Dpto. de Ingeniería Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Fundamentos de Programación I

Apuntes de la Asignatura

Índice general

I Teoría **1**

1. Introducción a la Programación **1-1**
Arquitectura Básica de los computadores . . . 1-3
Introducción a los Sistemas Operativos . . . 1-6
Introducción a los Lenguajes de Programación . . . 1-9

2. Programación Estructurada **2-1**
Resolución de Problemas y Algoritmos . . . 2-1
Introducción al Lenguaje C . . . 2-10

3. Codificación de la Información **3-1**
3.1. Introducción . . . 3-1
3.1.1. Los Sistemas Posicionales . . . 3-1
3.2. Codificación de Enteros sin Signo . . . 3-2
3.2.1. La aritmética binaria . . . 3-3
3.2.2. Otras Bases de Numeración . . . 3-3
3.2.3. El Desbordamiento en los Enteros sin Signo . . . 3-5
3.3. Codificación de los Enteros con Signo . . . 3-6
3.3.1. Codificación en Complemento a Dos . . . 3-6
3.3.2. El Desbordamiento en los Enteros con Signo . . . 3-7
3.4. Codificación de Números con Decimales . . . 3-7

4. Tipos, Constantes y Variables **4-1**
4.1. Tipos de Datos . . . 4-1
4.1.1. Los Tipos de Datos en C . . . 4-2
4.1.2. Tipos Enteros sin Signo . . . 4-2
4.1.3. Tipos Enteros con Signo . . . 4-2
4.1.4. Tipos con Decimales . . . 4-3
4.1.5. Tipo Lógico . . . 4-3
4.1.6. Tipo Carácter . . . 4-4
4.2. Constantes . . . 4-4
4.2.1. Constantes enteras . . . 4-5
4.2.2. Constantes en coma flotante . . . 4-5

4.2.3. Constantes de carácter . . . 4-6
4.2.4. Cadenas de caracteres . . . 4-6
4.3. Variables . . . 4-7
4.3.1. Declaración de Variables . . . 4-7
4.3.2. Utilización del Valor de una Variable . . . 4-8
4.3.3. Asignación de Valores a Variables . . . 4-8
4.3.4. Tiempo de Vida de las Variables . . . 4-9
4.3.5. Visibilidad de las variables . . . 4-10

5. Expresiones **5-1**
5.1. Introducción . . . 5-1
5.2. Operadores Aritméticos . . . 5-2
5.2.1. Operaciones con valores de tipo char . . . 5-3
5.3. Operadores Relacionales . . . 5-3
5.3.1. Precisión de los Números con Decimales . . . 5-4
5.4. Operadores Lógicos . . . 5-4
5.5. Operadores de Manejo de Bits . . . 5-5
5.6. Asignación . . . 5-6
5.7. Asignación Compacta . . . 5-7
5.8. Conversión Forzada . . . 5-8
5.9. Operadores de Autoincremento y Autodecremento . . . 5-9
5.9.1. Efectos secundarios de ++ y -- . . . 5-10
5.10. Operador Condicional . . . 5-11
5.11. Operador `sizeof` . . . 5-11
5.12. Reglas de Prioridad . . . 5-12

6. Estructuras de Control **6-1**
6.1. Introducción . . . 6-1
6.2. Estructuras Secuenciales . . . 6-2
6.3. Estructuras Selectivas . . . 6-2
6.3.1. Estructura Selectiva Simple: `if else` . . . 6-3
6.3.2. Sentencias Selectivas Simples Anidadas . . . 6-4
6.3.3. Ejemplo . . . 6-5
6.3.4. Estructura Selectiva Múltiple: `switch` . . . 6-6
6.4. Estructuras Repetitivas . . . 6-9
6.4.1. Sentencia `while` . . . 6-9
6.4.2. Sentencia `do while` . . . 6-11
6.4.3. Sentencia `for` . . . 6-12
6.5. Ejemplos de Uso . . . 6-15
6.5.1. Lectura del teclado . . . 6-15
6.5.2. Solución de una ecuación de primer grado . . . 6-15

7. Funciones **7-1**

7.1. Funciones: Introducción . . . 7-1
7.1.1. La función `main` . . . 7-2
7.2. Definición de una Función . . . 7-2
7.2.1. El identificador . . . 7-3
7.2.2. Los parámetros . . . 7-3
7.2.3. El Cuerpo de la Función . . . 7-3
7.2.4. El Tipo de la Función . . . 7-4
7.3. Declaración de una Función . . . 7-4
7.4. Utilización de una Función . . . 7-5
7.4.1. Correspondencia de parámetros . . . 7-5
7.5. Recursividad . . . 7-7
7.6. Compilación Separada y Descomposición en Ficheros . . . 7-8
7.6.1. Funciones y ficheros de cabecera . . . 7-9
7.7. El preprocesador de C. . . . 7-9
7.7.1. La directiva `include` . . . 7-10
7.7.2. La directiva `define` . . . 7-10
7.7.3. Directivas condicionales . . . 7-12
7.8. Resultado final . . . 7-13

8. Punteros **8-1**
8.1. Introducción . . . 8-1
8.1.1. Declaración de Punteros . . . 8-2
8.1.2. El valor `NULL` en los Punteros . . . 8-3
8.2. Operadores Específicos de Punteros . . . 8-3
8.2.1. Prioridades de los Operadores de Punteros . . . 8-4
8.3. Aritmética de punteros . . . 8-5
8.3.1. Operaciones entre punteros y enteros . . . 8-5
8.3.2. Resta de dos Punteros . . . 8-6
8.3.3. Comparaciones de Punteros . . . 8-6
8.4. Punteros a Punteros . . . 8-7
8.5. Uso de Punteros para Devolver más de un Valor . . . 8-7

9. Tablas **9-1**
9.1. Introducción . . . 9-1
9.2. Declaración de las Tablas . . . 9-2
9.2.1. Inicialización de Tablas de Caracteres . . . 9-2
9.3. Acceso a los Elementos de una Tabla . . . 9-3
9.4. Recorrer los Elementos de una Tabla . . . 9-4
9.5. Utilización de Constantes Simbólicas . . . 9-4
9.6. Punteros y Tablas . . . 9-5
9.6.1. Diferencias y Usos . . . 9-7
9.7. Tablas como Resultado de una Función . . . 9-8

9.8. Tablas Multidimensionales . . . 9-8
9.8.1. Inicialización de Tablas Multidimensionales . . . 9-9
9.9. Tablas de punteros . . . 9-9
9.9.1. Tablas de punteros y tablas multidimensionales . . . 9-10
9.10. Tablas como Parámetros de Funciones . . . 9-10
9.10.1. Tablas Multidimensionales como Parámetros . . . 9-11
9.10.2. Tablas de Punteros como Parámetros . . . 9-13
9.10.3. Argumentos en la Línea de comandos . . . 9-13

10. Tipos Agregados **10-1**
10.1. Estructuras . . . 10-1
10.1.1. Declaración de Estructuras y Variables . . . 10-1
10.1.2. Inicialización de Estructuras . . . 10-3
10.1.3. Punteros a Estructuras . . . 10-4
10.1.4. Acceso a los campos de una estructura . . . 10-4
10.1.5. Estructuras como Campos de Estructuras . . . 10-5
10.1.6. El Operador de Asignación en las Estructuras . . . 10-6
10.1.7. Tablas de Estructuras . . . 10-7
10.1.8. Estructuras como Argumentos de Funciones . . . 10-8
10.1.9. Estructuras como Resultado de una Función . . . 10-8
10.2. Uniones . . . 10-8
10.2.1. Declaración de Uniones . . . 10-8
10.2.2. Inicialización de una Unión . . . 10-9
10.2.3. Acceso a los campos de una unión . . . 10-9
10.3. Campos de bits . . . 10-9
10.4. El tipo Enumerado . . . 10-11
10.4.1. Declaración de Enumerados . . . 10-11
10.4.2. Declaración de Variables de Tipo Enumerado . . . 10-12
10.5. La Definición de Nuevos Tipos . . . 10-12
10.5.1. Uso del typedef . . . 10-13

11. Funciones de Biblioteca **11-1**
11.1. Introducción . . . 11-1
11.2. Reserva dinámica de memoria . . . 11-2
11.2.1. Función de reserva **`malloc`** . . . 11-2
11.2.2. Función de reserva **`calloc`** . . . 11-3
11.2.3. Función de liberación **`free`** . . . 11-3
11.2.4. La reserva dinámica y las tablas multidimensionales . . . 11-4
11.3. Entrada y Salida en C. . . . 11-5
11.3.1. Funciones de apertura y cierre . . . 11-6
11.3.2. Funciones de entrada y salida . . . 11-6
11.3.3. Salida con formato: función **`fprintf`** . . . 11-8

11.3.4. Entrada con formato: función **fscanf** . . . 11-9
11.3.5. Funciones de lectura/escritura binaria . . . 11-10
11.3.6. Funciones especiales . . . 11-10
11.3.7. Flujos estándar: **stdin**, **stdout** y **stderr** . . . 11-11
11.3.8. Errores frecuentes en el uso de las funciones de E/S . . . 11-11
11.4. Funciones de Cadenas de Caracteres . . . 11-15
11.4.1. Funciones de copia . . . 11-15
11.4.2. Funciones de encadenamiento . . . 11-15
11.4.3. Funciones de comparación . . . 11-15
11.4.4. Funciones de búsqueda . . . 11-15
11.4.5. Otras funciones . . . 11-16

II Anexos **11-17**

1. El Entorno del Centro de Cálculo **1-1**
1.1. Introducción . . . 1-1
1.2. Arranque del sistema . . . 1-1
1.3. Funcionamiento del Entorno . . . 1-2
1.3.1. El terminal . . . 1-2
1.4. Los dispositivos de memoria externos . . . 1-3
1.4.1. Conexión . . . 1-4
1.4.2. Desconexión . . . 1-4

2. El Sistema Operativo UNIX **2-1**
2.1. Introducción . . . 2-1
2.2. Estructura de directorios y sistema de ficheros . . . 2-2
2.2.1. La estructura de directorios . . . 2-2
2.2.2. Permisos . . . 2-3
2.3. Formato general de las órdenes . . . 2-4
2.3.1. Ayuda en línea . . . 2-4
2.3.2. Operaciones sobre el sistema de ficheros . . . 2-4
2.3.3. Operaciones sobre directorios . . . 2-7
2.3.4. Comandos para visualización de ficheros . . . 2-8
2.3.5. Comandos informativos . . . 2-9
2.3.6. Otros comandos . . . 2-10
2.4. Funciones especiales del intérprete de comandos . . . 2-11
2.4.1. Caracteres ordinarios y caracteres de control . . . 2-11
2.4.2. Metacaracteres . . . 2-12
2.4.3. Redireccionamiento de la entrada/salida . . . 2-12
2.5. Procesos . . . 2-13
2.5.1. Intercomunicación entre procesos . . . 2-13
2.5.2. Control de trabajos . . . 2-13

2.5.3. Órdenes de información sobre procesos y trabajos . . . 2-14
2.5.4. Órdenes de cambio de estado de un trabajo . . . 2-15

3. El compilador `gcc` 3-1
3.1. Introducción . . . 3-1
3.1.1. Etapas en la generación de un ejecutable . . . 3-2
3.2. El comando **`gcc`** . . . 3-4
3.2.1. Uso de la opciones **`-W`** y **`-Wall`** . . . 3-4
3.2.2. Uso de la opción -c . . . 3-5

4. El procesador de órdenes `make` 4-1
4.1. Introducción . . . 4-1
4.2. Descripción . . . 4-1
4.3. Sintaxis . . . 4-2
4.4. El Comportamiento de make . . . 4-3
4.4.1. Objetivos sin Dependencias . . . 4-5
4.4.2. Forzar la reconstrucción completa . . . 4-5
4.5. Macros en makefiles . . . 4-5

5. El depurador de programas `gdb` 5-1
5.1. Introducción . . . 5-1
5.2. Llamada al depurador gdb . . . 5-2
5.2.1. El intérprete de comandos de **`gdb`** . . . 5-3
5.3. Órdenes de ayuda . . . 5-4
5.3.1. Orden **`help`** . . . 5-4
5.3.2. Orden **`info`** . . . 5-4
5.3.3. Orden **`show`** . . . 5-4
5.3.4. Orden **`shell`** . . . 5-4
5.4. Ejecución de programas bajo gdb: orden **`run`** . . . 5-4
5.5. Puntos de parada (*breakpoints*) . . . 5-5
5.5.1. Establecimiento de puntos de parada . . . 5-5
5.5.2. Información sobre los puntos de parada . . . 5-5
5.5.3. Deshabilitar un punto de parada . . . 5-6
5.5.4. Habilitar un punto de parada . . . 5-6
5.5.5. Borrado de un punto de parada . . . 5-6
5.5.6. Ejecución automática en la parada . . . 5-6
5.5.7. Continuación de la ejecución . . . 5-7
5.6. Examinar los Ficheros Fuente . . . 5-7
5.7. Examinar la Pila . . . 5-8
5.8. Examinar los datos . . . 5-9

6. Guía de Estilo. Consejos y normas de estilo para programar en C 6-1
6.1. Introducción . . . 6-1

6.2. El fichero fuente 6-2
6.3. Sangrado y separaciones 6-3
6.4. Comentarios 6-4
6.5. Identificadores y Constantes 6-5
6.6. Programación Modular y Estructurada 6-6
6.7. Sentencias Compuestas 6-6
6.7.1. Sentencia **`switch`** 6-7
6.7.2. Sentencia **`if-else`** 6-7
6.8. Otros 6-7
6.9. Organización en ficheros 6-8
6.9.1. Ficheros de cabeceras 6-8
6.10. La sentencia **`exit`** 6-9
6.11. Orden de evaluación 6-9
6.12. Los operadores **`++`**, **`--`** 6-9
6.13. De nuevo los prototipos 6-10

III Ejercicios **6-13**

3. Codificación de la Información **3-1**

4. Tipos, Constantes y Variables **4-1**

5. Expresiones **5-1**

6. Estructuras de Control **6-1**

7. Funciones **7-1**

8. Punteros **8-1**

9. Tipos Agregados **9-1**

Parte I

Teoría

Tema 1

Introducción a la Programación

Índice

Arquitectura Básica de los computadores **1-3**
Introducción a los Sistemas Operativos **1-6**
Introducción a los Lenguajes de Programación **1-9**

Introducción

- Informática: ciencia que estudia todo lo referente al procesamiento de la información de forma automática.
- Ordenador: máquina destinada a procesar información.
- Programa: secuencia de instrucciones que describe cómo ejecutar cierta tarea.

ait

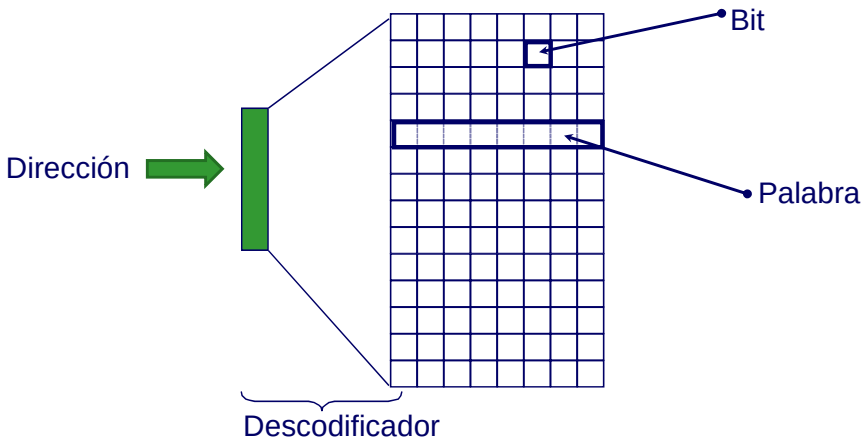
Arquitectura Básica de los Computadores

Memoria Principal

- También llamada central
- Almacena datos e instrucciones
- Convierte al ordenador en una máquina de propósito general
- De acceso directo y rápido
- Operaciones básicas: lectura y escritura

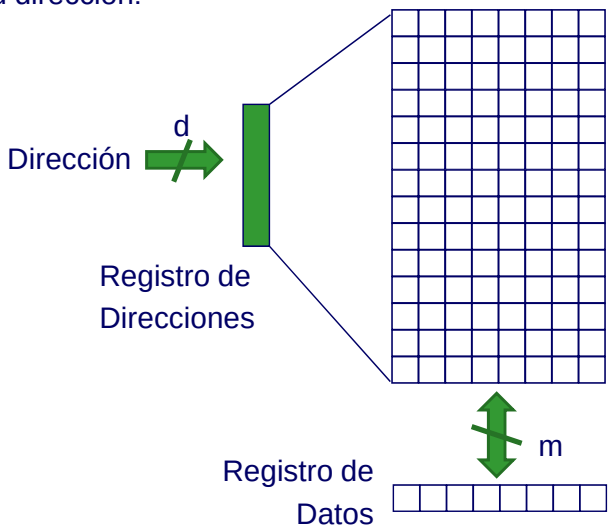
La memoria principal

- La memoria de un ordenador tiene organización matricial: cada fila una “palabra de memoria”



Direccionamiento

- Procedimiento para seleccionar una palabra de memoria mediante su dirección:



Direccionamiento

- Registro de direcciones:
  - tamaño suficiente para direccionar todas las posiciones (direcciones o filas) de la memoria.
- $2^d \geq$ Núm. direcciones memoria
  - d = 2 => 2^2 palabras
  - d = 3 => 2^3 palabras
  - d = 10 => 2^{10} palabras = 1024
- Tam_memoria = Num_palabras * tam_palabra

Unidades de medida de la capacidad de información

- bit: Unidad mínima de información.
  - 0/1, ON/OFF. Se representa con b.
- octeto o byte (8 bit)
  - Se representa con B.
- kilobyte (1024 B ó 2^{10}B)
  - Se representa con kB.
- megabyte (1024 kB ó 2^{10}kB ó 2^{20}B)
  - Se representa con MB.
- gigabyte (1024 MB ó 2^{10}MB ó 2^{20}kB ó 2^{30}B)
  - Se representa con GB.

ait

Introducción a los Sistemas Operativos

Introducción

- Sistema Operativo: programa (o conjunto de programas) que:
  - controla todos los recursos de la computadora
  - proporciona la base sobre la cual pueden escribirse todos los programas de aplicación.
- Presenta la máquina al usuario como una interfaz o máquina virtual.
- Gestiona los recursos hardware.

Sistema Operativo

- Facilita las tareas de programación
  - proporciona lenguaje de control o comandos
  - detecta errores
  - facilita operaciones de E/S
  - gestiona operaciones con ficheros
- Gestiona recursos internos de la máquina:
  - asignación de recursos
  - resolución de conflictos, prioridades..
  - chequeo del sistema

Estructura de niveles de un ordenador

<table>
<tr><td colspan="3">Programas de Aplicación</td></tr>
<tr><td>Compilador/
Intérprete</td><td>Editores</td><td>Intérprete
de comandos</td></tr>
<tr><td colspan="3">Sistema Operativo</td></tr>
<tr><td colspan="3">Lenguaje Máquina</td></tr>
<tr><td colspan="3">Dispositivos Físicos</td></tr>
</table>

Gestión de la Memoria

- Un programa debe estar en memoria para ser ejecutado.
  - El S.O (o parte de él) también debe estar.
  - Puede haber varios programas en memoria.
- Para esto debe proporcionar mecanismos de gestión de estas zonas
  - Asigna espacio en memoria a los procesos cuando éstos la necesiten
  - Los libera cuando terminan.

ait

Introducción a los Lenguajes de Programación

Lenguajes de Programación

- Introducción. Historia
- Clasificación.
- Ensamblador
- Lenguajes de Alto Nivel
- Traductores
- Editores, enlazadores y Cargadores.
- Entornos

Introducción

- Programa
  - conjunto ordenado de instrucciones (susceptible de ser entendido por un ordenador)
- Lenguaje de programación:
  - conjunto de símbolos
  - normas para la correcta combinación de estos

Evolución Histórica

- Lenguaje Máquina
- Lenguaje Ensamblador
- 50´s: Fortran, Algol, Lisp
- 60's: Cobol, Basic
- 70's: Pascal, C
- 80's: C++
- 90's: Java

Clasificaciones

- Según proximidad al lenguaje máquina:
  - de bajo nivel: ensamblador, máquina
  - de alto nivel: Pascal, C
- Según su propósito:
  - de propósito general: Pascal, C
  - de propósito específico: Cobol, RPG
- Según su orientación:
  - al procedimiento, imperativos: C, Pascal
  - al problema, declarativos: Prolog
  - a objetos: C++, Java

Lenguaje Máquina

- Es el directamente inteligible por la máquina.
- Sus instrucciones:
  - son cadenas binarias.
  - dependen del hardware de la computadora
  - difieren de una a otra.

Lenguaje Máquina

- Ventajas:
  - Rapidez.
  - No necesita traducción.
- Desventajas:
  - Dificultad.
  - Lentitud en la codificación.
  - Sólo válidos en una máquina.

Lenguaje de Bajo Nivel

- Más fácil de usar que el lenguaje máquina.
- Sigue dependiendo de la máquina.
- Lenguaje de bajo nivel por excelencia:
  - **ensamblador** (assembly language).
- Mejoras respecto al lenguaje máquina:
  - Nemotécnicos para las instrucciones (ADD, SUB, …)
  - Nombres simbólicos para el manejo de datos y posiciones de memoria.

Lenguajes de Alto Nivel (I)

- Diseñados para ser legibles:
  - Se aprenden, modifican, etc. mejor que los lenguajes máquina y ensamblador.
- Son independientes de la máquina (no dependen del hardware):
  - Son *portables (o transportables)*: pueden utilizarse con poca o ninguna modificación en diferentes tipos de computadoras.

Lenguajes de Alto Nivel (y II)

- Objetivos
  - Solución de problemas
  - Simplicidad
  - Eficiencia
  - Legibilidad
- Características
  - palabras reservadas
  - reglas sintácticas
  - uso de variables y constantes
  - operadores aritméticos y lógisos
  - control del flujo del programa: bifurcaciones, bucles..

Ventajas e Inconvenientes

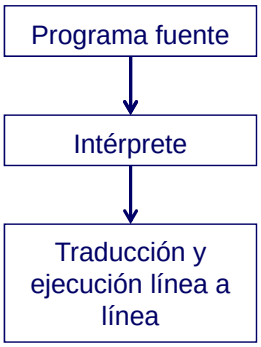
- Ventajas:
  - Menor tiempo de formación de programadores.
  - Basados en reglas sintácticas similares a los lenguajes humanos.
  - Mayor facilidad para modificar y poner a punto los programas.
  - Reducción en el coste de los programas.
  - Portables.
- Inconvenientes:
  - Necesita diferentes traducciones.
  - No se aprovechan los recursos internos de la máquina.
  - Aumento de la ocupación de memoria.
  - El tiempo de ejecución de los programas es “mucho” mayor.

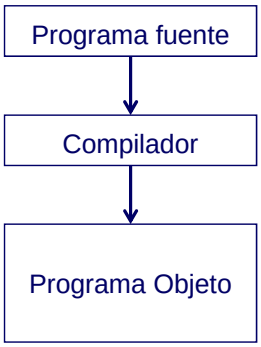
Traductor

- Es un programa que *traduce* un programa en un lenguaje de programación a su correspondiente en lenguaje máquina.
  - Además puede detectar errores y optimizar.
- Los traductores se dividen en:
  - Ensambladores: de lenguaje ensamblador a lenguaje máquina.
  - Compiladores: de alto nivel a máquina
  - Intérpretes: traducen instrucción a instrucción.

Intérprete vs. Compilador

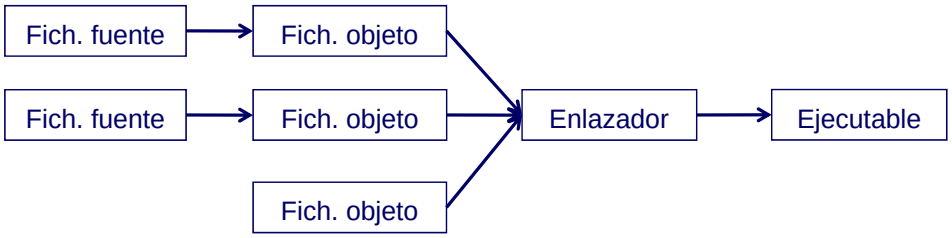
- Intérprete: traductor que toma el programa fuente y lo traduce y ejecuta línea a línea.
  - Basic, Java, Smalltalk
- Compilador: traduce los programas fuente escritos en lenguaje de alto nivel a lenguaje máquina.
  - C, C++, pascal, FORTRAN, COBOL.





Compilación y sus fases

- La **compilación** es el proceso de traducción de programas fuente a programas objeto.
- Para obtener el programa máquina real (o ejecutable) se debe usar el montador o **enlazador** (*linker*).



Entorno de Programación

Tema 2

Programación Estructurada

Índice

Resolución de Problemas y Algoritmos **2-1**
Introducción al Lenguaje C **2-10**

ait

Resolución de Problemas y Algoritmos

Introducción

- El objetivo fundamental es enseñar a resolver problemas mediante un ordenador.
- Un programador es primero una persona que resuelve problemas.
- La resolución de problemas deberá ser rigurosa y sistemática.
- La metodología necesaria para resolver problemas mediante programas se llama metodología de programación.
- El eje central de esta metodología es el algoritmo.

Algoritmo

- Un algoritmo es un método para resolver un determinado problema.
- El algoritmo es previo al programa.
- Un programa es un algoritmo expresado en un lenguaje de programación.
- Características fundamentales que debe cumplir un algoritmo:
  - Ser **preciso**: indicar el orden de realización paso a paso.
  - Ser **determinista**: siempre se comporte igual.
  - Ser **finito**: debe terminar en algún momento.

Diseño de un algoritmo

- Debe ser descendente (top-down):
  - dividir el problema en subproblemas.
- Debe presentar refinamiento sucesivo, con descripción detallada.
- La especificación debe ser independiente del lenguaje y puede ser:
  - mediante diagramas de flujo
  - mediante pseudocódigo

Ejemplo de algoritmo

- Especificaciones:
  - Calcular el valor medio de una serie de números positivos que se leen del teclado.
  - Un valor cero o negativo indicará el final de la serie.
- Algoritmo:
  1. Inicializar contador de números C y suma S a cero
  2. Leer un número N
  3. Si el número N es positivo
     a. Sumar el número N a la suma S
     b. Incrementar en 1 el contador de números C
     c. Ir al paso 2
  4. Calcular el valor medio (S/C)
  5. Escribir el valor medio
  6. Fin

Fases en el desarrollo del Software

- Análisis
  - Comprensión del problema
- Diseño del programa
- Codificación del programa
- Prueba del programa
- Mantenimiento

Análisis

- Descripción clara y completa del problema, haciendo énfasis en los siguientes aspectos:
  - entradas, salidas, ejecución.
- Para ello se usan:
  - técnicas sencillas como la descripción verbal o escrita del problema.
  - O más complejas como son los lenguajes de especificación formal.

Diseño

- Es la elaboración de pasos concretos para resolver el problema.
- Objetivos:
  - Minimizar el coste, tamaño y tiempo de ejecución.
  - Obtener la máxima facilidad de uso, fiabilidad, flexibilidad y sencillez de mantenimiento.

Otras fases

- Codificación: Consiste en formular el problema en algún lenguaje de programación.
- Prueba:
  - Puesta en funcionamiento.
  - Detección de errores de codificación.
  - Puede consumir más del 50% del tiempo total de desarrollo.
- Mantenimiento: Etapa destinada al desarrollo de mejoras, ampliaciones y refinamiento de las aplicaciones desarrolladas.

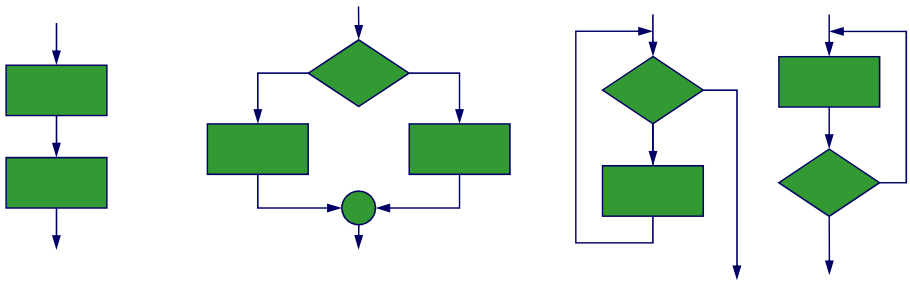
Programación Modular

- El programa se dividirá en módulos independientes, cada uno de los cuales ejecutará una determinada tarea.
- Existirá el llamado módulo principal que será el encargado de transferir el control a los demás módulos.
- Al ser independientes se pueden codificar y probar simultáneamente.
- Ventajas:
  - claridad
  - transportabilidad
  - modificabilidad
  - reducción del coste de desarrollo

Programación Estructurada

- Usa solamente tres estructuras de control: bloques consecutivos con un única entrada y una única salida.

Secuencial Selectiva Repetitiva

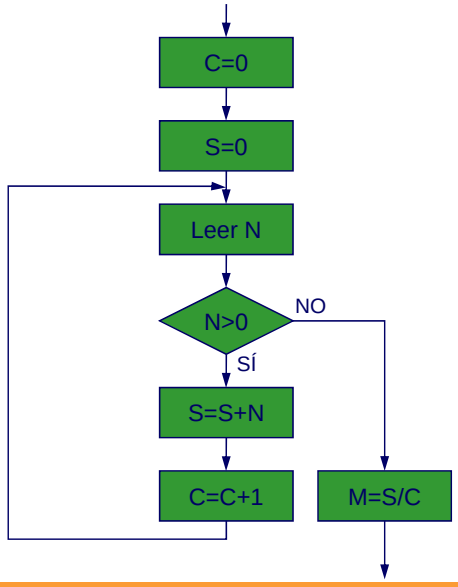


Programación Estructurada

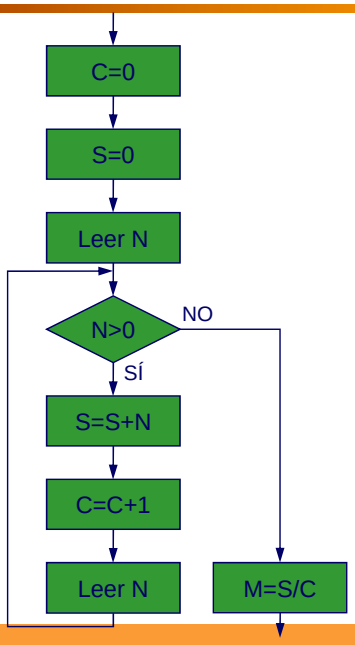
- La programación estructurada se debe usar conjuntamente con la programación modular.
- El empleo de técnicas de programación estructurada disminuye el tiempo de:
  - verificación
  - depuración
  - mantenimiento

  de las aplicaciones.

Ejemplo de la media



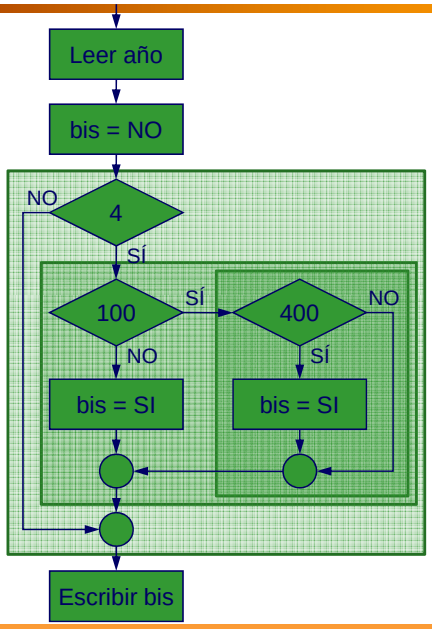
Ejemplo correcto de la media



Otro ejemplo: años bisiestos

- Especificaciones:
  - Un año es bisiesto si:
    - es múltiplo de 4 sin ser múltiplo de 100
    - es múltiplo de 400
- Algoritmo:
  1. Leer el año
  2. Suponemos que el año no es bisiesto
  3. Si el número es múltiplo de 4
     1. Si el número no es múltiplo de 100
        1. SÍ es bisiesto
     2. Si sí es múltiplo de 100:
        1. Si el número es múltiplo de 400
           1. SÍ es bisiesto
  4. Escribir si es bisiesto
  5. Fin

Años bisiestos: diagrama de flujo



ait

Introducción al Lenguaje C

Índice

- Introducción y evolución
- Características del lenguaje C.
- Palabras reservadas, identificadores y delimitadores.
- Estructura de un Programa en C.
- Ejemplo de un programa en C.

Introducción Histórica

- Lenguaje de programación de alto nivel
  - Desarrollado en los años setenta por Dennis Ritchie en Bell Telephone Laboratories, Inc.
- Originalmente para la programación de sistemas.
  - Se probó efectivo y flexible para cualquier tipo de aplicación.
- Primer gran programa escrito en C: UNIX.
- Tiene algunas características de bajo nivel:
  - acceso directo a memoria
  - permite mejorar la eficiencia.

Evolución del C

- Primeras implementaciones comerciales:
  - diferían en parte de la definición original.
  - Estandarizado por ANSI (comité ANSI X3J11):
    - Primera versión en 1989.
    - Versión actual de 1999
- En los 80 aparece C++:
  - adopta todas las características de C
  - incorpora nuevos fundamentos
  - base para la programación orientada a objetos

Características de C (I)

- Lenguaje estructurado de propósito general:
  - Características de lenguajes de alto nivel ...:
    - sentencias de control
    - tipos de datos.
    - palabras reservadas, etc..
  - ... y características adicionales de bajo nivel:
    - Manipulación de direcciones de memoria: punteros.
    - manipulación a nivel de bits.
    - sin restricciones (a veces cómodo, pero peligroso)

Características de C (II)

- Modular:
  - División de un programa en módulos
  - Se pueden compilar de forma independiente.
- Conciso:
  - Repertorio de instrucciones pequeño.
  - Gran número de operadores
  - Numerosas funciones de librería
- Compilado, no interpretado.

Identificadores

- Definición:
  - Nombres que permiten hacer referencia a los objetos (constantes, variables, funciones) que se usan en un programa.
- Reglas de formación:
  - Formado por caracteres alfanuméricos y '_'.
  - El primer carácter debe ser una letra
    - el subguión (_) está permitido pero no se debe usar.
  - Distinción entre mayúsculas y minúsculas:
    - `Cont` ≠ `cont` ≠ `CONT`

Palabras Reservadas

- Significado especial para el compilador.
- No se pueden usar como identificadores

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Delimitadores

- Son símbolos que permiten al compilador separar y reconocer las diferentes unidades sintácticas del lenguaje.
- Los principales delimitadores son:
  - `;` es necesario para finalizar sentencias o declaraciones.
  - `,` separa dos elementos consecutivos de una lista.
  - `()` enmarca una lista de parámetros.
  - `[]` enmarca la dimensión o el subíndice de una tabla.
  - `{}` enmarca un bloque de instrucciones o una lista de valores iniciales.

Estructura de un programa en C

```
<Órdenes de preprocesado>
<Declaración de prototipos de funciones>

int main ()
{
  <declaración de variables;>
  /* comentarios */
  <instrucciones ejecutables;>

  return 0;
}

<definición de otras funciones>
```

Comentarios

```
/* ... */
// ...
```

- Importantísimos en la realización de cualquier programa
- Dos formas de escribirlos:
  - Cuando ocupan más de una línea:
    ```
    /*esto es un comentario
      de más de una línea */
    ```
  - Cuando ocupan una sola línea:
    ```
    // Esto es otro comentario
    /* aunque también sirve así */
    ```
  - Pueden acompañar a sentencias en la misma línea:
    ```
    int contador; // Número de muestras para la media
    int suma;     /* Suma de las muestras */
    ```


Órdenes de Preprocesado

- Instrucciones especiales dentro de un programa, también llamadas directivas.
  - Se ejecutan antes del proceso de compilación.
- Las directivas más utilizadas son las de:
  - Declaración de importaciones
  - Definición de constantes

Declaración de Importaciones

```
#include <nombreFichero.h>
#include "nombreFichero.h"
```

- Incluye el contenido del archivo indicado:
  - Antes de la compilación.
- Utilidad:
  - Declarar funciones, constantes, … que pueden usarse en el programa:
    - De archivos de biblioteca:

```
#include <stdio.h>    // General de entrada/salida
#include <string.h>   // Manejo de cadenas de caracteres
#include <math.h>     // Matemáticas
```

    - De otros módulos del programa

```
#include "misFunciones.h"
```


Definición de Constantes

```
#define identificador cadena
```

- Sustituye el identificador por la cadena de caracteres:
  - En todos los lugares que aparezca en el programa.
  - Antes de la compilación.
  - Ejemplos:

```
#define MAXIMO        999
#define PI            3.1416
#define ERROR         "Se ha producido un error"
```

Sentencias

- Cada sentencia:
  - debe terminar en **;**
  - puede ocupar más de una línea.
- Puede haber varias sentencias en una línea:
  - No en esta asignatura: **¡Prohibido!**

Funciones

- Además de las funciones de biblioteca, el programador puede definir sus propias funciones que realicen determinadas tareas.
- El uso de funciones definidas por el programador permite dividir un programa grande en varios pequeños.
- Un programa en C se puede modular mediante el uso inteligente de las funciones.
- El uso de funciones evita la necesidad de repetir las mismas instrucciones de forma redundante.
- Una función se invoca al nombrarla, seguida de una lista de argumentos entre paréntesis.

Función `main`

- Indica el punto de comienzo de ejecución del programa.
- Hace las funciones de hilo conductor o “director de orquesta”
- Todo programa C debe contener **una y sólo una** función `main`.

Ejemplo de Programa en C

```
#include <stdio.h>
#define  FACTOR      2

int doble(int valor);

int main()
{
   int indice = 2;

   printf(“El doble de %d es %d\n”, indice, doble(indice));
   return 0;
}

int doble(int valor)
{
   return (valor * FACTOR);
}
```

Compilación y Prueba

- Editamos el fichero y lo guardamos con el nombre **`ejemplo.c`**
  - La extensión debe ser siempre:
    - **`.c`** para los ficheros fuente
    - **`.h`** para los ficheros de cabecera
  - Cualquier editor de texto
    - **NO** un procesador de texto (Word, WordPad, …)
- Compilamos el fichero:

```
$> gcc -c ejemplo.c
```

  - Se genera el fichero objeto **`ejemplo.o`**
- Enlazamos para obtener el ejecutable:

```
$> gcc -o ejemplo ejemplo.o
```

  - Se genera el fichero ejecutable **`ejemplo`**
- Ejecutamos el programa:

```
$> ./ejemplo
El doble de 2 es 4
```

Tema 3

Codificación de la Información

Índice

3.1. Introducción **3-1**
3.1.1. Los Sistemas Posicionales 3-1
3.2. Codificación de Enteros sin Signo **3-2**
3.2.1. La aritmética binaria 3-3
3.2.2. Otras Bases de Numeración 3-3
3.2.3. El Desbordamiento en los Enteros sin Signo 3-5
3.3. Codificación de los Enteros con Signo **3-6**
3.3.1. Codificación en Complemento a Dos 3-6
3.3.2. El Desbordamiento en los Enteros con Signo 3-7
3.4. Codificación de Números con Decimales **3-7**

3.1. Introducción

Utilizamos los ordenadores para procesar información de forma rápida y eficiente. Para ello es necesario que le proporcionemos los datos que queremos que procese, y que éste nos entregue los resultados obtenidos. Pero los ordenadores almacenan la información de forma diferente a la nuestra: mientras que nosotros manejamos conceptos como números enteros, números reales, alfabeto, colores, ... los ordenadores sólo entienden secuencias de unos y ceros.

