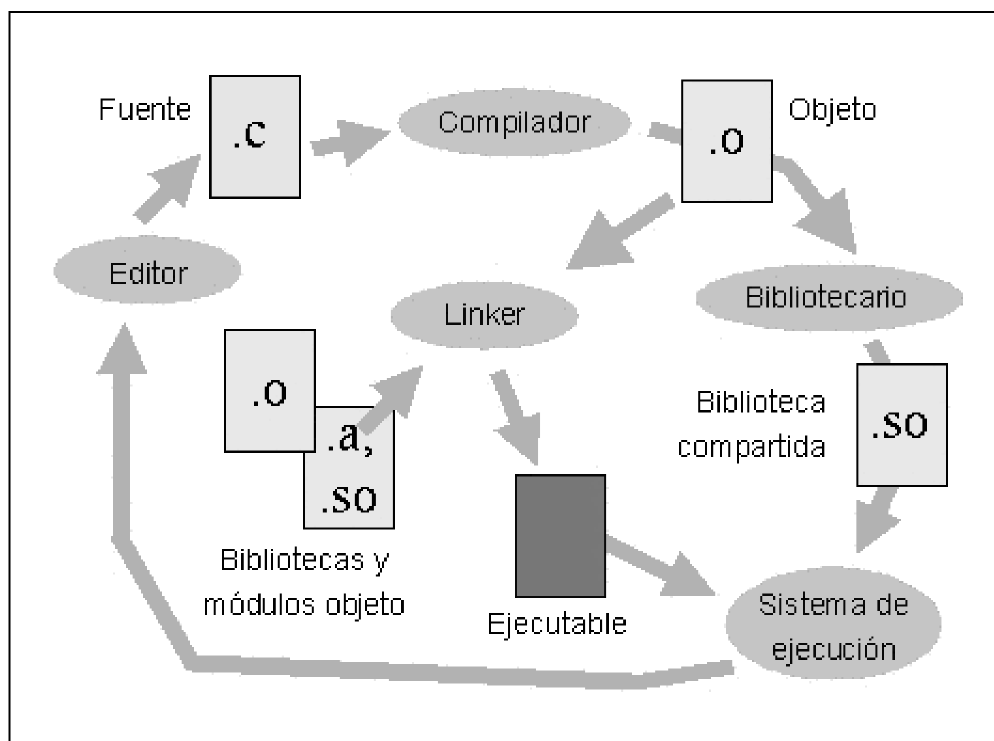
El compilador acepta un archivo fuente, posiblemente relacionado con otros (una **unidad de traducción**), y genera con él un **módulo objeto**. Este módulo objeto contiene porciones de código ejecutable mezclado con referencias, aún no resueltas, a variables o funciones cuya definición no figura en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

Linkeditor o *linker*

El linkeditor recibe como entrada un conjunto de módulos objeto y busca resolver, o enlazar, las referencias simbólicas en ellos, buscando la definición de las variables o funciones faltantes en los mismos objetos o en bibliotecas. Estas pueden ser la biblioteca standard u otras provistas por el usuario. Cuando encuentra la definición de un objeto buscado (es decir, de una variable o función), el linker 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.

Bibliotecario

El bibliotecario es un 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.



El ciclo de compilación produce un ejecutable a partir de archivos fuente

El utilitario Make

Un proyecto de programación se compone de un conjunto de varios archivos fuente, archivos *header* o de inclusión, posiblemente módulos objeto resultantes de compilaciones anteriores, y bibliotecas. Para automatizar el proceso de desarrollo se puede recurrir al comando **make**, que cuenta con un cierto conocimiento y aplica una serie de reglas para saber qué acciones tomar a fin de producir un programa ejecutable.

En particular, **make** adivina, por la extensión de cada archivo, cuáles archivos son fuentes, cuáles son módulos objeto, etc. Tomando en cuenta las fechas de última modificación de cada archivo, es capaz de saber cuáles archivos deben recompilarse o relinkearse para generar un ejecutable actualizado. El programa **make** efectuará únicamente las tareas necesarias para actualizar el proyecto con el mínimo trabajo posible.

A estas reglas por defecto que sigue **make**, el usuario puede agregar otras; por ejemplo, si el proyecto incluye módulos compilables en otros lenguajes. En este caso deberá indicar la manera de llamar al compilador indicado. Estas reglas agregadas, así como la enumeración de los archivos que componen el proyecto, son especificadas en un archivo especial, llamado en UNIX *makefile*.

El primer ejemplo

El clásico ejemplo de todas las introducciones al lenguaje C es el programa **hello.c**:

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

- Este programa minimal comienza con una **directiva de preprocesador** que indica incluir en la unidad de traducción al archivo *header* **stdio.h**. Este contiene, entre otras cosas, la declaración (o **prototipo**) de la función de salida de caracteres **printf**. 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). 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").

Sometemos este programa al comando **cc**:

```
$ cc hello.c
```

El resultado debería ser un nuevo archivo llamado **a.out**. Este es el ejecutable deseado.

Otra manera de compilar el programa es simplemente llamando al comando **make**:

```
$ make hello
```

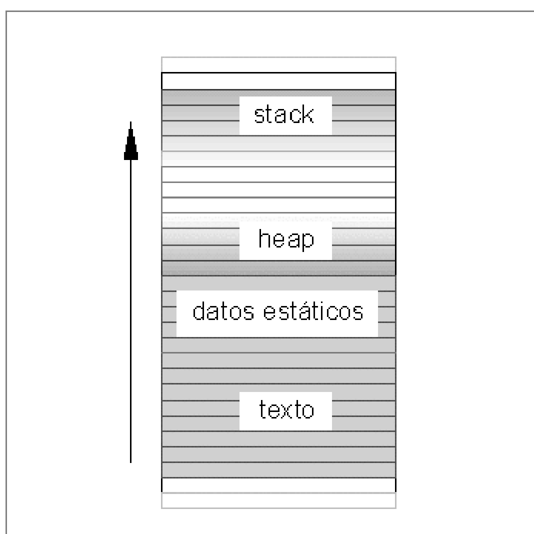
El comando **make** contiene la inteligencia para buscar, en el directorio activo, archivos fuente llamados **hello.*;** determinar (a partir de la extensión) que el hallado se trata de un programa en C; invocar con una cantidad de opciones default al compilador **cc**, y renombrar la salida con el nombre **hello**. Este será el ejecutable deseado.

Se 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).

Mapa de memoria de un programa

El programa compilado queda contenido en un archivo, pero al ser invocado, se carga en memoria y allí se despliega en una cantidad de secciones de diferentes tamaños y con distintas funciones. La manera como se distribuyen realmente estas secciones en la memoria física depende fuertemente de la forma de administración de memoria del sistema operativo para el cual está construido. Sin embargo, el siguiente modelo ideal puede servir de referencia para ilustrar algunas particularidades y problemas que irán surgiendo con el estudio del lenguaje.

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



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

La región de código contendrá el **texto** 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 C fuente 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).

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.

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).

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. En este caso, el límite del heap se irá desplazando hacia las direcciones superiores. Es su responsabilidad, también, liberar esta memoria cuando no ya sea necesaria, ya que 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.

Volveremos sobre este modelo en varias ocasiones para explicar algunas cuestiones especiales.

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. Localice en su sistema directorios que contengan bibliotecas, fuentes, módulos objeto.

5. Tomemos un archivo conteniendo un módulo objeto o una biblioteca. ¿Qué vemos con los comandos siguientes?

```
od -bc archivo | more
strings archivo | more
```

6. Investigue en su sistema UNIX, usando el comando **man**, el funcionamiento y las opciones de los comandos **make**, **cc, ld, ar**.

7. ¿Qué pasaría si un programa C **no** contuviera una función **main**()? ¿Serviría de algo?

8. Copie, compile y ejecute el programa **hello.c** del ejemplo. Verifique en cada momento del proceso en qué lugar del ciclo de compilación se encuentra y qué herramientas está utilizando.

9. Edite el programa **hello.c** y modifíquelo según las pautas que siguen. Interprete los errores de compilación. Si resulta un programa ejecutable, vea qué hace el programa.

- 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

2. El preprocesador

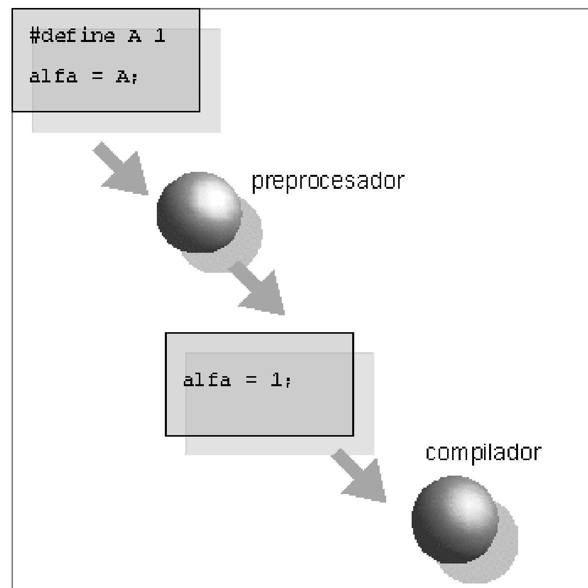
El compilador C tiene un componente auxiliar llamado **preprocesador**, que actúa en la primera etapa del proceso de compilación. Su misión es buscar, en el texto del programa fuente entregado al compilador, ciertas directivas que le indican realizar alguna tarea a nivel de texto. Por ejemplo, inclusión 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 manualmente por el usuario, utilizando los comandos de un editor de texto ("incluir archivo" y "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.

Directivas de preprocesador

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 a 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.

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.



El preprocesador realiza ediciones automáticas, en línea, de los fuentes antes de entregar el resultado al compilador.

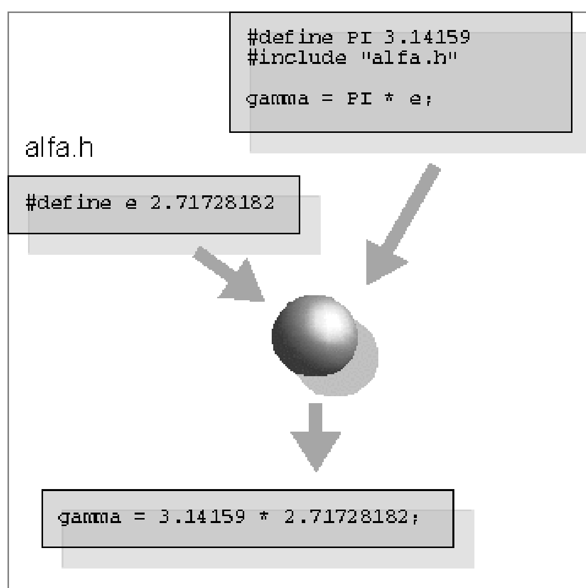
Una de las funciones del preprocesador es sustituir **símbolos**, o cadenas de texto dadas, por otras. La directiva **#define** establece la relación entre los símbolos y su expansión o cadena a sustituir.

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 serán sustituidas 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 después de la aparición de la directiva que la define, y no antes. Tampoco puede utilizarse en una unidad de traducción diferente (los símbolos de preprocesador no se "propagan" entre unidades de traducción salvo por el uso de directivas de inclusión).

Las directivas para incluir archivos suelen darse al principio de los programas, porque 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. Un archivo fuente, junto con todos los archivos que incluya, es llamado una unidad de traducción.

En los párrafos anteriores, nótese que decimos **declarar funciones**, y no **definirlas**; la diferencia es importante y se verá con detalle 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 los tipos y cantidad de los argumentos, etc.

No se considera buena práctica de programación colocar la **definición** de una función de uso frecuente en un header. Esto forzaría 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 linkedición.



La directiva **#include** hace que el preprocesador inserte y preprocese otros archivos en el punto donde se indica la directiva. El resultado de preprocesar el archivo incluido puede ser definir otros símbolos y macros, o aun incluir otros archivos.

Los archivos destinados a ser incluidos son habitualmente llamados archivos de cabecera o *headers*.

Las directivas de inclusión son anidables, es decir, pueden incluirse headers que a su vez contengan directivas de inclusión.

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.

Ejemplos

1) Si el programa dice:

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

Es mucho más claro poner:

```
#define PI     3.14159
#define RADIO  20.299322
a=2*PI*RADIO;
```

2) Con estas 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 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;
```

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

```
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)
```

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.

3) Con las directivas condicionales:

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

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

En el primer caso, 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. En el segundo caso, 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.

4) 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
```

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.

Ejercicios

1. Dé ejemplos de directivas de preprocesador:

- para incluir un archivo proporcionado por el compilador
- para incluir un archivo confeccionado por el usuario

- para definir una constante numérica
- para compilar un segmento de programa bajo la condición de estar definida una constante
- idem bajo la condición de ser no definida
- idem bajo la condición de que un símbolo valga una cierta constante
- 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__**.

3. Tipos de datos y expresiones

Las expresiones se construyen, en general, conectando mediante operadores elementos diversos, tales como identificadores de variables, constantes e invocaciones de funciones. Cada uno de estos elementos 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 para contener valores. Estos espacios de memoria son de diferentes "tamaños" (cantidades de bits) de acuerdo al tipo de dato de la subexpresión.

Así, las expresiones 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.