Debemos encontrar pues un mecanismo que permita trasladar la información tal como nosostros la manejamos a como puede manejarla el ordenador, y viceversa. A este mecanismo se le denomina **codificación de la información**.

A la información codificada por un ordenador se le denomina **dato**. A las diferentes formas de codificar la información que entiende un lenguaje de programación se le denominan **tipos de datos**.

En este capítulo veremos cómo podemos codificar los números naturales (enteros positivos) y los números enteros.

3.1.1. Los Sistemas Posicionales

Los sistemas posicionales son aquellos en los que el valor de un dígito depende de su posición dentro de la secuencia de dígitos. Un dígito tendrá menos valor conforme más a la derecha esté, y más valor conforme más a la izquierda se encuentre.

Ejemplo: El sistema de numeración decimal que utilizamos habitualmente es un ejemplo de sistema posicional: un 3 en la posición de las unidades (la de más a la derecha) vale tres, mientras que el mismo 3 en la posición de las centenas (la tercera empezando por la derecha) vale trescientos.

La **base** de numeración, b, determina cuántos símbolos son necesarios para la representación de los números.

Ejemplo: El sistema de numeración decimal que utilizamos habitualmente es un ejemplo de sistema posicional: un 3 en la posición de las unidades (la de más a la derecha) vale tres, mientras que el mismo 3 en la posición de las centenas (la tercera empezando por la derecha) vale trescientos.

Ejemplo: En el caso de la base decimal son necesarios diez símbolos: los dígitos numéricos del 0 al 9.

Conocida la base b de un sistema posicional, un dígito d en la posición p vale $d \times b^p$, siendo $p = 0$ para el dígito de más a la derecha, $p = 1$ para el inmediato por la izquierda, ...

Ejemplo: $4532 = 4000 + 500 + 30 + 2 = 4 \times 10^3 + 5 \times 10^2 + 3 \times 10^1 + 2 \times 10^0$

El número de cantidades diferentes que pueden representarse con n dígitos vale b^n.

Ejemplo: Si disponemos de cuatro cifras en sistema decimal, podremos escribir 10000 números (10^4), desde el 0 hasta el 9999.

3.2. Codificación de Enteros sin Signo

Los números enteros sin signo se codifican utilizando la codificación binaria.

La codificación binaria es un **sistema posicional en base dos**. Necesita por tanto dos símbolos para la representación de cantidades, y se utilizan el 0 y el 1. En lo sucesivo, para indicar que un número está en binario, terminaremos dicho número con el subíndice 2.

Ejemplo: El número 10 está escrito en decimal, mientras que el número 10_2 está escrito en binario.

Como corresponde a un sistema posicional en base dos, el total de números distintos que pueden representarse con n cifras es 2^n, en el intervalo $[0, 2^n - 1]$.

Para obtener la codificación binaria de un número decimal tenemos que dividir sucesivamente por la base de numeración (base 2) y quedarnos con el último cociente y los restos de cada una de las divisiones.

Ejemplo: Para encontrar la representación en binario de 25, tendríamos que realizar las siguientes operaciones:

$$\begin{array}{r|l}
25 & 2 \\ \hline
05 & 12 \quad 2 \\ \hline
1 & 0 \quad 6 \quad 2 \\ \hline
 & \quad\;\; 0 \quad 3 \quad 2 \\ \hline
 & \quad\;\;\;\; 1 \quad 1
\end{array}$$

Por lo que la codificicación binaria de 25 será 11001_2.

Si lo que queremos es obtener el valor decimal correspondiente a un número binario, aplicamos la expresión correspondiente al sistema posicional.

Ejemplo: $10101_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 16 + 4 + 1 = 21$

3.2.1. La aritmética binaria

Al igual que para el sistema decimal, podemos establecer tablas aritméticas para el sistema binario. Las tablas son mucho más simples que las del sistema decimal, porque sólo disponemos de dos símbolos, el 0 y el 1, y no de diez como en el caso decimal.

Las tablas de sumar y multiplicar en binario serían:

+	0	1
0	0	1
1	1	10

×	0	1
0	0	0
1	0	1

3.2.2. Otras Bases de Numeración

Aunque los ordenadores utilizan únicamente dos símbolos para codificar la información, hay dos bases de numeración que son ampliamente utilizadas cuando hablamos de lenguajes de programación, que son la octal (base 8) y la hexadecimal (base 16). La principal utilidad de estas dos bases es que es mucho más simple pasar de ellas a decimal (y viceversa) de lo que sucedía en el caso binario, pero se sigue teniendo una imagen en binario de la información.

El Sistema de Numeración Octal

El sistema de numeración octal es un sistema posicional en base 8. Utiliza los símbolos del 0 al 7 para su representación. Para indicar que un número está escrito en octal, lo terminaremos con el subíndice 8.

Ejemplo: El número 732 está escrito en decimal, mientras que el número 732_8 está escrito en octal.

Por tratarse de un sistema posicional, pasar de decimal a octal consistirá en dividir sucesivamente por 8 (que es la base) y quedarnos con el último cociente y los sucesivos restos; y pasar de octal a decimal, en aplicar la expresión correspondiente a los sistemas posicionales.

Ejemplo: Para obtener la representación octal de 1512, haríamos:

$$\begin{array}{r|l}
1512 & 8 \\ \cline{2-2}
71 & 189 \;\; \underline{|\;8} \\ \cline{3-3}
72 & 29 \;\; \underline{23 \;|\; 8} \\ \cline{4-4}
0 & 5 \;\; 7 \;\; 2
\end{array}$$

por lo que la representación octal de 1512 es 2750_8. Para encontrar la representación decimal de 2750_8, haremos:

$$2750_8 = 2 \times 8^3 + 7 \times 8^2 + 5 \times 8^1 = 1024 + 448 + 40 = 1512$$

Una vez que tenemos la representación octal de un número, pasarlo a binario es inmediato, puesto que cada cifra octal se corresponde con exactamente tres cifras binarias. Es decir, podremos realizar el cambio de base sin realizar ninguna operación, ni de multiplicación ni de división. La siguiente tabla recoge las conversiones a realizar entre octal y binario:

Octal	Binario	Octal	Binario
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

Ejemplo: Si queremos obtener la representación binaria de 1512, una vez que sabemos que su representación octal es 2750_8, es muy simple: basta con sustituir cada símbolo octal por la secuencia correspondiente de tres símbolos binarios, según la tabla anterior:

$$\begin{array}{cccc} 2 & 7 & 5 & 0 \\ 0\ 1\ 0 & 1\ 1\ 1 & 1\ 0\ 1 & 0\ 0\ 0 \end{array}$$

Si hubiésemos realizado la conversión de decimal a binario directamente, habríamos necesitado 10 divisiones, en lugar de las tres que necesitamos pasando previamente por octal.

Si lo que queremos es obtener la representación decimal de un número binario, también podemos utilizar el octal como paso intermedio, reduciendo a un tercio el número de operaciones necesarias. Para ello habrá que agrupar de tres en tres los bits, comenzando por la derecha, y añadiendo ceros a la izquierda si fuera necesario. A continuación, sustituiremos cada grupo de tres bits por el dígito octal correspondiente. Para finalizar, sólo tendremos que pasar a decimal el número octal obtenido.

Ejemplo: Si queremos obtener la representación decimal de 10111101000_2, realizaremos los siguientes pasos:

- separamos en grupos de tres, comenzando por la derecha, y añadiendo un cero a la izquierda:

$$0\ 1\ 0 \quad 1\ 1\ 1 \quad 1\ 0\ 1 \quad 0\ 0\ 0$$

- sustituimos cada grupo de tres bits por el dígito octal correspondiente:

$$2 \quad 7 \quad 5 \quad 0$$

- obtenemos el valor decimal correspondiente:

$$2750_8 = 2 \times 8^3 + 7 \times 8^2 + 5 \times 8^1 = 1024 + 448 + 40 = 1512$$

El Sistema de Numeración Hexadecimal

El sistema de numeración hexadecimal es un sistema posicional en base 16. Necesita 16 símbolos, que son del 0 al 9 y de la A a la F (mayúsculas o minúsculas indistintamente). Para indicar que un número está escrito en hexadecimal, lo terminaremos añadiéndole el subíndice 16.

Ejemplo: El número 9723 está escrito en decimal, mientras que el número 9723_{16} está escrito en hexadecimal.

Al igual que sucede para el octal, la ventaja del sistema de numeración hexadecimal estriba en que permite obtener de forma directa la representación binaria de un número.

La única diferencia con el sistema octal visto anteriormente (aparte de los símbolos utilizados) es que cada símbolo hexadecimal se corresponde con cuatro símbolos binarios, según la siguiente tabla:

Hexadecimal	Decimal	Binario	Hexadecimal	Decimal	Binario
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

Podemos utilizar la base hexadecimal como paso intermedio entre las bases decimal y binaria:

- en el sentido de decimal a binario, dividiremos sucesivamente por 16, y obtendremos la representación hexadecimal del número. A continuación sustituiremos cada dígito hexadecimal por la secuencia de cuatro bits correspondiente.

- en el sentido de binario a decimal, separaremos los dígitos binarios en grupos de cuatro, comenzando por la derecha, y añadiendo ceros a la izquierda si fuera necesario. Sustituiremos cada grupo de cuatro bits por el símbolo hexadecimal correspondiente, y por último aplicaremos la expresión general del sistema posicional.

Ejemplo: Para obtener la representación hexadecimal de 1512, haríamos:

$$\begin{array}{r|l|l} 1512 & 16 & \\ \cline{2-2} 072 & 94 & 16 \\ \cline{3-3} 08 & 14 & 5 \end{array}$$

por lo que la representación hexadecimal de 1512 es $5E8_{16}$. Si ahora sustituimos cada dígito hexadecimal por su secuencia de cuatros bits, tenemos:

$$\begin{array}{ccc} 5 & E & 8 \\ 0\ 1\ 0\ 1 & 1\ 1\ 1\ 0 & 1\ 0\ 0\ 0 \end{array}$$

Si lo que queremos es la representación decimal de 1110101010_2, primero agrupamos en grupos de cuatro, añadiendo ceros a la izquierda, y sustituimos cada grupo por el dígito hexadecimal correspondiente:

$$\begin{array}{ccc} 0\ 0\ 1\ 1 & 1\ 0\ 1\ 0 & 1\ 0\ 1\ 0 \\ 3 & A & A \end{array}$$

Por último, la expresión del sistema posicional:

$$3AA_{16} = 3 \times 16^2 + 10 \times 16 + 10 = 768 + 160 + 10 = 938$$

Hay otra ventaja inherente a los sistemas de numeración octal y hexadecimal: es posible conocer el valor de un bit determinado sin necesidad de hacer la conversión a binario. Simplemente debemos saber qué digito hexadecimal incorpora el bit que interesa, y descodificar dicho dígito.

Ejemplo: Si queremos saber el valor del séptimo bit (empezando por la derecha, que sería el menos significativo) de $7FE542E_{16}$, sólo tenemos que saber que, puesto que cada dígito hexadecimal equivale a cuatro bits, el bit que interesa estaría integrado en el segundo (empezando por la derecha) dígito hexadecimal, 2. Sólo tendríamos que obtener la correspondencia en binario de dicho dígito, 0010, y puesto que el séptimo bit sería el tercero empezando por la derecha, el valor deseado es 0.

3.2.3. El Desbordamiento en los Enteros sin Signo

El número de dígitos que utilicemos para almacenar un entero sin signo determina el mayor valor que se puede almacenar en él; y esto puede ser una fuente de errores a la hora de programar. Lo vamos a ver con un ejemplo.

Supongamos un dato que utiliza ocho bits. Como hemos visto anteriormente, el número de valores diferentes que podrá almacenar será $2^8 = 256$, desde 0 a 255. Supongamos que el valor actual es 255 (11111111_2). ¿Qué sucede si a ese número le añadimos 1?. Utilizando la aritmética binaria, el resultado será:

$$\begin{array}{r} 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ 1\ + \\ \hline 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

pero como el dato sólo usa 8 bits para su almacenamiento, el bit más significativo (el de más a la izquierda) se pierde y nos quedamos con los últimos ocho dígitos del número binario, es decir, 00000000_2. Por lo tanto, el resultado que hemos obtenido de sumarle 1 a 255 ha sido 0, lo cual, evidentemente, es un error. A esto se le conoce como desbordamiento.

Para evitarlo, es indispensable que conozcamos qué rango de valores puede tomar un determinado dato, y seleccionar siempre un tipo de dato que tenga capacidad suficiente.

Ejemplo: Si queremos representar el mes del año (del 1 al 12), evidentemente tendremos suficiente con cuatro bits ($2^4 = 16 > 12$). Sin embargo, si queremos representar el número de páginas de un libro necesitaremos al menos 11 bits ($2^{11} = 2048$).

3.3. Codificación de los Enteros con Signo

Para codificar números enteros con signo se utiliza la codificación de complemento a 2. Esta codificación presenta ventajas sobre otras posibles codificaciones de números con signo, entre las que podemos destacar:

- aprovecha al máximo el rango de números que se pueden representar con un determinado número de dígitos.
- restar dos números equivale a sumar un número y el complemento a dos del otro, simplificando la estructura de la unidad aritmético-lógica.

3.3.1. Codificación en Complemento a Dos

Para codificar un número en complemento a dos se realizan las siguientes operaciones:

1. se toma el valor absoluto del número (el número sin signo).
2. se codifica en binario dicho valor absoluto.
3. si el número era positivo, ésa es la codificación del número.
4. si el número era negativo, se invierten todos los bits, y se le suma 1.

Esta codificación permite representar, con n dígitos binarios, un total de 2^n números distintos, en el intervalo $[-2^{n-1}, 2^{n-1} - 1]$. Y con una particularidad: el bit más significativo (el de más a la izquierda) representa el signo del número, de forma que si este bit vale 1 el número es negativo, y en caso contrario es positivo (si consideramos que el cero es un valor positivo).

Ejemplo: La representación en complemento a dos de -13 utilizando un octeto, sería:

- obtención del valor absoluto: 13
- codificación en binario: 00001101_2
- inversión de los bits: 11110010_2
- suma de una unidad: 11110011_2

Para obtener la representación decimal de un número dado codificado en complemento a dos, el proceso a realizar es muy similar:

1. se observa el bit más significativo para comprobar si el número es positivo (bit igual a cero) o negativo (bit igual a uno). Si el número es positivo, se pasa al punto 4. Si es negativo, se sigue con el 2.
2. se invierten todos los bits.
3. se le suma uno al número.
4. se convierte a decimal la representación binaria obtenida.
5. se le pone el signo adecuado.

Ejemplo: Para obtener la representación decimal de 11001110_2, sabiendo que el octeto representa un número entero con signo, haremos:

1. Observamos el bit más significativo: $\mathbf{1}1001110_2$. Como es un 1, el número es negativo.
2. Invertimos todos los bits: 0 0 1 1 0 0 0 1
3. Le sumamos una unidad:
$$\begin{array}{r} 0\ 0\ 1\ 1\ 0\ 0\ 0\ 1 \\ 1\ + \\ \hline 0\ 0\ 1\ 1\ 0\ 0\ 1\ 0 \end{array}$$
4. Obtenemos el número decimal cuya representación binaria es 00110010_2:
$00110010_2 = 2^5 + 2^4 + 2^1 = 32 + 16 + 2 = 50$
5. Le colocamos el signo adecuado: como el número era negativo, el resultado será -50.

3.3.2. El Desbordamiento en los Enteros con Signo

Al igual que sucede en los enteros sin signo, los enteros con signo también pueden verse afectados por el problema del desbordamiento. Supongamos un dato de ocho dígitos, y con un valor almacenado de 127, que codificado en complemento a dos quedaría como 01111111_2. Si a este valor le sumamos una unidad, obtendríamos:

$$\begin{array}{r} 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ 1\ + \\ \hline 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

Si ahora interpretamos el resultado, como el bit más significativo es un 1, consideramos que el número es negativo, por lo que para descodificarlo tendremos que invertir todos los bits:

$$0\ 1\ 1\ 1\ 1\ 1\ 1\ 1$$

y sumarle 1:

$$\begin{array}{r} 0\ 1\ 1\ 1\ 1\ 1\ 1\ 1 \\ 1\ + \\ \hline 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0 \end{array}$$

que equivale a -128. Es decir, el resultado de sumarle 1 a 127 ha sido -128.

3.4. Codificación de Números con Decimales

Los números con decimales en C se codifican en mantisa fraccionaria normalizada. Dicha codificación es muy parecida a la notación científica de las calculadoras: se utiliza un número con un único número distinto de cero a la izquierda de la coma (que se denomina **mantisa**), seguido de una potencia de diez (que se denomina **exponente**.

Ejemplo: En notación científica, 175,8376 lo pondríamos como $1,758376 \times 10^2$, y 0,00000012 como $1,2 \times 10^{-7}$. En el primer caso, la mantisa sería $1,758376$ y el exponente 2, mientras que en el segundo caso la mantisa sería $1,2$ y el exponente -7.

Las diferencias básicas con esta representación son las siguientes:

- el signo se codifica mediante un bit: 0 para números positivos y 1 para negativos.
- la mantisa se codifica en binario en valor absoluto, y de tal forma que el primer dígito distinto de cero (que necesariamente será un 1) esté inmediatemente a la izquierda de la coma decimal.

- el exponente lo es para una potencia de 2, y también se codifica en binario.

Ejemplo: Para codificar 175,8376 en binario, primero lo pasamos a binario, $10101111,110101_2$, y después desplazamos la coma las posiciones necesarias para que sólo quede un dígito a la izquierda de la coma: $1,0101111110101_2 \times 2^7$.

Se codificaría por una parte el signo (un 0 por ser un número positivo), por otro lado la mantisa, $1,0101111110101_2$, y por otro lado el exponente, 7.

Al contrario de lo que sucede para los números enteros, la codificación de los números con decimales no es exacta en la mayoría de las ocasiones, por lo que es importante no perder de vista esta circunstancia a la hora de utilizarlos.

Ejemplo: Si se suman los valores 0,333333, 0,333333 y 0,333334 con una calculadora que sólo tenga cuatro cifras significativas, el resultado será:

$$0,3333 + 0,3333 + 0,3333 = 0,9999$$

que evidentemente no es el exacto.

La precisión de los números con decimales viene determinada por el tamaño de la mantisa: a mayor tamaño de mantisa, mayor precisión y viceversa.

Tema 4

Tipos, Constantes y Variables

Índice

4.1. Tipos de Datos **4-1**
4.1.1. Los Tipos de Datos en C 4-2
4.1.2. Tipos Enteros sin Signo 4-2
4.1.3. Tipos Enteros con Signo 4-2
4.1.4. Tipos con Decimales 4-3
4.1.5. Tipo Lógico 4-3
4.1.6. Tipo Carácter 4-4
4.2. Constantes **4-4**
4.2.1. Constantes enteras 4-5
4.2.2. Constantes en coma flotante 4-5
4.2.3. Constantes de carácter 4-6
4.2.4. Cadenas de caracteres 4-6
4.3. Variables **4-7**
4.3.1. Declaración de Variables 4-7
4.3.2. Utilización del Valor de una Variable 4-8
4.3.3. Asignación de Valores a Variables 4-8
4.3.4. Tiempo de Vida de las Variables 4-9
4.3.5. Visibilidad de las variables 4-10

4.1. Tipos de Datos

Cada lenguaje de programación incorpora una serie de tipos de datos básicos, que son los que entiende directamente. Cada tipo de dato permite determinar los siguientes aspectos:

- el tamaño que dicho tipo va a ocupar en la memoria del ordenador.
- el rango de valores que puede almacenar dicho tipo.
- la forma en que se almacenan en memoria los diferentes valores.
- las operaciones que pueden realizarse con él.

En este tema nos centraremos únicamente en los tres primeros apartados, dejando el último para un tema posterior.

4.1.1. Los Tipos de Datos en C

En C hay tres tipos de datos básicos:

1. los datos de tipo numérico. Dentro de este tipo se distinguen tres subtipos:
   a) los enteros sin signo: para representar números enteros que no pueden ser negativos.
   b) los enteros con signo: para representar números enteros que pueden ser tanto positivos como negativos.
   c) los decimales: para representar cantidades que pueden tener decimales.
2. los datos de tipo lógico
3. los datos de tipo carácter

4.1.2. Tipos Enteros sin Signo

Para representar un entero sin signo en C se utiliza la palabra reservada **`unsigned`**. El estándar define varios tipos básicos de enteros sin signo:

- **`unsigned char`**
- **`unsigned short int`** o **`unsigned short`**
- **`unsigned int`** o **`unsigned`**
- **`unsigned long int`** o **`unsigned long`**
- **`unsigned long long int`** o **`unsigned long long`**

Todos los enteros sin signo se codifican en binario.

Los tipos anteriores sólo se diferencian en la cantidad de memoria utilizada para su almacenamiento (y por lo tanto en el rango de números que pueden representar). Su tamaño exacto no está definido en el estándar, y puede variar de una máquina a otra (y en la misma máquina, de un compilador a otro). Lo que sí se garantiza es que el tamaño utilizado por cada uno de los tipos es al menos igual que el tipo que le precede (es decir, un **`unsigned int`** puede ser del mismo tamaño que un **`unsigned short`**, pero nunca será de menor tamaño).

Ejemplo: En la máquina en la que se han redactado estos apuntes (un Pentium IV), y para la versión 3.4.1 del compilador gcc, los tamaños utilizados por cada uno de los tipos anteriores es:

- 1 octeto para el tipo **`unsigned char`**.
- 2 octetos para el tipo **`unsigned short int`**.
- 4 octetos para el tipo **`unsigned int`**.
- 4 octetos para el tipo **`unsigned long int`**.
- 8 octetos para el tipo **`unsigned long long int`**.

4.1.3. Tipos Enteros con Signo

Para cada uno de los enteros sin signo enumerados en el apartado anterior, el estándar define un correspondiente entero con signo. Se obtienen eliminando la palabra reservada **`unsigned`** de los tipos anteriores, con la excepción del tipo **`char`**, en el que hay que sustituir dicha palabra reservada por la palabra reservada **`signed`**:

- **signed char**
- **short int** o simplemente **short**
- **int**
- **long int** o simplemente **long**
- **long long int** o simplemente **long long**

Todos los enteros con signo se codifican en complemento a dos.

La diferencia entre los diferentes tipos radica, como en el caso de los enteros sin signo, en la cantidad de memoria utilizada para su almacenamiento (y por lo tanto en el rango de números que pueden representar). Su tamaño exacto no está definido en el estándar, y puede variar de una máquina a otra (y en la misma máquina, de un compilador a otro). Lo que sí se garantiza es que el tamaño utilizado por cada uno de los tipos es al menos igual que el tipo que le precede (es decir, un **int** puede ser del mismo tamaño que un **short**, pero nunca será de menor tamaño).

Ejemplo: En la máquina en la que se han redactado estos apuntes (un Pentium IV), y para la versión 3.4.1 del compilador gcc, los tamaños utilizados por cada uno de los tipos anteriores es:

- 1 octeto para el tipo **signed char**.
- 2 octetos para el tipo **short int**.
- 4 octetos para el tipo **int**.
- 4 octetos para el tipo **long int**.
- 8 octetos para el tipo **long long int**.

4.1.4. Tipos con Decimales

En C hay tres tipos para representar los números reales:

- **float**
- **double**
- **long double**

Todos los tipos con decimales se codifican en mantisa fraccionaria normalizada.

El estándar tampoco define el tamaño utilizado para el almacenamiento de estos tipos de datos, aunque sí especifica que un **double** tendrá al menos el mismo tamaño que un **float**, y un **long double** al menos el mismo tamaño que un **double**.

Ejemplo: En la máquina en la que se han redactado estos apuntes (un Pentium IV), y para la versión 3.4.1 del compilador gcc, los tamaños utilizados por cada uno de los tipos anteriores es:

- 4 octetos para un **float**.
- 8 octetos para un **double**.
- 12 octetos para un **long double**.

4.1.5. Tipo Lógico

Aunque desde el estándar del 99 existe un tipo lógico en C, que se designa con la palabra reservada **_Bool**, habitualmente se utilizan los enteros (con y sin signo) para representar datos booleanos (que sólo tienen dos valores: VERDADERO y FALSO), utilizando el siguiente criterio:

- Se considera que el dato es VERDADERO si tiene un valor distinto de cero (sea positivo o negativo).
- Se considera que el dato es FALSO si tiene un valor igual a cero.

4.1.6. Tipo Carácter

Se utilizan los datos de tipo carácter cuando se desea almacenar un carácter perteneciente al conjunto de caracteres básico de ejecución (conjunto finito y ordenado de caracteres que el ordenador es capaz de reconocer).

En C el tipo carácter se representa con la palabra reservada **`char`**.

Para codificar los diferentes caracteres, al ser estos un conjunto finito y bien definido, se utilizan tablas de correspondencia, donde a cada carácter se le asigna una determinada secuencia de bits que lo representa.

Aunque no está normalizado, en la mayoría de los casos se utiliza la codificación según el estándar ASCII, pudiendo reconocerse los siguientes subconjuntos:

- Caracteres alfabéticos: **`a b c ... z A B C ... Z`**
- Caracteres numéricos: **`0 1 2 ... 9`**
- Caracteres no imprimibles: espacio, tabulador, cambio de línea ...
- Caracteres especiales: **`+ - * / ^ . , ; < > $ ?`**

La siguiente tabla recoge la codificación ASCII. Nótese que en dicha tabla no aparecen algunos símbolos del español, como las vocales acentuadas o la 'ñ'. Eso significa que los códigos de dichos caracteres no están definidos en dicha codificación, y pueden visualizarse bien en algunas máquinas, y mal en otras.

Bits 3-0	Bits 6-4							
	000	001	010	011	100	101	110	111
0000	NULL	DLE		0	@	P	`	p
0001	SOH	DC1	!	1	A	Q	a	q
0010	STX	DC2	"	2	B	R	b	r
0011	ETX	DC3	#	3	C	S	c	s
0100	EOT	DC4	$	4	D	T	d	t
0101	ENQ	NAK	%	5	E	U	e	u
0110	ACK	SYN	&	6	F	V	f	v
0111	BEL	ETB	'	7	G	W	g	w
1000	BS	CAN	(	8	H	X	h	x
1001	HT	EM	)	9	I	Y	i	y
1010	LF	SUB	*	:	J	Z	j	z
1011	VT	ESC	+	;	K	[	k	{
1100	FF	FS	,	<	L	\	l	|
1101	CR	GS	-	=	M	]	m	}
1110	SO	RS	.	>	N	^	n	~
1111	SI	US	/	?	O	_	o	DEL

El tamaño del tipo carácter suele ser de un octeto, aunque eso dependerá del conjunto de caracteres que componen el alfabeto de ejecución.

4.2. Constantes

Decimos que son **constantes** aquellos datos que no son susceptibles de ser cambiados durante la ejecución de un programa.

En C existen cuatro tipos básicos de constantes:

- constantes enteras

- constantes en coma flotante
- constantes de carácter
- cadena de caracteres

4.2.1. Constantes enteras

Las constantes enteras se pueden escribir en tres sistemas numéricos diferentes: decimal (base 10), octal (base 8) y hexadecimal (base 16).

Una constante entera decimal es aquella que comienza con un dígito decimal distinto de cero al que sigue cualquier secuencia de dígitos decimales.

Ejemplo: Son constantes decimales **`1`**, **`743`** ó **`32767`**.

Una constante entera octal es aquella que comienza por un 0, y va seguida de cualquier secuencia de dígitos octales (tomados del conjuntos del 0 al 7).

Ejemplo: Son constantes octales **`0`**, **`0102`** ó **`0777`**.

Una constante entera hexadecimal es aquella que comienza por 0x ó 0X, y va seguida de cualquier secuencia de dígitos hexadecimales (tomados del conjunto de dígitos del **`0`** al **`9`**, y caracteres de la **`a`** a la **`f`** o de la **`A`** a la **`F`**).

Ejemplo: Son constantes hexadecimales **`0x0`**, **`0X12ef`** ó **`0xAAaA`**.

Las constantes enteras pertenecen, por defecto, al tipo **`int`**. Si se desea modificar este aspecto, se les puede añadir un sufijo con el siguiente significado:

- Si incorpora una **`U`** (o **`u`**), indica que se corresponde con un **`unsigned int`**.
- Si incorpora una **`L`** (o **`l`**), indica que se corresponde con un **`long int`**.
- Si incorpora una **`LL`** (o **`ll`**), indica que se corresponde con un **`long long int`**.

El sufijo **`U`** puede combinarse con el sufijo **`L`**, dando lugar a constantes de tipo **`unsigned long int`** (sufijo **`UL`** o **`LU`**), o con el sufijo **`LL`**, dando lugar a constantes de tipo **`unsigned long long int`** (**`ULL`** o **`LLU`**).

Ejemplo: Serían constantes enteras válidas **`0x12efUL`** (constante hexadecimal del tipo **`unsigned long int`**), **`1234l`** (constante decimal de tipo **`long int`**), **`0177lU`** (constante octal de tipo **`unsigned long int`**) ó **`1235LLu`** (constante decimal de tipo **`unsigned long long int`**).

Los valores máximos de las constantes vienen determinados por el valor máximo correspondiente al tipo de dato al que representan.

4.2.2. Constantes en coma flotante

Una constante en coma flotante es un número decimal que contiene un punto decimal o un exponente (o ambos). Para separar la mantisa del exponente se utiliza el símbolo **`e`** (o **`E`**). El exponente deberá ser un número entero (positivo o negativo).

Ejemplo: Según la definición, serían constantes en coma flotante **`32.`**, **`.23`**, **`32e7`** ó **`23.5E-12`**. No lo serían, en cambio **`1`** (puesto que no tiene punto decimal ni exponente), **`1,000.0`** (puesto que incorpora una coma, que es un carácter ilegal) ni **`2E+10.2`** (puesto que el exponente no es un entero).

Las constantes en coma flotante son, por defecto, de tipo **`double`**. Si se desea modificar el tipo, se le puede añadir el sufijo **`f`** (o **`F`**) que la convierte en **`float`**, o el sufijo **`l`** (o **`L`**) que la convierte en **`long double`**.

Ejemplo: Serían constantes en coma flotante **32e7L** ó **.5E-2f**

Los valores máximos de las constantes de coma flotante vienen determinados por el máximo valor del tipo de dato que representan.

4.2.3. Constantes de carácter

Una constante de carácter es un sólo carácter, encerrado con comillas simples. Una constante de tipo carácter almacena el valor entero que representa el carácter indicado según el código de representación utilizado. Esto permite escribir programas en C que no dependan del código utilizado por la máquina.

Ejemplo: Son constantes de tipo carácter **'A'**, **'X'**, **'3'** o **'/'**

Algunos caracteres no pueden representarse de esa manera, y por eso necesitan utilizar un mecanismo alternativo. Estos caracteres se representan mediante secuencias de escape: códigos especiales de dos caracteres, el primero de los cuales siempre es la barra invertida (****). Algunas de estas secuencias de escape son:

'\'' representa el carácter comilla simple.

'\\' representa el carácter barra invertida.

'\"' representa el carácter comilla doble (aunque también valdría **'"'**).

'\?' representa el carácter cierre de interrogación (también vale **'?'**).

'\t' representa el carácter tabulador horizontal.

'\n' representa el carácter nueva línea.

'\r' representa el carácter retorno de carro.

Ejemplo: No serían constantes de carácter válidas **'as'** ni **'\\a'**.

También es posible especificar un carácter mediante su código en octal o hexadecimal:

'\ooo' representa el carácter cuyo código en octal es **ooo**. Serían secuencias válidas **'\0'**, **'\50'** o **'\126'**

'\xhh' representa el carácter cuyo código en hexadecimal es **hh**. Serían secuencias válidas **'\x9'** o **'\x50'**

No obstante, la utilización de valores octales o hexadecimales para representar caracteres presupone el conocimiento del código utilizado, y limita la utilización del programa realizado a máquinas que utilicen dicho código. Por eso **se desaconseja** la utilización de este mecanismo para la especificación de constantes de carácter.

4.2.4. Cadenas de caracteres

Una cadena de caracteres es una secuencia de cero o más caracteres encerrados entre comillas dobles. Las comillas dobles no forman parte de la cadena, sólo sirven para delimitarla.

Ejemplo: Son constantes de cadena:

- **"cadena de caracteres"**
- **"Fundamentos de la Programación\n"**
- **"El carácter barra invertida es \\"**
- **"Las comillas dobles hay que escaparlas: \""**

La representación interna de una cadena tiene un carácter nulo (**'\0'**) al final, de modo que el almacenamiento físico es uno más del número de caracteres escritos en la cadena.

Es importante destacar que no son equivalentes **'D'** y **"D"**. En el primer caso estamos representando el carácter d mayúscula, mientras que en el segundo caso estamos representando una cadena de caracteres de un solo carácter (el carácter d mayúscula). Un carácter tiene un valor entero equivalente, mientras que una cadena de un solo carácter no tiene un valor entero equivalente y de hecho consta de dos caracteres, que son el carácter especificado y el carácter nulo (**'\0'**).

4.3. Variables

Las variables se utilizan para almacenar valores que pueden cambiar en el transcurso de la ejecución del programa. Sin embargo solamente pueden tomar valores de un tipo: el tipo al que pertenecen.

Ejemplo: Si una variable es de tipo entero sin signo, podrá contener valores enteros positivos, pero no negativos; si es de tipo entero con signo, podrá contener valores enteros (positivos o negativos), pero no números con decimales. Si es de tipo con decimales, podrá contener también valores con decimales.

Se designan mediante un nombre al que se llama **identificador**. El identificador de una variable puede estar formado por cualquier secuencia de caracteres alfabéticos (se excluyen las vocales acentuadas y la ñ) tanto en mayúsculas como en minúsculas, caracteres numéricos y el carácter especial _, pero comenzando siempre por un carácter alfabético.

Ejemplo: Serían identificadores válidos de variables **i**, **num_veces** o **paso2**. No serían identificadores válidos **2oPaso**, porque empieza por un número, **_recordar**, porque empieza por el carácter de subrayado, ni **año**, porque contiene el carácter ñ.

Es muy conveniente que el identificador utilizado para designar una variable tenga relación con aquello que representa (velocidad, resultado, suma, volumen, ...), para que al encontrarla dentro del código podamos tener una idea de qué representa dicha variable.

Ejemplo: Si en un programa hacemos referencia a la variable **i**, no sabremos exactamente a qué nos estamos refiriendo. Sin embargo, si la llamamos **diaSemana**, de un simple vistazo podemos entender para qué la estamos usando.

Existen tres operaciones básicas que pueden realizarse con una variable:

- declararla.
- utilizar el valor almacenado.
- asignarle un nuevo valor.

4.3.1. Declaración de Variables

Hemos visto anteriormente que dependiendo del tipo de dato que estemos utilizando, la ocupación en memoria y la forma de codificar la información es diferente.

Por eso, si en un punto del programa nos encontramos con la expresión **2 * temperatura**, de la variable **temperatura** no podríamos saber ni cómo está codificada la información que almacena ni cuánto ocupa en memoria dicha variable. Y no podremos saberlo porque tanto una cosa como otra dependen del tipo de dato, y con la información de la sentencia no podemos determinar a qué tipo de dato pertenece la variable **temperatura**.

Por eso en C, para poder utilizar una variable es necesario haberla declarado previamente. Declarar una variable es sencillamente dar cuenta de su existencia explícitamente en algún lugar del programa dedicado a tal efecto, especificando el tipo de dato al que pertenece.

Una declaración consta de un tipo de dato seguido de uno o más nombres de variables separados por comas, y finalizando con un punto y coma.

Ejemplo: Para declarar dos variables de tipo **`unsigned int`**, de nombres **`mes`** y **`anio`**, pondríamos:

```
unsigned int mes, anio;
```

A pesar de estar permitida la declaración de varias variables en una única sentencia (como en el ejemplo anterior), es aconsejable utilizar una sentencia distinta para declarar cada variable. El resultado es idéntico, y mucho más legible (y por tanto menos propenso a errores).

Ejemplo: Para declarar las mismas dos variables del ejemplo anterior, sería preferible escribir:

```
unsigned int mes;
unsigned int anio;
```

Ejemplo: A continuación siguen algunas declaraciones de variables:

- Para declarar una variable que contendrá el código del carácter leído desde el teclado:

```
char caracter_de_teclado;
```

- Para declarar una variable que almacene el número de sumandos de una suma:

```
unsigned int numSumandos;
```

- Para declarar una variable que va a almacenar la temperatura de una sala:

```
float temperatura;
```

Es importante destacar que la declaración de una variable tiene un doble efecto:

1. Por una parte, permite que en el resto del programa, cuando nos refiramos a ella por su identificador, podamos saber qué tipo de dato almacena dicha variable.
2. Por otro lado, permite reservar una zona de memoria, exclusiva y del tamaño adecuado, para dicha variable.

4.3.2. Utilización del Valor de una Variable

Para poder utilizar el valor almacenado en una variable en un momento determinado, basta con poner el nombre de la variable en el lugar en que necesitemos dicho valor.

Ejemplo: Si queremos obtener el doble de la temperatura de una sala, que está almacenada en la variable **`temperatura`**, no tendremos más que poner: **`2 * temperatura`**.

4.3.3. Asignación de Valores a Variables

Para asignar un valor a una variable tendremos que utilizar el operador de asignación (que se verá más adelante). En general, con dicho operador almacenaremos en una variable el resultado de evaluar una expresión.

Ejemplo: Para asignar a una variable el cuadrado de un determinado número, pondremos:

```
cuadrado = numero * numero;
```

Hay que tener en cuenta que si una variable se declara pero no se le asigna ningún valor tendrá en general un valor indeterminado, y por lo tanto los resultados obtenidos al utilizar su valor serán impredecibles:

Ejemplo: La siguiente porción de código no dará ningún error al generar el ejecutable, pero los resultados obtenidos serán indeterminados (podemos tener diferentes resultados si ejecutamos el código varias veces):

```
unsigned int numero;
unsigned int cuadrado;

cuadrado = numero * numero;
```

Por ello es muy importante que toda variable que vaya a ser usada en un programa tenga un valor previamente asignado. Como la única condición que exige el lenguaje C es que toda variable sea declarada antes de ser utilizada, la mejor forma de conseguir que toda variable tenga siempre un valor asignado es realizar esta asignación en la propia declaración de la variable.

Para realizar la asignación de un valor a una variable en su declaración, tendremos que poner el tipo de dato, el nombre de la variable, el operador de asignación, **=**, y una expresión que proporcione el valor inicial para dicha variable.

Ejemplo: En este ejemplo puede observarse como, desde el momento en que una variable ha sido declarada puede ser utilizada, incluso en la declaración de otra variable:

```
unsigned int numero = 2;
unsigned int cuadrado = numero * numero;
```

4.3.4. Tiempo de Vida de las Variables

El **tiempo de vida** de una variable determina cuándo se crea y se destruye. Con caracter general, una variable en C sólo existe durante la ejecución del bloque[1] en el que se ha definido.

Esto significa que la variable se crea al declararla dentro de un bloque, y se destruye al salir del bloque, lo que trae consigo una serie de consecuencias:

- la primera, y más importante, es que el valor de una variable en sucesivas ejecuciones del bloque es completamente independiente.
- la segunda, que el valor de inicialización especificado en la declaración de la variable se aplica en cada una de las ejecuciones del bloque. Si no existe inicialización en la declaración de la variable, el valor inicial es indeterminado; esto hace que sea muy aconsejable dar **siempre** un valor inicial a cada variable en su declaración.
- la tercera es que las modificaciones que hagamos sobre la variable dentro del bloque en la que está definidas quedan sin efectos una vez que finalice este.

A estas variables se les conoce como **variables automáticas** o **variables locales**, y son las que habitualmente utilizaremos en los programas.

[1] Recuerde que un bloque no es más que un conjunto de sentencias encerradas entre llaves (**{}**).

La Especificación `static`

Existe un procedimiento para evitar que las variables se creen y destruyan cada vez que se entra y se sale de un bloque: anteponer a la declaración de la variable la palabra reservada **`static`**. A las variables especificadas de esta forma se les denomina **variables estáticas**.

Esta especificación tiene un triple efecto sobre la variable:

- hace que las variables se creen al inicio de la ejecución del programa, y no al declararla dentro de un bloque, como sucede con las variables locales.
- hace que se inicialicen al comienzo de la ejecución del programa: al valor indicado en la declaración de la variable, o a cero si no se indica nada. Sólo pueden utilizarse expresiones constantes para la inicialización de variables estáticas.
- hace que la variable no se destruya hasta que no finalice el programa.

Como consecuencia de su comportamiento, estas variables conservan su valor de una ejecución a la siguiente, lo que constituye su principal utilidad.

Ejemplo: Supongamos que queremos contar el número de veces que se llama a una determinada función. Podremos hacerlo mediante el siguiente código:

```
int funcion(...)
{
  static int numLlamadas = 0;
  ...

  numLlamadas = numLlamadas + 1;
  ...

  return numLlamadas;
}
```

Cada vez que se ejecute la función se sumará una unidad al contador de llamadas (**`numLlamadas`**).

Sin embargo, si no utilizáramos la especificación **`static`**, la variable **`numLlamadas`** se crearía de nuevo cada vez que se ejecutara la función, por lo que el valor devuelto sería siempre **`1`**.

Variables Globales

Una excepción a lo enunciado anteriormente lo constituyen las denominadas **variable globales**. Dichas variables se declaran fuera de cualquier bloque y su comportamiento, desde el punto de vista del tiempo de vida, es similar al de las variables estáticas: se crean al inicio de la ejecución del programa, se inicializan una sola vez, y se destruyen a la finalización del programa. En esta asignatura, las variables globales no están permitidas.

4.3.5. Visibilidad de las variables

La **visibilidad** o **alcance** de una variable indica dónde es posible utilizar una variable.

Evidentemente, el primer requisito para utilizar una variable es que exista. Aplicando este concepto a las variables locales, se deduce inmediatamente que las variables locales sólo serán visibles (y por lo tanto utilizables) dentro del ámbito del bloque en el que están declaradas. Esto supone que dos variables automáticas declaradas en bloques distintos son totalmente independientes, aunque tengan el mismo identificador.

Pero la visibilidad también se ve afectada por los bloques anidados. Imaginemos el siguiente fragmento de código:

```
{
  // Comienzo de un bloque
  int num = 100;
  int i   = 0;
  ...
  num = num + 10;
  ...
    {
      // Comenzamos un nuevo bloque dentro del anterior
      int num = 0;
      ...
      num = num + 4;
    }
  num = num * 2;
  ...
}
```

Tenemos definida dos variables con el mismo nombre (**num**) en dos bloques diferentes, siendo uno de ellos interior al otro. En la primera utilización de **num** (**num = num + 10;**) no hay ninguna duda sobre a qué variable nos estamos refiriendo: sólo está definida la del bloque exterior, y por tanto será a esa a la que aplicaremos el operador.

En la segunda utilización de la variable **num** (**num = num + 4;**) ya estamos dentro del bloque más interno, y por tanto tendremos definidas las dos variables **num**. En este caso, siempre nos referimos a la variable definida en el bloque más interior, haciendo invisible la variable definida en el bloque más exterior.

En la tercera utilización de la variable **num** (**num = num * 2;**), la segunda variable **num** ya no existe, puesto que estamos fuera del bloque en el que se definió. Por tanto aquí tampoco existe ambigüedad: nos estamos refiriendo a la primera variable **num** definida.

Las Variables Estáticas

¿Pero qué sucede con las variables estáticas?. Hemos visto que dichas variables "viven" durante toda la ejecución del programa, pero esto no significa que puedan utilizarse en cualquier sitio del programa: su visibilidad también está restringida al bloque en el que se han definido (o al fichero en el que estén definidas, en caso de estar definidas fuera de cualquier bloque).

Las Variables Globales

Caso aparte constituyen las variables globales. Estas variables existen durante toda la vida de la aplicación, y además son visibles en cualquier sitio del programa, aunque para ello es necesario que se utilice una declaración especial en los ficheros en los que se quiere utilizar pero no se ha definido: se repite la definición de la variable (sin inicialización) anteponiéndole la palabra reservada **extern**.

Desde el punto de vista de la detección y corrección de errores, es muy conveniente que la visibilidad de las variables sea lo más reducida posible: podemos controlar el valor de dicha variable centrándonos en el contexto en el que es visible. Por esta razón es muy desaconsejable la utilización de variables globales en cualquier programa. Si además tenemos en cuenta que **en ningún caso** puede considerarse indispensable la utilización de variables globales en un programa, en el contexto de esta asignatura las variables globales **ESTÁN PROHIBIDAS**.

Tema 5

Expresiones

Índice

5.1. Introducción **5-1**
5.2. Operadores Aritméticos **5-2**
5.2.1. Operaciones con valores de tipo char 5-3
5.3. Operadores Relacionales **5-3**
5.3.1. Precisión de los Números con Decimales 5-4
5.4. Operadores Lógicos **5-4**
5.5. Operadores de Manejo de Bits **5-5**
5.6. Asignación **5-6**
5.7. Asignación Compacta **5-7**
5.8. Conversión Forzada **5-8**
5.9. Operadores de Autoincremento y Autodecremento **5-9**
5.9.1. Efectos secundarios de ++ y -- 5-10
5.10. Operador Condicional **5-11**
5.11. Operador `sizeof` **5-11**
5.12. Reglas de Prioridad **5-12**

5.1. Introducción

En cualquier programa es necesario procesar los datos de entrada para generar los datos de salida. El elemento básico del que dispone C para procesar los datos son las expresiones, cuyo significado es muy similar al utilizado en otras materias como el álgebra.

Las expresiones son combinaciones de operandos, operadores y paréntesis, que al evaluarse dan como resultado un valor.

Los operandos determinan los valores que se utilizan para realizar una operación determinada. Pueden ser:

- Constantes
- Variables
- Funciones (del estilo de las utilizadas en álgebra; se estudiarán más adelante).
- Otra expresión

Los operadores determinan las operaciones a realizar. Pueden ser:

- Aritméticos
- Relacionales
- Lógicos
- De manejo de bits
- De asignación
- De asignación compacta
- De autoincremento y autodecremento
- Condicional.

Los paréntesis se utilizan para definir en qué orden deben evaluarse las subexpresiones que aparecen en una expresión.

Ejemplo: Una posible expresión sería:

```
9.23 * (7.3 + cos(angulo/2))
```

donde:

- **9.23**, **7.3** y **2** serían constantes
- **angulo** sería una variable
- *****, **+** y **/** serían operadores
- **cos()** sería una función

5.2. Operadores Aritméticos

Toman como valores operandos numéricos y producen como resultado un valor numérico. Los operadores aritméticos definidos en C son los siguientes:

- Operadores unarios (que se aplican a un único operando):
  - **+**: no tiene efecto aparente.
  - **−**: convierte en positivos los valores negativos, y en negativos los positivos.
- Operadores binarios (que se aplican a dos operandos):
  - **+**: operador suma
  - **−**: operador resta
  - *****: operador multiplicación
  - **/**: operador división: devuelve el cociente de la división del operando de la izquierda por el de la derecha. Si ambos operandos son enteros, el operador es la división entera (despreciando el resto); en cualquier otro caso, la división es con decimales.
  - **%**: operador módulo; devuelve el resto resultante de dividir el operando de la izquierda por el de la derecha. Ambos operandos deben ser necesariamente valores de tipo entero. El resultado también es entero.

Los operadores aritméticos se evalúan de izquierda a derecha.

Cuando se realiza una operación aritmética, los tipos de los operandos no tienen por qué ser los mismos (exceptuando el operador `%`, que necesita que ambos sean de tipo entero). El tipo resultante de evaluar una expresión aritmética es el tipo del operador de mayor rango[1] en la expresión; es lo que se denomina promoción automática.

Ejemplo: Si operamos un **`long double`** con un **`float`**, este último se convierte internamente en **`long double`** antes de realizar la operación, y el resultado sería de tipo **`long double`**. Si operamos un **`float`** con un **`long int`**, este último se convierte internamente en **`float`** antes de realizar la operación, y el resultado es de tipo **`float`**.

5.2.1. Operaciones con valores de tipo char

No hay que olvidar que los valores de tipo **`char`** son también numéricos, y por tanto pueden aplicárseles los operadores aritméticos aquí descritos.

Sin embargo, hay dos operaciones que resultan de especial interés:

- Si restamos dos valores de tipo **`char`**, obtenemos como resultado un entero que determina la separación entre los códigos de los dos caracteres. Esto es especialmente útil para convertir un carácter en su correspondiente representación numérica: se obtiene restando al carácter que contiene el código del dígito en cuestión el carácter que contiene el código del dígito **`0`**:
  `'8' - '0' = 8`.
- Si a un valor de tipo **`char`** le sumamos o restamos un valor entero, obtendremos otro valor de tipo **`char`** que representa el carácter cuyo código se diferencia del primero en tantas unidades como le hayamos sumado o restado. Por ejemplo: **`'a' + 2 = 'c'`** y **`'c' - 2 = 'a'`**.

Cualquier otra operación aritmética que utilice valores de tipo char como operandos se trataría como si fuera una operación entre enteros.

5.3. Operadores Relacionales

Toman como operandos valores numéricos, y producen como resultado un valor entero. Los operadores de relación, todos binarios, son:

- **`<`**: devuelve 1 si el valor del operando de la izquierda es menor que el de la derecha, y 0 en otro caso.
- **`>`**: devuelve 1 si el valor del operando de la izquierda es mayor que el de la derecha, y 0 en otro caso.
- **`<=`**: devuelve 1 si el valor del operando de la izquierda no es mayor que el de la derecha, y 0 en otro caso.
- **`>=`**: devuelve 1 si el valor del operando de la derecha no es mayor que el de la izquierda, y 0 en otro caso.
- **`==`**: devuelve 1 si los valores de los operadores de izquierda y derecha son iguales, y 0 en otro caso.
- **`!=`**: devuelve 1 si los valores de los operadores de izquierda y derecha son distintos, y 0 en otro caso.

[1] Un número tiene mayor rango cuanto mayor es el máximo número que se puede representar con dicho tipo. Según eso, el tipo de mayor rango en C será el **`long double`**, seguido del **`double`**, seguido del **`float`**, seguido del **`unsigned long long int`**, ..., mientras que el de menor rango será el **`signed char`**.

Ejemplo: Si **i**, **j** y **k** son tres variables de tipo **int**, con valores **1**, **2** y **3** respectivamente, tendríamos:

Expresión	Resultado
`i < j`	1
`(i + j) >= k`	1
`(j + k) > (i + 5)`	0
`k != 3`	0
`j == 2`	1

Los operadores relacionales se evalúan de izquierda a derecha. La cual significa que, suponiendo que la variable **i** valga **5**, la expresión:

```
7 > i > 2
```

daría como resultado **0** (puesto que **7** es efectivamente mayor que **5**, y daría como resultado **1**, pero **1** no es mayor que **2**, por lo que el resultado final sería **0**). De la misma forma, la expresión:

```
2 <= i <= 4
```

daría como resultado **1** (puesto que **2** es menor que **5**, y daría como resultado **1**, y **1** es menor que **4**, por lo que el resultado final sería **1**).

Conviene recordar que las variables de tipo **char** también son tipos numéricos, y por tanto pueden utilizarse como operandos de las expresiones relacionales. En ese caso, las comparaciones se realizarán en función del código que representa al carácter en cuestión, y no del carácter mismo. Para el código ASCII (y en general para cualquier sistema de codificación de caracteres) se pueden establecer las siguientes relaciones:

- `'0' < '1' < '2' < ... < '9'`
- `'A' < 'B' < 'C' < ... < 'Z'`
- `'a' < 'b' < 'c' < ... < 'z'`

Pero no debe suponerse ninguna otra relación a la hora de codificar (como por ejemplo que `'0' < 'a'`, o que `'A' < 'a'`).

5.3.1. Precisión de los Números con Decimales

Cuando se utilizan los operadores de relación de igualdad o desigualdad con operandos con decimales (**float**, **double** o **long double**), hay que tener cuidado con la precisión de estos, puesto que puede ser que el resultado esperado no sea exactamente el obtenido.

Por ejemplo, si sumamos tres números decimales, **0.1**, **0.2**, **0,7**, el resultado de la suma puede ser **0.9999999999998**, que evidentemente no es **1**, de forma que la comparación con el operador de igualdad daría como resultado un **0**, puesto que la expresión es falsa.

5.4. Operadores Lógicos

Toman operandos numéricos, y producen como resultado un valor entero. Los operadores lógicos son:

- **!**: operador unario, que representa el **NO de la lógica proposicional; devuelve 0 si el operando es distinto de cero, y 1 si es igual a cero.**

- **`&&`: operador binario, que representa el Y de la lógica proposicional; devuelve 1 si los dos operandos son distintos de cero, y 0 en caso contrario. Si el operando de la izquierda vale cero, el operando de la derecha no se evalúa y el resultado de la operación es 0.**

- **`||`: operador binario, que representa el O de la lógica proposicional; devuelve 0 si los dos operandos son iguales a cero, y 1 en caso contrario. Si el operando de la izquierda vale distinto de cero, el operando de la derecha no se evalúa y el resultado de la operación es 1.**

Ejemplo: Si **`i`** es una variable de tipo **`int`** y valor **`7`**, **`f`** es una variable de tipo de **`float`** y valor **`5.5`**, y **`c`** es una variables de tipo **`char`** y valor **`'w'`**, tendríamos:

Expresión	Resultado
`(i >= 6) && (c == 'w')`	1
`!((i >= 6) && (c == 'w'))`	0
`(f < 11) && (i >= 100)`	0
`(c != 'p') || ((i + f) <= 10)`	1

Los operadores lógicos se evalúan de izquierda a derecha.

5.5. Operadores de Manejo de Bits

Sólo admiten operandos de tipo entero, y se aplican bit a bit. Los operadores de manejo de bit son:

- **`~`**: operador unario; invierte cada bit del operando; los bits que valían cero pasan a valer 1, y los que valían uno pasan a valer 0.

- **`&`**: operador binario; pone a 1 los bits si ambos son uno, y a 0 si alguno es igual a cero.

- **`|`**: operador binario; pone a 0 los bits si ambos son cero, y a 1 si alguno es igual a uno.

- **`^`**: operador binario; pone a 0 los bits si son iguales, y a 1 si son distintos.

- **`<<`**: operador binario; desplaza a la izquierda el operando de la izquierda tantas veces como especifique el operando de la derecha. Los bits que 'aparecen' por la derecha se rellenan con ceros.

- **`>>`**: operador binario; desplaza a la derecha el operando de la izquierda tantas veces como especifique el operando de la derecha. Los bits que 'aparecen' por la izquierda son indefinidos (pueden rellenarse con ceros o con unos).

Lo mejor para utilizar correctamente estos operadores es obtener la representación del número en binario, realizar las operaciones, y volver a codificar en decimal si se desea.

Ejemplo:
```
~126 = ~0111 1110 = 1000 0001 = 129
126 & 3 = 0111 1110 & 0000 0011 = 0000 0010 = 2
126 | 3 = 0111 1110 | 0000 0011 = 0111 1111 = 127
126 ^ 3 = 0111 1110 ^ 0000 0011 = 0111 1101 = 125
126 << 3 = 0111 1110 << 3 = 1111 0000 = 240
126 >> 3 = 0111 1110 >> 3 = xxx0 1111 = indeterminado
```

5.6. Asignación

El operador de asignación admite dos operandos; el de la izquierda debe ser un identificador de variable, mientras que el de la derecha puede ser cualquier expresión, del mismo tipo que la variable. Este operador, que en C se representa por el símbolo **=**, permite asignar un valor a una variable; el valor asignado será el resultado de la expresión.

Ejemplo: Serían expresiones válidas:
a = 7 + 5;
a = a + 3;
No sería una expresión de asignación válida
a + 1 = 5;
puesto que el operando de la izquierda es una expresión, y no una variable.

Como la asignación es en sí una expresión, también devuelve un valor: el asignado a la variable.

Ejemplo: Una sentencia válida sería:
b = (a = 7 + 5);
y tanto **a** como **b** tomarían el valor **12**.

Una particularidad de este operador es que se evalúa de derecha a izquierda, por lo que las asignaciones de más a la derecha se pueden aplicar a asignaciones de más a la izquierda.

Ejemplo: La expresión
a = b = c = d = 1;
es equivalente a:
a = (b = (c = (d = 1)));

Sin embargo, este tipo de asignaciones 'en cascada' suelen inducir a error y convierten el código en poco legible, por lo que su utilización debe reducirse al máximo. Es mucho mejor utilizar una sentencia para cada asignación.

Ejemplo: La expresión en 'cascada' del ejemplo anterior sería mucho más adecuado escribirla como:
d = 1;
c = d;
b = c;
a = b;

Cuando se realiza la asignación, el resultado de la expresión se convierte al tipo del operando de la izquierda. Si el tipo del operando de la izquierda es de mayor rango que el de la derecha, no hay ningún problema. Sin embargo, si es al revés sí pueden producirse alteraciones en los valores de los datos:

- si un tipo entero se asigna a otro tipo entero pero de menor rango, se copian los bits menos significativos, pudiéndose producir resultados erróneos.
- si un valor en coma flotante se asigna a una variable de tipo entero, se trunca antes de realizar la asignación.
- si un valor en coma flotante se asigna a una variable en coma flotante, se puede producir un redondeo que será tanto mayor cuanto menor sea la precisión del tipo de la variable.

```
#include <stdio.h>

int main()
{
   signed char c;
```

```
   int i;
   float f = 1.1111111111;

   printf("Asignacion a entero de menor capacidad\n");
   c = 255;
   printf("\tc vale %d\n", c);

   printf("Asignacion de punto flotante a entero\n");
   i = 3.3;
   printf("\tasignando 3.3 i vale %d\n", i);
   i = 3.9;
   printf("\tasignando 3.9 i vale %d\n", i);
   i = -3.9;
   printf("\tasignando -3.9 i vale %d\n", i);

   printf("Asignacion de punto flotante a punto flotante\n");
   printf("\tf vale %.10f\n", f);

   return 0;
}
```

Si compilamos y ejecutamos este programa (que hemos denominado **`truncado.c`**) obtendríamos lo siguiente:

```
local> gcc -W -Wall -o truncado truncado.c
local> ./truncado
Asignacion a entero de menor capacidad
   c vale -1
Asignacion de punto flotante a entero
   asignando 3.3 i vale 3
   asignando 3.9 i vale 3
   asignando -3.9 i vale -3
Asignacion de punto flotante a punto flotante
   f vale 1.1111111641
local>
```

Es importante destacar que las asignaciones que puedan dar lugar a pérdida de información (por redondeo o por truncado) no son ilegales, y por lo tanto el compilador nos va a dejar realizarlas; pero deben utilizarse con muchísimo cuidado.

5.7. Asignación Compacta

Existen operadores en C que permiten simultáneamente realizar una operación con una variable, y asignar el resultado a esa misma variable. Al igual que en el operador de asignación visto anteriormente, los operadores de asignación compacta toman dos operandos, siendo el de la izquierda una variable, y el de la derecha una expresión.

En general, los diferentes operadores de asignación compacta son de la forma:

*op***=**

donde *op* puede ser cualquiera de los siguientes: **+**, **−**, *****, **/**, **%**, **&**, **|**, **^**, **»** o **«**. No debe aparecer ningún espacio en blanco entre el operador y el símbolo **=**.

Son operadores binarios que necesitan como operador a la izquierda un identificador de variable, y a la derecha cualquier expresión compatible con el operador que se esté utilizando. La forma general de utilización de un operador de asignación compacta es:

*var op***=** *expresion*

que es equivalente a:

var **=** *var op (expresion)*

A la vista de la expresión equivalente, puede observarse que, independientemente de la prioridad del operador que estemos utilizando, la expresión siempre se evalúa antes de proceder con la operación final y la asignación.

Los operadores de asignación compacta, al igual que sucedía con el operador de asignación:

- también forman una expresión, y devuelven como resultado el valor asignado.
- también se evalúan de derecha a izquierda.

```
#include <stdio.h>

int main()
{
  int i = 3;
  int j = 2;

  printf("i vale %d, y j vale %d\n", i, j);
  i += 5;
  printf("Tras i += 5, i vale %d, y j vale %d\n", i, j);
  i -= j;
  printf("Tras i -= j, i vale %d, y j vale %d\n", i, j);
  i /= 2;
  printf("Tras i /= 2, i vale %d, y j vale %d\n", i, j);
  j *= i;
  printf("Tras j *= i, i vale %d, y j vale %d\n", i, j);
  i %= j - 2;
  printf("Tras i %%= j-2, i vale %d, y j vale %d\n", i, j);

  return 0;
}
```

Si compilamos y ejecutamos este programa (que hemos denominado **compacta.c**) obtendríamos lo siguiente:

```
local> gcc -W -Wall -o compacta compacta.c
local> ./compacta
i vale 3, y j vale 2
Tras i += 5, i vale 8, y j vale 2
Tras i -= j, i vale 6, y j vale 2
Tras i /= 2, i vale 3, y j vale 2
Tras j *= i, i vale 3, y j vale 6
Tras i %= j-2, i vale 3, y j vale 6
local>
```

Es importante no perder de vista que en la asignación compacta, primero se evalúa la expresión de la derecha, y después se realiza la operación con la variable de la izquierda, finalizando con la asignación. Por tanto, la expresión

x = x * y + 1

no es equivalente a

x *= y + 1

ya que si desarrollamos esta última, la expresión equivalente sería

x = x * (y + 1)

5.8. Conversión Forzada

Ya hemos visto que cuando se trata de una asignación, se realiza una conversión de tipos, de forma que el resultado de evaluar la expresión se convierte al tipo de la variable antes de realizar la asignación. Hay

situaciones en las que es necesario realizar un cambio de tipo sin involucrar una asignación; podría ser el caso, por ejemplo, de una comparación, o de un parámetro para pasarlo a una función.

En esos casos es necesario aplicar el operador unario de cambio de tipo (suele aparecer como *cast* en la bibliografía), cuyo formato es:

(nuevo_tipo) expresion

Este operador convierte el resultado de la expresión al tipo indicado entre paréntesis. Un caso especialmente útil es cuando deseamos realizar una división con decimales entre dos enteros, o una división entera entre números con decimales.

```
#include <stdio.h>

int main()
{
  float resultado = 0;
  float decimal   = 5.0;
  int   entero    = 2;

  resultado = decimal / entero;
  printf("decimal/entero da %f\n", resultado);
  resultado = (int) decimal / entero;
  printf("(int) decimal/entero da %f\n", resultado);

  return 0;
}
```

Si compilamos y ejecutamos este programa (que hemos denominado **conversion.c**) obtendríamos lo siguiente:

```
local> gcc -W -Wall -o conversion conversion.c
local> ./conversion
decimal/entero da 2.500000
(int) decimal/entero da 2.000000
local>
```

5.9. Operadores de Autoincremento y Autodecremento

Existen dos operadores en C que permiten utilizar el valor de una variable, y a la vez incrementarla (o decrementarla) en una unidad. Son operadores unarios, y su único operando debe ser una variable.

Los operadores son:

- **++** incrementa el valor de la variable en una unidad.
- **--** decrementa el valor de la variable en una unidad.

La particularidad de estos operadores radica en que tienen dos comportamientos, dependiendo de si se colocan delante (prefijos) o detrás (postfijo) de la variable que hace de operando. En ambos casos, se modifica el valor de la variable en una unidad y se devuelve el valor de la misma; la diferencia entre ambos radica en cuándo se modifica el valor de la variable:

- Si se utiliza como prefijo, primero se modifica el valor de la variable (incrementándola o decrementándola, dependiendo del operador) y se obtiene como valor de la expresión el valor ya modificado.
- Si se utiliza como postfijo, primero se obtiene el valor de la variable (que es el resultado de la expresión) y posteriormente se modifica el valor de la variable, incrementándola o decrementándola.

```
#include <stdio.h>

int main()
{
  int i = 1;
  int j = 0;

  printf("i vale %d, y j vale %d\n", i, j);
  j = ++i;
  printf("Tras j = ++i, i vale %d, y j vale %d\n", i, j);
  j = --i;
  printf("Tras j = --i, i vale %d, y j vale %d\n", i, j);
  j = i++;
  printf("Tras j = i++, i vale %d, y j vale %d\n", i, j);
  j = i--;
  printf("Tras j = i--, i vale %d, y j vale %d\n", i, j);

  return 0;
}
```

Si compilamos y ejecutamos este programa (que hemos denominado **autos.c**) obtendríamos lo siguiente:

```
local> gcc -W -Wall -o autos autos.c
local> ./autos
i vale 1, y j vale 0
Tras j = ++i, i vale 2, y j vale 2
Tras j = --i, i vale 1, y j vale 1
Tras j = i++, i vale 2, y j vale 1
Tras j = i--, i vale 1, y j vale 2
local>
```

5.9.1. Efectos secundarios de ++ y --

Los operadores de autoincremento y autodecremento pueden resultar muy útiles, pero presentan efectos secundarios que a menudo hacen que el programa no se comporte como debe, y sea muy difícil encontrar el motivo.

Algo similar sucede en el paso de parámetros a funciones, como se verá en capítulos posteriores.

Por lo tanto se aconseja tener mucha precaución en la utilización de los operadores de autoincremento y autodecremento, y en ningún caso utilizarlos:

- con variables que se empleen más de una vez en una expresión. Por ejemplo, si **num** vale **5**, en la siguiente expresión:

  num / 2 + 5 * (3 + num++)

  no está garantizado cuál de las veces que aparece **num** en la expresión se evaluará antes. Si es la segunda (la que aparece con el operador de autoincremento), la primera tomaría para **num** el valor **6**; mientras que si se evalúa la primera, tomaría **5**

- en argumentos de funciones, si la variable sobre la que opera se utiliza en más de un argumento. Esto se verá más adelante, pero el principio que se aplica es el mismo del caso anterior.

- como parte del segundo operando en una expresión lógica. En una expresión como:

  (i > 7) && (j++ > 2)

  si **i** es mayor que **7**, **j** incrementaría su valor tras la expresión, pero si **i** fuera menor o igual a **7**, no se evaluaría el segundo operando de la expresión y **j** no modificaría su valor.

5.10. Operador Condicional

El operador condicional es un operador ternario, con tres argumentos numéricos, que proporciona una forma alternativa para escribir una construcción condicional.

La sintaxis del operador condicional es:

`expresion1 ? expresion2 : expresion3`

El resultado de la expresión será:

- **`expresion2`**, si **`expresion1`** es verdadera (distinta de 0), y en este caso **no se evaluaría `expresion3`**.
- **`expresion3`**, si **`expresion1`** es falsa (igual a 0), y en este caso **no se evaluaría `expresion2`**.

En el siguiente ejemplo:

`(i < 0) ? 0 : 100`

si el valor de **`i`** es negativo, la expresión valdrá **`0`**, pero si no lo es, devolverá **`100`**.

5.11. Operador `sizeof`

Se ha comentado ya que en C, el tamaño de los diferentes tipos de datos puede depender de la máquina en la que ejecutamos el código e incluso del compilador que estemos utilizando.

Sin embargo, hay situaciones (algunas de las cuales veremos de forma exhaustiva más adelante) en las que es necesario conocer el tamaño de un determinado tipo en tiempo de ejecución. Para eso, el C incorpora un operador especial, que es el operador **`sizeof`**.

Es un operador unario, que admite como único operando el nombre de un tipo de datos o el identificador de una variable. Devuelve un valor numérico que representa la cantidad de memoria necesaria para el tipo de datos que se le pasa como parámetro. Se mide en unidades de **`char`**, por lo que, por definición, **`sizeof(char)`** devolverá siempre **`1`**[2].

El siguiente programa permite obtener para una máquina y un compilador determinados, el tamaño correspondiente a cada uno de los tipos enteros sin signo del estándar C:

```
#include <stdio.h>

int main()
{
  printf("El tamano de un unsigned char es %d\n",
         sizeof(unsigned char));
  printf("El tamano de un unsigned short int es %d\n",
         sizeof(unsigned short int));
  printf("El tamano de un unsigned int es %d\n",
         sizeof(unsigned int));
  printf("El tamano de un unsigned long int es %d\n",
         sizeof(unsigned long int));
  printf("El tamano de un unsigned long long int es %d\n",
         sizeof(unsigned long long int));

  return 0;
}
```

Si compilamos y ejecutamos este programa (que hemos denominado **`tamanio.c`**) en la máquina en la que se redactan estos apuntes, obtendríamos lo siguiente:

[2] El tamaño en bits de un **`char`** viene definido por la constante simbólica **`CHAR_BIT`**, definida en **`limits.h`**.

```
local> gcc -W -Wall -o tamanio tamanio.c
local> ./tamanio
El tamano de un unsigned char es 1
El tamano de un unsigned short int es 2
El tamano de un unsigned int es 4
El tamano de un unsigned long int es 4
El tamano de un unsigned long long int es 8
local>
```

5.12. Reglas de Prioridad

Los operadores tienen reglas de precedencia (o prioridad) y asociatividad que determinan exactamente la forma de evaluar las expresiones.

Considérese la expresión:

1 + 2 * 3

Existirían dos posibilidades de evaluar dicha expresión: primero la suma y después la multiplicación, con lo que el resultado obtenido sería **9**, o primero la multiplicación y después la suma, con lo que el resultado sería **7**. La duda se resuelve sabiendo que el operador ***** tiene mayor prioridad que el operador **+**, por lo que primero se ejecuta la multiplicación y después la suma:

1 + (2 * 3)

y por lo tanto el resultado es **7**.

La situación está clara cuando mezclamos operadores de distinta prioridad pero, ¿qué sucede si mezclamos operadores de la misma prioridad? En ese caso utilizamos el concepto de asociatividad. La asociatividad determina el orden en que se realizarán las operaciones cuando utilizamos operadores de igual prioridad. Así, por ejemplo, considerando que la multiplicación y la división tienen la misma prioridad, la expresión:

6 / 3 * 2

podría evaluarse de dos formas: primero la división y luego la multiplicación, con lo que obtendríamos **4** como resultado, o primero la multiplicación y después la división, lo que daría **1** como resultado. Sin embargo, la duda se resuelve aplicando la asociatividad de ambos operadores: de izquierda a derecha, lo que supone que primero se aplica el operador de más a la izquierda, y a continuación los que le siguen por la derecha. Es decir, la forma en la que se evaluará la expresión es:

(6 / 3) * 2

y el resultado que se obtendrá será **4**.

La siguiente tabla recoge la prioridad y la asociatividad de los operadores vistos anteriormente; serán más prioritarios mientras antes aparezcan en la tabla. Dentro de los de igual prioridad, se evaluarán según la asociatividad indicada (la asociatividad se aplica al orden en que aparecen los operadores en la expresión, no al orden en el que aparecen en la tabla):

PRIORIDAD	OPERADORES	ASOCIATIVIDAD
Mayor	Paréntesis `()`	⇒

	Operadores unarios postfijos
	`++ --`
	Operadores de llamadas a función
	`()`
	Operadores de tablas y estructuras
	`[] . ->`
		⇒

	Operadores unarios prefijos
	`! ~ ++ -- + - (tipo) sizeof()`
	Operadores de punteros
	`* &`
		⇐

	Operadores multiplicativos `* / %`	⇒
	Operadores aditivos `+ -`	⇒
	Operadores de desplazamiento: `<< >>`	⇒
	Operadores relacionales `< <= > >=`	⇒
	Operadores de igualdad `== !=`	⇒
	Operador AND `&`	⇒
	Operador XOR `^`	⇒
	Operador OR `|`	⇒
	Operador lógico Y `&&`	⇒
	Operador lógico O `||`	⇒
	Operador condicional `?:`	⇐
Menor	Operadores de asignación `= += -= *= /= %= &= ^= |= <<= >>=`	⇐

Podemos observar en la tabla que la máxima prioridad la presentan los paréntesis, por lo que siempre que queramos evaluar una expresión de forma distinta a la establecida por las prioridades y la asociatividad de los operadores, haremos uso de los mismos.

Ejemplo: Si en `1 + 2 * 3` queremos que se aplique antes la suma que la multiplicación, debemos poner `(1 + 2) * 3`. Y si en `6 / 3 * 2` queremos aplicar antes la multiplicación que la división, tendremos que escribir `6 / (3 * 2)`

También es conveniente la utilización de los paréntesis en caso de duda, o para dejar más claro en el código cómo se va a evaluar una expresión. En definitiva: ante la duda, utilice siempre los paréntesis.

En la tabla aparecen algunos operadores que se verán más adelante.

Tema 6

Estructuras de Control

Índice

6.1. Introducción **6-1**
6.2. Estructuras Secuenciales **6-2**
6.3. Estructuras Selectivas **6-2**
6.3.1. Estructura Selectiva Simple: `if else` 6-3
6.3.2. Sentencias Selectivas Simples Anidadas 6-4
6.3.3. Ejemplo 6-5
6.3.4. Estructura Selectiva Múltiple: `switch` 6-6
6.4. Estructuras Repetitivas **6-9**
6.4.1. Sentencia `while` 6-9
6.4.2. Sentencia `do while` 6-11
6.4.3. Sentencia `for` 6-12
6.5. Ejemplos de Uso **6-15**
6.5.1. Lectura del teclado 6-15
6.5.2. Solución de una ecuación de primer grado 6-15

6.1. Introducción

En los ejemplos de programas que se han visto hasta ahora, las instrucciones seguían una estructura secuencial: se ejecutaba una sentencia, tras la finalización se ejecutaba la siguiente, y así sucesivamente hasta alcanzar el final del programa.

Evidentemente, esta estructura no es válida para todos los casos. Imaginemos que necesitamos obtener la suma de los 100 primeros números naturales. Con lo visto hasta ahora, la única posibilidad consiste en utilizar 100 instrucciones, una a continuación de otra, sumando un número diferente en cada una de ellas.

Otro ejemplo en el que no es suficiente con la estructura secuencial es aquel en el que sólo debemos realizar una operación si se cumple una condición previa: por ejemplo, un programa que controla un cajero automático, y sólo entrega el dinero (y lo resta del saldo) si la cantidad solicitada es inferior o igual al saldo disponible.

Realmente, lo difícil es encontrar un programa que pueda realizarse únicamente con la estructura secuencial. Por eso, el lenguaje C (y todos los lenguajes de programación en general) incorporan una serie de estructuras de control que permiten resolver fácilmente los ejemplos mostrados anteriormente.

Las estructuras que incorpora el C basadas en la programación estructurada son:

- secuenciales

- selectivas
- repetitivas

Se debe evitar el uso de cualquier otra estructura que no sea una de las anteriores, ya que conduce a código no estructurado. Contrariamente a lo que pudiera parecer en un primer momento, esto no supone ninguna limitación a la hora de escribir un programa, ya que se puede demostrar que cualquier programa se puede realizar utilizando únicamente estas estructuras.

En concreto, está totalmente prohibido en el ámbito de la asignatura la utilización de las sentencias **goto** (que realiza un salto incondicional) y **continue** (que fuerza una nueva iteración dentro de de una estructura repetitiva), y sólo se permite la utilización de la sentencia **break** (que fuerza la salida de una estructura selectiva o repetitiva) en el ámbito de la sentencia selectiva múltiple, que se estudiará más adelante.

6.2. Estructuras Secuenciales

En un programa en C, las sentencias se ejecutan una tras otra en el orden en el que están escritas. El fin de una sentencia marca el comienzo de la siguiente.

```
#include <stdio.h>

/* Obtiene en grados Celsius una temperatura dada en grados
   Fahrenheit, segun la expresion C = (5/9) * (F-32) */

int main()
{
  float fahrenheit;
  float celsius;

  printf("Temperatura en grados Fahrenheit: ");
  scanf("%f", &fahrenheit);
  celsius = (fahrenheit - 32) * 5 / 9;

  printf("%f grados fahrenheit son %f grados celsius\n",
   fahrenheit, celsius);

  return 0;
}
```

Para poder considerar un grupo de sentencias como una sola, podemos encerrarlas entre llaves. A esta construcción se le denomina **bloque**, y puede aparecer en cualquier lugar en el que puede aparecer una sentencia. Recuerde además que la formación de bloques determinan la visibilidad y el tiempo de vida de las variables que se definen en él.

6.3. Estructuras Selectivas

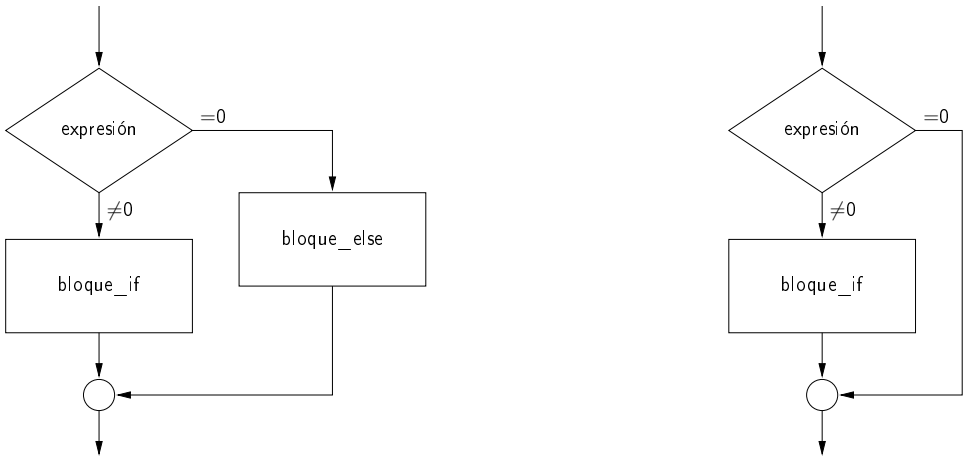
También llamadas condicionales; permiten que ciertas sentencias se ejecuten o no en función de una determinada condición.