Declaración de variables

Los tipos básicos son:

char (un elemento de tamaño "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;
```

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, 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 que 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 nuevo valor de **a** se trunca a 16 bits, volviendo 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.

Operaciones con expresiones de distintos tipos

En una expresión en C pueden aparecer componentes de diferentes tipos. Durante la evaluación de una expresión donde sus 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 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 esa expresión 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 con enteros 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 (más adelante las damos con precisión), estas reglas dicen que el compilador hará estrictamente las conversiones necesarias para llevar todos los operandos al tipo del mayor. El resultado de evaluar una operación aritmética será del tipo del mayor de sus operandos, en el sentido del tamaño en bits de cada objeto de datos.

A veces esto no es lo deseado. 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.

Cómo recuperar el valor correcto, con decimales, de la división? Declarar **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 constantes como constante en punto flotante (con punto decimal, o en notación exponencial):

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

Operador *cast*

La segunda consiste en forzar explícitamente una conversión de tipo, con un importante operador llamado ***cast***, así:

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

Da lo mismo aplicar el operador **cast** a cualquiera de las constantes. Sin embargo, lo siguiente no es útil:

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

Aquí el operador **cast** se aplicará a la expresión ya evaluada como entero, con lo que volvemos a tener un 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.). Esta 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.

Observaciones

- Nótese que no existen tipos **boolean** ni **string**. Más adelante veremos cómo manejar datos de estas clases.
- El tipo char no está restringido a la representación de caracteres, como en Pascal. 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.

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.
- 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 š%Š. 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 son:

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

Ejemplos

- Este programa escribe algunos valores con dos especificaciones 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
```

- El programa siguiente escribe el mismo valor en doble precisión pero con diferentes modificadores del campo correspondiente.

```
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=%.10lf\n",d);
}
```

Salida:

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

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?

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 por 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;  /* conversión int/double */
    printf("x = %10.6le\ti = %d\n",x,i);
    x = 14.999;
    i = x;  /* conversión double/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 resutlados apropiados:

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


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

4. Constantes

Las constantes numéricas en un programa C pueden escribirse en varias bases.

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 un computador 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, 43l
```

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 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 o **bss**.

Así, una constante string equivale a una dirección de memoria**: la dirección donde está almacenado el primer carácter**. El carácter '\0' del final señaliza el fin de la cadena, y sirve como protocolo para las funciones de biblioteca standard que se ocupan de manejo de strings.

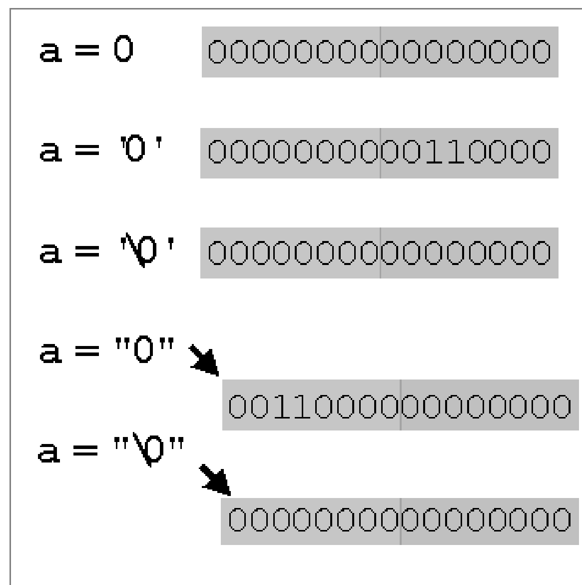
¿Hay diferencias entre '\0', '0' y "0"? Muchas.

- La primera constante es un entero. Su valor es 0.
- La segunda es de carácter. Ocupa un objeto de datos de 8 bits de tamaño. Su valor es 48 decimal en computadoras cuyo juego de caracteres esté basado en ASCII.
- La tercera es una constante string, y se evalúa a una dirección. 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 48 decimal (el primer y único carácter del string) y a continuación viene un byte 0 (señal de fin del string).

Si tenemos las declaraciones y asignaciones siguientes:

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

La primera asignación es perfectamente válida y equivale a **a=0**. La segunda también es correcta y equivale a **b=48** en computadores basados en ASCII. La tercera 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 referencia, indirectamente, a un dato (una dirección). Se dice que la variable y la constante string **tienen diferente nivel de indirección.**



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.

Constantes de carácter

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 pensamos 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).

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.

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.

Ejemplos

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

En este ejemplo, los valores de las constantes son ENE = 1, FEB = 2, MAR = 3, etc.

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.

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

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

Ejercicios

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

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

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.

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 diferencia 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

```
int m;
int fun1()
{
    int m;

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

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

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

Cada vez que **fun2()** asigna el resultado de **fun1()** a **j**, está utilizando la misma variable **j**, porque es externa; pero cada invocación de **fun1()** crea una nueva variable **k**, la que se destruye al terminar esta función.

Ejemplo

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

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 modificar, para la variable **k**. Si lee su valor, encontrará el contenido que pueda haberle quedado de la invocación anterior. Si lo modifica, la invocación siguiente encontrará ese valor en **k**. Este ejemplo muestra que alcance y vida no son propiedades equivalentes en C. La verdadera diferencia entre ambas formas de **k** es la **clase de almacenamiento**; en el primer caso, **k** es local y automática; en el segundo, **k** es local pero estática.

Clases de almacenamiento

Dependiendo de cómo son almacenados los contenidos de las variables, 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.**

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**.

Automáticas

Esta clase abarca exclusivamente las variables, declaradas localmente a una función, que no sean declaradas **static**. 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.

Registro

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

Ejemplo

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

Aquí **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 como una *recomendación* al compilador, ya que no hay seguridad 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 arreglos, variables de control de lazos, etc.

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 por parte del programador.

¿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

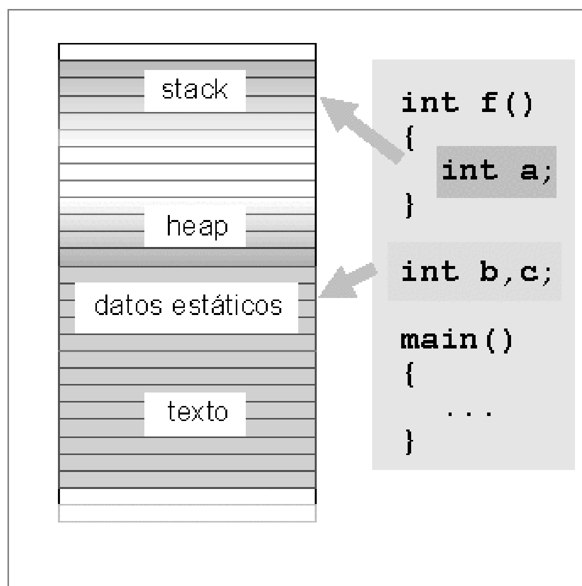
```
int veces()
{
        static int vez=0;
        return ++vez;
}
int fun()
{
        while(veces() <= 50) {
                ...
        }
}
```

Aquí el lazo **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 a cada instancia de la misma.

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 el stack. 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*.

Ejemplo

```
int fun1()
{
   int a, b;
   fun2(a,b);
}


int fun2()
{
   int c;
   ...
}
```

	stack
Antes de entrar a fun1()	-
	-
	-
Al entrar a fun1()	**a**
	b
	-
Al entrar a fun2()	**a**
	b
	c
Al salir de fun2() y volver a fun1()	**a**
	b
	-
Al salir de fun1()	-
	-
	-

Liga

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 **las variables externas y las funciones**, de modo que todas éstas pueden ser referenciadas desde otras unidades de traducción.

Ejemplo

alfa.c	beta.c	gamma.c

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

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.

En el ejemplo dado, fun1(), fun2() y fun3() están definidas en diferentes unidades de traducción que 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 hace que el linker resuelva las referencias pendientes a fun1() y a fun2() (que no están definidas en alfa.c). Por motivos similares **gamma**necesita de **beta** al momento de linkedición.

En la práctica logramos esto de varias maneras; o bien con:

```
$ cc alfa.c beta.c gamma.c -o alfa
```

Que significa "compilar separadamente los tres fuentes, linkeditarlos juntos y al ejecutable resultado renombrarlo como **alfa**"; o bien con:

```
$ cc -c alfa.c
$ cc -c beta.c
$ cc -c gamma.c
$ cc alfa.o beta.o gamma.o -o alfa
```

Que es la misma tarea pero distribuida en etapas separadas. Una tercera manera es preparar un *makefile* indicando este modo de construcción e invocar a **make**.

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.

Ejemplo

<table>
<tr><th>iota.c</th><th>kappa.c</th><th>lambda.c</th></tr>
<tr><td><pre>main()
{
    fun1();
    fun2();
}</pre></td><td><pre>int fun1()
{
    ...
}

static int fun3()
{
    ...
}</pre></td><td><pre>int fun2()
{
    ...
    fun3();
    ....
}</pre></td></tr>
</table>

Este ejemplo 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.

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).

Declaración y definición de variables

Normalmente una **declaración** de variable (de la forma *especificación_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.

La excepción a esto son los objetos declarados **extern**. Cuando la declaración de una variable cualquiera aparece precedida del modificador **extern**, ésta indica el tipo asociado, pero no habilita al compilador para crear el objeto de datos; se trata de una variable cuya **definición** debe ser encontrada en otra unidad de traducción. La declaración **extern** tan sólo enuncia el tipo y nombre de la variable para que el compilador lo tenga en cuenta.

Una variable externa es visible desde todas las funciones de la unidad de traducción y 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

delta.c	eta.c

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

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).

Modificadores especiales

Const

El modificador **const** indica que una variable no será modificada. 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 inicialización */
a++;            /* el compilador no aprobará 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.

Por ejemplo, el ciclo siguiente 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 optimizar	"optimizado"

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

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.

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	**fuente2.c**

`int a;`
 	
`int fun1(int x)`
`{`
`    return 2 * x;`
`}`
	`main()`
	`{`
	`    a = 1;`
	`    printf("d\n", fun1(a));`
	`}`

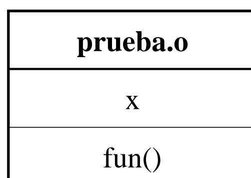
5. ¿Qué ocurre si un fuente intenta modificar una variable externa, declarada en otra unidad de traducción como **const**? Prepare, compile y ejecute un ejemplo.

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

header.h	**fuente1.c**	**fuente2.c**

`#include <stdio.h>`
`#define VALOR 6`
	`#include "header.h"`
	`main()`
	`{`
	`    static int c;`
	`    printf("%d\n",fun(c));`
	`}`
		`#include "header.h"`
		`int fun(int x)`
		`{`
		`    return VALOR * x;`
		`}`

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



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 siguientes. ¿Hay colisión de nombres? ¿Hay referencias

que el linker no pueda resolver? Cada grupo de fuentes, ¿puede producir un ejecutable?

	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(š%dŠ,
				    fun1(3));
				}

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.

6. Operadores

Operadores aritméticos

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

+, -, *, / (operaciones aritméticas usuales)

% (operador módulo)

++, -- (incremento y decremento)

Ejemplo

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

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.

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

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

La división de **j** por **k** es **entera**.

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.

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.

Ejemplos

Sentencias	Resultado	
	a	b
a=5; b=++a;	6	6
a=5; b=a++;	6	5

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

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

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

Operadores de relación

== (igual)

<, >, <=, >=

!= (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 válida y asigna el valor 38 a la variable a! La expresión **a=38**, como se verá enseguida, es legal como expresión lógica y su valor de verdad es TRUE.

Operadores lógicos

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

Ejemplo

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

es una expresión lógica (*a igual a b o no c menor que 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".**

Ejemplos

- **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.

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.

Operadores de bits

~ (negación)

&, | (and, or)

^ (or exclusivo)

>>, << (desplazamiento a derecha y a izquierda)

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

Ejemplos

```
unsigned char a;

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

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

Ejemplos

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

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

Operadores especiales

= (asignación)

= (inicialización)

?: (ternario)

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

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. Si tenemos:

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

entonces, si **expresion_1** es verdadera, el ternario se evaluará a **expresion_2**, y ese valor será asignado a la variable a. Si **expresion_1** es falsa, a quedará con el valor **expresion_3**.

Ejemplo

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

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

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 **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. El remedio es secuencializar la ejecución dividiendo las expresiones en sentencias:

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

Resumen

La tabla siguiente 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.

() (llamada a función) **[]** (indexación de arreglo) **->** **.** (acceso a miembros de estructuras)
! ~ (negación y complemento a uno) ++ **--** (incremento y decremento) ***** (indirección) **&** (dirección de) **()** (cast, conversión forzada de tipo) **sizeof** (tamaño de variables o tipos) + unario, **-** unario
***** **/** **%** (aritméticos)
+ **-** (suma y resta)
<< >> (desplazamiento de bits)
< <= >= > (de relación)
== != (igual y distinto de)
& (AND de bits)
^ (XOR de bits)
| (OR de bits)
&& (AND lógico)
|| (OR lógico)
?: (operador ternario)
= (asignación) += **-=** *= /= **%=** **&=** ^= **|=** <<= >>= (abreviados)
, (operador coma)

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. Usando cualquiera de las anteriores escriba una macro IMPAR(x).

5. 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.

6. Escriba una macro ENTRE(x,y,z) que devuelva TRUE si y sólo si **y** está entre **x** y **z**. Usela para escribir una macro RECT(p,q,w,x,y,z) que devuelva TRUE si y sólo si el punto (p,q) pertenece al interior del rectángulo cuyos vértices opuestos son (w,x) e (y,z). Suponga que el rectángulo está expresado en forma canónica (con **w < y** y **x < z**).

7. Declare un **unsigned char a** y asígnele la constante 255. Trate de predecir el resultado de calcular **!a, ~a, a & x, a | x, a ^ x**, donde x es un entero menor que cero, cero y mayor que cero. Escriba un programa para verificar sus sospechas.

8. Escriba macros para expresar la conjunción, la disyunción, la negación y la implicación de proposiciones.

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

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

11. Utilice la macro del punto anterior para sumar $2^3 + 2^2$. Verifique el resultado.

12. Utilizando la macro de los puntos anteriores podemos simular un tipo de datos conjunto. Utilizaremos variables enteras largas (long int) para representar conjuntos. Los elementos de los conjuntos se denotarán por un número entre **0** y **sizeof(long) - 1**. La pertenencia del i-ésimo elemento a un conjunto se denotará por el i-ésimo bit activo en la variable que representa al conjunto. Escribir macros:

- AGREGAR(a,A) y ELIMINAR(a,A) que incorporen y quiten el elemento a al conjunto A.
- PERTENECE(a,A) que diga si un elemento pertenece a A.

- UNION(A,B) e INTERSECCION(A,B) que devuelvan respectivamente la unión e intersección de conjuntos.
- DIFERENCIA(A,B) que devuelva A-B.
- COMPLEMENTO(A) que devuelva el complemento de un conjunto.
- VACIO(A) que diga si un conjunto es vacío.
- DIFSIM(A,B) que exprese la diferencia simétrica de dos conjuntos.

13. Convertir una cantidad de tiempo, dada en segundos, a horas, minutos y segundos.

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

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

15. Reescribir la expresión utilizando el operador ternario:

(c == d) && (a < 5) || (b > 4)

16. Los años bisiestos son **los divisibles por cuatro, salvo aquellos que siendo divisibles por 100, no lo son por 400**. Escribir un conjunto de macros que devuelvan TRUE si un año es bisiesto y FALSE en caso contrario.

17. Reescribir utilizando abreviaturas:

```
a = a + 1;
b = b * 2;
b = b - 1;
c = c - 2;
d = d % 2;
e = e & 0x0F;
a = a + 1; b = b + a;
a = a - 1; c = c * a;
```

7. Estructuras de control

Las estructuras de control del C no presentan, en conjunto, grandes diferencias con las del resto de los lenguajes estructurados del tipo de Pascal. En general, se corresponden casi uno a uno, con las diferencias sintácticas de cada caso.

En los esquemas siguientes, donde figura una **sentencia** puede reemplazarse por varias sentencias encerradas entre llaves (un bloque).

Estructura alternativa

Formas típicas de la estructura alternativa son:

```
if(expresión)
    sentencia;

if(expresión)
    sentencia;
else
    sentencia;
```

Ejemplos

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

if(c >= 2 || func(b) < 0)
    if(d)
        c++;
    else
        c += 2;
```

En el segundo ejemplo aparecen estructuras anidadas. La cláusula **else** se aparea con el **if** más interno, salvo que se utilicen llaves, así:

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

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

Estructuras repetitivas

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

Estructura while

```
while(expresión)
    sentencia;
```

Estructura do...while

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.

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

Ambas estructuras ejecutan su cuerpo de sentencias **mientras la expresión resulte verdadera**.

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(inicialización; condición_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**:

```
inicialización;
while(condición_mientras) {
    sentencia;
    incremento;
}
```

Aunque el uso más común de las sentencias de incremento es 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.

Ejemplos

Este 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**.

La propiedad de que toda asignación tiene un valor como expresión (el valor asignado) permite escribir estructuras de control tales como

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

La anterior es una forma muy sintética de la clásica estructura de Pascal:

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

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,

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

equivale a los lazos anteriores.

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 menos claro:

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

Estructura de selección

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(expresión_entera) {
    case expresión_constante1:
        sentencias;
        break;
    case expresión_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

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;
}
```

Tanto si **m** vale 4, 6, 9, como 11, asignamos 30 a **d**. Al no haber un **break** intermedio, el control cae hasta la asignación d=30.

La estructura **switch** tiene varias limitaciones con respecto a su análogo el **case** de Pascal. 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).

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(expresión) {
    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).

La 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.

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).

Ejercicios

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

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

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

for( ; ; )
    a++;
```

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

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

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

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

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


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

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

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

for(x = 32; x<55; x += 3);

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. Implementar las sentencias que determinen si un año es bisiesto (años divisibles por 4 salvo aquellos que siendo divisibles por 100 no lo son por 400).

8. Usando el punto anterior imprimir una cartilla de años bisiestos entre 1100 y 2004 inclusive.

9. Implementar sentencias para imprimir el calendario para un mes dado. Suponga que se tiene una función **diasem**()que devuelve un entero indicando el día de la semana para una terna (día, mes, año) dada. El valor devuelto indicará 0=domingo, 1=lunes, etc.

10. Usando los puntos anteriores imprimir el calendario para un año dado completo.

11. Problema de Sissa: el tablero de ajedrez tiene 8 x 8 casilleros. El rey pone un grano de trigo en el primer casillero, luego dos en el segundo, cuatro en el tercero, etc. Calcular la cantidad total de granos de trigo sobre el tablero.

12. Imprimir la tabla de los diez primeros números primos (sólo divisibles por sí mismos y por la unidad).

13. Escribir las sentencias para calcular el factorial de un entero.

14. Retomar el ejercicio de simulación de conjuntos mediante operaciones de bits del capítulo 5, e implementar una función que imprima qué elementos pertenecen a un conjunto dado.

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.

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**:

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

```
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**:

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

```
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

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. En este ejemplo, 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;
        ...
}
```

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 $\mathbf{e}^{5}$:

```
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.

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.

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);
}
```

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.

7. Escribir una función como la anterior pero donde el primer parámetro sea **long** y el otro, como antes, de tipo **char**. Escribir el programa de prueba como en el punto anterior.

8. Escribir una función para calcular la raíz cuadrada aproximada con la fórmula de Newton:

nueva = (anterior + primera / anterior) / 2

Detener el cómputo cuando la diferencia entre nueva y anterior es menor que un umbral pequeño dado.

9. Una cierta función matemática está definida como sigue:

f(x,y) = x - y si x<0 o y <0

f(x,y) = f(x-1, y) + f(x, y-1) en otro caso

Escribir una función C de dos argumentos que la evalúe.

10. Utilizar la función anterior en un programa que imprima una tabla de los valores de la función para todo x e y entre 0 y 10.

9. Arreglos y variables estructuradas

Casi todos los lenguajes proveen una manera de combinar variables simples en alguna forma de agregación. 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 **arreglo, vector o array**. Sus elementos se acceden indexando el bloque con expresiones enteras entre corchetes. En C, los arreglos **se indexan a partir de 0.**

Ejemplos

```
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];
```

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.

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

double saldo[10] = { 150.40, 170.20 };
    /* inic. incompleta */

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

Errores frecuentes

Los siguientes son errores muy comunes y lamentablemente el lenguaje C no provee ayuda para prevenirlos:

1. Indexación fuera de límites

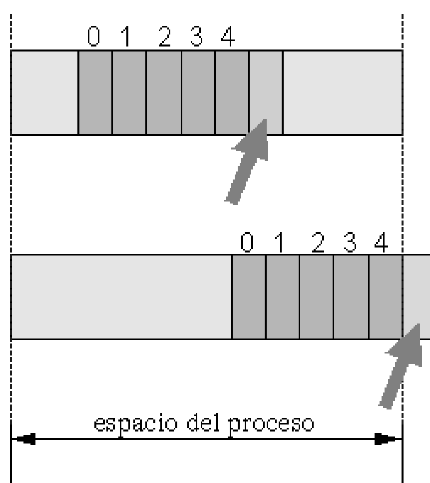
La dimensión dada en la declaración del arreglo dice **cuántos elementos tiene**. Esto **no** quiere decir que exista un elemento del arreglo **con ese índice** (porque se numeran a partir de 0):

```
int tabla[5];

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

La última instrucción equivale a acceder a un elemento fuera de los límites del arreglo. Este programa, erróneo, compilará sin embargo correctamente. 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 el acceso, a pesar de superar los límites del arreglo, no cae fuera del espacio 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.



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.

2. 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.

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

Esta instrucción no es compilable. Debe reemplazarse por:

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

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;
```

Estructuras y uniones

Las variables estructuradas 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.

Ejemplos

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

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

Las declaraciones de más arriba no definen variables, con espacio de almacenamiento, sino que simplemente enuncian un nuevo tipo que puede usarse en nuevas declaraciones de variables. 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 Pérez", 30};
c2.p = p1;
```

Una declaración con inicialización completa:

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

También 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 };
```

Una función que recibe y devuelve estructuras:

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

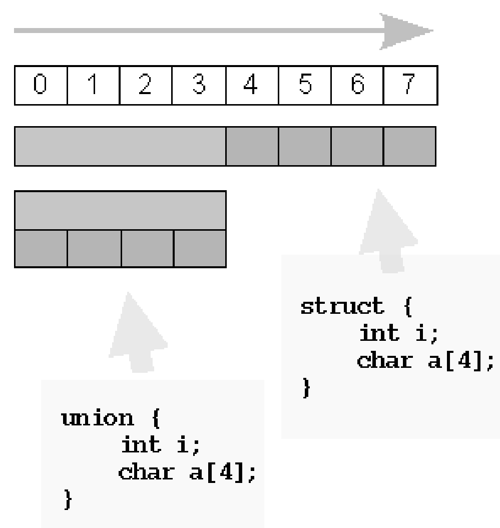
Uniones

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. 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];
```



El diagrama ilustra los diferentes desplazamientos u offsets a los que se ubican los miembros en una **union** y en una **struct**, suponiendo una arquitectura subyacente donde los ints miden cuatro bytes.

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;
};
```

En este ejemplo "inventamos" 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 que permite encenderlo o apagarlo, consultar su disponibilidad (online u offline), 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.

Nuestro dispositivo imaginario podría encenderse, esperar a que se ponga online, tomar el promedio de diez lecturas de estado y apagarse, con las instrucciones siguientes. Como se ve, no hay diferencia de acceso con las estructuras.

```
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;
```

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. Consulte el manual del sistema para conocer la función rand() que genera un entero al azar. Generar enteros al azar entre 0 y 99 hasta que se hayan generado todos al menos una vez.

7. Calcular la suma y el producto de dos matrices.

8. Escribir una función que ordene ascendemente por cualquier algoritmo un arreglo de ints.

9. Declarar una estructura **punto** conteniendo coordenadas x e y de tipo double. Dar ejemplos de inicialización y de asignación.

10. Declarar una estructura **segmento** conteniendo dos estructuras **punto**. Dar ejemplos de inicialización y de asignación. Dar una función que calcule su longitud.

11. Declarar una estructura fecha conteniendo tres enteros d, m y a. Escribir funciones para asignar valores a una estructura fecha, para calcular diferencia de días entre fechas, para obtener la fecha posterior en **n** días a una dada, para saber el día de la semana de una fecha dada del año 2001 sabiendo que el 1/1/2001 fue lunes.

12. ¿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 };
```

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.

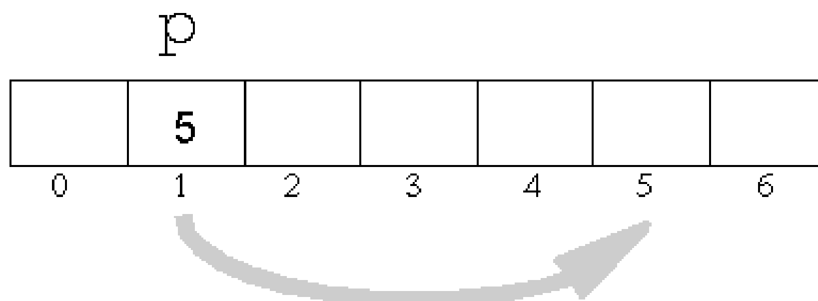
Para manipular direcciones se utilizan en C variables especiales llamadas **apuntadores o punteros**, que son aquellas capaces de contener direcciones. En la declaración de un apuntador se especifica el tipo de los objetos de datos cuya dirección contendrá.

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. 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.

Las cuestiones fundamentales a entender para no caer en estos errores son los conceptos de **direcciones** y **punteros**, así como dominar la **sintaxis de las declaraciones de punteros** para asegurarse de que escribimos lo que se pretende lograr.