En C existen dos tipos de estructuras selectivas:

- la simple: **if else**.
- la múltiple: **switch**.

6.3.1. Estructura Selectiva Simple: `if else`

La estructura selectiva simple responde al siguiente diagrama de flujo:



La sintaxis en C de dicha estructura es:

```
if (expresion)
  bloque_if
else
  bloque_else
```

donde la parte correspondiente al **`else`** es opcional.

El funcionamiento de la estructura selectiva simple es el siguiente:

1. se evalúa la expresión que acompaña a la cláusula **`if`**
2. Si la expresión es cierta (el valor de la expresión es distinto de cero), se ejecuta el bloque que sigue a continuación y se termina.
3. Si la expresión es falsa (el valor de la expresión es igual a cero) y existe la cláusula **`else`**, se ejecuta el bloque que sigue a la cláusula **`else`** y se termina.

o dicho de otra forma:

- el bloque que sigue a la cláusula **`if`** sólo se ejecuta si el valor de la expresión es distinto de cero.
- si existe una cláusula **`else`**, el bloque que sigue a dicha cláusula sólo se ejecuta si el valor de la expresión es igual a cero.

Ponga atención en que lo que sigue a un **`if`** o a un **`else`** es un bloque, y recuerde que un bloque es o una única sentencia, o un grupo de sentencias encerradas entre llaves. Un problema muy común cuando se utiliza una estructura selectiva simple consiste en utilizar más de una sentencia (sin agruparlas en un bloque) bien en el bloque del **`if`** bien en el del **`else`**:

```
if (numero < 0)
  negativo = 1;
  suma -= 1;
else
  negativo = 0;
  suma += 1;
```

En el primer caso (varias sentencias en el bloque del **`if`**), el compilador detectaría el problema, puesto que habría un **`else`** que no se corresponde con ningún **`if`** (tal y como está escrito, la sentencia selectiva simple

terminaría con la ejecución de la sentencia **negativo = 1**). La solución, una vez detectado el problema, es simple: formar un bloque con las dos sentencias encerrándolas entre llaves:

```
if (numero < 0)
  {
    negativo = 1;
    suma -= 1;
  }
else
   negativo = 0;
   suma += 1;
```

El segundo caso es más grave, ya que no es ningún error de sintaxis. Desde el punto de vista del compilador, la sentencia selectiva simple finalizaría con la ejecución de la sentencia **negativo = 0**, y la siguiente sentencia se ejecutaría siempre, independientemente de si **numero** es menor o no que cero. En consecuencia, si **numero** es menor que cero, las sentencias que se ejecutarían serían:

```
  negativo = 1;
  suma -= 1;
  suma += 1;
```

y si no es menor que cero:

```
  negativo = 0;
  suma += 1;
```

Encontrar este error, como se puede comprender, puede resultar muy laborioso. Por eso es muy conveniente escribir los programas en algún editor que sea capaz de entender la sintaxis de C e indente adecuadamente cada sentencia, en función del bloque al que pertenezca. Con un editor de este tipo, el resultado que habríamos obtenido habría sido:

```
if (numero < 0)
  {
    negativo = 1;
    suma -= 1;
  }
else
  negativo = 0;
suma += 1;
```

donde puede observarse, por la situación de la sentencia **suma += 1**, que no pertenece al bloque de la cláusula **else**, como debería.

La solución, al igual que en el caso anterior, pasa por encerrar entre llaves (y formar un bloque) las sentencias que forman parte del ámbito del **else**:

```
if (numero < 0)
  {
    negativo = 1;
    suma -= 1;
  }
else
  {
    negativo = 0;
    suma += 1;
  }
```

6.3.2. Sentencias Selectivas Simples Anidadas

Dentro de los bloques del **if** o del **else** en una sentencia selectiva simple pueden utilizarse a su vez sentencias selectivas simples, dando lugar a sentencias selectivas simples anidadas.

En este caso, hay que tener en cuenta que la sentencia **else** se asocia siempre al **if** más cercano.

Por ejemplo, en el siguiente fragmento de código:

```
if (n > 0)
  if (a > b)
    z = a;
  else
    z = b;
```

la cláusula **else** se asocia a **if (a > b)**. Si quisiéramos que se aplicara al primer **if**, bastará con encerrar entre llaves el segundo **if**:

```
if (n > 0)
  {
    if (a > b)
      z = a;
  }
else
  z = b;
```

De hecho es muy conveniente, en el caso de sentencias de selección simples anidadas, encerrar siempre entre llaves los bloques correspondientes a cada una de las cláusulas **if** y **else** que aparezcan, para dejar claro el ámbito de cada una de ellas.

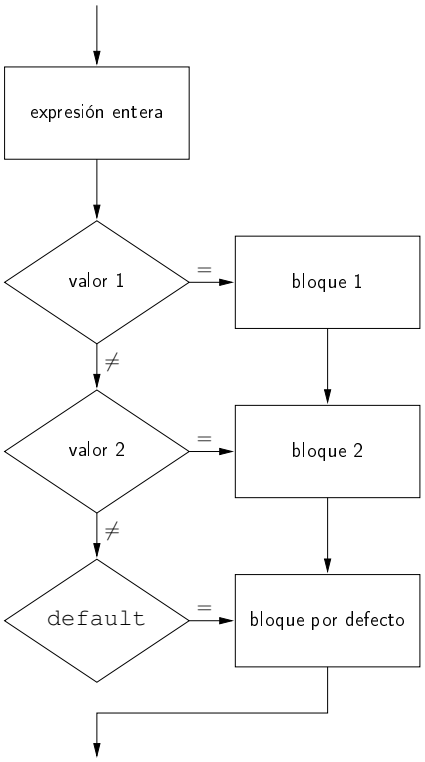
6.3.3. Ejemplo

Como ejemplo de utilización de estructura selectiva simple, escribiremos un programa en C que indique si un año es bisiesto, utilizando el algoritmo ya usado en el tema 2:

```
#include <stdio.h>

/* Calcula si un anio es bisiesto o no.
   Un anio es bisiesto si es divisible por 4 pero no por 100,
   excepto aquellos divisibles por 400. */

int main()
{
  int year     = 0;
  int bisiesto = 0;

  printf("Introduzca el anio: ");
  scanf("%d", &year);

  if (0 == (year % 4))
    {
      if (0 == (year % 100))
        {
          if (0 == (year % 400))
            bisiesto = 1;
        }
      else
        bisiesto = 1;
    }

  if (1 == bisiesto)
    printf("El anio %d es bisiesto\n", year);
  else
    printf("El anio %d NO es bisiesto\n", year);

  return 0;
}
```

6.3.4. Estructura Selectiva Múltiple: `switch`

En la estructura selectiva múltiple, se puede seleccionar entre varias alternativas según el valor de una expresión entera. El diagrama de flujo de esta estructura sería el siguiente:



La sintaxis en C de dicha estructura selectiva múltiple es:

```
switch (expresion)
{
  case exprConst1:
    listaProp1
  case exprConst2:
    listaProp2
  case exprConstN:
    listaPropN
  default:
    propDefault
}
```

El funcionamiento de esta estructura es como sigue:

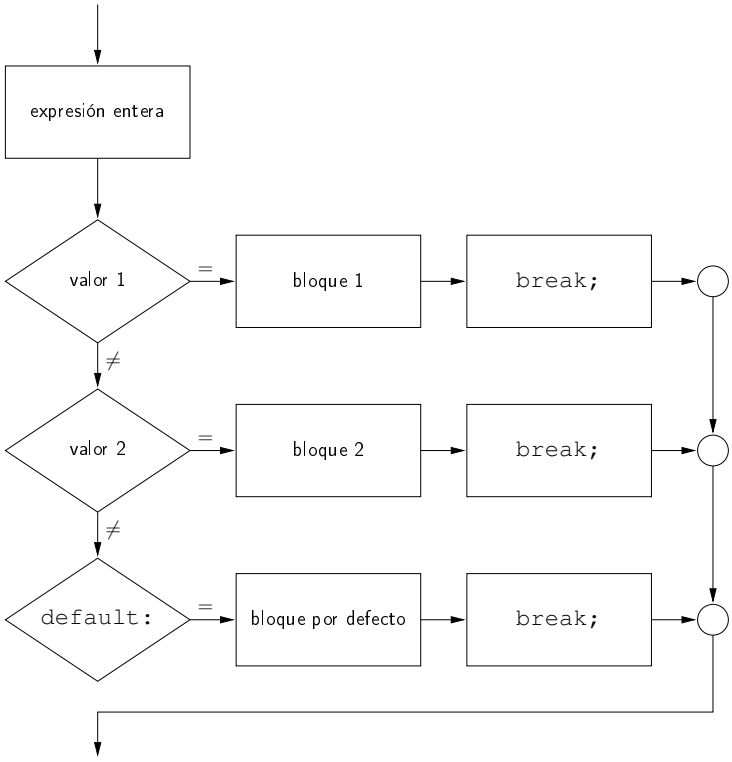
1. Se evalúa la expresión que acompaña al **`switch`**. Esta expresión se evalúa como expresión entera.
2. Se compara el valor obtenido, secuencialmente, con los valores que acompañan los diferentes **`case`**, deteniéndose en el primero que coincida. Estos valores deberán ser siempre **expresiones constantes enteras**.

Existe una etiqueta especial, **`default`**, que siempre es cierta, por lo que se utiliza para contemplar el resto de casos que no se han considerado en los **`case`** anteriores. Su utilización es opcional; es decir, no es necesario que todas las estructuras selectivas múltiples la incorporen. Es muy importante que esta etiqueta, si se utiliza, se sitúe como la última del bloque, puesto que las que se pongan detrás de ella nunca serán consideradas (como se ha obtenido la coincidencia en una etiqueta, se deja de comprobar el resto de etiquetas).

Cuando en una estructura de selección múltiple se encuentra concordancia con un **case**, se ejecutan todas las sentencias que se encuentren a partir de este hasta el final de la estructura selectiva múltiple (incluyendo todos los **case** que aparezcan a continuación del coincidente). Lo que normalmente se desea no es esto, sino que se ejecute sólo hasta la aparición del siguiente **case**. La solución consiste en poner un **break** como última sentencia dentro de cada **case**, lo que hace que se finalice la ejecución de la estructura selectiva múltiple:

```
switch (expresion)
{
  case exprConst1:
    listaProp1
    break;
  case exprConst2:
    listaProp2
    break;
  case exprConstN:
    listaPropN
    break;
  default:
    propDefault
    break;
}
```

lo que se correspondería con el siguiente diagrama de flujo:



El siguiente programa determina el número de días del mes indicado, agrupándolos según tengan 28, 30 ó 31 días:

```
#include <stdio.h>

/* Programa que lea un numero de mes (en el rango de 1 a 12)
   e indique el numero de dias del mes. */
```

```
int main ()
{
  int mes   = 0;
  int ndias = 0;

  printf("Introduzca el numero de un mes (1-12): ");
  scanf("%2d", &mes);

  switch (mes)
    {
    case 2:
      ndias = 28;
      break;
    case 4:
    case 6:
    case 9:
    case 11:
      ndias = 30;
      break;
    default:
      ndias = 31;
      break ;
    }
  printf("\tEl mes %d tiene %d dias.\n", mes, ndias);

  return 0;
}
```

Observe cómo tras la última sentencia de cada bloque aparece una sentencia **break** para dar por finalizada la ejecución de la estructura selectiva múltiple (incluyendo el caso por defecto, **default**).

El caso por defecto (**default**) suele utilizarse para detectar casos no válidos o erróneos, evitando que se produzcan errores en la ejecución. El siguiente ejemplo determina si un número entre el 1 y el 10 es par o impar; el caso por defecto se ha utilizado en este programa para los números que están fuera de ese rango:

```
#include <stdio.h>

/* Determina si un numero entre el 1 y el 10 es par o impar. */

int main()
{
  int num = 0;

  printf("\nIntroduzca el numero: ");
  scanf("%d", &num);

  switch (num)
    {
    case 1:
    case 3:
    case 5:
    case 7:
    case 9:
      printf("\n El numero %d es impar\n", num);
      break;
    case 2:
    case 4:
    case 6:
    case 8:
    case 10:
      printf("\n El numero %d es par\n", num);
      break;
    default:
      printf("\n FUERA DE RANGO\n");
```

```
      break;
    }

  return 0;
}
```

6.4. Estructuras Repetitivas

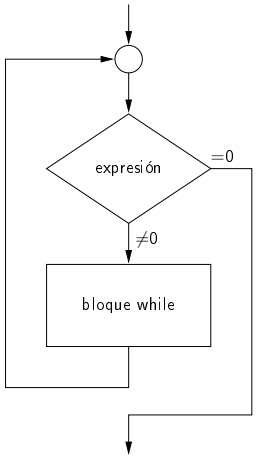
En este tipo de estructuras, se repite un conjunto de instrucciones en función de una condición. La principal diferencia entre las diferentes estructuras repetitivas consiste en qué punto se realiza la comprobación de la condición.

En C existen tres tipos de estructuras repetitivas:

- **`while`**
- **`do while`**
- **`for`**

6.4.1. Sentencia `while`

La sentencia **`while`** responde al siguiente diagram de flujo:



La sintaxis en C de dicha sentencia es:

```
while (expresion)
  bloque
```

El funcionamiento es el siguiente:

1. se evalúa la expresión que acompaña a la cláusula **`while`**
2. Si la expresión es cierta (el valor de la expresión es distinto de cero), se ejecuta el bloque que sigue a continuación.
3. se vuelve al primer paso, y se repite el proceso.

Algunos aspectos de interés sobre esta estructura serían:

- Puede que el bloque que sigue al **while** no se ejecute ninguna vez. Si la primera vez que se calcula la condición el resultado es cero (falso), no se pasaría a ejecutar el bloque, y se pasaría directamente a la sentencia que siga a la sentencia **while**.
- Alguno de los valores que determinan la condición debe ser modificado dentro del bloque. Si no fuera así y la condición fuera cierta (distinta de cero) la primera vez que la comprobáramos, pasaríamos a ejecutar el bloque, y ya no saldríamos nunca de él puesto que la condición seguiría siendo cierta de forma indefinida.
- En la condición, es conveniente utilizar los operadores de rango (**<**, **>**, **<=** o **>=**) en lugar de los operadores de igualdad y desigualdad (**==** o **!=**). Con estos últimos estamos esperando un valor exacto, y si por cualquier motivo no se produce (por ejemplo, porque estemos saltando de tres en tres) el bucle se convierte en infinito; si usamos los primeros, esta circunstancia se evita.

Al igual que sucedía en el caso de las sentencias selectivas simples, es un error muy común querer utilizar más de una sentencia dentro del bloque del **while** pero no encerrarlas entre llaves formando un bloque, como en el siguiente ejemplo:

```
#include <stdio.h>

int main()
{
  int num  = 0;
  int suma = 0;

  while (10 > num)
    num++;
    suma += num;
  printf("La suma hasta el %d vale %d\n", num, suma);

  return 0;
}
```

Si lo compilamos y ejecutamos, obtenemos el siguiente resultado:

```
salas@318CDCr12:~$ gcc -W -Wall -o whileMal whileMal.c
salas@318CDCr12:~$ ./whileMal
La suma hasta el 10 vale 10
salas@318CDCr12:~$
```

Como puede observarse, y en contra de lo que pudiera parecer a simple vista, el fragmento de código anterior no suma los diez primeros números (del 1 al 10), sino únicamente el último (10); a pesar de que el compilador no ha detectado ningún error. Eso es porque el bloque del **while** está formado únicamente por la sentencia **num++**, que es la que se repite 10 veces. Una vez finalizado el bucle **while** es cuando se ejecuta la sentencia **suma += num**, que lógicamente sólo se ejecuta una vez. La forma correcta sería la siguiente:

```
#include <stdio.h>

int main()
{
  int num  = 0;
  int suma = 0;

  while (10 > num)
    {
      num++;
      suma += num;
    }
  printf("La suma hasta el %d vale %d\n", num, suma);

  return 0;
```

```
}
```

Si lo compilamos y ejecutamos, obtenemos el siguiente resultado:

```
salas@318CDCr12:~$ gcc -W -Wall -o whileBien whileBien.c
salas@318CDCr12:~$ ./whileBien
La suma hasta el 10 vale 55
salas@318CDCr12:~$
```

Se pueden evitar estas confusiones si siempre agrupamos las sentencias que forman el ámbito de aplicación del **`while`** entre llaves, aunque sea una única sentencia. O utilizando un editor de textos que entienda la sintaxis de C e indente adecuadamente cada sentencia, en función del bloque a que pertenezca.

Valores Extremos en la Condición

En las estructuras repetitivas en general, y como caso particular en la sentencia **`while`**, es muy importante comprobar que los valores extremos de la condición (el primer y el último valor para los que se ejecuta el bloque que acompaña a la cláusula **`while`**) son los adecuados.

Para ello es de gran utilidad repasar mentalmente (o con ayuda de papel y lápiz) qué sucede en la condición de la sentencia **`while`** la primera vez que se llega a ella y la última vez que se realiza la comprobación (la vez que se termina la sentencia).

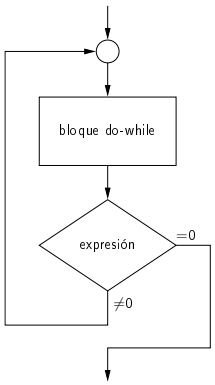
Vamos a utilizar como ejemplo para ver esto el del apartado anterior, de suma de los 10 primeros números. La primera vez que llegamos al **`while`**, el valor de **`num`** vale **`0`** (lo acabamos de inicializar), que cumple la condición **`(10 > num)`**, por lo que entraremos a ejecutar el bloque. Incrementamos el valor de **`num`** (que ahora pasará a valer **`1`**) y lo sumamos. Comprobamos pues que el primer valor del bucle es el correcto. Supongamos ahora que **`num`** vale **`9`**; la condición se sigue cumpliendo, y por tanto entramos en el bucle: incrementamos **`num`**, que ahora pasa a valer **`10`**, y lo sumamos. En la siguiente iteración, la condición no se cumple y saldremos de la sentencia **`while`**. Hemos comprobado por tanto que efectivamente hemos sumado los números del 1 al 10.

Lo anterior se resume en la siguiente tabla:

`num`	Condición	Valor sumado	**`num`**
0	cierta	1	1
...			
9	cierta	10	10
10	falsa		

6.4.2. Sentencia `do while`

El diagrama de flujo de la sentencia **`do while`** es:



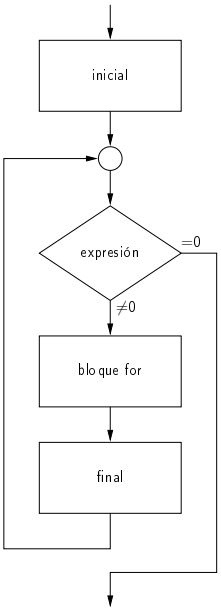
La sintaxis en C de esta sentencia es la siguiente:

```
do
  bloque
while (expresion);
```

A diferencia de la sentencia **`while`**, en esta estructura repetitiva primero se ejecuta el bloque y posteriormente se comprueba la condición. Por tanto, el bloque de una sentencia **`do while`** se ejecuta siempre al menos una vez. Se recomienda usar siempre las llaves para delimitar el bloque de sentencias a ejecutar dentro del bucle. Esto no es necesario cuando el bloque está formado por una única sentencia.

6.4.3. Sentencia `for`

El diagrama de flujo de la sentencia **`for`** es:



La sintaxis en C de esta sentencia es:

```
for (inicial;
     expresion;
     final)
  bloque
```

El funcionamiento de esta sentencia es el siguiente:

1. se ejecuta el bloque **`inicial`**.
2. se evalúa la expresión.
3. si es falsa (igual a cero), se finaliza la ejecución de la sentencia.
4. si es verdadera (distinta de cero):
   - se ejecuta el bloque.
   - se ejecuta el bloque **`final`**.
   - se vuelve al paso 2.

En una sentencia **for** puede omitirse cualquiera de los tres campos, pero es imprescindible mantener los **;** de separación. Si se omite el segundo campo, la condición se da por cierta, y se obtiene un **bucle infinito**.

En el siguiente ejemplo se utiliza una sentencia **for** para calcular la suma de todos los números pares entre 0 y 1000:

```
#include <stdio.h>

/* Programa que obtiene la suma de los numeros pares
   comprendidos entre 2 y 1000. */

int main()
{
  int i;
  int suma = 0;

  for ( i = 2; i <= 1000; i += 2 )
    suma += i;
  printf("La suma de los pares hasta 1000 es %d\n", suma);

  return 0;
}
```

Pueden introducirse varias sentencias tanto en el bloque **inicial** como en el bloque **final**. En ese caso, las sentencias van separadas por comas, ejecutándose de izquierda a derecha:

```
#include <stdio.h>

/* Programa que obtiene la suma de los numeros pares
   comprendidos entre 2 y 1000. */

int main()
{
  int i;
  int suma;

  for ( suma = 0, i = 2; i <= 1000; suma += i, i += 2)
    ;
  printf("La suma de los pares hasta 1000 es %d\n", suma);

  return 0;
}
```

Obsérvese el **;** aislado que aparece tras la cláusula **for**. Esto es necesario porque una sentencia **for** siempre incluye un **bloque for**. Si no hubiéramos incluido ese **;** (que representa una sentencia vacía), el bloque del **for** lo hubiera constituido la sentencia **printf**, que se habría ejecutado 500 veces.

En el siguiente ejemplo, la sentencia **for** se utiliza para implementar la función que eleva un número a una potencia:

```
#include <stdio.h>

/* Calcula la potencia n de un numero num */

int main()
{
  int num       = 0;
  int exponente = 0;
  int potencia  = 1;
  int i         = 0;     /* var. de control del bucle */

  printf("\nIntroduzca el numero y exponente: ");
  scanf("%d %d", &num, &exponente);

  for (i = 1; i <= exponente; i++)
```

```
    potencia *= num;

  printf("%d elevado a %d vale %d\n", num, exponente, potencia);

  return 0;
}
```

Estructuras Repetitivas Anidadas

Una estructura puede contener otra dentro, cumpliendo que estén totalmente anidadas:

```
for (......)
  {
    for (..........)
      {
        while (...)
          {
            ...........
          }
      }
  }
```

En bucles anidados cada alteración del bucle externo provoca la ejecución completa del bucle interno, como puede comprobarse en el siguiente ejemplo, que imprime las tablas de multiplicar:

```
#include <stdio.h>

/* Imprime tablas de multiplicar */

int main()
{
  int i = 0;
  int j = 0;

  /* Primera aproximacion */
  for (i = 1; i <= 10; i++)
    for (j = 1; j <= 10; j++)
      printf("%d x %d = %d\n", i, j, i*j);

  /* Mas presentable */
  for (i = 1; i <= 10; i++)
    {
      printf("\nTabla del %d\n=============\n", i);
      for (j = 1; j <= 10; j++)
        printf("%d x %d = %d\t", i, j, i*j);
    }

  /* Ahora las tablas en paralelo.
     OJO a como se recorren los indices. */
  for (i = 1; i <= 10; i++)
    {
      printf("\n");
      for (j = 1; j <= 10; j++)
        printf("%d x %d = %d\t", j, i, i*j);
    }
  printf("\n");

  return 0;
}
```

6.5. Ejemplos de Uso

6.5.1. Lectura del teclado

El siguiente programa lee desde el teclado carácter a carácter, mediante la función **getchar**, y escribe lo leído en la pantalla mediante la función **putchar**. El proceso finaliza cuando se recibe el carácter de **fin de entrada**, **EOF**, que se obtiene pulsando simultáneamente las teclas **Ctrl** y **D**.

```
#include <stdio.h>

/* Programa que copia la entrada estandar a la salida estandar */

int main()
{
  int c = 0;      /* Almacena caracteres */

  while ((c = getchar()) != EOF)
    putchar(c);

  return 0;
}
```

6.5.2. Solución de una ecuación de primer grado

El siguiente programa calcula la solución de una ecuación de primer grado del tipo

$$ax + b = 0$$

Para ello solicita los dos parámetros necesarios, y resuelve la ecuación considerando de forma separada los casos especiales.

```
#include <stdio.h>

/* Resuelve una ecuacion de primer grado:
   0 = ax + b
   a <> 0 => x = -b/a
   a == 0 y b <> 0 => solucion imposible
   a == 0 y b == 0 => sol. indeterminada */

int main()
{
  int   a = 0;    /* Coeficiente de la variable independiente */
  int   b = 0;    /* Termino independiente */
  float x = 0;    /* Solucion */

  printf("\nIntroduzca los coeficientes a y b: ");
  scanf("%d %d", &a, &b);
  if (0 != a)
    {
      x = -(float) b / a;
      /* Razonar el resultado sin la conversion forzada */
      printf("\nLa solucion es %f\n", x);
    }
  else if (b)
    /* La condicion anterior equivale a esta otra: 0 != b */
    printf("\n Solucion imposible\n");
  else
    printf("\n Solucion indeterminada\n");

  return 0;
}
```

Tema 7

Funciones

Índice

7.1. Funciones: Introducción **7-1**
7.1.1. La función `main` 7-2
7.2. Definición de una Función **7-2**
7.2.1. El identificador 7-3
7.2.2. Los parámetros 7-3
7.2.3. El Cuerpo de la Función 7-3
7.2.4. El Tipo de la Función 7-4
7.3. Declaración de una Función **7-4**
7.4. Utilización de una Función **7-5**
7.4.1. Correspondencia de parámetros 7-5
7.5. Recursividad **7-7**
7.6. Compilación Separada y Descomposición en Ficheros **7-8**
7.6.1. Funciones y ficheros de cabecera 7-9
7.7. El preprocesador de C. **7-9**
7.7.1. La directiva `include` 7-10
7.7.2. La directiva `define` 7-10
7.7.3. Directivas condicionales 7-12
7.8. Resultado final **7-13**

7.1. Funciones: Introducción

Los programas de ejemplo que se han visto hasta ahora han sido todos ellos muy simples. Eso nos ha permitido codificar fácilmente la solución al problema planteado.

Sin embargo, ya se indicó en su momento que para resolver un problema era muy importante poder descomponerlo en problemas más pequeños, puesto que la resolución de cada uno de estos sería más fácil.

También hay situaciones en las que una tarea tiene que realizarse más de una vez dentro de un programa; en estos casos, interesa escribir el código una única vez, pero poder ejecutarlo tantas veces como sea preciso.

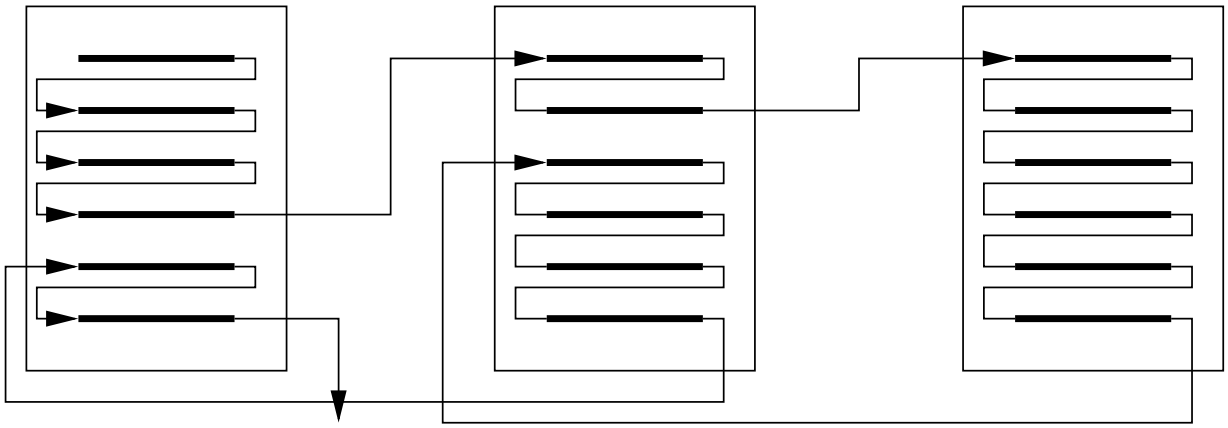
Tanto una cosa (la descomposición en problemas más simples) como la otra (la repetición de una tarea en varias ocasiones), se consigue en C con las funciones. Una **función** no es más que un **conjunto de instrucciones que realiza una determinada tarea**.

Un aspecto importante a considerar en las llamadas a funciones en C es que el hilo de ejecución no se pierde. Cuando desde un punto del programa llamamos a una función, la ejecución sigue en el punto en que hemos

definido la función; cuando esta última termina, la ejecución vuelve al punto desde el que se llamó a la función, continuando la ejecución normal.

Por supuesto, este mecanismo puede repetirse de forma reiterada, de forma que desde una función se puede llamar a una función, en la que a su vez se llame a una función, ... pero siempre manteniendo que tras la finalización de la ejecución de una función, el hilo de ejecución vuelve a la función que hizo la llamada.

Esto puede observarse en la siguiente figura:



En C, las funciones no se pueden anidar; es decir, no puede definirse una función dentro de otra función. Eso significa que todas las funciones están definidas al mismo nivel, en el más exterior dentro de un fichero, y por lo tanto su visibilidad y alcance son los mismos que los de las variables definidas a ese nivel.

7.1.1. La función `main`

Un programa en C no es más que un conjunto de funciones. Entre todas las funciones debe decidirse por qué función se empieza la ejecución. En C, la ejecución siempre debe comenzar por la función **`main`**. Esto quiere decir que cualquier programa que hagamos debe contener al menos la función **`main`**.

De la misma forma, un programa finalizará cuando se termine la ejecución de la función **`main`**.

La función **`main`** debe devolver un valor entero; si no es así, el compilador dará un aviso.

7.2. Definición de una Función

Para poder utilizar funciones dentro de un programa es imprescindible que antes especifiquemos qué tarea va a realizar. Para ello utilizamos lo que se denomina **definición** de la función.

La definición de una función consiste en especificar:

- cómo se llama la función: el **identificador**.
- los valores, y el tipo de dato al que pertenecen, que necesita para su ejecución: los **parámetros**.
- el tipo del valor que devuelve (si es que devuelve algún valor): el **tipo**.
- qué es lo que realmente hace: el **cuerpo**.

El esquema general de la definición de una función es:

```
tipo identificador(tipo1 parametro1, tipo2 parametro2, ...)
{
  cuerpo
}
```

Ejemplo: En la siguiente función, que calcula el mayor de entre dos valores:

```
int calculaMax(int a, int b)
{
  int resultado = a;

  if (b > a)
    resultado = b;

  return resultado;
}
```

el identificador de la función es **`calculaMax`**, devuelve un **`int`**, necesita dos parámetros, ambos de tipo **`int`**, y su cuerpo sería el conjunto de sentencias encerradas entre las llaves.

7.2.1. El identificador

El identificador de una función permite identificar de forma única la función a la que estamos haciendo referencia dentro del programa.

Para formar el identificador de una función se utilizan los mismos criterios de formación que los vistos para los identificadores de variables.

En un programa no puede haber dos funciones con el mismo identificador, a no ser que las funciones se encuentren en ficheros diferentes y ambas tengan restringida su visibilidad mediante la utilización de la cláusula **`static`**.

7.2.2. Los parámetros

Cuando definimos una función para poder reutilizarla, es muy importante disponer de algún mecanismo que nos permita modificar los valores utilizados por la función en sus cálculos. El mecanismo que se utiliza en C para permitir la comunicación entre la función llamante y la función llamada son los **parámetros**.

Ejemplo: Si definimos una función para que devuelva el mayor valor entre dos posibles, necesitamos poder decirle a la función entre qué dos valores queremos que realice la comparación. En el ejemplo anterior, los parámetros serían **`int a`** e **`int b`**.

Cada uno de los parámetros de una función se define exactamente igual a como se definiría una variable: indicando su tipo y un identificador que permita referirse a él en el cuerpo de la función. Los diferentes parámetros de una función van encerrados entre paréntesis y separados por comas.

Los parámetros que se utilizan en la definición de una función se denominan **parámetros formales**. A todos los efectos (visibilidad y tiempo de vida), los parámetros formales de una función son como variables locales definidas en el ámbito de la definición de la función. Eso significa que las variables se crearán al iniciar la ejecución de la función, y se destruirán al finalizar.

7.2.3. El Cuerpo de la Función

El cuerpo de la función lo constituye el bloque en el que se incluirán las sentencias necesarias para que la función realice el objetivo encomendado. Dichas sentencias, como corresponde a cualquier bloque, irán encerradas entre llaves.

Ejemplo: El cuerpo de la función **calculaMax** sería:

```
{
  int resultado = a;

  if (b > a)
    resultado = b;

  return resultado;
}
```

7.2.4. El Tipo de la Función

Las funciones en C pueden devolver como máximo un único valor tras su ejecución. Al tipo de valor que puede devolver es a lo que se denomina **tipo** de la función. El tipo de la función antecede al identificador en la definición. Si la función no devuelve ningún valor, se dice que la función es de tipo **void**.

Ejemplo: El tipo de la función **calculaMax** definida anteriormente sería **int**, puesto que el valor que devuelve (**resultado**) es de dicho tipo.

Para que una función devuelva un valor, se utiliza la sentencia **return**, cuya sintaxis es:

return expresion;

siendo **expresion** una expresión del mismo tipo que la función.

Aunque el estándar permite que se utilice la sentencia **return** en cualquier punto del cuerpo de una función y tantas veces como se desee, la utilización indiscriminada de dicha sentencia hace difícilmente legible el código, por lo que en el ámbito de la asignatura las funciones sólo pueden tener **una** sentencia **return**, que debe ser necesariamente **la última** del cuerpo de la función.

Si es necesario (porque el tipo de la expresión que acompaña a la sentencia **return** no se corresponda con el tipo de la función) debe realizarse una conversión forzada de tipo antes de devolver el valor:

```
float funcion ()
{
  int res;
  .........
  .........
  return (float) res;
}
```

En las funciones de tipo **void**, no se debe utilizar ninguna sentencia **return**.

7.3. Declaración de una Función

Para poder utilizar una función en un programa, es necesario que **antes de su uso** el compilador conozca las características de la función (tipo, identificador y parámetros).

Este cometido lo cumple perfectamente la definición de la función; sin embargo no siempre es posible tener la definición de una función antes de cada llamada (puesto que la definición sólo puede hacerse en un punto del programa, pero puede utilizarse en muchos), por lo que se utiliza la **declaración** de la función. A la declaración de una función también se le denomina **prototipo**.

La declaración de una función se hace de la misma forma que la definición, pero sustituyendo el cuerpo de la misma por el carácter de fin de sentencia **;**.

Ejemplo: La declaración de la función **calculaMax** anterior es:

```
int calculaMax(int a, int b);
```

El lenguaje permite que en los parámetros se especifique sólo el tipo; sin embargo, en el ámbito de la asignatura, y para mayor claridad, la declaración de una función debe ser idéntica a su definición (exceptuando el cuerpo, claro está).

Hay una excepción a esta regla, en caso que la función no admita parámetros. En ese caso es necesario indicarlo en la declaración utilizando la palabra reservada **void**, en lugar de no poner nada entre los paréntesis.

Ejemplo: La declaración de una función **menu**, que no devuelve nada y no admite parámetros, sería:

```
void menu(void);
```

La declaración de una función, al contrario que la definición, puede aparecer tantas veces como sea necesario dentro de un programa. Es además conveniente que la declaración aparezca antes que la propia definición.

7.4. Utilización de una Función

Para utilizar una función (o hacer una llamada a la función), escribiremos el identificador de la función, y colocaremos entre paréntesis los **valores** que deseemos para los parámetros. Los valores de los parámetros que utilizamos en la llamada se denominan **parámetros reales** de la función. En general, podremos utilizar como parámetro cualquier expresión del mismo tipo que el parámetro formal correspondiente.

Ejemplo: Podrían ser llamadas a la función **calculaMax** declarada y definida anteriormente (supuesto que **uno** y **dos** son variables de tipo **int**):

```
calculaMax(12, 5);
calculaMax(uno, dos);
calculaMax(3 * uno, 5 * dos);
```

Si la función devuelve algún valor (es decir si su tipo es distinto de **void**), la función **puede** utilizarse en expresiones en los mismos lugares en los que podría utilizarse una expresión de su mismo tipo.

Ejemplo: Así, en el caso que nos ocupa, podríamos poner:

```
uno = 2 * calculaMax(3 * uno, 5 * dos)
```

7.4.1. Correspondencia de parámetros

Cuando se realiza una llamada a una función, antes de comenzar su ejecución se evalúan las expresiones que forman los parámetros reales, y los valores obtenidos se asignan a las variables que representan los parámetros formales. Sólo entonces comienza la ejecución de la función.

La asignación entre parámetros reales y parámetros formales se realiza siempre según el orden en el que aparecen tanto en la llamada como en la definición: el primer parámetro real de la llamada se asignará al primer parámetro formal de la definición, el segundo al segundo, y así sucesivamente. A esto se le denomina correspondencia posicional.

Por tanto es imprescindible que el número de parámetros reales coincida exactamente con el de parámetros formales, debiendo también coincidir en el orden y en el tipo de ambas clases de parámetros.

Ejemplo: En el siguiente programa:

```
#include <stdio.h>

/* Declaracion de la funcion */
int calculaMax(int a, int b);

int main()
{
  int i = 7;
  int j = 4;
  int maximo;

  maximo = calculaMax(i, j);
  printf("Maximo de %d y %d es %d\n", i, j, maximo);
  maximo = calculaMax(2 * i, 4 * j);
  printf("Maximo de %d y %d es %d\n", 2 * i, 4 * j, maximo);

  return 0;
}

/* Definicion de la funcion */
int calculaMax(int a, int b)
{
  int resultado = a;

  if (b > a)
    resultado = b;

  return resultado;
}
```

tenemos dos llamadas a la función **calculaMax**. En la primera invocación tendríamos la siguiente ejecución equivalente:

```
int calculaMax()
{
  int a = 7;
  int b = 4;
  int resultado = a;

  if (b > a)
    resultado = b;

  return resultado;
}
```

mientras que en la segunda tendríamos:

```
int calculaMax()
{
  int a = 14;
  int b = 16;
  int resultado = a;

  if (b > a)
    resultado = b;

  return resultado;
}
```

Del ejemplo anterior podemos extraer dos consecuencias muy importantes:

- La primera es que puesto que los parámetros formales actúan como variable locales, los cambios que hagamos sobre los parámetros formales en la función llamada no tienen ningún efecto sobre los valores de las variables utilizadas como parámetros reales en la llamada a la función, puesto que dichos cambios se están realizando sobre una copia del valor de los parámetros reales.