Operadores especiales

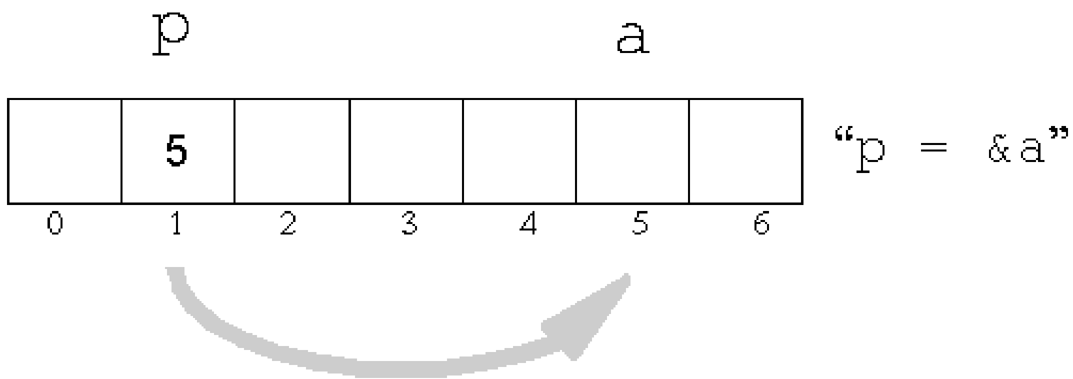
Para manipular punteros se hacen necesarios dos operadores especiales:

&	**Operador de dirección**
*	**Operador de indirección**

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

```
p = &a;
```

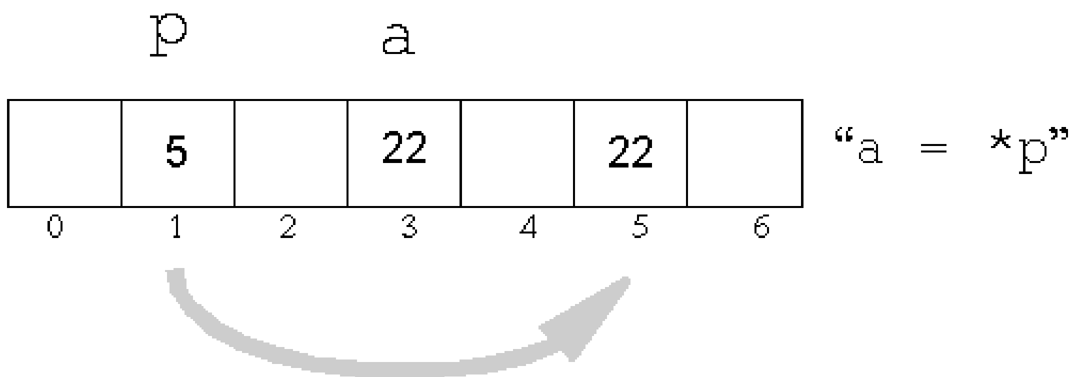
puede leerse: **"asignar a p la dirección de a".**



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

```
a = *p;
```

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



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

En general, si p = &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. Las instrucciones

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

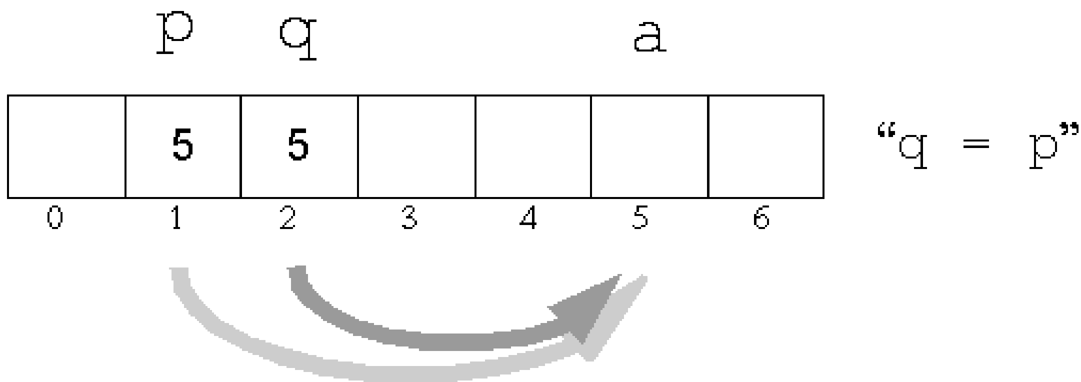
equivalen a

```
a = 1;
```

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.

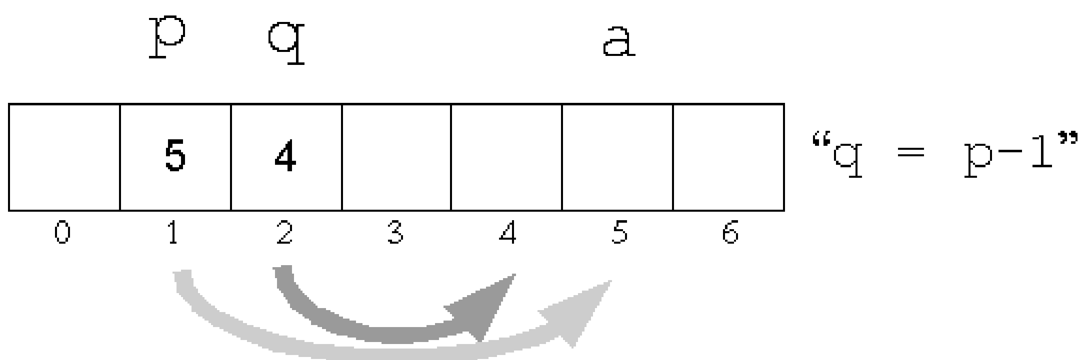
Asignación entre punteros



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.

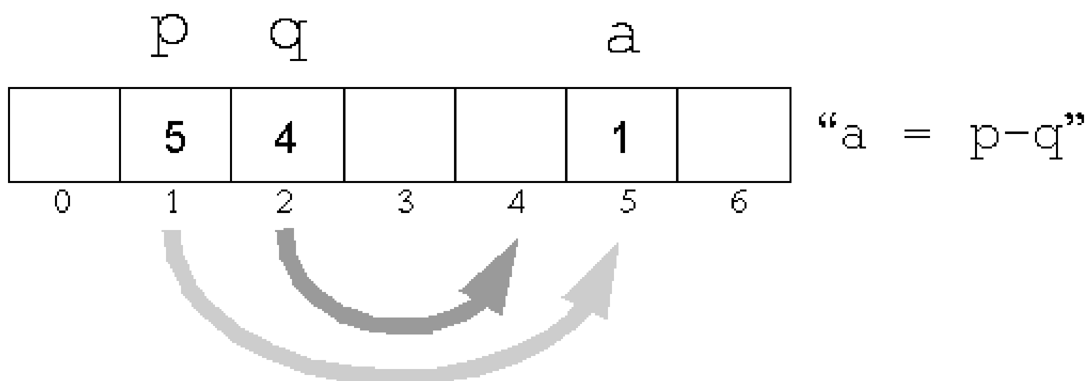


Es decir que **sumar (o restar) una unidad a un puntero, lo incrementa (decrementa) en tantos bytes como mida el objeto al cual apunta.**

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 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**.

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;
```

Ejemplos

- Algunas manipulaciones con arreglos y punteros:

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

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 está definida */
```

- Los dos segmentos siguientes realizan exactamente la misma tarea.

```
int i:
long tabla[10];
for(i = 0; i < 10; i++)
    suma += tabla[i];

long *p;
long tabla[10];
for(p = tabla; p < tabla+10; p++)
    suma += *p;
```

Punteros y cadenas de texto

Posiblemente el caso más extendido del uso de punteros sea cuando se necesita trabajar con cadenas de texto, o *strings*. En C, éstas son análogos bastante cercanos de los arreglos de caracteres, aunque con diferencias importantes. En la inicialización de punteros, las constantes string son un caso especialmente frecuente.

Las **constantes string** son un caso particular de cadenas de texto: son todas aquellas secuencias de caracteres (eventualmente la secuencia vacía) entre comillas. Cuando el compilador C encuentra una constante string, copia los caracteres entre comillas encontrados a un espacio de almacenamiento y termina la secuencia de caracteres con un 0 binario (un byte con contenido binario 0). Esta

representación interna se propagará al programa una vez instalado en memoria al momento de ejecución. El carácter cero final tiene la misión de funcionar como señal de terminación para aquellas funciones de biblioteca standard que manejan strings (copia de cadenas, búsqueda de caracteres, comparación de cadenas, etc.).

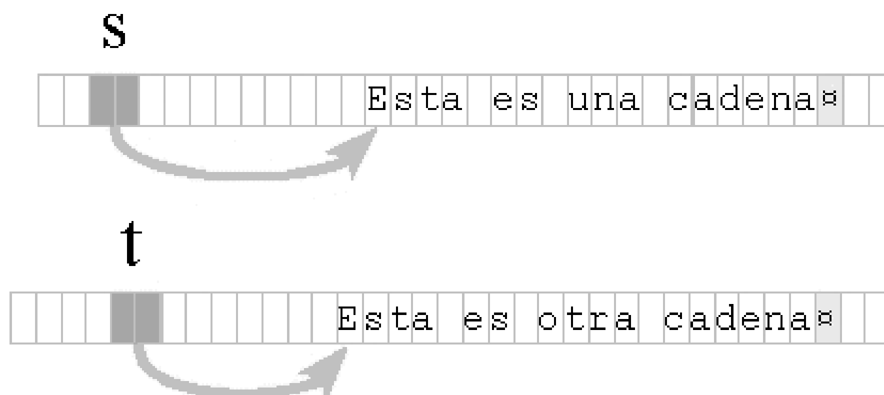
Gracias a esta representación, las cadenas no tienen una longitud máxima; pero es necesario cumplir con el protocolo de terminarlas con su cero final para poder utilizar las funciones que manipulan strings. Debido a esta representación interna algunas veces se las ve mencionadas con el nombre de cadenas ASCIIZ (caracteres ASCII seguidos de cero).

La segunda cosa que hace el compilador C con una constante string es reemplazar la referencia a la constante, en el texto bajo compilación, por la dirección del almacenamiento asignado. De esta manera, en el ejemplo, 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.

Ejemplo

```
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**.

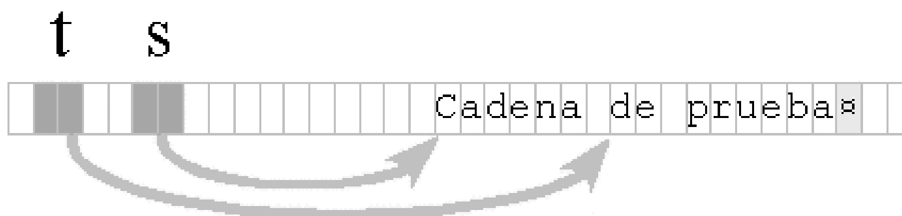
Ejemplo

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

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
```

Pasaje por referencia

En C, donde todo pasaje de parámetros 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 implica entregar **la dirección** del objeto.

Ejemplos

- **Modificación de un objeto externo a una función.**

La función **f2()** debe poner a cero una variable entera, 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           */
}
```

- **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);
```

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 sea exactamente el argumento **real** (arreglo o puntero): la función únicamente recibe una dirección y no sabe cuál es la naturaleza real del objeto exterior a ella.

Ejemplo

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)
{
    for( ; *p != 0; p++)
        if(*p == c)
            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.

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 longs llamados 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.

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

5. Compilar y ejecutar:

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

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

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

6. ¿Qué imprimirán estas sentencias?

```
char *s = "ABCDEFG";
printf("%s\n", s);
printf("%s\n", s + 0);
printf("%s\n", s + 1);
printf("%s\n", s + 6);
printf("%s\n", s + 7);
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:

- Calcular la longitud de una cadena.
- 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.
- Comparar dos cadenas, devolviendo 0 si son iguales, un valor negativo si la primera es lexicográficamente menor, un valor positivo si es mayor.
- Idem pero comparando hasta el n-simo carácter y con las mismas convenciones de salida.
- Idem pero ignorando las diferencias entre mayúsculas y minúsculas.
- 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:

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

10. Escribir funciones para:

- Rellenar una cadena con un carácter dado, hasta que se encuentre el 0 final, o hasta alcanzar n iteraciones.
- Pasar una cadena a mayúsculas o minúsculas.
- Copiar una cadena en una dirección de memoria dada, suponiendo que el destino tiene espacio suficiente.

- Idem los primeros n caracteres de la cadena.

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

10b. Temas avanzados de apuntadores y direcciones

Observaciones

Continuando la unidad, insistimos sobre las causas de errores más frecuentes.

1. Punteros sin inicializar

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

Ejemplo

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

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. El problema es que el contenido de p es impredecible. Si p es una variable local, su clase de almacenamiento es **auto** y por lo tanto contiene *basura*, salvo inicialización explícita. Si p es externa, es inicializada a 0, 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 a veces sumamente 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.

La solución es asegurarse, siempre, que los punteros apuntan a lugares válidos del programa, asignándoles direcciones de objetos conocidos. Por ejemplo:

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

2. 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 exponen a continuación. No hay que dejarse engañar por ellos.

a) 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().

b) 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);
```

c) 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.

3. 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. 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**.

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". Aplicar el operador **&** a algo aumenta su nivel de indirección. Aplicar el operador ***** 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.

Esto es análogo a 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 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.

Ejemplos

```
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). La asignación **u = s** intenta asignar una dirección (la contenida en s) a un char. No se respeta el mismo nivel de indirección en ambos miembros de la asignación, 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.

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.

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

```
char *mes[] = { "Ene","Feb","Mar","Abr","May","Jun",
                "Jul","Ago","Sep","Oct","Nov","Dic" };
printf("Mes: %s\n",mes[6]);
```

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

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

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

Estructuras de datos recursivas y punteros

Las estructuras de datos recursivas se expresan efectivamente con 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;
}
```

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 buen momento para introducirla. El lenguaje C admite la generación de nuevos nombres para tipos estructurados mediante la primitiva **typedef**.

Ejemplo

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.

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

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

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. Ahora podemos hacer la asignación. Luego del uso se debe liberar la zona apuntada por p.

Para ser completamente correcto, el programa 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.

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 calcularse en tiempo de ejecución.

Una variante de malloc() es **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.

Ejemplo

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

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.

Ejemplos

- 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 cuya dirección tiene asignada:

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

Aplicación

La Biblioteca Standard contiene una función 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), 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 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 0
*p1 == *p2	0
*p1 > *p2	un número mayor que 0

La función de ordenamiento recibe un puntero a la función de comparación y la invoca repetidamente.

Ejemplo

```
#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);
}
```

Los argumentos formales declarados como **void ***expresan que puede tratarse de direcciones de objetos de cualquier tipo.

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);
```

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

```
int **p;    /* un puntero doble */
int *q;
int a;
q = &a;
p = &q;
**p = 8;    /* carga 8 en a */
```

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 para compatibilizarla con las funciones de tratamiento de strings de la biblioteca standard.

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

Ejemplos

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

main()
{
    char arreglo[];
    char *s;
    s = arreglo;
    gets(s); /* Ahora sí */
    gets(arreglo); /* Idéntico a lo anterior */
}

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

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

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

En el segundo ejemplo proveemos espacio legítimo donde apunte **s,** reservándolo estáticamente mediante la declaración del arreglo.

En el tercer ejemplo usamos asignación dinámica y liberación de memoria. En un sentido estricto, no es necesaria la liberación en este caso particular 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().

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. Escribir una función que calcule el valor entero asociado a una cadena de dígitos decimales (sugerencia: utilizar ejercicio de capítulo 4).

3. Idem hexadecimales (misma sugerencia).

4. 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.

5. Construir una función que reciba un arreglo de punteros a string y presente un menú en pantalla, devolviendo la opción seleccionada.

6. Utilizando la anterior, construir una función de menús que además de permitir la selección de una opción ejecute la función asociada a la opción seleccionada.

7. Construir una función que imprima una cadena en forma inversa. Muestre una versión iterativa y una recursiva.

8. 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.

9. Construir un programa que lea una sucesión de palabras y las almacene en un arreglo de punteros a carácter.

10. Ordenar lexicográficamente el arreglo de punteros del ejercicio 8.

11. Entrada/salida

El concepto de entrada/salida en C replica el de su ambiente nativo, el sistema operativo UNIX, donde todos los archivos son vistos como una sucesión de bytes, prescindiendo completamente de su contenido, organización o forma de acceso. Además, en UNIX los dispositivos de entrada o salida llevan asociados archivos lógicos, que son puntos de entrada implementados en software a dichos dispositivos. Toda la comunicación entre un programa y el mundo externo, ya sean archivos físicos o lógicos, se hace mediante las mismas funciones.

Lo anterior da como resultado que la abstracción de programación para los archivos en C es simplemente un **flujo** de bytes o **stream**, que se maneja con operaciones primitivas independientemente de cuál sea su origen y su destino.

Si bien el C no contiene palabras reservadas de entrada/salida (**E/S**), la biblioteca standard sí provee un rico conjunto de funciones de E/S, tan amplio que suele provocar confusión en quienes se aproximan por primera vez al lenguaje. Ofreceremos primeramente un resumen de las funciones de E/S standard y nos concentraremos luego en la E/S sobre archivos. Aunque la información dada aquí es suficiente para intentar la creación de programas simples, la E/S es un tema notablemente complejo, y es aconsejable tener a mano el manual de las funciones C de su sistema.

Para orientación del lector agregamos un mapa de la lección presente.

Funciones de entrada/salida			
E/S standard	**Sobre archivos**		
de caracteres	**ANSI C**	**De acceso directo**	**POSIX**
de líneas	de caracteres		
con formato	de líneas		
sobre strings	con formato		

Funciones de E/S standard

Los programas C reciben tres canales de comunicación con el ambiente abiertos antes de comenzar su ejecución. El uso típico de estos canales de comunicación es la lectura del teclado y la impresión sobre pantalla, aunque si el sistema operativo lo soporta, también posibilitan la poderosa técnica de la redirección. La biblioteca standard provee funciones mínimas para estos usos, quedando fuera de consideración algunas características indispensables en programas de producción, como seguridad, validación, o la posibilidad de organizar la salida en pantalla. Por ejemplo, no hay una forma canónica de borrar la pantalla en C, ya que ésta es una función que depende fuertemente de la plataforma donde se ejecute el programa. Las características faltantes en la E/S standard se compensan recurriendo a bibliotecas de terceras partes.

E/S standard de caracteres

Las funciones de E/S standard de caracteres son**getchar()** y **putchar().** El ejemplo siguiente es un programa que copia la entrada en la salida carácter a carácter. Puede usarse, con redirección, para crear o copiar archivos, como un clon del comando **cat** de UNIX.

```
#include <stdio.h>
main()
{
    int a;
    while((a = getchar()) != EOF)
        putchar(a);
}
```

E/S standard de líneas

Las funciones **gets()** y **puts()** leen de teclado e imprimen, respectivamente, líneas de caracteres terminadas por la señal de fin de línea **\n**. La función gets() debe recibir como argumento la dirección de un buffer o zona de memoria donde depositar los caracteres provenientes de entrada standard. Estos son normalmente tipeados por el usuario, pero pueden provenir de archivos o ser resultado de la ejecución -eventualmente concurrente- de otros programas, gracias a la redirección. Es un error muy frecuente ofrecer a gets() un puntero no inicializado. La función gets() se ha descripto en la unidad sobre apuntadores y direcciones.

Ejemplo

El mismo programa, pero orientado a copiar un stream de texto línea por línea. La constante BUFSIZ está definida en stdio.h y es el tamaño del buffer de estas funciones. Se puede sugerir esta elección para el buffer del programa, salvo que haya motivos para proporcionar otro tamaño.

```
#include <stdio.h>
main()
{
    char area[BUFSIZ];
    while(gets(area) != NULL)
        puts(area);
}
```

La función gets() elimina el \n final con que termina la línea antes de almacenarla en su buffer. La función puts() lo repone.

Hay que subrayar que, como el tamaño del buffer no es argumento para la función gets(), ésta **no conoce los límites** del área de memoria de que dispone para dejar los resultados de una operación de entrada, y por lo tanto no puede hacer verificación en tiempo de ejecución. Podría ocurrir que una línea de entrada superara el tamaño del buffer: entonces esta entrada corromperá algún otro contenido del espacio del programa. Esta condición se conoce como *buffer overflow* y el comportamiento en este caso queda indefinido, dando lugar, inclusive, a problemas de seguridad. Por este motivo gets() no es utilizada en programas de producción.

E/S standard con formato

Las funciones **printf()** y **scanf()** permiten imprimir e ingresar, respectivamente, conjuntos de datos **en formato legible**, descriptos por cadenas de formato. Las cadenas se componen de especificaciones de conversión y son simétricamente las mismas para ambas funciones. La función printf() y las cadenas

de formato han sido descriptas en la unidad correspondiente a tipos de datos.

Inversamente a printf(), la función scanf() buscará en la entrada standard patrones de acuerdo a las especificaciones de conversión. Generará representaciones internas para los datos leídos y los almacenará en variables. Para esto debe recibir las **direcciones** de dichas variables donde almacenar los elementos detectados en la entrada standard. Es un error frecuente ofrecerle, como argumentos, las variables, y no las **referencias** a las mismas.

Ejemplo

```
main()
{
    int a, long b;
    if(scanf("%d %ld", &a, &b) != 2)
        exit(1);
    printf("a=%d, b=%ld\n", a, b);
}
```

El valor devuelto por scanf() es la cantidad de datos leídos y convertidos exitosamente, y debería siempre comprobarse que es igual a lo esperado.

El uso de scanf() es generalmente problemático. La función scanf() consumirá toda la entrada posible, pero se detendrá al encontrar un error (una entrada que no corresponda a lo descripto por la especificación de conversión) y dejará el resto de la entrada sin procesar, en el buffer de entrada. Si luego otra función de entrada intenta leer, se encontrará con esta entrada no consumida, lo cual puede dar origen a problemas de ejecución difíciles de diagnosticar. El error parecerá producirse en una instrucción posterior a la invocación de scanf() culpable. Por esta razón suele ser difícil mezclar el uso de scanf() con otras funciones de entrada/salida. Además, no hay manera directa de validar que la entrada quede en el rango del tipo de datos destino.

El uso más recomendable de scanf() es cuando se la utiliza para leer, mediante redirección, un flujo generado automáticamente por otro programa (y que, por lo tanto, tiene una gramática rigurosa y conocida).

E/S standard sobre strings

La misma lógica de las funciones de E/S con formato sirve para que otras funciones lean variables con formato de un string, o impriman variables formateadas sobre una cadena. El efecto de **sprintf()** sobre su cadena argumento es el mismo que tendría printf() sobre salida standard. Por su parte **sscanf()** lee de un string **en memoria**, conteniendo datos en formato legible, y los recoge en representación binaria en variables, lo mismo que si scanf() los hubiera leído de entrada standard.

Ejemplo

```
main()
{
    char area[1024];
    int a; long b;
    sprintf(area, "%d %ld\n", -6534, 1273632);
    sscanf(area, "%d %ld", &a, &b);
    printf("%d %ld\n", a, b);
}
```

El resultado debería ser:

```
-6534 1273632
```

E/S sobre archivos

Diferentes sistemas operativos tienen diferentes **sistemas de archivos** y diferentes **modelos de archivos**. Los sistemas de archivos son los conjuntos de funciones particulares que cada sistema ofrece para acceder a los archivos y a la estructura de directorios que soporta. Los modelos de archivos son aquellas convenciones de formato u organización que son particulares de un sistema operativo o plataforma.

Por ejemplo, tanto DOS como UNIX soportan la noción de archivo de texto, pero con algunas diferencias a nivel de modelo de archivos. En ambos, un archivo de texto es una secuencia de líneas de texto, donde cada línea es una secuencia de caracteres terminado en fin de línea; pero en DOS, la convención de fin de línea es un par de caracteres (CR,LF) (equivalentes a ASCII 13 y 10) mientras que para UNIX, un fin de línea equivale solamente a LF (ASCII 10). Además DOS soporta la noción de carácter de fin de archivo (EOF o ASCII 26) mientras que no hay tal concepto en UNIX.

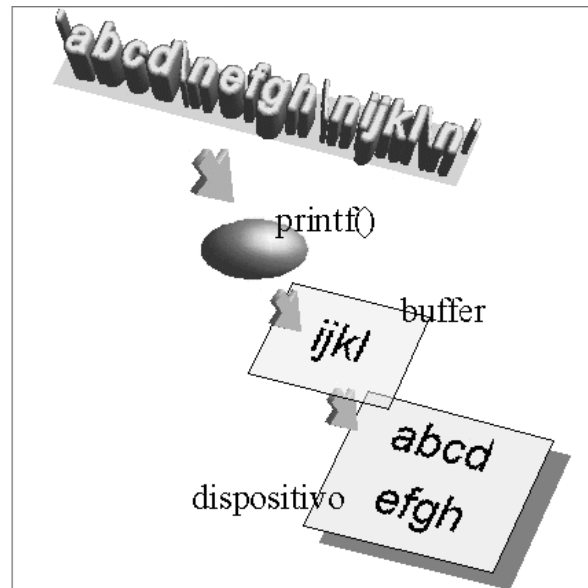
Por otro lado, diferentes sistemas de archivo proveen diferentes vistas sobre diferentes implementaciones. Un sistema operativo puede soportar o no la noción de directorio, o la de links múltiples; o puede fijar determinadas condiciones sobre los nombres de archivos, todo esto en función de la organización íntima de sus estructuras de datos.

Siendo un objetivo de diseño del lenguaje C el favorecer la producción de programas portables, el C contempla la forma de resolver estos problemas de manera fácil para los programadores. Las funciones de entrada/salida sobre archivos de la biblioteca standard están divididas en dos grandes regiones: el conjunto de funciones del C standard, también llamadas funciones de entrada/salida **bufferizada**, definidas por ANSI, y las funciones POSIX, también llamadas funciones de entrada/salida **de bajo nivel.**

Las funciones ANSI C tienen el objetivo de ocultar a los programas las particularidades de la plataforma, haciéndolos plenamente portables a pesar de las diferencias conceptuales y de implementación de entrada/salida entre los diferentes sistemas operativos. Estas funciones resuelven, por ejemplo, el clásico problema de las diferentes convenciones sobre los delimitadores de un archivo de texto. Es decir, están orientadas a resolver los problemas de incompatibilidad inherentes al **modelo de archivos**. En cambio, el esfuerzo de estandarización de POSIX apunta a establecer (aunque no solamente en lo referente a los archivos) una interfaz uniforme entre compilador y sistema operativo, proveyendo primitivas de acceso a los archivos con un comportamiento claramente determinado, independientemente de cuál sea el sistema operativo subyacente. Así, las funciones POSIX resuelven problemas de consistencia entre diferentes **sistemas de archivos**.

Las funciones del ANSI C son las más comúnmente utilizadas por el programador, pero se apoyan en funcionalidad suministrada por las funciones POSIX (de nivel más bajo), que también están disponibles y son las recomendadas cuando las restricciones del problema exceden a las funciones ANSI. La característica fundamental de las funciones ANSI es la entrada/salida *bufferizada.*Por ejemplo, una operación de escritura solicitada por una instrucción del programa no se efectiviza inmediatamente sino que se realiza sobre un buffer intermedio, administrado por las funciones de biblioteca standard y con su política propia de *flushing* o descarga al dispositivo. En cambio, las funciones POSIX hacen E/S directa a los dispositivos (o al menos, al sistema de E/S del sistema operativo) y por esto son las preferidas para la programación de drivers, servidores, etc., donde la

performance y otros detalles finos deban ser controlados más directamente por el programa.



Las funciones de entrada/salida bufferizada reciben argumentos a imprimir y los van depositando en un buffer o zona de memoria intermedia.

Cuando el buffer se llena, o cuando aparece un carácter de fin de línea, el buffer se descarga al dispositivo, escribiéndose los contenidos del buffer en pantalla, disco, etc.

Funciones ANSI C de E/S sobre archivos

Las funciones ANSI realizan todas las operaciones sobre archivos por medio de una estructura o bloque de control cuyo tipo en C se llama **FILE**. Esta estructura está definida en el header **stdio.h** y contiene, entre otras cosas, punteros a buffers para escritura y lectura. La primera operación necesaria es la **apertura** del archivo, que construye una estructura FILE, la inicializa con valores adecuados y devuelve un apuntador a la misma. El apuntador servirá para referenciarla durante todo el trabajo con el archivo y hasta que deba ser cerrado.

En la apertura del archivo corresponde indicar el **modo de acceso** (la clase de operaciones que se van a a hacer sobre él). Como algunos sistemas operativos (notoriamente, el DOS) distinguen entre archivos de texto y binarios, el ANSI C provee dos formas de apertura, para indicar cómo se va a tratar el archivo. Cuando un archivo se abre en **modo de texto**, durante las operaciones de lectura y escritura se aplicarán las conversiones de fines de línea y de fin de archivo propias de la plataforma. Para los archivos abiertos en **modo binario**, no se aplicarán conversiones.

En sistemas conformes a POSIX, como UNIX, no hay realmente diferencia entre los dos modos de apertura. Si se desea especificar una apertura de archivo en modo binario (para asegurar la portabilidad) se añade una **b** a la especificación de modo (por ejemplo, como en "wb+").

En el cuadro siguiente se resumen las especificaciones de modos de acceso en apertura de archivos.

Modos de acceso

"r"	Abre un archivo que ya existe para lectura. La lectura se realiza al inicio del archivo.
"w"	Se crea un nuevo archivo para escribir. Si el archivo existe, se inicializa y se sobreescribe.
"a"	Abre un archivo que ya existe para agregar información al final. Sólo se puede escribir a partir del final.
"r+"	Abre un archivo que ya existe para actualizarlo (tanto para lectura como para escritura).
"w+"	Crea un nuevo archivo para actualizarlo (lectura y escritura); si existe, lo sobreescribe.
"a+"	Abre un archivo para añadir información al final. Si no existe, lo crea.

Como en la entrada/salida standard, para manejar archivos tenemos funciones para E/S de caracteres, de líneas y con formato.

Funciones ANSI C de caracteres sobre archivos

Las funciones son **fgetc()** y **fputc()**. Ejemplo que copia un archivo:

```
#include <stdio.h>
main()
{
    FILE *fp1, *fp2;
    int a;
    if(((fp1 = fopen("ejemplo.txt","r")) == NULL) ||
                    ((fp2 = fopen("copia.txt","w")) == NULL))
        exit(1);
    while((a = fgetc(fp1)) != EOF)
        fputc(a, fp2);
    fclose(fp1);
    fclose(fp2);
}
```

Hay otras funciones dentro de esta categoría, como **ungetc()** que devuelve un carácter al flujo de donde se leyó.

Funciones ANSI C de líneas sobre archivos

Mismo ejemplo, en base a líneas, con las funciones **fgets()** y **fputs()**. La declaración de variables FILE * y las sentencias de apertura y cierre de archivos son idénticas al caso anterior. Enunciamos solamente el lazo principal.

```
    char area[BUFSIZ];
    while(fgets(area, BUFSIZ, fp1) != NULL)
        fputs(area, fp2);
```

Funciones ANSI C con formato sobre archivos

Existen funciones **fprintf()** y **fscanf()**, casi idénticas a printf() y scanf(), pero donde se especifica el stream de entrada o de salida. Las funciones printf() y scanf() pueden verse como el caso particular de las primeras donde el stream es **stdout** o **stdin**, respectivamente.

Ejemplo

```
#include <stdio.h>
main()
{
    FILE *fp1, *fp2;
    int a; long b;
    if(((fp1 = fopen("ejemplo.txt","r")) == NULL) ||
                ((fp2 = fopen("copia.txt","w")) == NULL))
        exit(1);
    if(fscanf(fp1, "%d %ld", &a, &b) != 2)
        exit(1);
    fprintf(fp2, "%d %ld\n", a, b);
    fclose(fp1);
    fclose(fp2);
}
```

El programa del ejemplo lee dos variables de un archivo y las escribe, en el mismo formato, en un segundo archivo.

Funciones ANSI C de acceso directo

Un conjunto muy útil de funciones ANSI permite el **acceso directo, aleatorio**, o **random**, sobre archivos, saltando por encima del modelo de E/S secuencial que domina al resto de las funciones. Las funciones básicas de acceso directo son **fread()** y **fwrite()**, que leen bloques de un tamaño dado y en una cantidad dada. Son ideales para lectura y escritura directa de estructuras de datos, ya se trate de elementos individuales u organizados en arreglos. Al poder posicionarse el puntero de lectura o escritura en zonas arbitrarias de los archivos, se logra la capacidad de E/S por registros con acceso aleatorio.

Ejemplo

```
struct registro {
    int dato1;
    long dato2;
} datos;
...
fseek(fp, 10L * sizeof(struct registro), SEEK_SET);
fread(&datos, sizeof(struct registro), 1, fp);
datos.dato1 = 1;
fseek(fp, 5L * sizeof(struct registro), SEEK_SET);
fwrite(&datos, sizeof(struct registro), 1, fp);
```

En el ejemplo suponemos que se han grabado en el archivo varios registros cuyo formato está representado por la estructura de la variable **datos**. La función **fseek()** posiciona el puntero de lectura en el *offset* **10 * sizeof(...)** (que debe ser un **long**), significando que necesitamos acceder al registro lógico 10 del archivo. A continuación se leen tantos bytes como mide un elemento de datos. Cambiando el tercer argumento de fread() podemos leer en un solo acceso un vector completo de estas estructuras en lugar de un elemento individualmente.

Luego de cambiar un valor del registro se lo vuelve a grabar, esta vez en un offset distinto (correspondiente al registro lógico 5).

La constante SEEK_SET indica que el posicionamiento solicitado debe entenderse como absoluto a partir del principio del archivo. Otras constantes son SEEK_CUR (posicionamiento a partir del offset actual) y SEEK_END (a partir del fin del archivo). El offset proporcionado puede ser negativo.

Sincronización de E/S

Una restricción importante de la E/S en ANSI C es que **no se pueden mezclar instrucciones de entrada y de salida sin que intervenga una operación intermedia de posicionamiento**. Es decir, una sucesión de invocaciones a **fwrite()** puede ser seguida de uno o más **fread()**, pero únicamente luego de un **fseek()** entre ambas. La operación de posicionamiento resincroniza ambos punteros y su ausencia puede hacer que se recupere *basura*.

Este posicionamiento puede ser nulo, como por ejemplo en **fseek(fp, 0L, SEEK_CUR)** que no varía en absoluto la posición de los punteros, pero realiza la sincronización buscada.

Resumen de funciones ANSI C de E/S

Podemos resumir lo visto hasta aquí con los prototipos de las funciones ANSI de E/S en el cuadro siguiente.

	E/S Standard	**E/S sobre archivos**
De caracteres	int getchar(); int putchar(int c);	int fgetc(FILE *stream); int fputc(int c, FILE *stream);
De líneas	char *gets(char *s); int puts(const char *s);	char *fgets(char *s, int n, FILE *stream); int fputs(const char *s, FILE *stream);
Con formato	int printf(const char *format, ...); int scanf(const char *format, ...);	int fprintf(FILE *stream, const char *format, ...); int fscanf(FILE *stream, const char *format, ...);
Sobre strings	int sprintf(char *s, const char *format, ...); int sscanf(char *s, const char *format, ...);	
De acceso directo		size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream); size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream);

Funciones POSIX de E/S sobre archivos

Las funciones POSIX son la interfaz directa del programa con las llamadas al sistema, o **system calls**. Las funciones POSIX que operan sobre archivos lo hacen a través de **descriptores de archivos.**Estos pertenecen a una tabla de archivos abiertos que tiene cada proceso o programa en ejecución y se corresponden con estructuras de datos del sistema operativo para manejar la escritura y la lectura en los archivos propiamente dichos.

La tabla de archivos abiertos de un proceso se inicia con tres archivos abiertos (correspondientes a los streams de entrada standard, salida standard y salida standard de errores) que reciben los descriptores números 0, 1 y 2 respectivamente. Los archivos que se abren subsiguientemente van ocupando el siguiente descriptor de archivo libre en esta tabla y por lo tanto reciben descriptores 3, 4, ..., etc. Cada nueva operación de apertura exitosa de un archivo devuelve un nuevo descriptor. Este número de descriptor se utiliza durante el resto de la actividad sobre el archivo.

Tanto las funciones ANSI C para archivos como las funciones POSIX de archivos manejan referencias obtenidas mediante la apertura y utilizadas durante toda la relación del programa con el archivo, pero las referencias son diferentes. La referencia al bloque de control utilizada por las funciones ANSI es de tipo **FILE ***, mientras que el descriptor de archivo POSIX es un **int**. Por este motivo **no se pueden mezclar** las llamadas a funciones de uno y otro grupo.

Sin embargo, sí es cierto que las estructuras de tipo FILE, referenciadas por un FILE *, como se dijo antes, se apoyan en funcionalidad aportada por funciones POSIX, y por lo tanto contienen un descriptor de archivo. Si se tiene un archivo abierto mediante una función POSIX, es posible, dado su descriptor, obtener directamente el stream bufferizado correspondiente para manipularlo con funciones ANSI. Esto se logra con la función **fdopen()**.

Los programas que utilicen funciones POSIX deben incluir los headers de biblioteca standard **unistd.h** y **fcntl.h**.

Ejemplo

```
#include <unistd.h>
#include <fcntl.h>
main()
{
    char area[1024];
    int fd1, fd2, bytes;
    if((fd1 = open("ejemplo.txt", O_RDONLY)) < 0)
        exit(1);
    if((fd2 = open("copia.txt",
                O_WRONLY|O_CREAT|O_TRUNC, 0660)) < 0)
        exit(1);
    while(bytes = read(fd1, area, sizeof(area)))
        write(fd2, area, bytes);
    close(fd1);
    close(fd2);
}
```

Este programa copia dos archivos usando funciones POSIX read() y write() aplicadas a los descriptores obtenidos con open().

Abre el primer archivo en modo de sólo lectura con el *flag*, u opción, O_RDONLY, en tanto que necesita escribir sobre el segundo, por lo cual utiliza el flag O_WRONLY. Además, para el segundo archivo, especifica otros flags que van agregados al primero y que son O_CREAT (si no existe, se lo crea) y O_TRUNC (si ya existe, se borran todos sus contenidos).

Los *flags* son nombres simbólicos para constantes de bits. Todas las combinaciones de flags posibles pueden expresarse como un OR de bits. Resumimos los flags más importantes existentes para este segundo argumento de open().

O_RDONLY	El archivo se abre para lectura solamente
O_RDWR	Se abre para escritura solamente
O_APPEND	El archivo puede ser leído o agregársele contenido
O_CREAT	Si el archivo no existe, se lo crea
O_EXCL	Si ya existe, se vuelve con indicación de error
O_WRONLY	Se abre para escritura solamente
O_TRUNC	Si existe se destruye antes de crearlo

El tercer argumento de open() tiene sentido sólo al crear un archivo. Sirve para especificar los permisos con los que será creado, siempre según el concepto de UNIX de permisos de **lectura, escritura y ejecución**, distribuidos en clases de usuarios. Para los sistemas operativos que no cuentan con estas nociones, el tercer argumento simplemente se ignora, pero se mantiene la interfaz POSIX para asegurar la portabilidad de los programas.

Nótese que las funciones ANSI C no permiten la especificación de estos permisos de creación.

Como para los flags, hay algunas constantes de bits útiles.

	Dueño	**Grupo**	**Otros**
Lectura	S_IRUSR (S_IREAD)	S_IRGRP	S_IROTH
Escritura	S_IWUSR (S_IWRITE)	S_IWGRP	S_IWOTH
Ejecución	S_IXUSR (S_IEXEC)	S_IXGRP	S_IXOTH
Los tres permisos	S_IRWXU	S_IRWXG	S_IRWXO

Ejemplo

```
int fd = open("prueba.dat", O_RDWR|O_CREAT, S_IRWXU|S_IRGRP);
```

Si el archivo prueba.dat no existe, se lo crea; se abre para lectura y escritura y con todos los permisos para el creador, pero sólo con permiso de lectura para el grupo del dueño. El resto de los usuarios no tiene ningún permiso sobre el archivo.

Para posicionar el puntero de lectura/escritura en un offset determinado, existe la función **lseek()**. El origen del desplazamiento se expresa, como en las funciones ANSI C de acceso directo, con las constantes SEEK_SET, SEEK_END y SEEK_CUR.

Ejemplo

Repetimos el ejemplo dado para las funciones ANSI donde se lee el registro lógico número 10 y tras una modificación se lo copia en el registro lógico 5, esta vez con funciones POSIX.