  Podríamos reescribir entonces la función **calculaMax** de la siguiente forma:

```
int calculaMax(int a, int b)
{
  if (b > a)
    a = b;

  return a;
}
```

- La segunda consecuencia es que, puesto que vamos a utilizar los parámetros reales para realizar una asignación sobre los parámetros formales, sólo podremos utilizar como parámetros aquellos elementos que pueden aparecer en el lado derecho de una asignación.

7.5. Recursividad

Ya sabemos que una función en C puede realizar llamadas a otras funciones. Un caso particular de esta técnica es aquel en el que desde una función se realiza una llamada a la propia función. A este caso se le denomina **recursividad**. La recursividad es posible en C porque en cada llamada a una función, esta copia los valores de los parámetros reales en los parámetros formales, por lo que cada llamada está utilizando sus propios valores.

La técnica de la recursividad se aplica a problemas que pueden definirse de modo natural de forma recursiva.

Ejemplo: Consideremos el caso de la función factorial: la definición del factorial de un número, **fact(n)** se define como:

$$\begin{array}{rl} \mathtt{1} & \text{si } \mathtt{n = 0} \\ \mathtt{n * fact(n - 1)} & \text{si } \mathtt{n > 0} \end{array}$$

Vemos que la especificación de la función hace uso de la definición de la misma, es decir: la función se llama a sí misma.

No todos los problemas admiten soluciones recursivas, y si la admiten muchas veces no son inmediatas. Sí podemos asegurar que toda solución recursiva tiene una solución iterativa equivalente, aunque no siempre es inmediato obtenerla.

En las definiciones de funciones recursivas es **imprescindible** que exista un **CASO BASE** que permita la terminación de la propia función y devuelva el control a la función que realizó la llamada. En caso contrario, el programa entraría en lo que se denomina un bucle infinito: la función estaría llamándose a sí misma de forma indefinida. Para que pueda aplicarse el caso base, es necesario que en cada una de las llamadas a sí misma que haga una función se modifiquen los valores de los parámetros.

Ejemplo: En el caso del factorial el caso base sería el factorial de 1, como se refleja en la definición.

Aunque el código de una función que utiliza técnicas recursivas pueda ser más corto, habitualmente consume más memoria durante la ejecución, y puede llegar a ser más lento en la ejecución, debido a que en cada llamada que la función se hace a sí misma es necesario copiar valores en variables y estas ocupan espacio en memoria, tanto más cuanto mayor sea el número de llamadas.

Ejemplo: Para ver el funcionamiento paso a paso de una función recursiva, estudiaremos la siguiente función, que obtiene el máximo común divisor de dos números enteros:

```
int mcd(int a, int b)
{
  int res;

  if (a == b)
    res = a;
  else if (a > b)
    res = mcd(b, a - b);
  else
    res = mcd(a, b - a);

  return res;
}
```

Si se llama a esta función con los parámetros reales **36** y **52** (en este orden), tendríamos la siguiente secuencia de ejecución:

	a	**b**	**Condición**	**Nueva llamada**
1ª llamada	36	52	a < b	**(a, b - a)**
2ª llamada	36	16	a > b	**(b, a - b)**
3ª llamada	16	20	a < b	**(a, b - a)**
4ª llamada	16	4	a > b	**(b, a - b)**
5ª llamada	4	12	a < b	**(a, b - a)**
6ª llamada	4	8	a < b	**(a, b - a)**
7ª llamada	4	4	a = b	caso base

7.6. Compilación Separada y Descomposición en Ficheros

Cuando un programa es muy grande y tiene muchas funciones, suele ser inmanejable tenerlas todas en un único fichero. Para evitar este inconveniente, C permite que un programa pueda dividirse en varios ficheros, dando lugar al concepto de **compilación separada**.

La compilación separada hace referencia al caso en que tengamos varios ficheros fuente para generar un ejecutable. Es un concepto distinto al de la modularidad, aunque muy relacionado: la compilación separada implica necesariamente modularidad, pero un programa modular puede estar compuesto por un único fichero, y por lo tanto no le es de aplicación la compilación separada.

La división del programa en varios ficheros tiene dos considerables ventajas:

- El programa queda más estructurado y organizado.
- Si se necesita modificar un fichero, sólo se compilaría el modificado, para posteriormente enlazarlo con el resto de ficheros objeto.

Cada fichero en los que se descompone un programa fuente conforma lo que se denomina una **unidad de compilación**. Las unidades de compilación deben poder ser compiladas separadamente.

Ejemplo: El programa visto en **`ejemplo.c`** podríamos descomponerlo en las siguientes unidades de compilación:

```
int main()
{
  int i = 7;
  int j = 4;
  int maximo;
```

```
  maximo = calculaMax(i, j);
  printf("Maximo de %d y %d es %d\n", i, j, maximo);
  maximo = calculaMax(2 * i, 4 * j);
  printf("Maximo de %d y %d es %d\n", 2 * i, 4 * j, maximo);

  return 0;
}
```

```
/* Definicion de la funcion */
int calculaMax(int a, int b)
{
  if (b > a)
    a = b;

  return a;
}
```

7.6.1. Funciones y ficheros de cabecera

Puesto que cada unidad de compilación debe poder compilarse por separado, en cada uno de los ficheros debemos tener las declaraciones de las funciones que se utilizan en el fichero antes de su uso.

Sin embargo, en este caso tendremos dispersas las definiciones de las diferentes funciones que conforman el programa. Eso nos obliga a tener que usar la declaración de las funciones en diferentes sitios. Para evitar tener que repetir dichas declaraciones en cada uno de los ficheros en los que se utiliza cada función, lo que se hace es asociar un fichero de cabecera (fichero con extensión **.h**) a cada fichero fuente (con extensión **.c**) en que hayamos dividido nuestro programa. La única excepción podría ser el fichero que contiene la definición de la función **main**.

En dicho fichero de cabecera se incluirán las declaraciones o prototipos de todas las funciones definidas en el correspondiente fichero fuente y que puedan ser utilizadas fuera del fichero en que se definen. Las funciones definidas en un fichero de código que no vayan a ser utilizadas en otros ficheros no deben aparecer en el fichero de cabecera, y además deberán etiquetarse con la palabra reservada **static** para limitar su visibilidad.

Ejemplo: El fichero de cabecera correspondiente al ejempo del cálculo del valor máximo sería:

```
int maximo(int a, int b);
```

Una vez creado el fichero de definiciones de funciones y el correspondiente fichero de cabecera, se deberá incluir este último:

- en el propio fichero de definiciones, para que el compilador pueda comprobar que los prototipos se corresponden con las definiciones.
- en los ficheros que utilicen dichas funciones.

Para realizar esta inclusión se utilizan las directivas del preprocesador de C.

7.7. El preprocesador de C.

El preprocesador de C trabaja con una serie de instrucciones especiales (directivas) que se ejecutan antes del proceso de compilación. El preprocesado es un paso previo a la compilación, aunque la llamada al preprocesador la realiza el mismo compilador, siendo completamente transparente al programador.

Existen varios tipos de directivas del preprocesador, de los cuales sólo veremos tres:

- el de inclusión de ficheros.
- el de definición de macros.
- el de compilación condicional.

Las directivas del preprocesador de C empiezan siempre por el carácter **#**.

7.7.1. La directiva `include`

La inclusión de archivos consigue que el proceso de compilación separada sea mucho más robusto y eficiente, al permitir que algo que debe utilizarse en varios ficheros (como serían las declaraciones de las funciones) se escriba una única vez en un fichero, y se utilice siempre ese fichero.

La directiva **`#include`** tiene dos formatos, dependiendo de cómo se encierre el fichero a incluir:

- si se utilizan los ángulos (**<**, **>**) se busca el fichero en los directorios predefinidos por la instalación.

  Ejemplo: Para incluir el fichero de cabecera por excelencia, **`stdio.h`**, tendríamos que utilizar la instrucción:

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

- si se utilizan las comillas dobles (**"**), se busca el fichero primero en el directorio de trabajo, y si no lo encuentra lo busca en los directorios predefinidos por la instalación.

  Ejemplo: Para incluir el fichero de cabecera **`cuentas.h`**, situado en el subdirectorio **`listas`**, pondríamos:

  ```
  #include "listas/cuentas.h"
  ```

La directiva **`#include`** puede utilizarse tanto en ficheros de código (**`.c`**) como en ficheros de cabecera (**`.h`**), pero **SÓLO** puede utilizarse para incluir ficheros de cabecera, **NUNCA** ficheros de código.

7.7.2. La directiva `define`

La directiva **`define`** permite definir **macros**, un mecanismo simple de sustitución para facilitar la mantenibilidad de un programa.

La macro hace coincidir un identificador (el *nombre* de la macro) con una secuencia de caracteres, según la siguiente estructura:

```
#define NOMBRE texto de reemplazo
```

La formación del nombre de la macro sigue las reglas de formación de identificadores en funciones y variables. Por convenio, los nombres de las macros irán siempre en mayúsculas. El texto de reemplazo comienza tras el último espacio que sigue al nombre, hasta el carácter de fin de línea; si se desea que el texto de reemplazo ocupe más de una línea, se debe terminar cada línea menos la última con el carácter de escape (****).

Realizan la sustitución del nombre indicado en la macro por la expresión indicada. Los cambios tienen efecto desde el momento de la definición hasta el final del archivo. La sustitución no se lleva a cabo dentro de cadenas entrecomilladas ni en partes de un elemento.

Las macros se suelen utilizar para definir símbolos que representes entidades que se derivan de la especificación del programa.

Ejemplo: Las macros nos permiten eliminar constantes que se utilizan en varios lugares de un programa. Por ejemplo, si queremos limitar la longitud de los nombres de ficheros a 8 caracteres, podemos poner:

```
#define LON_NOM_FIC  8
```

Si cambiamos de máquina y queremos modificar la longitud de los nombres de ficheros a 15 caracteres, sólo tendremos que modificar una línea (la de la definición de la macro), en lugar de todas las líneas en las que se utiliza dicho valor.

Macros con argumentos

Las macros también pueden tener argumentos, de forma que en cada sustitución pueda modificarse el texto de reemplazo.

En las macros con argumentos, el nombre va seguido de los argumentos entre paréntesis y separados por coma. En el texto de reemplazo, los argumentos van encerrados entre paréntesis, y en cada sustitución toman la forma especificada en la llamada.

```
#define NOMBRE(lista argumentos) texto reemplazo
```

Ejemplo: Si tenemos el siguiente fragmento de código:

```
#define MAX(A,B)     ((A) > (B) ? (A) : (B))
#define CUADRADO(x)  (x) * (x)
...
    x = MAX(p + q, r + s);
    i = CUADRADO(j + 1);
```

Tendríamos las siguientes sustituciones:

```
    x = ((p + q) > (r + s) ? (p + q) : (r + s));
    i = (j + 1) * (j + 1);
```

En este último caso puede observarse la importancia de los paréntesis rodeando los parámetros en el texto de reemplazo. Si no existieran dichos paréntesis, tendríamos la siguiente sustitución:

```
    i = j + 1 * j + 1;
```

que no es, obviamente, lo que se desea.

La definición de una macro puede hacerse tan compleja como se quiera; sin embargo, su uso debe restringirse lo más posible, puesto que puede hacer que un programa sea inmanejable, generando efectos secundarios indeseables y difícilmente detectables. En el ámbito de la asignatura, las macros con parámetros **NO PUEDEN USARSE**.

Anulación de macros

Si no se desea que una macro definida previamente tenga validez hasta el final del fichero, puede anularse la definición con la directiva **undef**:

```
#undef NOMBRE
```

siendo *NOMBRE* el identificador de la macro definida con anterioridad.

7.7.3. Directivas condicionales

Las directivas condicionales permiten tomar decisiones en tiempo de compilación que dependan de la definición de alguna macro.

```
#if expresion entera constante
  ...
#elif expresion entera constante 2
  ...
#elif expresion entera constante 3
  ...
#else
  ...
#endif
```

Las cláusulas **#elif** y **#else** son opcionales. Una directiva **#if** puede ser seguida por cualquier número de directivas **#elif**, pero sólo puede haber una **#else**.

En la expresión entera se pueden usar:

- macros del preprocesador (en ningún caso variables del programa) y constantes
- operadores aritméticos, de comparación y lógicos.
- el operador **defined**, que toma como argumento el nombre de una macro y devuelve 0 si no está definida y distinto de cero si sí lo está.

Si el resultado de la expresión es distinto de cero se ejecutan las siguientes líneas hasta llegar a un **#elif**, **#else** o **#endif**. Si es cero, se pasa a la siguiente directiva **#elif**, **#else** o **#endif** que aparezca.

Ejemplo: El siguiente fragmento visualiza en la salida estándar, en función del valor de la macro **DEPURANDO**, diferentes mensajes que permitan seguir la ejecución del programa.

```
#if DEPURANDO > 1
  printf("Estoy en la funcion prueba y voy a calcular %d\n", valor);
#elif DEPURANDO == 1
  printf("Estoy en prueba\n");
#endif
```

Existen dos directivas condicionales adicionales:

- **#ifdef NOMBRE**, que es equivalente a **#if defined(NOMBRE)**
- **#ifndef NOMBRE**, que es equivalente a **#if !defined(NOMBRE)**

Ejemplo: Usaremos las directivas condicionales para evitar lo que se denomina doble inclusión de los ficheros de cabecera. Todos los ficheros de cabecera que escribamos deben comenzar por una directiva **#ifndef** seguida de una directiva **#define**. El fichero de cabecera siempre finalizará con la directiva **#endif**:

```
#ifndef CABECERA_H
#define CABECERA_H
...
#endif
```

El nombre de la macro que utilizaremos será el del fichero en mayúsculas, substituyendo el **.** por _. La primera vez que en una compilación incorporemos el fichero de cabecera, la macro no estará definida, con lo cual la definiremos y continuaremos con el fichero. Las sucesivas veces que se intente incluir el mismo fichero, la macro ya estará definida y no tendrá ningún efecto.

7.8. Resultado final

Después de todo lo visto, el programa que calcula el valor máximo de dos enteros estaría formado por los siguientes ficheros:

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

int main()
{
  int i = 7;
  int j = 4;
  int maximo;

  maximo = calculaMax(i, j);
  printf("Maximo de %d y %d es %d\n", i, j, maximo);
  maximo = calculaMax(2 * i, 4 * j);
  printf("Maximo de %d y %d es %d\n", 2 * i, 4 * j, maximo);

  return 0;
}
```

```
#ifndef MAXIMO_H
#define MAXIMO_H

int calculaMax(int a, int b);

#endif
```

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

/* Definicion de la funcion */
int calculaMax(int a, int b)
{
  if (b > a)
    a = b;

  return a;
}
```

Tema 8

Punteros

Índice

8.1. Introducción **8-1**
8.1.1. Declaración de Punteros 8-2
8.1.2. El valor NULL en los Punteros 8-3
8.2. Operadores Específicos de Punteros **8-3**
8.2.1. Prioridades de los Operadores de Punteros 8-4
8.3. Aritmética de punteros **8-5**
8.3.1. Operaciones entre punteros y enteros 8-5
8.3.2. Resta de dos Punteros 8-6
8.3.3. Comparaciones de Punteros 8-6
8.4. Punteros a Punteros **8-7**
8.5. Uso de Punteros para Devolver más de un Valor **8-7**

8.1. Introducción

Como ya hemos visto, toda variable que se utilice en un programa en C ocupará una cierta cantidad de memoria.

Para conocer el valor de una determinada variable, el ordenador necesita tres datos:

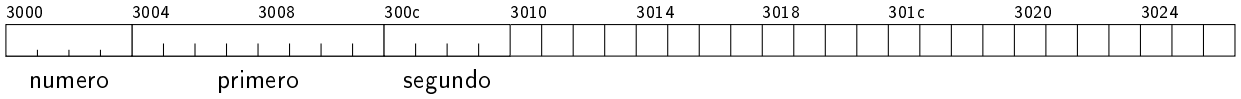
- en qué parte de la memoria está guardada esa variable.
- qué tamaño tiene.
- cómo está codificada.

Podemos ver la memoria del ordenador como una secuencia de octetos en los que la máquina va almacenando las variables que va a usar durante la ejecución de un determinado programa.

Por ejemplo, si tenemos una máquina en la que el tamaño de un tipo **int** es de 4 octetos, y un **float** de 8 octetos, las siguientes declaraciones:

```
int   numero;
float primero;
int   segundo;
```

podría reflejarse en la memoria de la máquina de la siguiente manera:



A la vista de la figura podemos observar que la variable **numero** comienza en la posición de memoria **0x3000**, la variable **primero** en la posición **0x3004**, y la variable **segundo** en la posición **0x300c**.

En C existe el concepto de **puntero**. Un puntero no es más que un valor que especifica una dirección de memoria. El puntero siempre tiene un tipo asociado, que es el de la variable a la que apunta. Es decir, que desde el punto de vista del C, un puntero a entero tiene un tipo distinto de un puntero a punto flotante, aunque tengan el mismo valor.

8.1.1. Declaración de Punteros

Para declarar un puntero en C utilizamos el carácter *****. Este carácter debe acompañar al tipo de variable al que apunta.

Ejemplo: Si queremos declarar un puntero a una variable de tipo entero, pondremos:

```
int * pEntero;
```

mientras que si queremos un puntero a una variable de tipo float pondremos:

```
float * pReal;
```

El objetivo de un puntero es poder acceder a una determinada variable. Por eso, cuando declaramos un puntero tenemos necesariamente que especificar de qué tipo es la variable a la que apunta el puntero. Esto además determina el tipo del puntero, de forma que dos punteros son del mismo tipo de datos si y sólo si apuntan a variables del mismo tipo.

Cuando un puntero almacena la dirección de comienzo de una variable, se dice que el puntero *apunta* a la variable.

Un tipo especial de puntero es el que se declara como puntero a tipo **void**:

```
void * puntero;
```

Es una forma de indicar que sólo sabemos que es un puntero, pero que no sabemos a qué tipo de variable apunta.

Vimos en un tema anterior que para poder utilizar una variable en un programa en C, primero debíamos declararla, lo que permitía a la máquina reservar memoria para dicha variable. Es importante destacar aquí que declarar un puntero a un determinado tipo **NO equivale a declarar una variable de dicho tipo**.

Ejemplo: Es decir, si en un programa hacemos la siguiente declaración:

```
int * pEntero;
```

eso no supone que hayamos reservado memoria para una variable de tipo **int**, sino para un puntero a una variable de dicho tipo. Si queremos una variable de tipo **int** tendremos que declararla de forma explícita.

En C, el tamaño de un puntero no está definido, sino que puede variar entre dos máquinas. Para conocer el tamaño de un puntero en una máquina determinada, podemos utilizar el operador **sizeof**, como en el siguiente programa:

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

```
int main()
{
  printf("El tamano de un puntero es %d\n", sizeof(char *));

  return 0;
}
```

Si lo compilamos y ejecutamos, obtendremos la siguiente salida:

```
salas@318CDCr12:~$ gcc -W -Wall -o tamPuntero tamPuntero.c
salas@318CDCr12:~$ ./tamPuntero
El tamano de un puntero es 4
salas@318CDCr12:~$
```

8.1.2. El valor NULL en los Punteros

Uno de los principales errores que se pueden cometer cuando se programa en C es utilizar punteros que no apuntan realmente a una variable. El principal problema de esta circunstancia es que es posible que los efectos de dicho error no se hagan visibles de forma inmediata, dificultando mucho encontrar dicho error y corregirlo. Para evitar esos problemas, es importante que un puntero **SIEMPRE** apunte a una variable. Hay circunstancias en las que no es posible saber en el momento de la declaración a qué variable tiene que apuntar el puntero; en esos casos, debe utilizarse un valor especial, **NULL**, como valor inicial de los punteros.

Ejemplo: Siguiendo esas indicaciones, la forma correcta de declarar los tres punteros anteriores, sería:

```
int   * pEntero = NULL;
float * pReal   = NULL;
void  * puntero = NULL;
```

Obsérvese que se utiliza el mismo valor para inicializar los tres tipos de punteros, porque los tres, independientemente de su tipo, contienen la misma información: una dirección de memoria.

No lo olvide: **TODOS** los punteros deben ser inicializados en su declaración a la dirección de memoria de la variable adecuada, o si no se conoce, al valor **NULL**.

El especificador de formato de la función **printf** para los punteros es **%p**, obteniendo el valor del puntero en hexadecimal. No hay ningún especificador de formato para **scanf**: no tiene sentido, y además es extremadamente peligroso, introducir el valor de un puntero por teclado.

8.2. Operadores Específicos de Punteros

Existen dos operadores específicos de manejo de punteros. Los dos son operadores prefijos unarios:

- **&**: operador de dirección. Se aplica como prefijo a cualquier **variable**, y devuelve la dirección en la que se encuentra almacenada la variable.

- *****: operador de contenido. Se aplica como prefijo a un **puntero**, y devuelve el valor contenido en la dirección de memoria almacenada en el puntero.

  Este operador tiene sentido porque en la declaración del puntero hemos especificado de qué tipo es la variable a la que apunta. Si no fuera así, el resultado de este operador sería indeterminado. De hecho, este operador no puede aplicarse si el puntero es **void ***, puesto que no puede saberse cómo está codificada la información en memoria.

Ejemplo: El siguiente programa es un ejemplo de utilización de los operadores de manejo de punteros:

```
#include <stdio.h>

int main ()
{
  int   origen  = 0;
  int * pOrigen = &origen;    /* pOrigen apunta a origen */

  printf("Origen vale %d, y esta en la direccion %p\n",
         origen, pOrigen);
  /* Podemos usar el puntero para ver el valor */
  printf("Origen vale %d\n", *pOrigen);
  origen = 1;
  /* Los cambios en el valor de la variable tambien se ven
     si usamos el puntero */
  printf("Origen vale %d\n", *pOrigen);
  /* Tambien podemos usar el puntero para cambiar el valor
     de la variable */
  *pOrigen = 2;
  /* Y el valor de la variable cambia */
  printf("Origen vale %d\n", origen);

  return 0;
}
```

Si lo compilamos y ejecutamos, obtenemos el siguiente resultado:

```
salas@318CDCr12:~$ gcc -W -Wall -o punteros punteros.c
salas@318CDCr12:~$ ./punteros
Origen vale 0, y esta en la direccion 0xbfa3ad08
Origen vale 0
Origen vale 1
Origen vale 2
salas@318CDCr12:~$
```

El valor del puntero puede variar de una máquina a otra, y dentro de la misma máquina, de una ejecución a otra, porque en cada momento reflejará en qué posición de memoria está la variable **origen**, y eso puede cambiar. Lo que no variará nunca es en una misma ejecución.

Observe que el operador **&** se aplica siempre a una variable, mientras que el operador ***** puede aplicarse a cualquier expresión de tipo puntero.

Ejemplo: Sería incorrecto por tanto escribir:

```
&(suma + 2);
```

porque estaríamos aplicando el operador de dirección a una expresión, mientras que sería totalmente correcto escribir:

```
*(&suma);
```

El resultado de esta última sentencia es, lógicamente, el valor de **suma**.

8.2.1. Prioridades de los Operadores de Punteros

Hemos visto que los operadores de punteros son operadores unarios, y como tal se comportan en temas de prioridad: presentan el segundo nivel de prioridad, y se evalúan de derecha a izquierda.

8.3. Aritmética de punteros

Existe una cierta aritmética que es aplicable a los punteros, y que difiere con respecto a la aritmética vista para los enteros o reales.

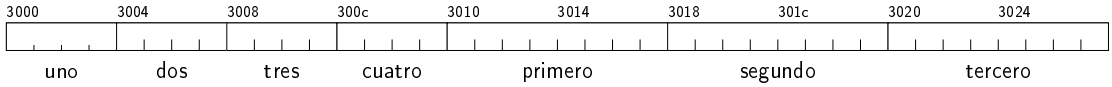
Las operaciones aritméticas admisibles para los punteros son:

- sumarle o restarle un entero, obteniendo como resultado un puntero del mismo tipo.
- restar dos punteros del mismo tipo, obteniendo como resultado un entero.

8.3.1. Operaciones entre punteros y enteros

La diferencia de la aritmética de punteros con respecto a la vista anteriormente es que los punteros se incrementan y decrementan en función del tipo de datos al que apuntan. La idea consiste en que si se le suma uno a un puntero, el nuevo valor del puntero apuntaría al siguiente elemento en la memoria.

Ejemplo: Supongamos el siguiente mapa de memoria en una máquina en la que los tipos **int** ocupan 4 octetos y los tipos **float** ocupan 8 octetos:



Si efectuamos la siguiente operación:

```
int * pEntero = &uno;
```

el valor del puntero será **0x3000**. Si le sumamos uno al puntero, y puesto que apunta a un tipo **int** que ocupa cuatro octetos, el resultado será que el puntero incrementará su valor en cuatro unidades, pasará a valer **0x3004** y ahora apuntará a **dos**. Si ahora le sumamos 2, incrementará su valor en ocho unidades, pasará a valer **0x300c** y apuntará a **cuatro**.

Si ahora volvemos a incrementar el puntero en una unidad, volverá a aumentar su valor en cuatro unidades, pasará a valer **0x3010**, y ahora apuntará a **primero**. Observe que ahora el puntero apunta a una variable de tipo **float**, pero **sigue siendo** un puntero a **int**. Por eso, si volvemos a sumarle 1, el puntero incrementará su valor en **4** unidades (las correspondientes al tamaño de un **int**, que es lo que refleja su tipo), pasando a valer **0x3014**, y no en 8 unidades (que sería el tamaño del **float** al que está apuntando en ese momento). Como resultado, el puntero apuntará a una zona de memoria que no se corresponde con ninguna variable.

Si con el mismo mapa de memoria efectuamos la siguiente operación:

```
float * pReal = &primero;
```

el valor del puntero será **0x3010**. Si le sumamos uno al puntero, y puesto que apunta a un tipo **float** que ocupa ocho octetos, el puntero incrementará su valor en ocho unidades, pasando a valer **0x3018** y apuntará a **segundo**.

Lo expresado para la suma es aplicable, lógicamente, al operador de autoincremento (**++**) y al operador de asignación compacta con el operador de suma (**+=**).

La operación de restar un entero a un puntero es muy similar (restar un número no es más que sumar un número negativo), de forma que cada vez que decrementamos en una unidad un puntero, su valor se ve reducido en el tamaño del tipo al que apunta.

Igual que antes, lo dicho para la resta de un entero es aplicable al operador de autodecremento (**--**) y al operador de asignación compacta con el operador de resta (**-=**).

Ejemplo: Siguiendo con la figura anterior, si tenemos los siguientes declaraciones:

```
int   * pEntero = &tres;
float * pReal   = &segundo;
```

pEntero valdrá **0x3008**, y **pReal** valdrá **0x3018**. Tras ejecutar:

```
pEntero -= 2;
pReal--;
```

el valor de **pEntero** será **0x3000** (apuntará a **uno**), y el valor de **pReal** será **0x3010** (apuntará a **primero**).

8.3.2. Resta de dos Punteros

Si de la suma de un puntero y un entero obtenemos un nuevo puntero del mismo tipo, de la resta de dos punteros (que deben ser del mismo tipo) obtendremos un entero.

Ejemplo: Sea de la expresión:

```
int * pAntiguo = &uno;
int * pNuevo   = NULL;

pNuevo = pAntiguo + 2;
```

si consideramos la sentencia anterior como una expresión algebraica, y pasamos **pAntiguo** al primer miembro, nos queda la expresión:

$$pNuevo - pAntiguo = 2$$

Es decir, la diferencia entre dos punteros es el número de veces que cabe entre los dos punteros el tipo de datos al que apuntan.

Recuerde que es imprescindible para poder realizar la operación que los dos punteros apunten al mismo tipo de datos. Además, las direcciones a las que apuntan los dos punteros deben estar separadas un número exacto de veces el tamaño del tipo al que apuntan; si no fuera así, el resultado de la operación es indeterminado.

8.3.3. Comparaciones de Punteros

Los punteros apuntan a una variable. Su valor, por tanto, es la dirección correspondiente a dicha variable, y no el propio valor de la variable.

Ejemplo: En el siguiente código:

```
int   uno  = 1;
int   dos  = 1;
int * pUno = &uno;
int * pDos = &dos;
```

si realizamos la comparación **pDos == pUno** obtendremos como resultado **0**, puesto que el valor de **pDos** es la dirección de la variable **dos**, y el valor de **pUno** es la dirección de la variable **uno**, que evidentemente son diferentes (cada variable ocupará una posición diferente en la memoria).

Si lo que queremos es comparar los valores de las variables, deberemos utilizar el operador de contenido.

Ejemplo: Si queremos comparar los valores de las variables **uno** y **dos** utilizando los punteros, tendríamos que utilizar la expresión ***pDos == *pUno**.

Lo mismo sucede para el resto de los operadores relacionales vistos con anterioridad.

8.4. Punteros a Punteros

Si declaramos una variable con la siguiente sentencia:

```
int * pEntero = NULL;
```

estamos declarando una variable, de nombre **pEntero**, y de tipo puntero a **int**. Es decir, que **pEntero** es una variable, y como a tal se le puede aplicar el operador de dirección, **&pEntero**, obteniendo la dirección de un puntero a **int**, es decir un puntero a puntero a **int**. Para declarar una variable de este tipo tendremos que utilizar dos veces el carácter *****:

```
int ** pPunteroEntero = NULL;
```

Esta operación podemos repetirla tantas veces como queramos, obteniendo cada vez un nivel más de indirección. Por cada nivel de indirección tendremos que utilizar un carácter ***** en la declaración de una variable.

Ejemplo: Así, si nos encontramos en un programa con esta declaración:

```
float *** pPuntPuntReal = NULL;
```

estaremos declarando una variable de nombre **pPuntPuntReal**, y de tipo puntero a puntero a puntero a **float**. Esta variable podemos utilizarla de diferentes formas en nuestro código, obteniendo diferentes tipos de datos. Así:

- si utilizamos **pPuntPuntReal** tendremos un tipo puntero a puntero a puntero a **float**.
- si utilizamos ***pPuntPuntReal** tendremos un tipo puntero a puntero a **float**.
- si utilizamos ****pPuntPuntReal** tendremos un tipo puntero a **float**.
- Por último, si utilizamos *****pPuntPuntReal** tendremos un tipo **float**.

Una regla muy simple para saber de qué tipo es una expresión con varios operadores de contenido aplicados a una variable con diversos grados de indirección es utilizar la declaración de dicha variable, y separar en ella la parte que nos encontramos en la expresión; lo que queda de la declaración es el tipo de la expresión.

Ejemplo: Utilizando el ejemplo anterior, cuya declaración era:

```
float *** pPuntPuntReal = NULL;
```

si en un programa nos encontramos con la expresión ****pPuntPuntReal**, aislamos esta parte en su declaración:

```
float *[** pPuntPuntReal] = NULL;
```

y lo que nos queda, **float *** sería el tipo de dicha expresión, es decir, un puntero a **float**.

El utilizar varios niveles de punteros hace que el código sea menos legible, por lo que debe evitarse su uso siempre que sea posible.

8.5. Uso de Punteros para Devolver más de un Valor

Puesto que la manipulación de los parámetros formales dentro de una función no tiene ningún efecto sobre los valores de los parámetros reales, una función sólo puede devolver un valor en cada llamada.

Sin embargo, hay circunstancias en las que se hace necesario que una función devuelva más de un valor (o lo que es equivalente, que modifique el valor de más de una variable). Por ejemplo, supongamos que queremos implementar una función que intercambie el valor de dos variables. Con lo visto hasta ahora, sería imposible en C.

Existe un método que nos permite resolver este problema. Para ello debemos pasar como parámetro **la dirección de la variable** que queramos modificar, y operar dentro de la función con el **contenido** de los parámetros formales. Lo que hacemos es pasarle la dirección de la variable: es decir, le estamos diciendo a la función dónde está el valor que queremos intercambiar, no cuál es el valor. Las modificaciones que se realicen en la función al contenido de los parámetros formales sí tienen efecto al finalizar la función, puesto que lo que deja de tener validez es la copia del puntero que se ha utilizado en la función.

Ejemplo: Una función que intercambia los valores de dos variables podría ser:

```
#include <stdio.h>

/* Esta definicion hace al mismo tiempo de declaracion */
void intercambio (int *x, int *y)
{
  int aux;

  aux = *x;
  *x = *y;
  *y = aux;
}

int main ()
{
  int a = 7;
  int b = 5;

  printf("Antes de intercambio (a,b)=(%d,%d)\n", a, b);
  intercambio(&a, &b);
  printf("Despues de intercambio (a,b)=(%d,%d)\n", a, b);

  return 0;
}
```

Si compilamos y ejecutamos el programa, tendremos el siguiente resultado:

```
salas@318CDCr12:~$ gcc -W -Wall -o intercambio intercambio.c
salas@318CDCr12:~$ ./intercambio
Antes de intercambio (a,b)=(7,5)
Despues de intercambio (a,b)=(5,7)
salas@318CDCr12:~$
```

Si pasáramos las variables como parámetros reales, las manipulaciones que la función realizara las haría sobre los parámetros formales, y las variables no se habrían modificado. Sin embargo, al pasarle la dirección de las variables actuamos sobre los valores apuntados por los parámetros (y por eso es necesario utilizar el operador de contenido, *****, dentro de la función), no sobre los propios parámetros.

Tema 9

Tablas

Índice

9.1. Introducción **9-1**
9.2. Declaración de las Tablas **9-2**
9.2.1. Inicialización de Tablas de Caracteres 9-2
9.3. Acceso a los Elementos de una Tabla **9-3**
9.4. Recorrer los Elementos de una Tabla **9-4**
9.5. Utilización de Constantes Simbólicas **9-4**
9.6. Punteros y Tablas **9-5**
9.6.1. Diferencias y Usos 9-7
9.7. Tablas como Resultado de una Función **9-8**
9.8. Tablas Multidimensionales **9-8**
9.8.1. Inicialización de Tablas Multidimensionales 9-9
9.9. Tablas de punteros **9-9**
9.9.1. Tablas de punteros y tablas multidimensionales 9-10
9.10. Tablas como Parámetros de Funciones **9-10**
9.10.1. Tablas Multidimensionales como Parámetros 9-11
9.10.2. Tablas de Punteros como Parámetros 9-13
9.10.3. Argumentos en la Línea de comandos 9-13

9.1. Introducción

Hemos visto que para representar una magnitud del mundo real en un programa utilizamos las variables. ¿Pero qué sucede si lo que queremos representar es el valor de un conjunto idéntico de variables, como por ejemplo las calificaciones de los alumnos de una clase?

Aunque es posible resolver este problema con lo que hemos visto hasta ahora, sin más que definir una variable por cada alumno:

```
int notaAlumno1;
int notaAlumno2;
...
int notaAlumno65;
```

y repetir el proceso de calcular la nota para cada uno de los alumnos, es evidente que este procedimiento puede resultar muy laborioso y propenso a errores.

Por ello, C incorpora un mecanismo para manejar este tipo de situaciones, que son las **tablas**.

Una tabla es un conjunto finito y ordenado de elementos del mismo tipo (**int**, **char**, **float**, ...) agrupados bajo el mismo nombre.

Su principal característica es el modo de almacenamiento en memoria: se realiza en posiciones consecutivas.

Una característica diferenciadora de las tablas con respecto a otros tipos de datos es que no soportan el operador de asignación; para asignar una tabla a otra será necesario asignar los elementos de la tabla uno a uno.

9.2. Declaración de las Tablas

La declaración de una tabla se hace indicando el tipo de todos los elementos de la tabla, seguido del nombre de la variable, seguido del número de elementos encerrados entre corchetes, **[]**, y terminando con un punto y coma.

Ejemplo: Si queremos declarar la tabla de las calificaciones de los 65 alumnos de una clase, tendríamos que poner (suponemos que las calificaciones son valores enteros, entre 0 y 10):

```
int notas[65];
```

Todos los elementos de la tabla así definida son de tipo **int**.

Podemos aprovechar la declaración de una tabla para inicializarla (como en el caso de las variables). Para eso tendremos que indicar un valor para cada uno de los elementos de la tabla, separándolos entre comas, y encerrados entre llaves.

Ejemplo: Para inicializar una tabla que contenga los días del mes, podríamos poner:

```
int diasMes[12] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
```

La asignación se realiza en orden ascendente del índice, de forma que si no hay suficientes valores para inicializar la tabla completa, se asignarán valores a los índices más bajos; los elementos para los que no haya valores se inicializarán a cero.

Si se realiza la inicialización de una tabla en la declaración, el C permite que no se especifique el tamaño de la tabla; lo que hará será contar cuántos valores de inicialización aparecen, y asumir que ese es el tamaño de la tabla.

Ejemplo: La tabla con los días de los meses del año podría declararse también de la siguiente forma:

```
int diasMes[] = { 31,28,31,30,31,30,31,31,30,31,30,31 };
```

La declaración de la tabla es idéntica a la vista anteriormente.

9.2.1. Inicialización de Tablas de Caracteres

Las cadenas de caracteres son un caso especial de tablas de caracteres; se diferencian en que las primeras siempre acaban en **\0**.

Esto permite que la inicialización de las tablas que vayan a contener cadenas de caracteres pueda hacerse de dos formas diferentes:

- mediante una secuencia de caracteres individuales, encerrado cada uno de ellos entre comillas simples, como una tabla de cualquier otro tipo. En este caso, no hay que olvidar asignar un **\0** tras el último carácter de la tabla.

- mediante una cadena de caracteres (secuencia de caracteres encerradas entre comillas dobles). En este caso no es preciso incluir el carácter **\0** al final de la cadena, pero sí hay que considerarlo a la hora de calcular el tamaño necesario de la tabla.

Ejemplo: Para inicializar una tabla de caracteres tenemos las siguientes posiblidades:

```
char nombre[8] = { 'P', 'r', 'i', 'm', 'e', 'r', 'o', '\0' };
char nombre[]  = { 'P', 'r', 'i', 'm', 'e', 'r', 'o', '\0' };
char nombre[8] = "Primero";
char nombre[]  = "Primero";
```

Ni que decir tiene que el método más utilizado es el último indicado.

También existe una especificación de formato especial para cadenas de caracteres en las funciones **printf** y **scanf**: **%s**. Esta especificación no es la especificación de una tabla de caracteres (que no tiene por qué terminar con el carácter **\0**), sino la especificación de una cadena de caracteres. Esto significa que, cuando la usemos con **scanf**, al terminar de escribir nosotros los caracteres, el sistema añadirá el **'\0'** final; y cuando la utilicemos con **printf**, el sistema sabrá que ha terminado de escribir la cadena de caracteres cuando encuentre el **'\0'** final.

Es muy importante recordar esto a la hora de solicitar una cadena de caracteres por teclado: al declarar la tabla que almacenará la cadena, debemos reservar sitio para todos los caracteres, y también para el **'\0'** que el sistema va a añadir.

9.3. Acceso a los Elementos de una Tabla

Para acceder a un elemento de una tabla, tendremos que utilizar el nombre de la tabla, y, encerrado entre corchetes, el índice del elemento al que queremos acceder, sabiendo que el primer elemento es el de índice **0**.