```
lseek(fd, 10L * sizeof(struct registro), SEEK_SET);
read(fd, &datos, sizeof(struct registro));
datos.dato1 = 1;
lseek(fd, 5L * sizeof(struct registro), SEEK_SET);
write(fd, &datos, sizeof(struct registro));
```

Ejercicios

1. Escribir una función que copie la entrada en la salida pero eliminando las vocales.

2. Escribir una función que reemplace los caracteres no imprimibles por caracteres **punto**.

3. Construir un programa que organice la salida de la función anterior para obtener un clon del comando **od** de UNIX.

4. Construir un programa que cuente la cantidad de palabras de un archivo, separadas por blancos, tabuladores o fin de línea.

5. Construir un programa que cuente la cantidad de caracteres y de líneas de un archivo.

6. Construir un programa que lea el listado de un directorio en formato largo (la salida del comando '**ls -l**') y devuelva la cantidad total de bytes ocupados.

7. Construir un programa que permita eliminar de un archivo las líneas que contengan una cadena dada.

8. Construir una función que pida por teclado datos personales (nombre, edad, ...) y los almacene en una estructura. Construir una función que imprima los valores recogidos por la función anterior.

9. Construir un programa que lea una cantidad de datos en lote y luego los imprima utilizando las funciones del ejercicio anterior. Generar un archivo de datos usando redirección.

10. Realizar el mismo programa del punto anterior pero efectuando toda la E/S sobre archivos. El programa deberá poder leer el archivo de datos del punto anterior.

11. Escribir una función que reciba como argumento dos enteros y devuelva un string de formato conteniendo una máscara de formato apropiada para imprimir un número en punto flotante. Por ejemplo, si se le dan como argumentos 7 y 2, deberá devolver el string "%7.2f". Aplicar la función para imprimir números en punto flotante.

12. Escribir sobre un archivo una variable int con valor 1 y una variable long con valor 2. Hacerlo primero con funciones de E/S con formato, y luego con funciones de acceso directo. Examinar en cada caso el resultado visualizando el archivo y opcionalmente con el comando **od -bc**, o con su programa clon realizado en esta práctica.

13. Defina una estructura básica simple para un registro, a su gusto. Puede ser un registro de información personal, bibliográfica, etc. Construya funciones para leer estos datos del teclado e imprimirlos en pantalla. Luego, usando funciones ANSI C, construya funciones para leer y escribir una de estas estructuras en un archivo, dado un número de registro lógico determinado.

14. Repita el ejercicio anterior reemplazando las funciones ANSI C por funciones POSIX.

15. Construya programas que utilicen las funciones anteriores, ANSI C o POSIX, y ofrezcan un menú de operaciones de administración: cargar un dato en el archivo, imprimir los datos contenidos en una posición determinada, listar la base generada completa, eliminar un registro, etc.

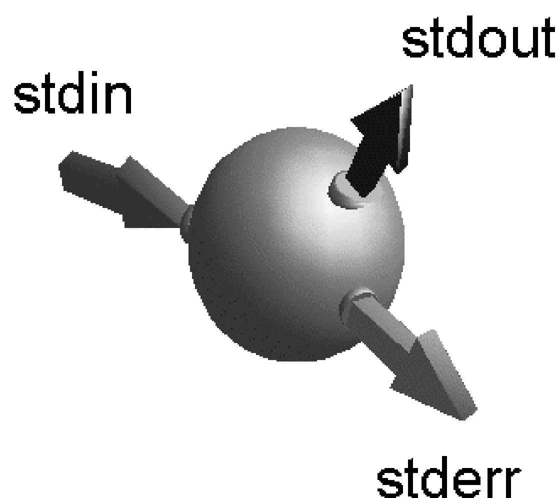
12. Comunicación con el ambiente

Entendemos por comunicación con el ambiente todas aquellas formas posibles de intercambiar datos entre el programa y el entorno, ya sea el sistema operativo, el shell del usuario u otro programa que lo haya lanzado. Una necesidad evidente de comunicación será recibir parámetros, argumentos u opciones de trabajo. Otras necesidades serán generar archivos con resultados, o comunicar una condición de error a la entidad que puso en marcha el programa.

Redirección y piping

Esta forma de comunicación en realidad no es específica del C sino que está implementada (hoy, en prácticamente todos los sistemas operativos) por el shell de usuario. Todos los programas en ejecución (o procesos) nacen con tres canales de comunicación abiertos: **entrada standard, salida standard y salida standard de errores**. Cuando el shell lanza un programa, por default le conecta estos tres canales con los dispositivos lógicos teclado, pantalla y pantalla respectivamente. El resultado es que el programa puede recibir caracteres por teclado e imprimir cadenas por pantalla haciendo uso de las funciones de entrada/salida corrientes.

Ahora bien, si el usuario indica al shell, en el momento de lanzar el programa, que desea reconectar alguno de estos canales con otros dispositivos lógicos o archivos, tenemos un fenómeno de **redirección**, que permite que el programa, sin cambio alguno, utilice las mismas funciones de entrada/salida para leer y generar archivos o comunicarse con dispositivos diferentes.

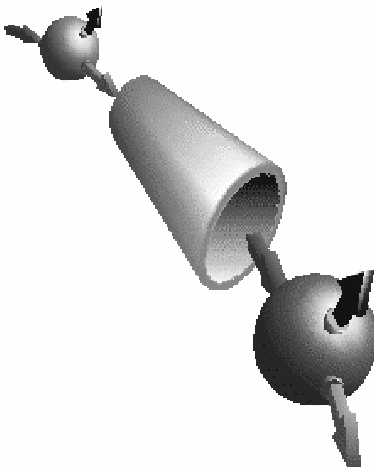


Los procesos reciben tres canales de comunicación abiertos por donde relacionarse con el ambiente. Mediante redirección se pueden crear archivos con el producto de su salida, o alimentarlos con el contenido de archivos preexistentes.

Otra alternativa es el **piping,** o entubamiento, que permite, con un solo comando de shell, el lanzamiento (en forma concurrente, si lo soporta el sistema operativo) de dos o más procesos con sus entradas y salidas interconectadas para funcionar acopladas. El shell se apoya en el sistema operativo para construir un **pipe,** o tubería temporaria, para conducir el flujo de datos entre los procesos que se

comunican.

El C adhiere a las convenciones de redirección y piping permitiendo manejar separadamente estos canales con sus funciones de biblioteca standard. Poder realizar piping entre procesos permite separar arquitecturalmente las funciones de un programa muy complejo, facilitando el desarrollo, aumentando la mantenibilidad y fomentando la reutilización de los programas escritos sin costo adicional de diseño o programación.



Los procesos pueden comunicarse a través de *pipes* o tuberías. El sistema operativo UNIX hace uso extensivo de esta capacidad proveyendo una gran cantidad de comandos sencillos que, combinados mediante piping, permiten crear poderosas herramientas sin necesidad de programación.

Para poder aprovechar estas capacidades solamente se requiere un protocolo común entre los programas que se comunicarán. Un medio para lograrlo, en aquellos programas que no son naturalmente cooperativos, es a veces construir adaptadores a nivel de shell. Estos son scripts generalmente sencillos que transforman un formato de datos en otro, facilitando la flexibilidad que no da el C por tratarse de un lenguaje compilado.

Los scripts, siendo interpretados, pueden ejecutarse directamente sin compilación. Pueden modificarse y probarse más rápidamente que los programas compilables, y la programación suele ser más flexible y poderosa. El costo asociado con el scripting es una menor velocidad de ejecución, lo que propone un estudio de cada caso, para optar entre scripting o programación ad hoc.

Ambientes como el moderno UNIX ofrecen numerosas herramientas y varios intérpretes de lenguajes de scripting, cada cual con mayores ventajas en un área determinada. Herramientas que es útil conocer son **grep**, **sed**, **diff**, **comm**, etc. El shell de usuario es normalmente una buena elección para scripting de tareas simples, poseyendo un lenguaje completo con manejo de variables, estructuras de control, arreglos, etc. Sin embargo, otros como **awk**, **Perl** o **Python** tienen mejores capacidades de manejo de cadenas, esencial para el trabajo que describimos, además de una sintaxis sumamente sintética y poderosa.

Variables de ambiente

El shell, responsable de recibir las órdenes del usuario para lanzar nuevos procesos, mantiene áreas de memoria reservadas para variables de ambiente que son accesibles a los nuevos procesos. Estas variables son simplemente pares (nombre, valor) de cadenas asociadas. Las variables de ambiente se pueden establecer y consultar con comandos de shell, desde la línea de comandos o desde un script; y lo mismo con funciones de biblioteca standard C desde un programa compilado. Los programas pueden consultar una variable de ambiente y decidir el curso de ejecución en función de su contenido; y pueden establecer sus valores para los procesos hijos que originen. Las variables de ambiente son una forma flexible de configurar el comportamiento de los programas.

Las funciones de manejo de variables de ambiente son **putenv()** y **getenv()** (POSIX). Ver también **setenv()** y **unsetenv()** (BSD 4.3).

Ejemplo

Estos comandos a nivel de shell colocan una variable y su valor en el ambiente. El comando **export** la hace visible a los procesos hijos.

```
$ DIR=/usr/local/programa
$ export DIR
```

Para leer la variable desde un programa C:

```
char directorio[50];
strcpy(directorio, getenv("DIR"));
```

Argumentos de ejecución

Un programa puede recibir argumentos al momento de ejecución, dados en la línea de comandos. El protocolo para recibir argumentos se ha diseñado para ser lo más general posible. Cada argumento en la línea de comandos es una cadena, independientemente del tipo de los datos, y se accede desde el programa como un puntero a carácter. Es responsabilidad del programa hacer las conversiones a los tipos esperados.

Los argumentos son recibidos por main() con las siguientes convenciones:

- main() espera dos parámetros, un entero y un arreglo de punteros a carácter.
- El primer parámetro representa la cantidad total de argumentos en la línea de comandos, incluido el nombre del programa.
- Los elementos del segundo parámetro son punteros a cadenas, terminadas en '\0', representando cada argumento recibido (incluyendo el nombre del programa).

Ejemplo

```
main(int argc, char *argv[])
{
    if(argc != 3)
        printf("Debe dar nombre y edad del usuario\n");
    else
        printf("Nombre: %s Edad: %d\n", argv[1], atoi(argv[2]));
}
```

Este programa se invocaría como:

```
$ programa Alicia 26
Nombre: Alicia Edad: 26
```

Salida del programa

Cada programa ha sido lanzado por algún otro, por lo común el shell del usuario. El programa puede seguir diferentes caminos de ejecución, encontrar errores, condiciones en las cuales es imposible proseguir, etc. Al momento de finalización del programa, puede ser interesante que el programa que le dio origen recoja alguna indicación de este estado final. El C tiene la capacidad (porque la tiene el sistema operativo) de devolver un entero, cuyo significado queda completamente librado al programador. El programa originador debe interpretar este código de retorno, que es una convención entre ambos programas. Es costumbre, aunque para nada obligatoria, devolver un 0 en caso de terminación exitosa, y números diferentes de cero para diferentes casos de terminación con error, al estilo de los protocolos de las funciones de biblioteca standard.

Esta característica es especialmente útil en el contexto de un script donde necesitamos determinar si se debe proseguir la ejecución en base al estado retornado por un programa invocado.

La función para terminar el programa devolviendo una señal de estado es **exit**(). Si no se dan argumentos, el valor devuelto queda indefinido.

Ejemplo

```
main(int argc, char *argv[])
{
    if(argc < 3) {
        printf("Insuficientes argumentos\n");
        exit(1);
    }
    procesar(argv[1],argv[2]);
    exit(0);
}
```

Opciones

Es muy común encontrar comandos del sistema operativo que aceptan un conjunto, a veces muy vasto, de opciones. Las opciones, si están presentes, se reconocen por comenzar con guiones, y deben ser los primeros argumentos dados al programa.

La convención usual en UNIX de expresar las opciones con un signo guión y letras, y opcionalmente argumentos numéricos, ha llevado a definir funciones de biblioteca standard para manejar conjuntos de opciones.

Ejemplo

```
#include <getopt.h>
#include <unistd.h>
extern char *optarg;
extern int optind, opterr, optopt;

int debug;

main(int argc, char *argv[])
```

```
{
    char *optstring="RrTtV:v:";
    int c;

    opterr=0;
    while((c=getopt(argc, argv, optstring)) != EOF)
        switch(c) {
            case 'v':
            case 'V':
                debug=atoi(optarg);
                printf("Nivel de debugging: %d\n",debug);
                break;
            case ':':
                printf("Falta valor numerico\n");
                exit(1);
                break;
            case 'R':
            case 'r':
                printf("Recibiendo\n");
                recibir(argv[optind]);
                break;
            case 'T':
            case 't':
                printf("Transmitiendo\n");
                transmitir(argv[optind]);
                break;
            case '?':
                printf("Mal argumento\n");
                break;
        }

}
```

El programa podría usarse tanto para transmitir como para recibir archivos observando un nivel de salida de debugging conveniente. Podría invocarse como:

```
$ transferir -v 2 -T archivo.txt
```

La función **getopt()** es quien va recogiendo las opciones vistas en la línea de comandos y devolviéndolas como caracteres separados. La variable string **optstring** contiene las opciones válidas. Para aquellas opciones (como V en el ejemplo) que pueden asumir un modificador numérico, se ubica un símbolo "dos puntos" a continuación en el string optstring. El valor para la opción numérica se recibe en la variable **optarg**.

Si ocurre un error sintáctico en el procesamiento de las opciones, la rutina devuelve el carácter '?' y emite un mensaje de error por salida de errores standard. Si no se desea emitir este mensaje, se hace **opterr**=0.

Las funciones recibir() y transmitir() obtienen el nombre del archivo del arreglo de argumentos argv[], indexándolo con la variable **optind**, que queda apuntando al siguiente elemento en la línea de comandos.

Ejercicios

1. Escribir un programa que imprima una secuencia de números consecutivos donde el valor inicial, el valor final y el incremento son dados como argumentos.

2. Mismo ejercicio pero donde los parámetros son pasados como variables de ambiente.

3. Mismo ejercicio pero donde los parámetros son pasados como opciones.

4. Programar una calculadora capaz de resolver cálculos simples como los siguientes:

```
$ casio 3 + 5
8
$ casio 20 * 6
120
$ casio 5 / 3
1
```

5. Agregar la capacidad de fijar precisión (cantidad de decimales) como una opción:

```
$ casio -d2  5 / 3
1.66
```

6. Manteniendo la capacidad anterior, agregar la posibilidad de leer una variable de ambiente que establezca la precisión default. Si no se da la precisión como opción, se tomará la establecida por la variable de ambiente, pero si se la especifica, ésta será la adoptada. Si no hay definida una precisión se tomará 0. Ejemplo:

```
$ casio 10 / 7
1
$ PRECISION_CASIO=5
$ export PRECISION_CASIO
$ casio 10 / 7
1.42857
$ casio -d2 10 / 7
1.42
```

7. Retomar ejercicios de programación de prácticas anteriores, agregándoles opciones. Por ejemplo, el programa para eliminar líneas de un archivo (práctica 11) admite una opción para elegir líneas **conteniendo** o **no conteniendo** una cadena. El programa que cuenta palabras de un archivo (misma práctica) puede recibir opciones o variables de ambiente especificando **cuáles serán los separadores** entre palabras.

13. Biblioteca Standard

La Biblioteca Standard no forma parte del C, estrictamente hablando, pero todos los compiladores contienen una implementación, a veces con agregados o pequeñas variantes. Desde la oficialización del ANSI C, los contenidos de la biblioteca standard se han estabilizado y se puede contar con el mismo conjunto de funciones en todas las plataformas.

Las funciones de la biblioteca standard están agrupadas en varias categorías. Para poder utilizar cualquiera de las funciones de cada categoría, es necesario incluir en el fuente el *header* asociado con la categoría. Esto **no implica** incluir los **textos** de las funciones en la unidad de traducción, sino simplemente incorporar los **prototipos** de las funciones de la biblioteca. Es decir, incluir un header de biblioteca standard no **define** las funciones que se van a usar, sino que las **declara**. La resolución de las referencias a las funciones o variables de biblioteca standard quedan pendientes hasta la linkedición.

Las categorías y los headers de la biblioteca standard son los siguientes. Una implementación de C puede aportar muchísimos otros headers más específicos. Con esta información no pretendemos reemplazar al manual del compilador sino orientar a los primeros pasos en el uso de la Biblioteca Standard.

Categoría	header
Macros de diagnóstico de errores	<assert.h>
Macros de clasificación de caracteres	<ctype.h>
Variables y funciones relacionadas con condiciones de error	<errno.h>
Características de la representación en punto flotante	<float.h>
Rangos de tipos de datos, dependientes de la plataforma	<limits.h>
Definiciones relacionadas con el idioma y lugar de uso	<locale.h>
Funciones matemáticas	<math.h>
Saltos no locales	<setjmp.h>
Manejo de señales	<signal.h>
Listas de argumentos variables	<stdarg.h>
Definiciones de algunos tipos y constantes comunes	<stddef.h>
Entrada/salida	<stdio.h>
Varias funciones útiles	<stdlib.h>
Operaciones sobre cadenas	<string.h>
Funciones de fecha y hora	<time.h>

Ya hemos visitado la mayoría de las funciones de la categoría de E/S. Nos quedan por explorar algunas otras de importancia.

Funciones de strings (<stdio.h>)

No existiendo un tipo de datos string en C, se lo implementa como un arreglo de caracteres, dado por su dirección inicial, y terminado en el carácter especial '\0'. Todas las funciones de strings de BS hacen uso de este protocolo de fin de string. Muchas de ellas han sido implementada en las prácticas de capítulos anteriores.

Aquí las más importantes. Consultar también manual de las funciones **strspn(), strcspn(), strpbrk(), strstr(), strtok().**

prototipo	significado	ejemplo
`char *strcpy(s, ct)`	Copia la cadena ct sobre s, incluyendo el '\0'. Devuelve s.	`char alfa[10];` `strcpy(alfa, "cadena");`

`char *strncpy(s, ct, n)`	Copia ct sobre s hasta n caracteres. Rellena con '\0' hasta el final si ct tiene menos de n caracteres.	`char alfa[10];`
		`strncpy(alfa, "cadena", 4);`
		`strncpy(alfa, otro, sizeof(alfa));`

`char *strcat(s, ct)`	Concatena la cadena ct al final de s. Debe garantizarse espacio para realizar la operación.	`char alfa[10] = "abc";` `strcat(alfa, "def");`
`char *strncat(s, ct)`	Idem hasta n caracteres.	
`int strcmp(cs, ct)`	Compara las cadenas. Devuelve <0 si cs<ct, 0 si cs==ct, >0 si cs>ct.	`if(strcmp(alfa,"abcdef") == 0)` `    printf("iguales\n");`
`int strncmp(cs, ct)`	Idem hasta n caracteres.	

`char *strchr(cs, c)`	Devuelve un apuntador a la primera ocurrencia del carácter c en la cadena cs, o bien NULL si no se lo halla.	`char *p;`
		`char r[] = "casualidad";`
		`p = strchr(r, 's');`
		`strncpy(p, "us", 2);`

`char *strrchr(cs, c)`	Idem la última ocurrencia.	
`size_t strlen(cs)`	Devuelve la longitud de la cadena, sin contar el '\0' final.	`for(i=0; i < strlen(s); i++)` `    printf("%c\n", s[i]);`

Listas de argumentos variables (<stdarg.h>)

Es posible definir funciones que reciban una cantidad arbitraria de parámetros reales. Para esto se prepara un encabezado de la función con los parámetros reales fijos que se desee y se indican los restantes, variables, mediante puntos suspensivos. Se recuperan los demás con macros especiales definidas en este header.

Lamentablemente estas macros no permiten la creación de funciones **sin** argumentos fijos. Existe otro paquete de argumentos variables, definido en **varargs.h**, que sí lo permite; pero que no está comprendido en el estándar ANSI C y que no es compatible con stdarg.h.

Ejemplo

```
#include <stdarg.h>
int sumar(int cuantos, ...)
{
    va_list ap;
    int suma=0;

    va_start(ap, cuantos);
    for(i=0; i<cuantos; i++)
        suma += va_arg(ap, int);
    va_end(ap);
    return suma;
}
```

Que se utilizaría como:

```
main()
{
    printf("Resultado 1: %d\n", sumar(3, 4, 5, 6));
    printf("Resultado 2: %d\n", sumar(2, 100, 2336));
}
```

Funciones de tratamiento de errores (<errno.h> y <assert.h>)

Esta zona de la Biblioteca provee indispensables herramientas de debugging. La variable externa **errno** es un entero que toma un valor de acuerdo a condiciones de error provocadas por cualquiera de las funciones de la BS, y de acuerdo a una catalogación de errores que depende de la función. Si una función ANSI C devuelve un valor indicador de error (como NULL donde debería devolver un puntero, o negativo donde debería devolver un positivo), la variable errno contendrá más explicación sobre el motivo del error. Se consulta con las funciones **strerror()** o **perror()**. La función perror() admite una cadena arbitraria para indicar, por ejemplo, el lugar del programa donde se produce el error. Imprimirá esta cadena más la descripción del problema.

La macro **assert** sirve para detener la ejecución cuando se alcanza un estado imposible para la lógica del programa. Para usarla adecuadamente es necesario identificar invariantes en el programa (condiciones que no deban jamás ser falsas). Si al evaluarse la macro resulta que su condición argumento es falsa, assert() aborta el programa indicando nombre del archivo fuente y línea donde estaba originalmente la llamada. Un programa en producción no debería fallar debido a assert.

Ejemplos

```
#include <errno.h>
if(open("noexiste", O_RDONLY) < 0)
    perror("Error en apertura");


#include <assert.h>
...
assert(restantes >= 0);
```

Funciones de fecha y hora (<time.h>)

Existen dos tipos definidos en el header time.h para manejar datos de fechas. Por un lado, se tiene el tipo **struct tm**, que contiene los siguientes elementos que describen un momento en el tiempo:

```
struct tm {
int tm_sec,     /* segundos 0..59 */
    tm_min,     /* minutos 0..59 */
    tm_hour,    /* horas 0..23 */
    tm_mday,    /* día del mes 1..31 */
    tm_mon,     /* meses desde enero 0..11 */
    tm_year,    /* años desde 1900 */
    tm_wday,    /* días desde el domingo 0..6 */
    tm_yday,    /* días desde enero 0..365 */
    tm_isdst;   /* flag de ahorro diurno de luz */
};
```

Por otro lado, existe un segundo formato de representación interna de fechas, **time_t,** que es simplemente un entero conteniendo la cantidad de segundos desde el principio de la era UNIX ("*the epoch*"), acaecido el 1/1/1970 a la hora 0 UTC. Este formato es el usado por la función **time**() que da la hora actual.

Este formato entero puede convertirse a struct tm y viceversa con funciones definidas en esta zona de la BS, como **mktime()** y **gmtime()**.

El contenido de una estructura **tm** se puede imprimir con gran variedad de formatos con la función **strftime()**, que acepta una cantidad de especificaciones al estilo de printf(). Las funciones **ctime()** y **asctime()** son más sencillas. Devuelven una cadena conteniendo una fecha en formato normalizado (como el que aparece en los mensajes de correo electrónico). La primera recibe un puntero a time_t; la segunda, un puntero a struct tm.

Ejemplo

```
time_t t;
struct tm *stm;

t = time(NULL);              /* recoge la hora actual */
printf("%s\n",ctime(&t));    /* imprime en formato standard */

char area[100];
stm = gmtime(&t);            /* convierte t a struct tm */
strftime(area,sizeof(area), /* prepara string segun formato del usuario */
    "%A %b %d %H",stm);
printf("%s\n",area);         /* lo imprime */
```

Funciones matemáticas (<math.h>)

Aquí se encuentran las habituales funciones aritméticas avanzadas, trigonométricas y logarítmicas.

sin(x)	seno de x	
cos(x)	coseno de x	
tan(x)	tangente de x	
asin(x)	arco seno de x	Devuelve valores en el rango [-pi/2, pi/2]
acos(x)	arco coseno de x	En el rango [0, pi]
atan(x)	arco tangente de x	Devuelve valores en el rango [-pi/2, pi/2]
atan2(y,x)	arco tangente de y/x	En el rango [-pi, pi]
sinh(x)	seno hiperbólico de x	
cosh(x)	coseno hiperbólico de x	
tanh(x)	tangente hiperbólica de x	
exp(x)	función exponencial de base *e*	
log(x)	logaritmo natural	
log10(x)	logaritmo de base decimal	
pow(x,y)	x^y	Para x=0 debe ser y>0. Si x<0, debe ser y entero
sqrt(x)	raíz cuadrada de x	Debe ser x>=0
ceil(x)	menor entero no menor que x	Devuelve un double
floor(x)	mayor entero no mayor que x	Devuelve un double
fabs(x)	Valor absoluto de x	
ldexp(x,n)	Devuelve x * 2^n	
frexp(x,exp)	Divide a x en una potencia entera de 2, que se almacena en el lugar apuntado por exp, y devuelve una mantisa en el intervalo [½, 1].	
modf(x, ip)	Divide a x en parte entera fraccionaria. El argumento ip debe ser un puntero a entero.	
fmod(x,y)	Residuo de punto flotante de x/y.	

Funciones utilitarias (<stdlib.h>)

El header **stdlib.h** agrupa las declaraciones de varias funciones no relacionadas entre sí, y que sirven a varios fines. Solamente las nombramos y encarecemos la lectura del manual.

- Funciones de conversión: las funciones **atoi(), atol(), atof(), strtol(), strtod(), strtoul()**, toman cadenas representando números y generan los elementos de datos del tipo correspondiente.
- Se pueden generar números aleatorios con **rand()** y **srand().**
- Aquí también se declaran las funciones de asignación de memoria como **malloc(), calloc(), realloc(), free().**

- Para manejar datos en memoria con eficiencia se puede recurrir a **qsort()** y **bsearch()**, que ordenan una tabla y realizan búsqueda binaria en la tabla ordenada.

Clasificación de caracteres (<ctype.h>)

El header **ctype.h** contiene declaraciones de macros para averiguar la pertenencia de un carácter a determinados conjuntos. Son todas booleanas salvo las últimas que devuelven ints.

macro	devuelve TRUE cuando
isalnum(c)	isalpha(c) o isdigit(c) son TRUE
isalpha(c)	isupper(c) o islower(c) son TRUE
iscntrl(c)	c es un carácter de control
isdigit(c)	c es un dígito decimal
isgraph(c)	c es un carácter imprimible y no isspace(c)
islower(c)	c es una letra minúscula
isprint(c)	c es un carácter imprimible, incluyendo el caso en que isspace(c) es TRUE
ispunct(c)	c es imprimible pero no espacio, letra ni dígito
isspace(c)	c es espacio, fin de línea, tabulador
isupper(c)	c es letra mayúscula
isxdigit(c)	c es dígito hexadecimal
tolower(c)	devuelve c en minúscula
toupper(c)	devuelve c en mayúscula

Ejercicios

1. Utilizar la función de cantidad variable de argumentos definida más arriba para obtener los promedios de los 2, 3, ..., n primeros elementos de un arreglo.

2. Construir una función de lista variable de argumentos que efectúe la concatenación de una cantidad arbitraria de cadenas en una zona de memoria provista por la función que llama.

3. Construir una función de cantidad variable de argumentos que sirva para imprimir, con un formato especificado, mensajes de debugging, conteniendo nombres y valores de variables.

4. Construir un programa que separe la entrada standard en palabras, usando las macros de clasificación de caracteres. Debe considerar como delimitadores a los caracteres espacio, tabulador, signos de puntuación, etc.

5. Dadas dos fechas y horas del día, calcular su diferencia. Utilizar las funciones de BS para convertir a tipos de datos convenientes e imprimir la diferencia en años, meses, días, horas, etc.

6. Generar fechas al azar dentro de un período de tiempo dado.