Ejemplo: Para acceder a la calificación del quinto alumno, tendríamos que poner:

```
notas[4]
```

El índice de una tabla siempre debe ser una expresión entera.

Puesto que el primer elemento de la tabla es el de índice **0**, el último será el de índice una unidad menor que el tamaño de la tabla.

El principal problema que tiene la utilización de tablas en C radica en que el compilador sabe siempre cuál es el primer elemento de la tabla (el de índice **0**), pero no sabe qué tamaño tiene la tabla. Podemos utilizar un índice para acceder a un elemento de la tabla mayor que su tamaño, y el compilador no detectaría el error. Por eso hay que tener mucho cuidado al escribir programas, asegurándonos de que no sobrepasamos los límites de las tablas.

Ejemplo: En la tabla de los días del mes, en un programa podríamos escribir:

```
diasMes[12]
```

y no se detectaría que ese elemento ya no pertenece a la tabla, puesto que el último elemento de la tabla es el de índice 11.

El operador **[]** tiene la máxima prioridad (exceptuando los paréntesis), y se evalúan de izquierda a derecha.

9.4. Recorrer los Elementos de una Tabla

Para recorrer los elementos de una tabla, la estructura repetitiva **for** es la ideal. Nos permite realizar una asignación inicial (que el valor del índice sea 0), una operación tras cada iteración (incrementar el índice en una unidad), y una comparación de control (que no se haya superado el límite de la tabla).

Ejemplo: El siguiente código calcula la media de las calificaciones de **10** alumnos, que se introducirán por teclado:

```
#include <stdio.h>

int main()
{
  int   i;
  float notas[10];
  float media = 0;

  for ( i = 0; i < 10; i++ )
    {
      printf("\nNota de alumno: %d: ", i);
      scanf(" %f", &notas[i]);
    }

  for ( i = 0; i < 10; i++ )
    media += notas[i];
  media /= 10;

  printf("\nLa media es %f\n", media);

  return 0;
}
```

Obsérvese que en la condición de control de la estructura repetitiva **for** hemos puesto **i < 10**, y no **i <= 10**, puesto que en una tabla, el índice del último elemento es uno menos que el tamaño de la tabla.

9.5. Utilización de Constantes Simbólicas

Como hemos observado en el ejemplo anterior, la dimensión de una tabla es un valor que se utiliza más de una vez: tenemos que utilizarla para declarar la tabla, y la necesitamos cada vez que queremos recorrer la tabla para obtener o modificar los valores de cada uno de los elementos. Por eso es muy importante que se utilicen constantes simbólicas para este fin, utilizando la cláusula **#define** del preprocesador. De esta forma, si en un momento determinado queremos cambiar el tamaño de una tabla, sólo tendremos que modificar una línea de código, eliminando una importante fuente de errores.

También es importante que el nombre de la constante simbólica sea significativo, de forma que su simple lectura nos indique qué significado tiene dicha constante en el programa.

Ejemplo: El siguiente programa calcula las calificaciones ponderadas de un curso:

```
#include <stdio.h>

int main()
{
  float notas[65];
  int   i      = 0;
  float maxima = 0.0;

  for ( i = 0; i < 65; i++)
    {
      printf("Calificacion del alumno %d: ", i);
      scanf(" %f", &notas[i]);
```

```
    }
  for ( i = 0; i < 65; i++)
    if (notas[i] > maxima)
      maxima = notas[i];
  for ( i = 0; i < 65; i++)
    notas[i] = notas[i] * 10 / maxima;
  for ( i = 0; i < 65; i++)
    printf("La nota final del alumno %d es %f\n", i, notas[i]);

  return 0;
}
```

Si aparece un nuevo alumno, tendremos que buscar en el código todas las apariciones de **65** y cambiarlas por **66**.

Si utilizamos una constante simbólica:

```
#include <stdio.h>

#define NUMALUMNOS 65

int main()
{
  float notas[NUMALUMNOS];
  int   i      = 0;
  float maxima = 0.0;

  for ( i = 0; i < NUMALUMNOS; i++)
    {
      printf("Calificacion del alumno %d: ", i);
      scanf("%f", &notas[i]);
    }
  for ( i = 0; i < NUMALUMNOS; i++)
    if (notas[i] > maxima)
      maxima = notas[i];
  for ( i = 0; i < NUMALUMNOS; i++)
    notas[i] = notas[i] * 10 / maxima;
  for ( i = 0; i < NUMALUMNOS; i++)
    printf("La nota final del alumno %d es %f\n", i, notas[i]);

  return 0;
}
```

sólo tendremos que cambiarla en un sitio, y además perfectamente localizado.

Debe observarse que el nombre dado a la constante simbólica es significativo. Sería un error utilizar la siguiente sentencia para definir la constante:

```
#define SESENTAYCINCO 65
```

Primero, porque no nos proporcionaría ninguna información sobre para qué se utiliza, y segundo, porque si tenemos que modificar su valor, representaría un contrasentido.

9.6. Punteros y Tablas

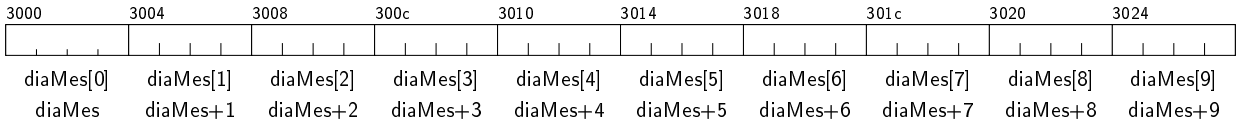
Por definición, el identificador de una tabla representa la dirección del primer elemento de la tabla.

Ejemplo: Las expresiones **`diaMes[0]`**, y **`*diaMes`** son equivalentes, y pueden utilizarse indistintamente en cualquier expresión.

Una de las características de las tablas es que almacenan sus valores en memoria de forma consecutiva. Como se vio al hablar de la aritmética de punteros, si esto sucedía se podía ir incrementando el valor del puntero

para acceder a las sucesivas variables.

Aplicando estos dos conceptos, podemos encontrar que si **diaMes** apunta al primer elemento de la tabla, **diaMes + 1** apuntará al segundo, **diaMes + 2** al tercero, y así sucesivamente, como podemos ver en la siguiente figura:



donde la fila inferior representa la dirección del elemento, y la superior el valor del elemento. Por tanto, en una tabla podemos decir que siempre se cumple que **tabla[indice]** es idéntico a ***(tabla+indice)**.

Ejemplo: El cálculo de la nota media podría escribirse también de la siguiente forma:

```
#include <stdio.h>

#define NUMALUMNOS 10

int main()
{
  int   i;
  float notas[NUMALUMNOS];
  float media = 0;

  for ( i = 0; i < NUMALUMNOS; i++ )
    {
      printf("\nNota de alumno: %d: ", i);
      scanf("%f", notas + i);
    }

  for ( i = 0; i < NUMALUMNOS; i++ )
    media += *(notas + i);
  media /= NUMALUMNOS;

  printf("\nLa media es %f\n", media);

  return 0;
}
```

donde simplemente se han sustituido la dirección de los elementos (**notas + i** en lugar de **&notas[i]**) y los valores de los elementos (***(notas + i)** en lugar de **notas[i]**).

Aunque el nombre de una tabla sea equivalente a la dirección del primer elemento, esto no significa que sea una variable de tipo puntero; y no lo es porque no es una variable, sino una constante. Por eso podemos utilizarla para acceder a los elementos de la tabla, pero no para modificar su valor.

Ejemplo: La expresión **diaMes++** es errónea, puesto que estaríamos intentando modificar el valor de una constante.

Para recorrer una tabla modificando el valor de un puntero, debemos utilizar una variable auxiliar, que inicializaremos con la dirección del primer elemento de la tabla, y que posteriormente iremos modificando.

Ejemplo: Con esta filosofía, el cálculo de la nota media quedaría de la siguiente forma:

```
#include <stdio.h>

#define NUMALUMNOS 10

int main()
```

```
{
  int     i;
  float   notas[NUMALUMNOS];
  float * pAux  = notas;
  float   media = 0;

  for ( i = 0; i < NUMALUMNOS; i++, pAux++ )
    {
      printf("\nNota de alumno: %d: ", i);
      scanf(" %f", pAux);
    }

  pAux = notas;
  for ( i = 0; i < NUMALUMNOS; i++, pAux++ )
    media += *pAux;
  media /= NUMALUMNOS;

  printf("\nLa media es %f\n", media);

  return 0;
}
```

9.6.1. Diferencias y Usos

Aunque el nombre con el que se declara una tabla representa la dirección del primer elemento de la tabla, no hay que confundir el concepto de tabla con el de puntero. La principal diferencia es que una tabla reserva memoria para almacenar la información, mientras que un puntero no, lo que hace que trabajar con punteros o con tablas sea diferente.

Ejemplo: Si se tienen las siguientes declaraciones:

```
char mensaje1[] = "Valor inadecuado";
char *mensaje2 = "Valor inadecuado";
```

en el primer caso estamos declarando una tabla de caracteres que inicializamos con una cadena de caracteres. En el segundo caso estamos declarando un puntero a carácter, que inicializamos con la dirección de una cadena de caracteres.

Las consecuencias de estas dos declaraciones son varias:

- cuando declaramos una tabla estamos reservando memoria para dicha tabla, y si la inicializamos estamos copiando los valores asignados sobre la tabla creada. Eso supone que pueden modificarse los valores de los elementos de la tabla.

  Cuando declaramos un puntero y realizamos una asignación, estamos copiando una dirección, y por lo tanto no podemos modificar la cadena (puesto que es una constante). El problema se agrava en tanto que el error no se detecta en tiempo de compilación (es sintácticamente correcto) sino de ejecución.

  Ejemplo: Podría ponerse **`mensaje1[0] = 'v';`**, pero no **`mensaje2[0] = 'v';`**.

- por otra parte, al declarar una tabla su nombre representa la dirección del primer elemento de la tabla, pero no es un puntero sino una constante, y por ello no puede modificarse.

  Ejemplo: No podríamos tener expresiones del tipo **`mensaje1++`**, aunque sí sentencias del tipo **`mensaje2++`**, puesto que éste sí es un puntero.

9.7. Tablas como Resultado de una Función

El valor que devuelve una función puede utilizarse dentro de expresiones; como no se han definido operadores para las tablas, no pueden utilizarse dentro de expresiones. Esto hace que no puedan devolverse tablas como resultado de la ejecución de una función.

Lo más que puede devolver una función sería el identificador de una tabla, que representa la dirección del primer elemento; es decir, estamos devolviendo un puntero y no una tabla. Hay que tener especial cuidado con este tipo de operaciones, puesto que si la tabla se ha definido dentro de la función (es una variable automática), dicha variable se destruye en el momento de la finalización de la función; estaríamos devolviendo un puntero a una zona de memoria que ya no existe, generándose un error de difícil localización.

9.8. Tablas Multidimensionales

En C se pueden definir también tablas multidimensionales. Para ello utilizaremos la definición de una tabla pero utilizando tantos índices encerrados entre corchetes como dimensiones se desea que tenga la tabla.

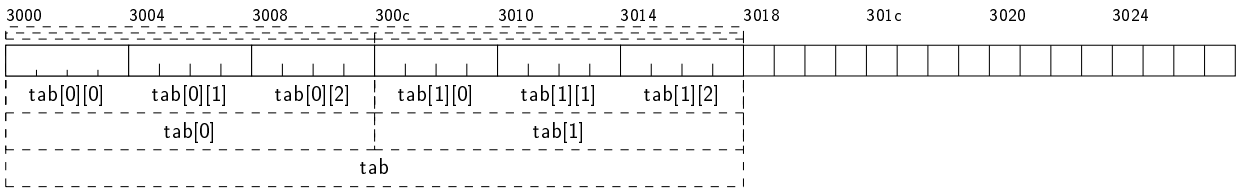
Ejemplo: Una tabla bidimensional se declararía en C de la siguiente forma:

```
int tab[2][3];
```

En realidad, lo que se declara es una tabla cuyos elementos son a su vez tablas, y así sucesivamente hasta completar las dimensiones definidas.

Ejemplo: En la declaración anterior se define una tabla de 2 elementos, siendo cada uno de los elementos una tabla de 3 elementos.

En la siguiente figura se recoge esa situación:



En la anterior figura podemos observar que **tab[0][0]** representa el primer elemento de la tabla bidimensional, que es un **int**; **tab[0]** representa la dirección de comienzo de dicho elemento, por lo que será un puntero a **int**; y **tab** representa la dirección de comienzo de **tab[0]**, que es un puntero a una tabla de 3 enteros.

En consecuencia, y siguiendo la aritmética de punteros, tendremos que **tab[0] + 1** apuntará el elemento siguiente al que apunta **tab[0]**, que es **tab[0][1]**, mientras que **tab + 1** apuntará al siguiente elemento al que apunta **tab**, que es **tab[1]**.

También podemos deducir de la figura anterior que **tab[0] + 3** apuntará al tercer elemento a partir del que apunta **tab[0]**, que es **tab[1][0]**.

Aplicando indistintamente el operador de índice o el operador de contenido y la aritmética de punteros, podemos decir que:

```
*(*(tab + 1) + 2) ≡ *(tab[1] + 2) ≡ tab[1][2] ≡ (*(tab + 1))[2]
```

Para saber de qué tipo es una expresión en la que se utiliza una tabla multidimensional, actuaremos de la misma forma que hicimos en el caso de varios punteros: en la declaración de la tablas, eliminamos la parte que aparece en una expresión; lo que queda es el tipo que representa.

Ejemplo: Si tenemos la siguiente declaración:

```
int notas[ALUMNOS][ASIGNATURAS][PARCIALES];
```

y nos encontramos con la expresión **notas[alumno][asignatura]**, si la marcamos en la declaración (considerando el número de **[]** que estamos utilizando):

```
int notas[ALUMNOS][ASIGNATURAS][PARCIALES];
```

podemos ver que el tipo de la expresión resultante es una tabla de **int** de **PARCIALES** elementos.

9.8.1. Inicialización de Tablas Multidimensionales

Puesto que las tablas multidimensionales en C no son más que tablas de tablas, esta estructura se debe respetar en la inicialización. Así, para inicializar una tabla bidimensional, habrá que inicializar una tabla, para lo que hay que encerrar los valores entre llaves, pero en la que cada elemento de esa tabla vuelve a ser una tabla, por lo que debemos volver a encerrar los valores entre llaves.

Ejemplo: La inicialización de la tabla del apartado anterior sería:

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

Podemos ver que hemos inicializado una tabla de 2 elementos, donde cada elemento es una tabla de 3 elementos.

Las tablas multidimensionales en C tienen una estructura lineal, por lo que dichas tablas también pueden inicializarse utilzando un único grupo de valores separados por comas y encerrados entre llaves. En este caso la asignación se realizará de forma secuencial.

Ejemplo: La inicialización anterior podría escribirse también como:

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

Al igual que sucede en las tablas de una dimensión, al inicializar una tabla multidimensional podemos eliminar uno de los índices, siempre el más externo (el que aparece antes en la declaración). La dimensión de ese índice más externo se tomará del número de elementos que aparezcan en la definición.

Ejemplo: Puede inicializarse la tabla anterior poniendo:

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

Es importante recordar que al inicializar una tabla puede suprimirse el primero de los índices, y sólo ése.

9.9. Tablas de punteros

Puesto que el puntero a un determinado tipo de variable es también un tipo, podemos definir tablas de punteros, de la misma forma que definimos tablas de enteros o de caracteres.

Ejemplo: Si escribimos la sentencia:

```
int *tabla[MAX];
```

estaremos definiendo una tabla de **MAX** elementos, en la que cada elemento es de tipo **int ***.

9.9.1. Tablas de punteros y tablas multidimensionales

De la misma forma que una tabla y un puntero son conceptos diferentes, también los son las tablas multidimensionales y las tablas de punteros. Las principales diferencias son:

1. reserva de memoria: en el caso de las tablas multidimensionales, en la declaración estamos reservando memoria para los elementos finales, mientras que en el caso de las tablas de punteros no, sería necesario tener memoria reservada en otro sitio para poder usarlo correctamente.

2. ubicación de la memoria: en las tablas multidimensionales, los elementos están dispuestos siempre en posiciones consecutivas de memoria; esto es así porque la reserva de memoria para los diferentes elementos se realiza de una sola vez, en el momento de la declaración de la variable. En las tablas de punteros, lo único que puede asegurarse es que son consecutivos los diferentes punteros, pero no la zona a las que apuntan.

9.10. Tablas como Parámetros de Funciones

Las tablas pueden pasarse como parámetros a las funciones. Cuando se hace esto, lo que realmente se pasa como parámetro es la dirección del primer elemento de la tabla, y no la tabla en sí misma; esto es consecuencia de la no existencia del operador de asignación para las tablas.

Puesto que el tamaño de la tabla no va implícito en la misma, siempre que se pase una tabla como parámetro a una función será necesario pasar también como parámetro el tamaño de la misma.

Ejemplo: El siguiente programa calcula la media de los alumnos en una funión aparte, a la que se le pasa la tabla de alumnos y el número de alumnos como parámetros:

```
#include <stdio.h>

#define NUMALUMNOS 10

float calculaMedia(float listaAlumnos[], int numAlumnos);

int main()
{
  int   i;
  float notas[NUMALUMNOS];

  for ( i = 0; i < NUMALUMNOS; i++ )
    {
      printf("\nNota de alumno: %d: ", i + 1);
      scanf(" %f", &notas[i]);
    }
  printf("\nLa media es %f\n", calculaMedia(notas, NUMALUMNOS));

  return 0;
}


float calculaMedia(float listaAlumnos[], int numAlumnos)
{
  int i;
```

```
  float media = 0;

  for ( i = 0; i < numAlumnos; i++ )
    media += listaAlumnos[i];
  media /= numAlumnos;

  return media;
}
```

Observe la declaración, la llamada y la definición de la función.

Puesto que el nombre de la tabla representa la dirección de su primer elemento, puede utilizarse indistintamente la notación de puntero (* tabla) o de tabla (tabla[]) tanto en la declaración como en la definición de una función que utiliza una tabla como parámetro.

Ejemplo: En el siguiente programa se ha utilizado la notación de puntero en la definición de la función, y la de tabla en la declaración y el cuerpo de la función:

```
#include <stdio.h>

#define NUMALUMNOS 10

float calculaMedia(float listaAlumnos[], int numAlumnos);

int main()
{
  int   i;
  float notas[NUMALUMNOS];

  for ( i = 0; i < NUMALUMNOS; i++ )
    {
      printf("\nNota de alumno: %d: ", i + 1);
      scanf(" %f", &notas[i]);
    }
  printf("\nLa media es %f\n", calculaMedia(notas, NUMALUMNOS));

  return 0;
}


float calculaMedia(float * listaAlumnos, int numAlumnos)
{
  int i;
  float media = 0;

  for ( i = 0; i < numAlumnos; i++ )
    media += listaAlumnos[i];
  media /= numAlumnos;

  return media;
}
```

No obstante, es preferible utilizar tanto en la definición como en la declaración la notación de tabla, porque explícitamente nos indica que el parámetro que se pasa es la dirección de una tabla, y no un simple puntero.

9.10.1. Tablas Multidimensionales como Parámetros

En el caso de las tablas multidimensionales se debe aplicar la misma técnica vista anteriormente:

- utilizar la tabla con todas sus dimensiones, pudiendo obviar el tamaño de la primera de ellas (la más interna).

- eliminar la primera dimensión (y únicamente la primera), utilizando en este caso la notación de punteros. En este caso, es imprescindible encerrar entre paréntesis el operador de contenido (*) y el nombre de la tabla, porque lo que se pasa como parámetro sería un puntero a tabla, y no una tabla de punteros.

En cualquiera de los dos casos será necesario también pasar como parámetros los tamaños de cada una de las dimensiones de la tabla multidimensional, para poder trabajar con ellas en el cuerpo de la función.

También se puede pasar como parámetro de una función una subtabla derivada de una tabla de más dimensones; en este caso tendremos que aplicar los criterios correspondientes a la tabla resultante (si es de una o de más dimensiones).

Ejemplo: En el siguiente código:

```
#include <stdio.h>

#define NUMALUMNOS 10
#define ASIGNATURAS 2

float calculaMediaAlumno(float listaNotas[],
                         int asignaturas);
float calculaMedia(float listaAlumnos[][ASIGNATURAS],
                   int numAlumnos,
                   int asignaturas);
float calculaMaxima(float (* listaAlumnos)[ASIGNATURAS],
                    int numAlumnos,
                    int asignaturas);

int main()
{
  int   i;
  int   j;
  float notas[NUMALUMNOS][ASIGNATURAS];

  for ( i = 0; i < NUMALUMNOS; i++ )
    for ( j = 0; j < ASIGNATURAS; j++ )
      {
        printf("\nNota de alumno: %d para asignatura %d: ", i + 1, j + 1);
        scanf(" %f", &notas[i][j]);
      }

  // Calculamos la media para cada uno de los alumnos
  for ( i = 0; i < NUMALUMNOS; i++ )
    printf("La media del alumno %d es %f\n", i + 1,
           calculaMediaAlumno(notas[i], ASIGNATURAS));

  // Calculamos la media de todo el curso
  printf("La media del curso es %f\n",
         calculaMedia(notas, NUMALUMNOS, ASIGNATURAS));

  // Calculamos la nota maxima del curso
  printf("La nota maxima del curso es %f\n",
         calculaMaxima(notas, NUMALUMNOS, ASIGNATURAS));

  return 0;
}


float calculaMediaAlumno(float listaNotas[],
                         int asignaturas)
{
  int   j;
  float media = 0;

  for ( j = 0; j < asignaturas; j++ )
    media += listaNotas[j];
```

```
  media /= asignaturas;

  return media;
}


float calculaMedia(float listaAlumnos[][ASIGNATURAS],
                   int numAlumnos,
                   int asignaturas)
{
  int   i;
  int   j;
  float media = 0;

  for ( i = 0; i < numAlumnos; i++ )
    for ( j = 0; j < asignaturas; j++ )
      media += listaAlumnos[i][j];
  media /= (numAlumnos * asignaturas);

  return media;
}


float calculaMaxima(float (* listaAlumnos)[ASIGNATURAS],
                    int numAlumnos,
                    int asignaturas)
{
  int   i;
  int   j;
  float maximo = 0;

  for ( i = 0; i < numAlumnos; i++ )
    for ( j = 0; j < asignaturas; j++ )
      if (listaAlumnos[i][j] > maximo)
        maximo = listaAlumnos[i][j];

  return maximo;
}
```

podemos observar los tres casos:

- la función **calculaMedia** es un ejemplo del primer procedimientos de pasar tablas multidimensionales como parámetros.
- la función **calculaMaxima** sería un ejemplo del segundo caso. Puede observarse que el nombre de la tabla y el operador de contenido se han encerrado entre paréntesis para que el tipo del parámetro sea puntero a tabla, en lugar de tabla de punteros.
- en la función **calculaMediaAlumno** se pasa como parámetro una subtabla de la tabla notas; en concreto, las calificaciones de las asignaturas de un alumno determinado.

9.10.2. Tablas de Punteros como Parámetros

Las tablas de punteros pueden utilizarse también como parámetros de funciones, utilizándose de la misma forma que las tablas unidimensionales.

9.10.3. Argumentos en la Línea de comandos

Un caso particular de tablas como parámetros de funciones lo constituye la función **main**.

Hasta ahora, se ha utilizado dicha función como si fuera una función sin parámetros que devuelve un entero. En realidad, la función **main** admite dos parámetros, el primero de ellos un entero, y el segundo una tabla

de punteros a cadenas de caracteres, con lo que su declaración sería:

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

Los identificadores de los parámetros formales pueden ser cualesquiera, como es habitual, pero suelen utilizarse siempre **argc** para el primero y **argv** para el segundo.

Estos parámetros de la función **main** nos permiten utilizar argumentos en la línea de comandos, pudiendo realizar varias ejecuciones con diferentes valores sin que sea necesario volver a compilar el programa.

El significado de los parámetros es el siguiente:

- El primero, **argc**, indica el número de elementos que componen la línea de comandos. Los elementos van separados entre sí por espacios. Si se desea que algún elemento incluya un espacio en blanco, será necesario encerrarlo entre comillas (simples o dobles).
- el segundo, **argv** es una tabla de punteros a cadenas de caracteres, cada uno de ellos apuntando, en orden, a uno de los elementos de la línea de comandos. El último elemento de argv siempre apunta a **NULL**, indicando que ya no hay más argumentos en la línea de comandos.

El comando en sí mismo forma parte de la línea de comandos, por lo que entrará en la cuenta de elementos, y será el primer elemento de la tabla de punteros.

Ejemplo: El siguiente programa escribe en la salida estándar el número de elementos en la línea de comandos, y a continuación, cada uno en una línea, cada uno de los elementos:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;

  printf("El numero de elementos es %d\n", argc);
  printf("Los elementos son:\n");
  for ( i = 0; i < argc; i++)
    printf("\t %s\n", argv[i]);

  return 0;
}
```

Si se realizan varias ejecuciones, obtendríamos los siguientes resultados:

```
salas@318CDCr12:~$  gcc -W -Wall -o linea lineaComandos.c
salas@318CDCr12:~$  linea
El numero de elementos es 1
Los elementos son:
        linea
salas@318CDCr12:~$  linea comandos uno dos tres
El numero de elementos es 5
Los elementos son:
        linea
        comandos
        uno
        dos
        tres
salas@318CDCr12:~$  linea comandos "uno dos tres"
El numero de elementos es 3
Los elementos son:
        linea
        comandos
        uno dos tres
```

Tema 10

Tipos Agregados

Índice

10.1. Estructuras **10-1**
10.1.1. Declaración de Estructuras y Variables 10-1
10.1.2. Inicialización de Estructuras 10-3
10.1.3. Punteros a Estructuras 10-4
10.1.4. Acceso a los campos de una estructura 10-4
10.1.5. Estructuras como Campos de Estructuras 10-5
10.1.6. El Operador de Asignación en las Estructuras 10-6
10.1.7. Tablas de Estructuras 10-7
10.1.8. Estructuras como Argumentos de Funciones 10-8
10.1.9. Estructuras como Resultado de una Función 10-8
10.2. Uniones **10-8**
10.2.1. Declaración de Uniones 10-8
10.2.2. Inicialización de una Unión 10-9
10.2.3. Acceso a los campos de una unión 10-9
10.3. Campos de bits **10-9**
10.4. El tipo Enumerado **10-11**
10.4.1. Declaración de Enumerados 10-11
10.4.2. Declaración de Variables de Tipo Enumerado 10-12
10.5. La Definición de Nuevos Tipos **10-12**
10.5.1. Uso del typedef 10-13

10.1. Estructuras

Una estructura es un conjunto finito de elementos de **cualquier tipo** agrupados bajo el mismo nombre, para hacer más eficiente e intuitivo su manejo. A cada elemento de la estructura se le denomina **campo**.

Al contrario de lo que sucede con las tablas, **no puede asegurarse en ningún caso** que los elementos de una estructura estén consecutivos en memoria.

10.1.1. Declaración de Estructuras y Variables

La utilización de estructuras en un programa en C tiene dos componentes:

- la declaración de la estructura, que consiste en enumerar los diferentes campos que la componen y asignarles un nombre.
- la declaración de variables que respondan a esta estructura.

El lenguaje nos permite completar cada uno de los componentes por separado o de forma conjunta.

Declaración de las Estructuras

Al contrario de lo visto hasta este momento, las estructuras pueden declararse, lo que significa que se especifica cuál es su estructura, pero no se define ninguna variable. Lógicamente esto sólo tiene utilidad si posteriormente se va a definir alguna variable que responda a dicha estructura.

Para declarar una estructura, utilizamos la palabra reservada **struct**, seguida de un identificador para la estructura, seguido de la declaración de cada uno de los campos encerrados entre llaves.

Ejemplo: Un ejemplo de la declaración de una estructura sería:

```
struct persona {
  char    nombre[10];
  int     edad;
  float   altura;
  int     sexo;
};
```

Estamos declarando una estructura que tiene cuatro campos: el primero es una tabla de 10 **char**, el segundo un **int**, el tercero un **float** y el cuarto otro **int**. A dicha estructura la hemos denominado **persona**. Y no hemos definido ninguna variable de este tipo.

Debe recordarse que al declarar una estructura sólo estamos describiendo la composición de la misma, y no estamos declarando ninguna variable, por lo que no estamos reservando ninguna memoria.

A partir de su declaración, podemos referirnos a un determinado tipo de estructura mediante la secuencia **struct <nombre>**, siendo *<nombre>* el identificador con el que hemos designado a la estructura en su declaración.

Es muy importante que los identificadores de los diferentes campos que componen una estructura tengan significado, de forma que su simple lectura nos indique qué esta almacenando ese campo. Si no se hace así, el manejo de estructuras dentro de un programa medianamente complejo se hace imposible.

La visibilidad de la declaración de una estructura sigue las mismas reglas que la visibilidad de las variables. Por tanto, si declaramos una estructura dentro de una función, sólo podremos utilizar dicho tipo de estructura en esa función.

Declaración de Variables de Tipo Estructura

Para declarar una variable de tipo estructura, tendremos que indicar primero el tipo de estructura de que se trate, y a continuación el nombre de la variable. La declaración de una variable de tipo estructura reserva espacio de memoria para todos y cada uno de los campos de la estructura.

Ejemplo: Para declarar una variable del tipo **struct persona** declarado antes, habrá que poner:

```
struct persona alumno;
```

Declaración simultánea de Estructuras y Variables

Pueden realizarse las dos operaciones anteriores (declaración de la estructura y declaración de variables) en una única instrucción. Para ello sólo tendremos que declarar la estructura, y antes de finalizar con el carácter

de fin de sentencia añadir las variables que se desean declarar.

Ejemplo: Con el siguiente código:

```
struct persona {
  char    nombre[10];
  int     edad;
  float   altura;
  int     sexo;
}  alumno;
```

en una única sentencia estamos declarando la estructura de nombre **persona** y además estamos declarando una variable de dicho tipo y de nombre **alumno**.

Si la estructura se va a utilizar en un único punto del programa (lo cual es muy improbable), puede eliminarse el identificador de la estructura, definiendo únicamente su composición (los campos que la componen) y declarando las variables necesarias de dicho tipo.

Ejemplo: El ejemplo anterior, según esta premisa, quedaría:

```
struct {
  char    nombre[10];
  int     edad;
  float   altura;
  int     sexo;
}  alumno;
```

Esta declaración, como queda dicho, nos imposibilita utilizar esta estructura en ningún otro punto del programa (no tenemos cómo identificarla), por lo que sólo es de utilidad en casos en los que ya no vaya a hacerse ninguna otra referencia a la estructura.

10.1.2. Inicialización de Estructuras

Podemos aprovechar la declaración de una variable de tipo estructura para inicializar cada uno de sus campos. Para ello, en la declaración, y tras el operador de asignación, se incluirán los valores correspondientes a cada uno de los campos, separados por comas, y encerrados entre llaves.

Ejemplo: Para inicializar una variable de tipo **struct persona** se pondría:

```
struct persona alumno = { "Pablo", 21, 1.8, 0};
```

Lógicamente, los tipos de los valores deben coincidir uno a uno con los tipos de los campos de la estructura.

Si no hubieran suficientes valores para todos los campos, se asignarán los que haya a los primeros campos, y el resto se inicializará a cero.

Ejemplo: Si la inicialización fuera:

```
struct persona alumno = { "Pablo", 21};
```

los valores para los campos **altura** y **sexo** se inicializarán a **0**.

Nótese que para poder inicializar una estructura debemos conocer de qué tipo son los campos y en qué orden están definidos; además, a pesar de que los campos no indicados se inicializan a cero, si utilizamos el segundo método el compilador avisa de que no se han especificado todos los campos de la estructura.

Para evitar estos inconvenientes existe una forma especial de inicialización de estructuras que la hace independiente del orden de definición. Para ello tendremos que incluir, encerrados entre llaves y separados por

comas, la secuencia de asignaciones, consistente cada asignación en un punto, seguido por el identificador del campo, el operador de asignación y el valor a asignar a dicho campo.

Ejemplo: Para la estructura anterior, sólo quisiéramos inicializar el nombre y la altura, podría ponerse:

```
struct persona alumno = { .altura = 1.8, .nombre = "Pablo" };
```

inicializándose a cero los dos campos restantes.

10.1.3. Punteros a Estructuras

A las variables de tipo estructura, como a cualquier otra variable, se le puede aplicar el operador de dirección, obteniendo en ese caso un puntero a una estructura. Podrán también existir, por tanto, variables de tipo puntero a estructura, que declararemos como siempre interponiendo un **`*`** entre el tipo de la estructura y el nombre de la variable.

Ejemplo: Para declarar un puntero a una estructura del tipo **`struct persona`**, que ha debido ser declarada con anterioridad, habría que escribir:

```
struct persona * pAlumno = NULL;
```

Debe recordarse que la declaración de un puntero **NUNCA** reserva memoria para la variable a la que apunta, por lo que para poder utilizarlo será preciso que apunte a una zona previamente reservada.

10.1.4. Acceso a los campos de una estructura

Es evidente que la utilización de estructuras podrá ser útil siempre y cuando podamos acceder a sus campos de forma individual. Para poder hacer esto, en C existen dos operadores, el operador **`.`** y el operador **`->`**. El primero se utiliza para acceder a los campos cuando tenemos una estructura, mientras que el segundo se utiliza cuando tenemos un puntero a estructura. En ambos casos, la parte izquierda del operador debe ser una expresión del tipo adecuado (estructura o puntero a estructura, respectivamente), y la parte derecha el nombre de uno de los campos de la estructura.

Ejemplo: El siguiente código muestra un ejemplo de acceso a los campos de una estructura:

```
#include <stdio.h>

struct persona {
  char    nombre[10];
  int     edad;
  float   altura;
  int     sexo;
};

int main()
{
  struct persona alumno = { "Pablo", 21, 1.8, 0};
  struct persona * pAlumno = &alumno;

  printf("El alumno se llama %s, y tiene %d anios de edad\n",
    alumno.nombre, pAlumno->edad);

  return 0;
}
```

Los dos operadores de acceso a los campos de una estructura tienen la misma prioridad y asociatividad que el operador **`[]`**, por lo que se evalúan de izquierda a derecha.

10.1.5. Estructuras como Campos de Estructuras

Puesto que para la definición de los campos de una estructura podemos utilizar cualquier tipo de datos, también podremos utilizar estructuras como campos.

Ejemplo: El siguiente fragmento de código declara primero una estructura de tipo **struct fecha**. A continuación declara una estructura de tipo **struct instante**, uno de cuyos campos es del tipo estructura definido con anterioridad. Por último, se declara e inicializa una variable de este último tipo de estructura, dando valores para cada uno de los campos, incluyendo el de tipo estructura:

```
struct fecha {
  int     dia;
  int     mes;
  int     ano;
  char    nombre_mes[4];
  char    dia_semana[4];
};
struct instante {
  int          seg;
  int          min;
  int          hora;
  struct fecha fec;
};
struct instante ahora = { 0,30,23, {1,12,2006,"dic","vie"}};
```

Para acceder al número de mes, tendremos primero que aplicar el operador **.** a la variable **ahora**, para obtener el campo **fec**. Como dicho campo vuelve a ser una estructura, podremos volver a utilizar el operador **.** para obtener el campo **mes**, obteniendo así el valor deseado:

```
ahora.fec.mes
```

Como el operador punto se evalúa de izquierda a derecha, no es necesario utilizar paréntesis.

Es imprescindible que la estructura que se utiliza como campo se declare con anterioridad a la que la utiliza.

También podemos utilizar como campos de estructura punteros a estructuras. En este caso debemos considerar, nuevamente, que un puntero no reserva memoria para los campos de la estructura, por lo que debemos inicializarlo a un valor adecuado antes de poder usarlo. En este caso no es imprescindible que la estructura a la que apunta el puntero haya sido declarada con anterioridad.

Ejemplo: El siguiente fragmento de código es similar al anterior, pero el campo de la estructura **instante** no es una estructura, sino un puntero:

```
struct fecha {
  int     dia;
  int     mes;
  int     ano;
  char    nombre_mes[4];
  char    dia_semana[4];
};
struct fecha hoy = {1, 12, 2006, "dic", "vie"};
struct instante {
  int           seg;
  int           min;
  int           hora;
  struct fecha * fec;
};
struct instante ahora = { 0, 30, 23, &hoy};
```

Si en este ejemplo queremos acceder al número de mes, tendremos primero que aplicar el operador **.** a la variable **ahora**, para obtener el campo **fec**. Como dicho campo es un puntero a una estructura, tendremos que utilizar el operador **->** para obtener el campo **mes**, obteniendo así el valor deseado:

```
ahora.fec->mes
```

Como los operadores **.** y **->** son de igual prioridad, y se evalúan de izquierda a derecha, en este caso tampoco son necesarios los paréntesis.

Un caso particular de esta última construcción lo constituyen las estructuras que tienen como uno de sus campos un puntero a una estructura del mismo tipo, que se denominan estructuras autoreferenciadas.

Ejemplo: La siguiente declaración es la de una estructura uno de cuyos campos es un puntero a la propia estructura:

```
struct fecha {
  int            dia;
  int            mes;
  int            ano;
  char           nombre_mes[4];
  char           dia_semana[4];
  struct fecha * siguiente;
};
```

Estas construcciones son especialmente útiles para ls definición de estructuras dinámicas de datos (que se verán en otra asignatura).

Sin embargo, no es posible que uno de los campos de una estructura sea una estructura del mismo tipo. ¿Intuye por qué?.

10.1.6. El Operador de Asignación en las Estructuras

Al contrario de lo que sucede con las tablas, sí es posible utilizar el operador de asignación con estructuras. Evidentemente, sólo podrá realizarse la asignación entre estructuras del mismo tipo.

La asignación consiste en copiar la zona de memoria ocupada por una estructura en la zona de memoria reservada para la otra. Esto permite que incluso se copien las tablas que puedan aparecer como campos de una estructura, a pesar de que la asignación no es un operador válido para las tablas.

Ejemplo: Si compilamos y ejecutamos el siguiente código:

```
#include <stdio.h>

int main()
{
  struct persona {
    char nombre[10];
    int  edad;
    float altura;
    int sexo;
  } alumno = { .nombre = "Pablo", .altura = 1.8 };
  struct persona alumno2 = alumno;

  alumno2.nombre[0] = 'T';
  printf("El alumno %s mide %f\n", alumno.nombre, alumno.altura);
  printf("El alumno %s mide %f\n", alumno2.nombre, alumno2.altura);

  return 0;
}
```

se tiene:

```
salas@318CDCr12:~$  gcc -W -Wall asignacion1.c
salas@318CDCr12:~$  a.out
El alumno Pablo mide 1.800000
```

```
El alumno Tablo mide 1.800000
salas@318CDCr12:~$
```

Hay que tener especial cuidado si se utilizan punteros como campos de la estructura. En ese caso, lo que se copia son los punteros, no los valores apuntados por éstos. Eso significa que, después de la asignación, los punteros de las dos estructuras tienen el mismo valor, lo que significa que apuntan al mismo sitio. Por lo tanto, si modificamos el contenido del valor apuntado, el efecto se verá en las dos estructuras.

Ejemplo: Si compilamos y ejecutamos el siguiente código:

```
#include <stdio.h>

int main()
{
  char cadena[10] = "Pablo";
  struct persona {
    char * nombre;
    int    edad;
    float  altura;
    int    sexo;
  } alumno = { .nombre = cadena, .altura = 1.8 };
  struct persona alumno2 = alumno;

  alumno2.nombre[0] = 'T';
  printf("El alumno %s mide %f\n", alumno.nombre, alumno.altura);
  printf("El alumno %s mide %f\n", alumno2.nombre, alumno2.altura);

  return 0;
}
```

se tiene:

```
salas@318CDCr12:~$  gcc -W -Wall asignacion2.c
salas@318CDCr12:~$  a.out
El alumno Tablo mide 1.800000
El alumno Tablo mide 1.800000
salas@318CDCr12:~$
```

10.1.7. Tablas de Estructuras

Los tipos agregados pueden combinarse entre sí. Ya hemos visto estructuras en estructuras, tablas de tablas y tablas en estructuras. Nos quedan por utilizar tablas de estructuras.

Para ello no tendremos más que declarar una tabla formada por elementos de algún tipo de estructura.

Ejemplo: Con el siguiente código:

```
#define TAM 40
#define NUMLIBROS 10

struct biblio
{
  char  titulo[TAM];
  char  autor[TAM];
  float precio;
};

struct biblio tab_libro[NUMLIBROS];
```

estamos declarando una tabla de **NUMLIBROS** elementos, siendo cada uno de los elementos una estructura de tipo **struct biblio**.

Para acceder a un campo de un elemento tendremos primero que seleccionar uno de los elementos de la tabla, utilizando el operador **[]**. El resultado será una estructura de tipo **struct biblio**, por lo que podremos utilizar el operador **.** para acceder a uno de sus campos. Si el campo seleccionado es una tabla, podremos además seleccinar uno de los elementos de la tabla, sin más que volver a utilizar el operador **[]**.

Ejemplo: Así, si queremos el tercer carácter del autor del primer libro, tendremos que poner:

```
tab_libro[0].autor[2]
```

Podemos hacer esto porque los dos operadores, **[]** y **.**, tienen la misma prioridad y se evalúan de izquierda a derecha.

10.1.8. Estructuras como Argumentos de Funciones

En el caso de las estructuras, puesto que sí existe el operador de asignación, sí pueden pasarse como argumentos a las funciones. Al igual que sucede con cualquier otro tipo de variables, en la función se trabajaría con una copia de la estructura, y no con la estructura original.

10.1.9. Estructuras como Resultado de una Función

De la misma forma, e igualmente por el hecho de que existe el operador de asignación para las estructuras, puede devolverse una estructura como resultado de una función. En este caso, la función podría utilizarse como parte derecha de una asignación, copiándose los valores de los campos a la estructura situada a la izquierda.

10.2. Uniones

Las uniones son tipos agregados en C, del estilo de las estructuras, en las que todos los campos comienzan en la misma posición de memoria. Se utilizan para considerar que una misma zona de memoria puede verse como de diferentes tipos, dependiendo del momento.

10.2.1. Declaración de Uniones

Al igual que las estructuras, podemos declarar la composición de una unión, la declaración de una variable de tipo unión, o ambas cosas a la vez. La forma de hacerlo es similar, simplemente cambiando la palabra reservada **struct** por **union**.

Ejemplo: Sea el siguiente fragmento de código:

```
#define MAX 128

/* Caso 1 */
union registro {
    int   entero;
    float real;
    char  cadena[MAX];
  };
/* Caso 2 */
union registro variable;
/* Caso 3 */
```

```
union tiposDeDatos {
    int   entero;
    float real;
    char  cadena[MAX];
  } tipoDato1;
```

En el primer caso estamos declarando una unión, en el segundo estamos declarando una variable de un tipo unión declarado anteriormente, y en el tercero estamos declarando la composición de una unión y una variable de dicho tipo simultáneamente.

10.2.2. Inicialización de una Unión

Se puede aprovechar el momento de la declaración de una variable de tipo unión para inicializarla. La forma de inicializarla coincide con la de una estructura, con una diferencia: puesto que todos los campos de una unión comienzan en la misma posición de memoria, si se inicializan varios campos, las sucesivas inicializaciones sobreescribirían las anteriores, por lo que sólo puede inicializarse uno de los campos. Si no se dice nada, se inicializa el primer campo; si se desea incializar un campo diferente al primero, se tendría que especificar en la inicialización el nombre del campo, precedido por un `.`.

Ejemplo: El siguiente código es un ejemplo de incialización de una unión:

```
#define MAX 128

union registro {
    int   entero;
    float real;
    char  cadena[MAX];
  };
union registro variable1 = { 123 };
union registro variable2 = { .real = 0.123 };
```

10.2.3. Acceso a los campos de una unión

Para el acceso a los campos de una unión tenemos los mismos operadores que en el caso de las estructuras: el `.` cuando lo que tenemos es una expresión de algún tipo de unión, y el `->` cuando lo que tenemos es una expresión de tipo puntero a algún tipo de unión.

10.3. Campos de bits

Por defecto, un campo de una estructura o una unión ocupa la totalidad del tipo de dicho campo. Sin embargo, dentro de una estructura o una unión, se puede definir el número de bits que ocupará un campo. Estos campos se denominan campos de bits. El formato general es:

```
struct nombre_campo {
tipo nombre_1 : longitud_1;
tipo nombre_2 : longitud_2;
...
tipo nombre_n : longitud_n;
}
```

Los tipos permitidos por el estándar son los tipos enteros **`_Bool`**, **`signed int`** y **`unsigned int`** (aunque algunos compiladores puede permitir otros). Lo más habitual es usar el tipo **`unsigned int`**, sobre todo cuando la longitud es 1. La longitud es un número no negativo que como máximo será la longitud del tipo utilizado en bits.

El compilador empaquetará todos los campos de la estructura lo más compacto posible. Así, una misma palabra de memoria puede contener varios campos siempre que estén todos completos y sean del mismo tipo.

Si no ponemos el nombre a un campo de bits, no podremos utilizarlo posteriormente pero el compilador reservará espacio para él. Además, si la longitud de un campo sin nombre es 0, se le está indicando al compilador que el siguiente campo de bits debe empezar en una palabra nueva.

La manera en que se disponen los distintos campos dentro de la palabra (derecha e izquierda) es dependiente de la máquina y el compilador.

El acceso a cada uno de los campos de bits se hace igual que al del resto de campos de una estructura o unión, y se pueden utilizar en operaciones aritméticas.

No es posible crear tablas de campos de bits ni tampoco obtener la dirección de memoria. Por tanto, el operador **&** no se puede utilizar con los campos de bits.

Los principales motivos de uso son:

- Si el almacenamiento es limitado, se pueden almacenar varias variables booleanas en un octeto.
- Ciertas interfaces de dispositivo transmiten información que se codifica en bits dentro de un octeto.
- Ciertas rutinas de encriptación necesitan acceder a los bits en un octeto.

Ejemplo: En una máquina con un tamaño de **unsigned int** de 16 bits, la siguiente declaración:

```
struct bits {
  unsigned int bit0:1;
  unsigned int bit1:1;
  unsigned int bit2:1;
  unsigned int bit3:1;
  unsigned int bit4:1;
  unsigned int bit5:1;
  unsigned int bit6:1;
  unsigned int bit7:1;
};
```

se vería en memoria de la siguiente forma:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<no usado>								bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0

mientras que la siguiente declaración:

```
struct bits2 {
  unsigned int campo1:3;
  unsigned int campo2:4;
  unsigned int : 0;
  unsigned int campo3:2;
  unsigned int : 4;
  unsigned int campo4:2;
};
```

se vería en memoria de la siguiente forma:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<no usado>									campo2				campo1		

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
<no usado>								campo4		<no usado>				campo3	

Un ejemplo de uso de este último ejemplo sería:

```
/* Creacion */
struct bits2 var;
/* Uso */
var.campo1 = 2;
```

```
var.campo3 = 1;
var.campo2 = var.campo1 + var.campo3;
```

La principal desventaja es que el código resultante puede no ser portable a otras máquinas o compiladores, ya que depende del tamaño de cada tipo del lenguaje, que como ya se sabe no viene determinado por el estándar. Además el código resultante puede ser más lento que si se hiciera con operaciones binarias.

Una alternativa recomendada para disponer de variables de un bit (semáforos), es el uso de **define**.

Ejemplo: Los "defines":

```
#define Nada      0x00
#define bitUno    0x01
#define bitDos    0x02
#define bitTres   0x04
#define bitCuatro 0x08
#define bitCinco  0x10
#define bitSeis   0x20
#define bitSiete  0x40
#define bitOcho   0x80
```

pueden ser utilizados para escribir el siguiente código:

```
if (flags & bitUno)          // si primer bit ON...
  {...}
flags |= bitDos;             // pone bit dos ON
flags &= ~bitTres;           // pone bit tres OFF
```

Se pueden usar esquemas similares para campos de bits de cualquier tamaño.

10.4. El tipo Enumerado

En muchas situaciones, necesitamos que una variable pueda tener un valor dentro de un conjunto pequeño de valores; sería el caso, por ejemplo, de especificar el mes del año. Para esos casos, el C contempla el tipo **enum**.

Dicho tipo nos permite definir un conjunto de valores que representarán algún parámetro cuyos valores sean discretos y finitos.

10.4.1. Declaración de Enumerados

Al igual que sucede para las estructuras y las uniones, con los enumerados podemos declarar un nuevo tipo, declarar una variable de tipo enumerado, o hacer las dos cosas a la vez.

Para declara un tipo enumerado se utiliza la palabra clave **enum**, seguida del identificador del nuevo tipo y, encerrados entre llaves, los identificadores válidos para este tipo.

Ejemplo: La siguiente declaración serviría para declarar un tipo de datos nuevo, **enum meses**, que tomaría valores para representar los meses del año:

```
enum meses {ene, feb, mar, abr, may, jun,
            jul, ago, sep, oct, nov, dic };
```

Nótese que lo que utilizamos son identificadores, no cadenas de caracteres. Internamente, el C convierte cada uno de estos identificadores en números enteros, empezando por el **0**, y asignando valores de forma

consecutiva. De esta forma, con la declaración anterior, al identificador **ene** se le habrá asignado el valor **0**, al identificador **feb** el valor **1**, ..., y al identificador **dic** el valor **11**.

El lenguaje permite modificar esta asignación de valores, sin más que asignarle un valor a un identificador. En ese caso, al identificador designado se le asigna el valor indicado, y a los sucesivos identificadores se le asignan los sucesivos valores.

Ejemplo: Con el siguiente código:

```
enum meses {ene = 1, feb, mar, abr, may, jun,
            jul, ago, sep, oct, nov, dic };
```

al identificador **ene** se le habrá asignado el valor **1**, al identificador **feb** el valor **2**, ..., y al identificador **dic** el valor **12**.

Esa asignación de valor forzada se puede realizar en cualquier identificador, no sólo en el primero, y tantas veces como se desee, incluso repitiendo valores.

Ejemplo: Esto queda patente en el siguiente código:

```
enum extensiones { Juan = 4103, Luis, Adolfo = 4103,
                   Laura = 4111 };
```

10.4.2. Declaración de Variables de Tipo Enumerado

Aunque la principal finalidad de los enumerados es definir un conjunto de constantes simbólicas, asignándoles valores específicos si se desea, el lenguaje permite definir variables de tipos enumerados. Internamente el sistema las convierte en variables de tipo **int**, por lo que le son aplicables todas las reglas y operaciones que son válidas para los **int**.

Además, los posibles valores de estas variables no están restringidos a los definidos por el tipo enumerado: es válido cualquier valor entero.

Ejemplo: El siguiente programa muestra que las variables de tipo enumerado no restringen sus valores a los identificadores definidos, y se comportan como enteros a todos los efectos:

```
#include <stdio.h>

int main()
{
  enum dias { lun, mar, mie, jue, vie, sab, dom };
  enum dias diasSemana = lun;

  printf("El dia de la semana es %d\n", diaSemana);
  diasSemana += dom;
  printf("El dia de la semana es %d\n", diaSemana);
  diasSemana += dom;
  printf("El dia de la semana es %d\n", diaSemana);

  return 0;
}
```

10.5. La Definición de Nuevos Tipos

Muchas veces, sobre todo cuando utilizamos tipos agregados, resultan tipos demasiado complejos y largos en su enunciación. El C permite asignar "sinónimos" a dichos tipos, de forma que sea mucho más fácil su

utilización (aunque es cierto que a veces ocultando información que puede resultar de ayuda a la hora de programar).

Estas definiciones se hacen con la cláusula **typedef**. Debe quedar claro que con esta cláusula se define un sinónimo de un tipo ya existente, no un nuevo tipo. Por tanto, a este tipo le serán de aplicación todo lo que pueda aplicarse al tipo original.

10.5.1. Uso del typedef

La sintaxis de la cláusula **typedef** es la misma que la utilizada para la declaración de variables (sin inicialización), anteponiéndole la palabra reservada **typedef**. El tipo definido es el que ocuparía la posición de la variable si se tratara de una declaración de variable.

Ejemplo: En el siguiente código:

```
#define MAX 10

typedef int lista[MAX];

lista notas;
```

se define un nuevo tipo, denominado **lista** que equivale a una tabla de **MAX** elementos de tipo **int**. En la última sentencia estamos declarando una variable de dicho tipo **lista**, por tanto la variable **notas** será una tabla de **MAX** elementos de tipo entero, y como tal podríamos tener expresiones como **notas + 1** o **notas[3]**.

Tema 11

Funciones de Biblioteca

Índice

11.1. Introducción **11-1**
11.2. Reserva dinámica de memoria **11-2**
11.2.1. Función de reserva **malloc** 11-2
11.2.2. Función de reserva **calloc** 11-3
11.2.3. Función de liberación **free** 11-3
11.2.4. La reserva dinámica y las tablas multidimensionales 11-4
11.3. Entrada y Salida en C. **11-5**
11.3.1. Funciones de apertura y cierre 11-6
11.3.2. Funciones de entrada y salida 11-6
11.3.3. Salida con formato: función **fprintf** 11-8
11.3.4. Entrada con formato: función **fscanf** 11-9
11.3.5. Funciones de lectura/escritura binaria 11-10
11.3.6. Funciones especiales 11-10
11.3.7. Flujos estándar: **stdin**, **stdout** y **stderr** 11-11
11.3.8. Errores frecuentes en el uso de las funciones de E/S 11-11
11.4. Funciones de Cadenas de Caracteres **11-15**
11.4.1. Funciones de copia 11-15
11.4.2. Funciones de encadenamiento 11-15
11.4.3. Funciones de comparación 11-15
11.4.4. Funciones de búsqueda 11-15
11.4.5. Otras funciones 11-16

11.1. Introducción

Uno de los principales objetivos de la programación modular es la reutilización de código: alguien escribe alguna función, de forma que cualquiera pueda utilizarla en su código.

En el estándar C se establecen una serie de funciones, denominadas de biblioteca, que cualquier compilador debe incorporar, de forma que cualquier programa que utilice dichas funciones pueda ser compilado sin errores en cualquier máquina.

En este tema se estudiarán algunas de estas funciones de biblioteca.

Para obtener una mayor información sobre cualquier función de biblioteca del estándar C utilice el comando **man**. En dicha ayuda encontrará el fichero de tipo `include` en el que se encuentra la declaración de la

función de biblioteca en cuestión, y que deberá incluir en todos los archivos que vayan a utilizar dicha función.

11.2. Reserva dinámica de memoria

Hemos visto que para poder utilizar una variable es necesario reservarle cierta cantidad de memoria para su uso exclusivo; eso se consigue habitualmente mediante la declaración de la variable.

Pero hay situaciones en las que no es posible declarar todas las variables que vamos a necesitar; por ejemplo, porque se trate de una tabla cuyo tamaño no conoceremos hasta que el programa no se ejecute, o por necesitar estructuras de datos dinámicas cuyo tamaño va cambiando durante la ejecución del programa.

En esos casos se hace imprescindible la utilización de las funciones de reserva dinámica de memoria que incorpora el estándar C.

11.2.1. Función de reserva **malloc**

`void * malloc(size_t`[1] **`tamanio);`

Esta función reserva la cantidad de memoria indicada por **tamanio**, y devuelve el puntero al comienzo de la zona de memoria asignada, o `NULL` si no se ha podido hacer la reserva.

Hay tres aspectos importantes a reseñar en esta función.

El primero es cómo calcular la memoria que necesitamos reservar. Eso dependerá de para qué queremos reservar la memoria, pero en cualquier caso siempre será de utilidad el operador **sizeof**. Así, por ejemplo, si queremos reservar memoria para una estructura de tipo `struct generic`, la cantidad de memoria a reservar será `sizeof(struct generic)`, mientras que si queremos reservar memoria para una tabla de enteros de **tam** elementos, la memoria a reservar será de `tam * sizeof(int)`.

El segundo aspecto a considerar es que devuelve un puntero de tipo genérico. La función **malloc** se utiliza para reservar memoria para cualquier tipo de datos, por lo que a priori no puede saber de qué tipo va a ser la zona de memoria devuelta. Por eso es imprescindible que el valor devuelto por **malloc** lo asociemos al tipo adecuado, mediante el operador de conversión forzada. Siguiendo con los dos ejemplos anteriores, tendríamos:

- en el primer caso el puntero devuelto lo sería a una estructura de tipo `struct generic`, por lo que la llamada correcta a la función **malloc** sería: `(struct generic *) malloc(sizeof(struct generic))`
- en el segundo ejemplo devolvería memoria reservada para una tabla de enteros, por lo que la llamada correcta sería: `(int *) malloc(tam * sizeof(int))`

El tercer aspecto a destacar es que la función devuelve `NULL` cuando no ha podido reservar la memoria. Puesto que para poder utilizar una variable necesitamos espacio reservado para la misma, *SIEMPRE* hay que comprobar que se ha podido realizar la reserva comprobando el valor devuelto por la función **malloc**.

Una llamada típica a la función **malloc** sería por tanto de la siguiente forma:

```
struct generic * paux = NULL;

paux = (struct generic *) malloc(sizeof(struct generic));
if (NULL == paux)
  /* Codigo que gestiona el error de falta de memoria */
else
  /* Codigo normal de la funcion */
```

[1] `size_t` es un tipo definido mediante **typedef** y que normalmente equivale a un `unsigned int`.

11.2.2. Función de reserva calloc

void * calloc(size_t numero, size_t tamanio);

La función **calloc** es análoga a **malloc**, con dos diferencias:

- La primera, visible en la propia declaración de la función, es que, mientras en **malloc** sólo teníamos un parámetro, y cuando queríamos reservar memoria para una tabla necesitamos realizar la multiplicación de forma explícita, en **calloc** esta multiplicación se hace implícitamente a partir de los dos parámetros con los que se llama.
- La segunda, no detectable en la declaración, es que la función **calloc** inicializa a cero toda la zona de memoria reservada.

Al igual que sucede con **malloc**, **calloc** devuelve NULL cuando no ha podido reservar la memoria, por lo que *SIEMPRE* hay que comprobar que se ha podido realizar la reserva comprobando el valor devuelto por la función **calloc**.

Una llamada típica a la función **calloc** sería por tanto de la siguiente forma (suponga que **tam** es un parámetro de la función en la que se incluye este fragmento de código):

```
int * paux = NULL;

paux = (int *) calloc(tam, sizeof(int));
if (NULL == paux)
  /* Codigo que gestiona el error de falta de memoria */
else
  /* Codigo normal de la funcion */
```

Comparando ambas funciones, es evidente que **malloc** es adecuada cuando queremos reservar memoria para un único elemento (generalmente una estructura), mientras que **calloc** lo es cuando queremos reservar memoria para un conjunto de elementos (habitualmente una tabla).

11.2.3. Función de liberación free

Con las funciones **malloc** y **calloc** reservamos memoria para uso exclusivo por el programa que realiza la llamada. Lógicamente, esa zona de memoria debemos reservarla única y exclusivamente durante el tiempo que sea necesario, de forma que cuando ya no la necesitemos pueda ser utilizada por otro programa.

Por lo tanto, igual que existen funciones para reservar memoria, existe una función para liberarla: la función **free**.

void free(void * puntero);

La función **free** libera la zona de memoria apuntada por **puntero**, que previamente ha debido ser reservada con **malloc** o **calloc**. Una vez realizada la llamada a **free**, la zona de memoria apuntada por **puntero** *NO PUEDE* volver a utilizarse. Si **puntero** vale NULL, la llamada a la función no tiene ningún efecto.

Aunque es evidente, no está de más señalar que el valor de **puntero** no se ve modificado por la llamada a **free**, como cualquier parámetro de cualquier función.

Como norma general, y para no mantener sin liberar ninguna zona de memoria, en los programas debe exitir una llamada a **free** por cada llamada a **malloc** o **calloc** existente.

Si llamamos a **free** con un valor que no haya sido asignado por una llamada a **malloc** o **calloc**, o con un valor que haya sido liberado con anterioridad, el sistema de gestión de memoria dinámica del sistema operativo puede corromperse; este problema es tanto más grave cuanto que sus efectos no se hacen visibles inmediatamente, sino transcurrido un tiempo variable, lo que hace muy difícil su localización. Por ello es conveniente que asignemos el valor NULL a cualquier puntero ya liberado, de forma que una posterior llamada a **free** con dicho puntero no tenga efectos nocivos. Y así protegemos además el programa de la

posible utilización de un puntero ya liberado, puesto que estaríamos utilizando un puntero igualado a `NULL` y la ejecución del programa se abortaría en ese punto exacto, pudiendo corregirlo.

Una llamada típica a la función **`free`** sería por tanto de la siguiente forma (suponga que **`paux`** almacena un valor devuelto por **`malloc`** o **`calloc`**):

```
free(paux);
paux = NULL;
```

11.2.4. La reserva dinámica y las tablas multidimensionales

Supongamos que en un determinado programa necesitamos reservar memoria para una tabla de `int` de dos dimensiones, con **`nfilas`** filas y **`ncolumnas`** columnas. Por lo visto en el tema de tablas, una tabla de esas características equivale a una sucesión de **`nfilas * ncolumnas`** `ints`, por lo que haríamos la reserva de la siguiente forma:

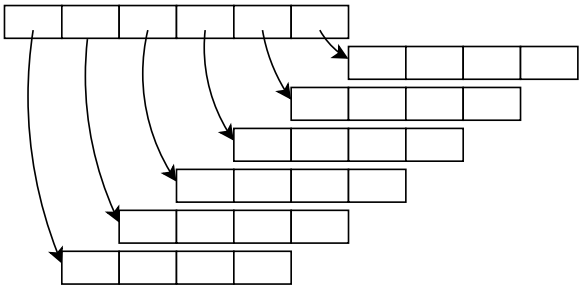
```
int * pTabla = NULL;

pTabla = (int *) calloc(nfilas * ncolumnas, sizeof(int));
if (NULL == pTabla)
  /* Codigo que gestiona el error de falta de memoria */
else
  /* Codigo normal de la funcion */
```

El acceso a los elementos de la tabla (que como puede observarse es una tabla de `int`) se hará calculando previamente la posición del elemento dentro de esa tabla; así, para acceder al elemento de la fila **`fila`** y columna **`columna`**, escribiremos:

```
pTabla[fila * ncolumnas + columna]
```

El acceso a esta tabla no podría hacerse mediante la utilización de doble índice puesto que no se conocía el número de columnas en tiempo de compilación.

Existe un segundo procedimiento para la reserva dinámica de tablas multidimensionales; este segundo método complica la reserva de memoria, pero simplifica el acceso a los elementos. Consiste en considerar la tabla multidimensional no como una tabla de tablas, sino como una tabla de punteros a tablas, según se recoge en la siguiente figura:



El código necesario para generar dicha estructura sería:

```
int ** reservaTabla(int nfilas, int ncolumnas)
{
  int ** pTabla = NULL;
  int    i      = 0;
  int    j      = 0;

  /* Reservamos memoria para la tabla de punteros a tabla */
  pTabla = (int **) calloc(nfilas, sizeof(int *));

  for ( i = 0; (NULL != pTabla) && (i < nfilas); i++ )        (1)
    {
      pTabla[i] = (int *) calloc(ncolumnas, sizeof(int));
      if (NULL == pTabla[i])
        {
          /* Si tenemos problemas en la reserva de un elemento,
             devolvemos la memoria previamente reservada ... */
          for ( j = i - 1; j >= 0; j-- )                      (2)
            free(pTabla[j]);                                  (3)
          /* ... incluyendo la tabla de punteros original */
          free(pTabla);
          pTabla = NULL;                                      (4)
        }
    }

  return pTabla;
}
```

(1) El **`if`** comprobando que se ha podido reservar memoria está implícito en la condición del **`for`**: si ha habido un error **`pTabla`** valdrá `NULL` y no se entrará en el bucle.

(2) La última reserva que hicimos correctamente fue la correspondiente al elemento anterior al actual; desde ese elemento (hacia atrás) tendremos que realizar la liberación de memoria.

(3) En este caso no es necesario igualar **`pTabla[j]`** a `NULL` puesto que la variable va a ser destruida.

(4) Al igualar **`pTabla`** a `NULL`, además de cumplir con lo enunciado en un apartado anterior para protegernos contra errores en ejecución, evitamos que vuelva a entrar en el bucle **`for`** y asignamos el valor que tiene que ser devuelto por la función en caso de error.

Cuando reservamos memoria para una tabla de esta forma, obtenemos un puntero a puntero a `int`, que como sabemos podemos tratar como el nombre de una tabla de punteros a `int`; si lo indexamos (**`pTabla[i]`**, por ejemplo), lo que obtenemos es un puntero a `int`, que podemos tratar como el nombre de una tabla de `int`; si lo volvemos a indexar (**`pTabla[i][j]`**, por ejemplo) obtendremos un `int`. Es decir, con este método de reserva de memoria dinámica para tablas multidimensionales podemos utilizar la indexación múltiple sin necesidad de conocer el valor máximo de ninguna de las componentes en tiempo de compilación.

11.3. Entrada y Salida en C.

Muchas aplicaciones requieren escribir o leer información de un dispositivo de almacenamiento masivo. Tal información se almacena en el dispositivo de almacenamiento en forma de un archivo de datos. Los archivos de datos permiten almacenar información de modo permanente, pudiendo acceder y alterar la misma cuando sea necesario.

En C existe un conjunto extenso de funciones de biblioteca para crear y procesar archivos de datos. En C podrá trabajar con ficheros de dos tipos:

- texto: se trabaja con secuencias de caracteres divididos en líneas por el carácter nueva línea (**`'\n'`**).

- binarios: se trabaja con una secuencia de octetos que representan zonas de memoria.

Cuando se trabaja con archivos el primer paso es establecer un área de búffer, donde la información se almacena temporalmente mientras se está transfiriendo entre la memoria del ordenador y el archivo de datos. Este área de búffer (también llamada *flujo* o *stream*) la gestiona la propia biblioteca estándar, y permite leer y escribir información del archivo más rápidamente de lo que sería posible de otra manera. Para poder intercambiar información entre el área de buffer y la aplicación que utiliza las funciones de biblioteca, se utiliza un puntero a dicha zona de buffer. En la siguiente declaración:

`FILE *ptvar;`

`FILE` (se requieren letras mayúsculas) es un tipo especial de estructura que establece el área de búffer, y **`ptvar`** es la variable puntero que indica el principio de este área. El tipo de estructura **`FILE`**, que está definido en el archivo de cabecera del sistema **`stdio.h`**, también se conoce como **descriptor de fichero**.

Las operaciones de entrada y salida en ficheros se realiza a través de funciones de la biblioteca estándar cuyos prototipos están en **`stdio.h`**, no existiendo palabras reservadas del lenguaje que realicen estas operaciones.

11.3.1. Funciones de apertura y cierre

Un archivo debe ser "abierto" para poder realizar operaciones sobre él. La apertura de un archivo asocia el archivo indicado a un descriptor de fichero, y devuelve su puntero. Este puntero es el que se deberá utilizar en lo sucesivo para cualquier operación sobre el fichero. Cuando se ha finalizado de operar sobre él, debe ser cerrado para que el sistema operativo libere los recursos que le tenía asignados.

Función `fopen`

`FILE *fopen(const`[2]` char *nombre, const char *modo);`

Esta función abre el archivo **`nombre`** y devuelve como resultado de la función el descriptor de fichero, o **`NULL`** si falla en el intento.

La cadena de caracteres **`modo`** determina cómo se abre el fichero; los valores más utilizados para dicho parámetro son los siguientes:

Modo	Operación	Si existe ...	Si no existe ...
`r`	lectura	lo abre y se posiciona al principio	error
`w`	escritura	borra el antiguo y crea uno nuevo	lo crea
`a`	escritura	lo abre y se posiciona al final	lo crea

Función `fclose`

`int fclose(FILE *fp);`

Esta función cierra el archivo referenciado por **`fp`**. Esto implica descartar cualquier buffer de entrada no leído, liberar cualquier buffer asignado automáticamente y cerrar el flujo. Devuelve **`EOF`** si ocurre cualquier error y **`0`** en caso contrario.

Antes de terminar cualquier programa, debe asegurarse de que ha cerrado todos los archivos que haya abierto previamente.

11.3.2. Funciones de entrada y salida

Función `fgetc`

`int fgetc(FILE *fp);`

[2] El calificador de tipo `const` indica que el valor del parámetro no se modifica dentro de la función llamada.

Esta función devuelve el siguiente carácter al último leído de **fp** como un **int**, o **EOF** si se encontró el fin de archivo o un error.

Función fgets

char * fgets(char *s, int n, FILE *fp);

Esta función lee hasta **n-1** caracteres (a partir del último carácter leído de **fp**) en la cadena **s**, deteniéndose antes si encuentra nueva línea o fin de fichero. Si se lee el carácter de nueva línea se incluye en la cadena **s**. La cadena **s** se termina con **'\0'**. La función devuelve **s**, o **NULL** si se encuentra fin de fichero o se produce un error.

Función fputc

int fputc(int c, FILE *fp);

Esta función escribe el carácter **c** (convertido a **unsigned char**) a continuación del último caracter escrito en **fp**. Devuelve el carácter escrito, o **EOF** en caso de error.

Función fputs

int fputs (const char *s, FILE *fp);

Esta función escribe la cadena s (no necesita que contenga **'\n'**, pero sí que termine en **'\0'**) en **fp**. Devuelve un número no negativo si no ha habido problemas, o **EOF** en caso de error.

Copia de un fichero de texto

El siguiente ejemplo realiza la copia de un fichero de texto en otro utilizando las funciones de lectura y escritura de caracteres vistas hasta ahora. Los nombres de los ficheros origen y destino se proporcionan por la línea de comandos:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  FILE * fent = NULL;
  FILE * fsal = NULL;
  int    caracter;

  if (argc != 3)
    puts("Se necesitan dos parametros: origen y destino\n");
  else
    {
      if ((fent = fopen(argv[1], "r")) == NULL)
        puts("No se puede abrir el fichero origen\n");
      else
        {
          if ((fsal = fopen(argv[2], "w")) == NULL)
            puts("No se puede abrir el fichero destino\n");
          else
            {
              // Leemos cada caracter del fichero de entrada ...
              while ((caracter = fgetc(fent)) != EOF)
                // ... y lo escribimos en el fichero de salida.
                fputc(caracter, fsal);
              fclose(fsal);
            }
          fclose(fent);
        }
```

```
    }

  return 0;
}
```

11.3.3. Salida con formato: función `fprintf`

Con las funciones vistas hasta ahora, sólo es posible imprimir secuencias de caracteres. Sin embargo, en muchas ocasiones es necesario imprimir datos numéricos, y con un cierto formato. Para lograr esto se utiliza la función:

`int fprintf(FILE *stream, const char *format, ...);`

Esta función escribe la cadena de formato **`format`** en **`stream`**. Devuelve el número de caracteres escritos, o un valor negativo si se produce algún error. La cadena de formato contiene dos tipos de objetos: caracteres ordinarios (que se copian en el flujo de salida), y especificaciones de conversión (que provocan la conversión e impresión de los argumentos). Cada especificación de conversión comienza con el carácter **`%`** y termina con un carácter de conversión.

Los caracteres de conversión más utilizados son los siguientes:

`c` para representar caracteres; el argumento se convierte previamente a **`unsigned char`**.

`u,o,X,x` para representar enteros sin signo en notación decimal, octal (sin ceros al principio), hexadecimal (sin el prefijo **`0X`** y con los dígitos **`A`** a **`F`** en mayúsculas), y hexadecimal (sin el prefijo **`0x`** y con los dígitos **`a`** a **`f`** en minúsculas), respectivamente.

`d` para representar enteros con signo en notación decimal.

`s` para representar cadenas de caracteres; los caracteres de la cadena son impresos hasta que se alcanza un **`'\0'`** o hasta que ha sido impreso el número de caracteres indicados por la precisión.

`f,E,e` para representar números de tipo **`double`**, en notación decimal, en notación científica con el exponente **`E`** en mayúscula, o en notación científica con el exponente **`e`** en minúscula, respectivamente. Por defecto se escriben seis dígitos significativos.

`G,g` para representar números de tipo **`double`**; es equivalente a la especificación **`E,e`** si el exponente del número expresado en notación científica es inferior a -4 o mayor o igual que la precisión (que por defecto es seis), y equivalente a **`f`** en caso contrario.

`p` se utiliza para imprimir el valor de un puntero (representación dependiente de la implementación).

Entre el **`%`** y el carácter de conversión pueden existir, en el siguiente orden:

1. **banderas**: modifican la especificación y pueden aparecer en cualquier orden

   `-` el dato se ajusta a la izquierda dentro del campo (si se requieren espacios en blanco para conseguir la longitud del campo mínima, se añaden *después* del dato en lugar de *antes*).

   `+` Cada dato numérico es precedido por un signo (**`+`** o **`-`**). Sin este indicador, sólo los datos negativos son precedidos por el signo **`-`** .

   `espacio` si el primer carácter no es un signo, se prefijará un espacio

   `0` Hace que se presenten ceros en lugar de espacios en blanco en el relleno de un campo. Se aplica sólo a datos que estén ajustados a la derecha dentro de campos de longitud mayor que la del dato

Con las conversiones tipo **o** y tipo **x**, hace que los datos octales y hexadecimales sean precedidos por **0** y **0x**, respectivamente. Con las conversiones tipo `e`, tipo `f` y tipo `g`, hace que se presenten todos los números en coma flotante con un punto, aunque tengan un valor entero. Impide el truncamiento de los ceros de la derecha realizada por la conversión tipo **g**.

número Un número que estipula un ancho mínimo de campo. El argumento convertido será impreso en un campo de por lo menos esta amplitud, y en una mayor si es necesario. Si el argumento convertido tiene menos caracteres que el ancho de campo, será rellenado a la izquierda (o derecha, si se ha requerido justificación a la izquierda) para completar el ancho de campo. El carácter de relleno normalmente es espacio, pero es `0` si está presente la bandera de relleno con ceros.

. Un punto, que separa el ancho de campo (número previo) de la precisión (número posterior).

número Un número, la precisión, que estipula el número máximo de caracteres de una cadena que serán impresos, o el número de dígitos que serán impresos después del punto decimal para conversiones `e`, `E` o `f`, o el número de dígitos significativos para conversiones `g` o `G`, o el número mínimo de dígitos que serán impresos para un entero (serán agregados ceros al principio para completar el ancho de campo necesario).

2. Un modificador de tipo:

hh indica que el argumento correspondiente va a ser impreso como **char** o **unsigned char**.

h indica que el argumento correspondiente va a ser impreso como **short** o **unsigned short**.

l indica que el argumento es **long** o **unsigned long**.

ll indica que el argumento es **long long** o **unsigned long long**.

L indica que el argumento es **long double**.

11.3.4. Entrada con formato: función `fscanf`

`int fscanf(FILE *fp, const char *format, ...);`

Esta función lee a través del descriptor de fichero **fp** bajo el control de **format**, y asigna los valores convertidos a través de los argumentos, cada uno de los cuales debe ser un puntero. La función devuelve **EOF** si se llega al final del archivo o se produce un error antes de la conversión; en cualquier otro caso devuelve el número de valores de entrada convertidos y asignados. La cadena de formato contiene especificaciones de conversión, que se utilizan para interpretar la entrada. La cadena de formato puede contener:

- espacios o tabuladores, que se ignoran.
- caracteres ordinarios (distintos de %), que se espera que coincidan con los siguientes caracteres que no son espacio en blanco del flujo de entrada.
- especificaciones de conversión, consistentes en **%**, un carácter optativo de supresión de asignación *****, un número optativo que especifica una amplitud máxima de campo, una **h**, **l** o **L** optativa que indica la amplitud del objetivo, y un carácter de conversión.

Una especificación de conversión determina la conversión del siguiente campo de entrada. Normalmente el resultado se sitúa en la variable apuntada por el argumento correspondiente, excepto en el caso que se indique supresión de asignación con ***** como en **%*s**. El campo de entrada simplemente se salta; no se hace asignación. Un campo de entrada está definido por una cadena de caracteres diferentes del espacio en blanco; se extiende hasta el siguiente carácter de espacio en blanco o hasta que el ancho de campo, si está especificado, se haya agotado.

Esto implica que **fscanf** leerá más allá de los límites de la línea para encontrar su entrada, ya que las nuevas líneas son espacios en blanco. Los caracteres de espacio en blanco son el blanco, tabulador, nueva línea, retorno de carro, tabulador vertical y avance de línea.

El carácter de conversión indica la interpretación del campo de entrada. El argumento correspondiente debe ser un puntero. Los caracteres de conversión pueden ser los siguientes:

d `int *`; entero decimal.

i `int *`; entero. El entero puede estar en octal (iniciado con `0`) o hexadecimal (iniciado con `0x` o `0X`).

o `int *`; entero octal (con o sin cero al principio).

u `unsigned int *`; entero decimal sin signo.

x `int *`; entero hexadecimal (con o sin `0x` o `0X`).

c `char *`; caracteres. Los siguientes caracteres de entrada se pondrán en la cadena indicada, hasta el número especificado por el ancho del campo; el valor por omisión es `1`. No se agrega **'\0'**. En este caso se suprime el salto normal sobre los caracteres de espacio en blanco; para leer el siguiente carácter que no sea blanco, hay que usar `%1s` (lee un carácter y lo guarda como una cadena).

s `char *`; cadena de caracteres que no es espacio en blanco (no entrecomillados). Apunta a una cadena de caracteres suficientemente grande para contener la cadena y un **'\0'** terminal que se le agregará.

e, f, g `float *`; número de punto flotante. El formato de entrada para los `float` es un signo optativo, una cadena de números posiblemente con un punto decimal, y un campo optativo de exponente con una `E` o `e` seguida posiblemente de un entero con signo.

n `int *`; escribe en el argumento el número de caracteres consumidos o leídos hasta el momento por esta llamada. No se lee entrada alguna. La cuenta de elementos convertidos no se incrementa.

11.3.5. Funciones de lectura/escritura binaria

Hasta ahora se ha estudiado la lectura/escritura en ficheros que son tratados como una secuencia de caracteres. Ahora se describen las funciones que se utilizan para tratar los ficheros como una secuencia de octetos, leyendo o escribiendo trozos de memoria.

Lectura binaria: función `fread`

`unsigned fread(void *ptr, size_t size, size_t nobj, FILE *fp);`

Esta función lee de **fp** en la cadena **ptr** hasta **nobj** objetos de tamaño **size** . La función devuelve el número de objetos leídos (este valor puede ser menor que el número solicitado).

Escritura binaria: función `fwrite`

`size_t fwrite(const void *ptr, size_t size, size_t nobj, FILE *fp);`

Esta función escribe de **ptr nobj** objetos de tamaño **size** en **fp** . Devuelve el número de objetos escritos, que es menor que **nobj** en caso de error.

11.3.6. Funciones especiales

Detección de fin de fichero: función `feof`

`int feof(FILE *fp);`

Esta función devuelve un valor diferente de cero si se ha llegado al final del archivo. No se detecta el final del fichero hasta que no se intente leer después de haber finalizado la lectura del fichero. Si se lee el último carácter NO se ha llegado al final del archivo. Para que esta función devuelva un valor diferente de cero hay que intentar leer después del último carácter.

Detección de error: función `ferror`

`int ferror(FILE *fp);`

Esta función devuelve un valor diferente de cero si se ha producido algún error.

11.3.7. Flujos estándar: `stdin`, `stdout` y `stderr`

Al hablar de ficheros se ha introducido el concepto de búfferes y descriptores o punteros a ficheros. Existen unos flujos de texto (*streams*) especiales que se abren automáticamente cuando un programa empieza a ejecutarse, y sus descriptores son:

`stdin` entrada estándar.

`stdout` salida estándar.

`stderr` salida de errores.

A partir de ese momento se puede utilizar estos identificadores (del tipo **`FILE *`**) con cualquiera de las funciones vistas previamente. Así, por ejemplo:

`printf("hola mundo");`

`scanf("%d", &edad);`

son equivalentes a

`fprintf(stdout, "hola mundo");`

`fscanf(stdin, "%d", &edad);`

La salida estándar y la salida de errores están por defecto dirigidas a la pantalla del ordenador.

11.3.8. Errores frecuentes en el uso de las funciones de E/S

El error más simple, y no por ello infrecuente, es olvidar que **`scanf`** recibe como parámetros las direcciones donde almacenar los valores que recibe de la entrada estándar.

```
float precio;
scanf("%f", precio);
```

en vez de

```
float precio;
scanf("%f", &precio);
```

Recuerde que cuando se trata de leer cadenas de caracteres se le pasa el nombre de la tabla.

```
char  nombre[25];
scanf("%s", nombre);
```

Otro error relacionado con el uso de punteros es que el puntero utilizado no apunte a una zona de memoria rerservada.

En definitiva, es necesario comprobar antes de utilizar una variable de tipo puntero si apunta a una zona de memoria válida, y con el tamaño suficiente.

scanf no es perfecto

Otro posible error en el uso de scanf es suponer que siempre funcione como nosotros esperamos (cosa que en general no se debe suponer de ninguna función, ya sea de biblioteca o codificada por nosotros, y especialmente si estamos depurando el programa). Por ejemplo, edite y compile el siguiente código:

```
#include <stdio.h>

int main()
{
  float saldo;
  char accion;

  fprintf(stdout, "Introduzca accion (D) y saldo\n");
  scanf(" %c %f", &accion, &saldo);
  if ('D' == accion)
    saldo = saldo * 2;
  else
    saldo = saldo / 2.0;
  fprintf(stdout, "Accion : %c\n", accion);
  fprintf(stdout, "Saldo : %f\n", saldo);

  return(0);
}
```

Este programa funcionará correctamente si introduce una cadena de caracteres y un número:

```
salas@318CDCr12:~$  a.out
Introduzca accion (D) y saldo
D 1000
Accion : D
Saldo : 2000.000000
salas@318CDCr12:~$
```

Pero si ejecuta el programa y al introducir el número se equivoca:

```
salas@318CDCr12:~$  a.out
Introduzca accion (D) y saldo
D q123
Accion : D
Saldo : -316884466973887584331540463616.000000
salas@318CDCr12:~$
```

El programa dejará valores inesperados en la variable **saldo** si el segundo valor a leer no es del tipo numérico. Este valor podrá variar en distintas ejecuciones.

Por lo tanto, habría que comprobar que **scanf** (o **fscanf**) devuelve tantos valores como se espera. Si no, existe la posibilidad de que las variables utilizadas en **scanf** conserven los valores previos, o incluso cualquier valor no esperado si no estaban iniciadas.

Observe también que **scanf** se detiene en la lectura del número en el momento que encuentra un espacio en blanco genérico o un carácter no numérico. En el siguiente ejemplo intentamos introducir la cantidad **1000** y tecleamos en vez de un cero final el carácter **o** :

```
salas@318CDCr12:~$ a.out
Introduzca accion (D) y saldo
D 100o
Accion : D
Saldo : 200.000000
salas@318CDCr12:~$
```

scanf lee el número 100, y continua su ejecución normalmente, ignorando el carácter '**o**'.

scanf y los espacios en blanco

Es peligroso abandonar espacios en blanco finales en la cadena de formato de **scanf** . Compruébelo con el siguiente programa, que deja un inocente espacio en blanco antes de terminar la cadena de formato:

```
#include <stdio.h>

int main()
{
   int precio;

   printf("Precio del libro?: ");
   scanf(" %d ", &precio);
   printf("\n %d \n ", precio);

   return 0;
}
```

scanf lee el precio, y a continuación se queda descartando los espacios en blanco (o tabuladores, caracteres de nueva línea, etc.). Parece que el programa esté bloqueado. Para terminar la lectura del precio es necesario que se teclee algún carácter distinto de espacio. Para corregir este programa basta eliminar ese espacio en blanco final.

Esta propiedad de **scanf** se puede utilizar para leer datos con un formato conocido previamente. Por ejemplo, una fecha de la forma **dd/mm/yyyy**: los tres números están separados por barras. La lectura se puede hacer con el siuiente programa:

```
#include <stdio.h>

int main()
{
   int dia;
   int mes;
   int year;

   printf("Fecha de nacimiento ? (dd/mm/yyyy): ");
   scanf(" %d/ %d/ %d", &dia, &mes, &year);
   printf("Fecha de nacimiento es %d- %d- %d \n",
            dia, mes, year);

   return 0;
}
```

scanf y fgets

Otro error en el uso de **scanf** es leer un número con **scanf(" %d", ...)** y a continuación intentar leer una cadena de caracteres con la función **fgets**: da la impresión de que el código se salta la llamada a la función **fgets**. Observe el siguiente ejemplo:

```
#include <stdio.h>
#define MAX 25

int main()
{
  int precio;
  char nombre[MAX];

  printf("Precio del libro?: ");
  scanf(" %d", &precio);
  printf("Nombre del libro?: ");
  fgets(nombre, MAX-1, stdin);
  printf("\nNombre: %s Precio: %d euros\n ", nombre, precio);

  return 0 ;
}
```

Si ejecutamos el programa generado, tendremos:

```
salas@318CDCr12:~$ a.out
Precio del libro?: 12
Nombre del libro?:
Nombre:                    Precio: 12 euros
salas@318CDCr12:~$ a.out
Precio del libro?: 26 El perfume
Nombre del libro?:
Nombre: El perfume Precio: 26 euros
salas@318CDCr12:~$ a.out
```

El comportamiento observado se debe a lo siguiente: en la primera ejecución, la sentencia **scanf** lee el precio del libro, y se detiene en el carácter de nueva línea, sin leerlo. La función **fgets** reanuda la lectura donde **scanf** la dejó, lee el carácter nueva línea, y como su condición de finalización es leer un carácter nueva línea detiene su ejecución, ignorando el resto de la entrada.

En la segunda ejecución, **scanf** lee el precio y se detiene al detectar un separador (en este caso un espacio en blanco, no nueva línea). **fgets** continúa leyendo donde **scanf** se quedó, y no se detiene hasta llegar al carácter de nueva línea.

Para no depender de cómo el usuario introduzca los datos, una opción es leer todos los caracteres de nueva línea pendientes antes de **fgets**, mediante la siguiente sentencia intercalada entre la lectura del precio y la lectura del título:

```
while (fgetc(stdin) != '\n');
```

scanf y los formatos numéricos

Existen unas pequeñas diferencias entre las especificaciones de formato de tipo numérico en **printf** y **scanf**. Se presentan en la siguiente tabla:

	printf	scanf
float	%f	%f
double	%f	%lf
long double	%Lf	%Lf

Lecturas de cadenas de caracteres

Para leer cadenas de caracteres con espacios en blanco no es posible usar **scanf**, puesto que considera los espacios en blanco separadores de campo. Suponga que desea leer el título de un libro con **scanf**: a priori no conoce cuantas palabras va a contener el título del libro, ni el autor, etc.

Para solucionar este 'problema' se utiliza la función **fgets**, que lee hasta encontrar un carácter de nueva línea (y no un espacio genérico como hace **scanf**).

Otro inconveniente de **scanf** para este tipo de situaciones es que no comprueba si se rebasan los límites de la cadena donde se almacena la entrada estándar. Ya se sabe que rebasar los límites del espacio de memoria reservado conduce a errores de memoria, muchos de ellos de difícil localización. Por eso es también preferible usar **fgets**, al que se le indica el tamaño máximo de almacenamiento.

Pero no hay que olvidar que **fgets** no sustituye el carácter de nueva línea **'\n'** por el terminador **'\0'**.

Otra solución a la lectura de datos genérica es leer líneas de texto completas (de fichero o de entrada estándar) con **fgets**. Una vez que tengamos una cadena de caracteres, podemos procesarla para determinar si es correcta o no, qué formato tiene, etc. Incluso se pueden procesar con la función **sscanf**, o con funciones creadas por el programador. **sscanf** funciona exactamente igual que **scanf** o **fscanf**, con la única diferencia que lee los datos de una cadena que se le pasa como primer parámetro:

```
int sscanf(char *cad, const char *format, ...)
```

11.4. Funciones de Cadenas de Caracteres

El uso de cadenas de caracteres es tan frecuente que la biblioteca estándar de C incorpora algunas funciones de manejo de cadenas. Sus declaraciones o prototipos se encuentran en **string.h**, y enunciaremos algunas de ellas.

11.4.1. Funciones de copia

char * strcpy(char *dest, const char *orig);

Copia la cadena **orig** en la cadena **dest**, incluyendo **'\0'**. Devuelve **dest**.

char * strncpy(char *dest, const char *orig, int n);

Copia hasta **n** caracteres de la cadena **orig** en la cadena **dest**. Devuelve **dest**. Rellena con **'\0'** si **orig** tiene menos de **n** caracteres.

11.4.2. Funciones de encadenamiento

char * strcat(char *inicio, const char *final);

Concatena la cadena **final** al final de la cadena **inicio**. Devuelve **inicio**.

char * strncat(char *inicio, const char *final, int n);

Concatena hasta **n** caracteres de la cadena **final** al final de la cadena **inicio**, terminando con **'\0'**. Devuelve **inicio**.

11.4.3. Funciones de comparación

Recuerde que no es posible en C comparar dos cadenas con el operador **==**. En su lugar se utilizan las funciones siguientes:

int strcmp(char *orig, char *dest);

Compara la cadena **orig** con la cadena **dest**. Devuelve un número negativo si **orig** es anterior a **dest**, un número positivo si **orig** es anterior a **dest**, o cero si las dos cadenas son iguales.

int strncmp(char *orig, char *dest, int n);

Compara hasta **n** caracteres de la cadena **orig** con la cadena **dest**. Devuelve un número negativo si **orig** es anterior a **dest**, un número positivo si **orig** es anterior a **dest**, o cero si las dos cadenas son iguales.

11.4.4. Funciones de búsqueda

char * strchr(char *s, char c);

Devuelve un puntero a la primera ocurrencia de **c** en **s**, o NULL si no está presente.

char * strrchr(char *s, char c);

Devuelve un puntero a la última ocurrencia de **c** en **s**, o NULL si no está presente.

int strlen(char *s);

Devuelve la longitud de **s**.

La función **strlen** devuelve la longitud efectiva de la cadena, sin contar el terminador. Luego para copiar una cadena en otra hay que reservar un carácter más que la longitud que devuelva **strlen**.

Por otra parte, las funciones de copia y concatenación de caracteres NO reservan espacio en memoria para las cadenas destino, y además suponen que el espacio reservado es suficiente.

11.4.5. Otras funciones

Otras funciones que pueden ser de interés, y que se encuentran declaradas en **stdlib.h** son:

int atoi (const char *s);

Devuelve el valor de **s** convertido a `int`.

double atof (const char *s);

Devuelve el valor de **s** convertido a `double`.

Parte II

Anexos

Anexo 1

El Entorno del Centro de Cálculo

Índice

1.1. Introducción **1-1**
1.2. Arranque del sistema **1-1**
1.3. Funcionamiento del Entorno **1-2**
1.3.1. El terminal 1-2
1.4. Los dispositivos de memoria externos **1-3**
1.4.1. Conexión 1-4
1.4.2. Desconexión 1-4

1.1. Introducción

En este anexo se presenta el entorno en el que se realizarán las prácticas correspondientes a la asignatura de Fundamentos de la Programación.

Las prácticas se realizan en los equipos disponibles en el Centro de Cálculo de la Escuela Técnica Superior de Ingenieros, por lo que esta explicación se centrará en el entorno que presentan dichos equipos.

1.2. Arranque del sistema

El sistema operativo linux está instalado en diversas salas del Centro de Cálculo de la Escuela Técnica Superior de Ingenieros.

Al encender cualquier equipo en alguna de estas salas, se presenta un menú con las diferentes opciones con las que pueden iniciarse los ordenadores del centro de cálculo. Para las prácticas de Fundamentos de la Programación, debe seleccionarse la opción etiquetada como `UBUNTU`.

Una vez seleccionada la opción correcta, el sistema se cargará en tres fases:

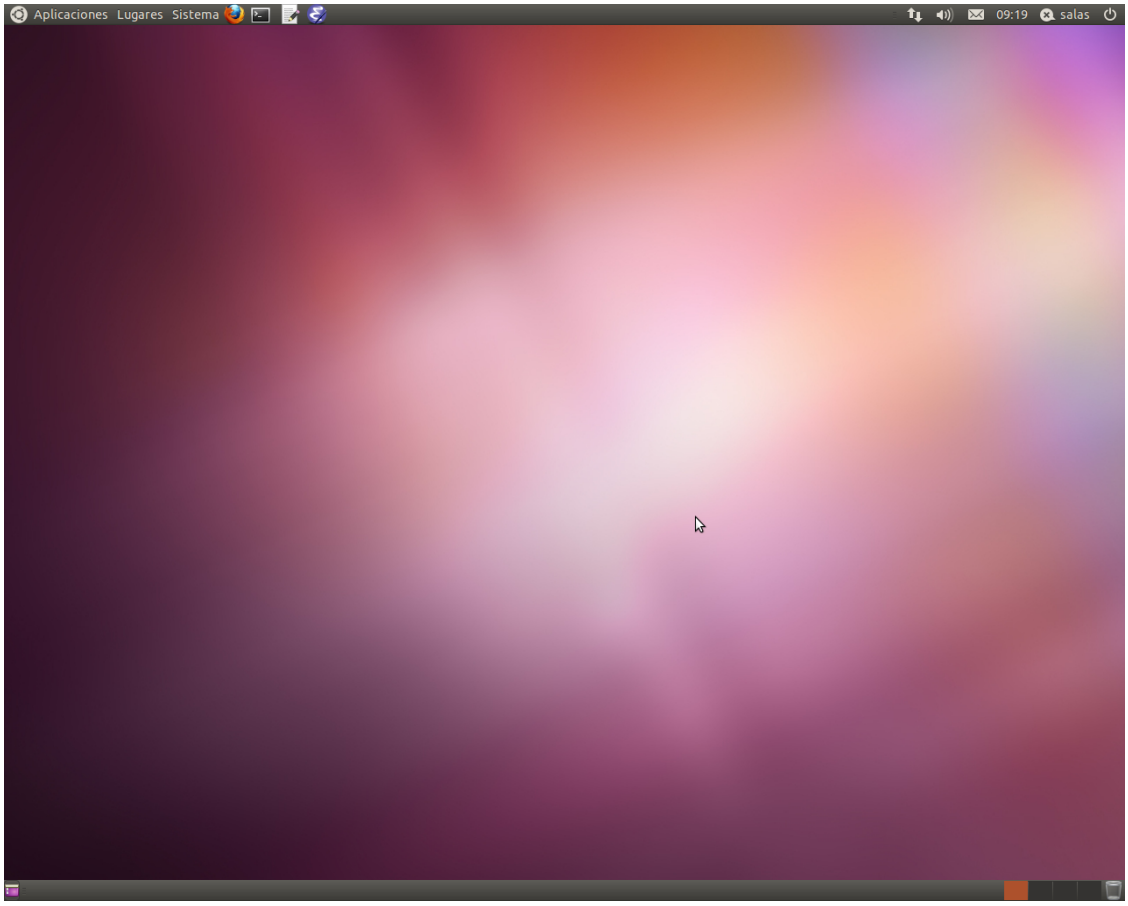
- en la primera fase, se actualiza la imagen del disco con el S.O. seleccionado. Este proceso puede ser lento si la imagen no está disponible en la caché del disco, así que sea paciente.
- en la segunda fase se arranca el sistema operativo linux, que es el que se utilizará en el laboratorio.
- en la tercera y última fase se inicia el entorno de ventanas utilizado, que en el caso del laboratorio de programación es Gnome.

1.3. Funcionamiento del Entorno

Una vez finalizada la carga del entorno de ventanas, puede observarse que la pantalla está dividida en tres zonas:

- una barra superior (barra de menú) mediante la que se accede a una serie de menús desplegables, a través de los cuales podemos lanzar diferentes aplicaciones.
- una barra de estado, en la parte inferior del terminal, en la que aparecerá una entrada por cada aplicación que tengamos ejecutándose.
- la zona central, denominada Escritorio, donde se ejecutarán las diferentes aplicaciones.

El aspecto del entorno se recoge en la siguiente figura:



1.3.1. El terminal

Todas las órdenes que debe realizar el sistema operativo se deben introducir por teclado. Para ello, es necesario que se utilice la aplicación **`gnome-terminal`**, un emulador de terminal que acompaña al entorno de ventanas utilizado.

Para arrancar la aplicación, bastará con pulsar sobre el icono de la barra de menú según se indica en la siguiente figura:

Cuando se inicia dicha aplicación, aparece el *prompt*, indicando que el procesador de órdenes está preparado para ejecutarlas. En el entorno del centro de cálculo, el *prompt* es:

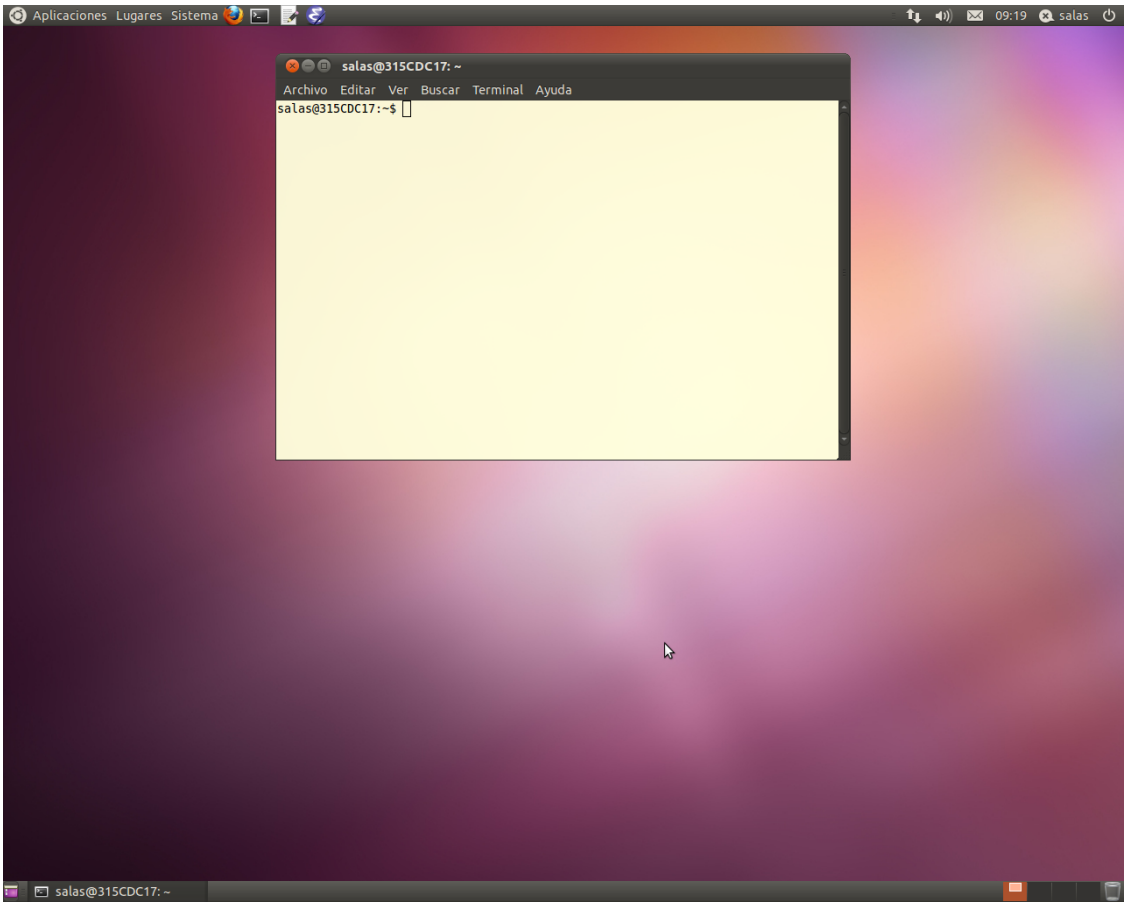
salas@<*sala*>CDCr<*num*>:<*directorio*>$

siendo <*sala*> el número de la sala en la que está el equipo, <*num*> el número del equipo dentro de la sala, y <*directorio*> el directorio en el que se está trabajando.

Ejemplo: Si nos encontramos en el directorio **/var/lib**, utilizando el ordenador número 12 de la sala 318 del Centro de Cálculo, el prompt sería:

salas@318CDCr12:/var/lib$

La siguiente figura presenta el entorno una vez arrancado el terminal, y a la espera de recibir las órdenes:



Cuando se teclea una orden, esta irá apareciendo en el terminal, a continuación del *prompt*. Durante la escritura se podrá editar la línea, corrigiendo lo que se considere necesario. Una vez terminada de escribir la orden, es necesario pulsar la tecla **<INTRO>** para que el sistema operativo la procese y la ejecute.

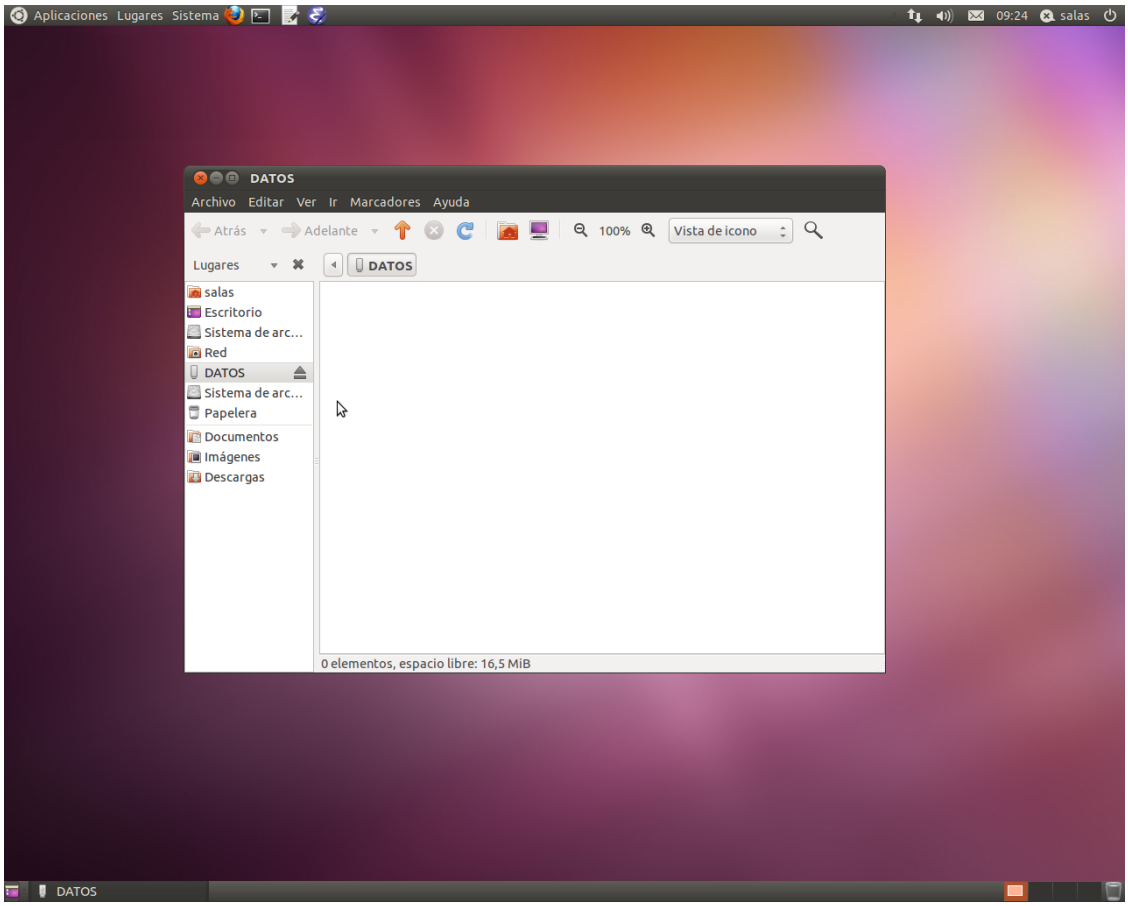
1.4. Los dispositivos de memoria externos

En los equipos del centro de cálculo los datos que se almacenan en el disco duro se pierden al desconectarlos, por lo que se hace necesario realizar una copia de seguridad del trabajo realizado en cada sesión antes de apagar el ordenador. La forma habitual de guardar los datos es utilizar un dispositivo de memoria externo con interfaz USB.

1.4.1. Conexión

Para acceder a un dispositivo de memoria USB es suficiente con conectar el dispositivo al ordenador. El equipo lo detecta automáticamente y abre un explorador de ficheros con el contenido del dispositivo.

La siguiente figura recoge el explorador de ficheros abierto cuando se inserta un dispositivo USB con la etiqueta **DATOS**:



Para hacer referencia al contenido del dispositivo desde la línea de comandos, se debe hacer referencia al directorio **/media/<*etiqueta*>**, siendo <*etiqueta*> la etiqueta del dispositivo. Si careciera de etiqueta, utilizaría la etiqueta **disk**.

1.4.2. Desconexión

Antes de retirar el dispositivo USB del ordenador, es necesario desmontarlo. No debe olvidar **NUNCA** realizar esta operación antes de desconectar el dispositivo, porque podría dar lugar a la pérdida de información.

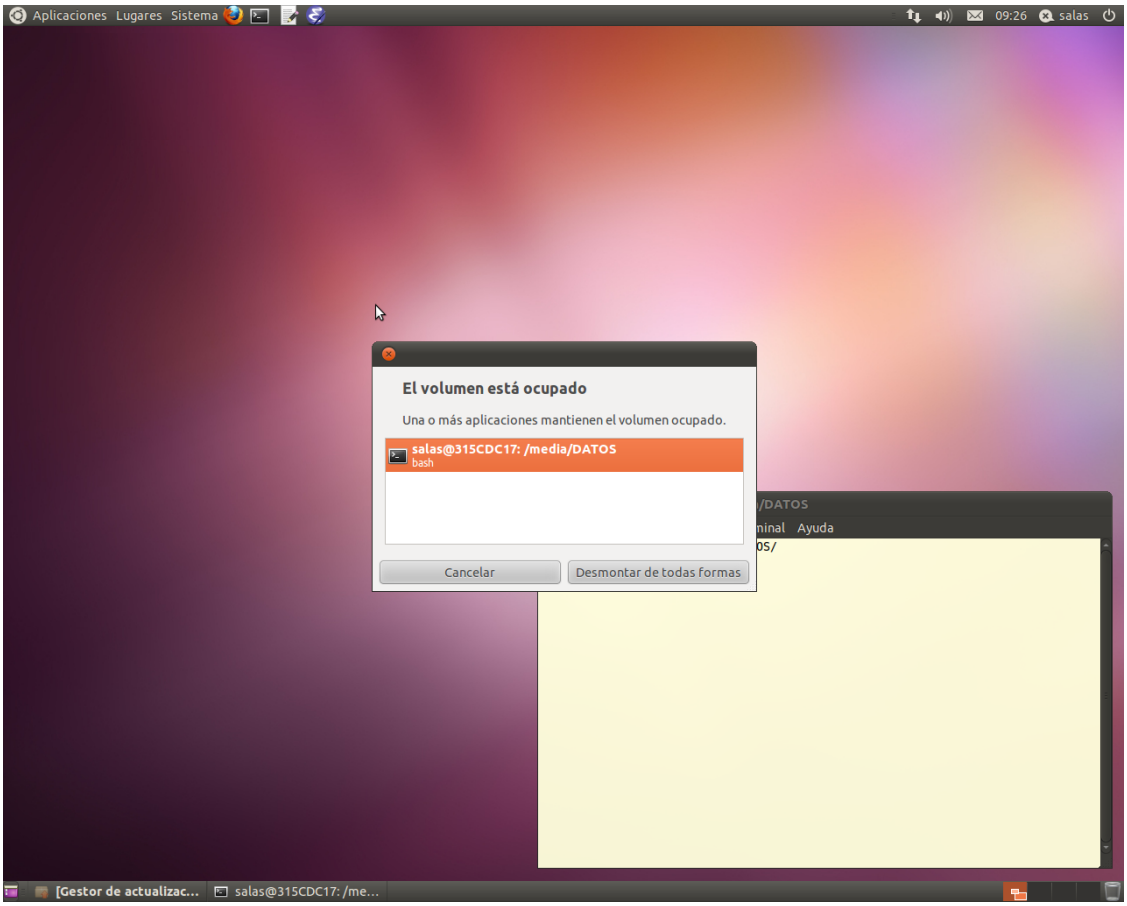
Para desmontar el dispositivo debe ejecutar en un terminal la siguiente orden:

umount /media/<*etiqueta*>

Para poder desmontar el dispositivo, el directorio sobre el que se ha montado (**/media/<*etiqueta*>**) no debe estar ocupado; esto significa que no haya ningún comando ejecutándose que utilice algún archivo de dicho directorio, o que exista algún terminal ubicado en dicho directorio. Si se intenta desmontar el dispositivo dándose alguna de estas circunstancias, el sistema operativo avisará de ello, dando la posibilidad de cancelar la operación o continuar con ella. Para evitar posibles problemas, recomendamos cancelar la

operación, corregir esta situación (esperando a que los comandos finalicen, y cerrando todos los terminales que estuvieran ubicados en el directorio) y volver a intentarlo.

La siguiente figura muestra el mensaje de error cuando se intenta desmontar el dispositivo; observe cómo el terminal de la figura está ubicado justamente en el directorio de montaje del dispositivo, que es lo que da lugar al error:



Si se cierra el terminal y se repite la orden, no se producirá el error y el dispositivo se desmontará normalmente.

Una vez desmontado el dispositivo, puede retirarlo del ordenador sin peligro.

Anexo 2

El Sistema Operativo UNIX

Índice

2.1. Introducción **2-1**

2.2. Estructura de directorios y sistema de ficheros **2-2**

2.2.1. La estructura de directorios 2-2

2.2.2. Permisos 2-3

2.3. Formato general de las órdenes **2-4**

2.3.1. Ayuda en línea 2-4

2.3.2. Operaciones sobre el sistema de ficheros 2-4

2.3.3. Operaciones sobre directorios 2-7

2.3.4. Comandos para visualización de ficheros 2-8

2.3.5. Comandos informativos 2-9

2.3.6. Otros comandos 2-10

2.4. Funciones especiales del intérprete de comandos **2-11**

2.4.1. Caracteres ordinarios y caracteres de control 2-11

2.4.2. Metacaracteres 2-12

2.4.3. Redireccionamiento de la entrada/salida 2-12

2.5. Procesos **2-13**

2.5.1. Intercomunicación entre procesos 2-13

2.5.2. Control de trabajos 2-13

2.5.3. Órdenes de información sobre procesos y trabajos 2-14

2.5.4. Órdenes de cambio de estado de un trabajo 2-15

2.1. Introducción

En este anexo se explica de manera resumida el conjunto de órdenes de linux más utilizado en el desarrollo de la asignatura.

Aunque las explicaciones de los comandos se refieren al sistema operativo linux, que es el que utilizan los ordenadores del Centro de Cálculo, casi en su totalidad son aplicables a cualquier sistema operativo del tipo UNIX.

2.2. Estructura de directorios y sistema de ficheros

Los ficheros sirven para guardar información, y normalmente están almacenados en disco magnético. Todo el sistema UNIX está basado en ficheros (por ejemplo las órdenes están en ficheros, y los propios directorios son ficheros). Cada fichero tiene un nombre, un contenido, un lugar para almacenarlo y cierta información de tipo administrativo, como por ejemplo quién es el dueño y cuál es su tamaño.

El nombre de un fichero puede tener más de 8 caracteres y normalmente está limitado a 255 (depende de la máquina). No se deben usar caracteres como el guión y el espacio. Se suele intercalar el carácter punto dentro del nombre, y a lo que aparece tras dicho punto se le denomina extensión; la extensión se suele utilizar para indicar qué tipo de información almacena el fichero.

2.2.1. La estructura de directorios

El sistema de ficheros UNIX está organizado en directorios con una estructura jerárquica. Existe un directorio básico llamado **directorio raíz** y que se especifica mediante el carácter **/**. Dentro de este puede haber subdirectorios y dentro de estos puede haber a su vez otros subdirectorios hasta todos los niveles que se quiera. Para separar los niveles también se usa el carácter **/**.

Dentro de un directorio NO pueden coexistir dos ficheros con el mismo nombre, aunque sí lo podrían hacer en directorios distintos. Por otra parte, los nombres de ficheros discriminan entre las mayúsculas y las minúsculas, de manera que **integrales.c** es distinto de **Integrales.c**, **INTEGRALES.C**, **integrales.C**, etc.

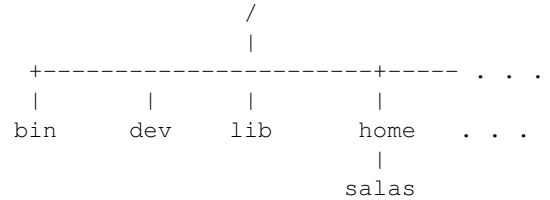
Cada usuario tiene un *directorio de inicio de sesión*, que en el caso de los alumnos de este laboratorio es el directorio **/home/salas**, y siempre existe un *directorio de trabajo*, o directorio actual, que es el directorio en el que se encuentra el procesador de órdenes en cada momento.

Al iniciar la sesión, el directorio de trabajo es el directorio de inicio de sesión. El directorio de trabajo se puede cambiar con la orden **cd** (*Change Directory*), pero el directorio de inicio de sesión será siempre el mismo. La orden **pwd** (*Print Working Directory*) muestra en pantalla el directorio de trabajo.

Para seleccionar un fichero dentro de una estructura de directorios, podemos utilizar cualquiera de los tres métodos siguientes:

- especificando un camino absoluto desde el directorio raíz. Para especificarlo de esta forma se empieza el camino con **/** indicando el directorio raíz, y se continúa con todos los directorios que hay por debajo de él hasta llegar al fichero que se quiere especificar, todos ellos separados por **/**.
- especificando un camino relativo desde el directorio de inicio de sesión. En este caso, la especificación del directorio comienza por **~/**, seguida de la ruta desde el directorio de inicio de sesión hasta el fichero deseado.
- especificando un camino relativo desde el directorio de trabajo. El directorio de trabajo se puede especificar con un punto (**.**). Para subir al nivel superior en la jerarquía de directorios se utilizan los dos puntos (**..**). Si el camino comienza con un nombre de directorio (sin **/** ni **~/**) se considera que el camino comienza en el directorio de trabajo.

Ejemplo: Supongamos por ejemplo que una parte de la estructura de directorios fuera similar a la siguiente:



```
                +--------+--------+---------+------ . . .
                |        |        |         |
               prog   fisica   calculo  circuitos   . . .
                |
          +-------------+
      fuentes     ejecutables
          |
    integrales.c
```

Si el directorio de inicio de sesión es **/home/salas**, y el directorio de trabajo actual es **/home/salas/prog/ejecuta**
podemos hacer referencia al fichero **integrales.c** de las tres formas siguientes:

- mediante su camino absoluto: **/home/salas/prog/fuentes/integrales.c**
- mediante su camino relativo desde el directorio de inicio de sesión: **~/prog/fuentes/integrales.c**
- mediante su camino relativo desde el directorio actual: **../fuentes/integrales.c**

En general, esta forma de especificar un camino es válida para cualquier orden de UNIX que necesite un nombre de directorio o fichero.

2.2.2. Permisos

En los sistemas operativos de tipo UNIX los ficheros y directorios presentan tres propiedades especiales: el **propietario**, el **grupo** y los **permisos**.

El propietario de un fichero, por defecto, es el usuario que lo ha creado, y el grupo de un fichero, el grupo al que pertenece el usuario que la ha creado.

Los permisos de los ficheros y directorios se organizan en tres conjuntos de tres tipos de permisos cada uno, lo que da un total de nueve permisos diferentes. Los tipos de permisos son:

1. permiso de lectura.
2. permiso de escritura.
3. permiso de ejecución (para los ficheros) o de búsqueda (para los directorios).

Y los conjuntos:

1. el usuario que es propietario del fichero.
2. otros usuarios que pertenecen al mismo grupo que el propietario.
3. el resto de usuarios.

Para representar estos nueve permisos se utilizan nueve caracteres, agrupados de tres en tres. El primer conjunto de tres caracteres representa los permisos del propietario, el segundo los del grupo, y el tercero el del resto de los usuarios.

Dentro de cada conjunto, el primer carácter representa el permiso de lectura, que puede ser **r** o **–**, el segundo representa el permiso de escritura, que puede ser **w** o **–**, y el tercero representa el permiso de ejecución (si se trata de un fichero) o de búsqueda (si se trata de un directorio), y que puede valer **x** o **–**.

Los permisos se asignan de forma independiente para cada una de los nueve casos posibles, de forma que un fichero puede tener permiso de lectura para todos los usuarios, pero permiso de ejecución sólo para el propietario.

Ejemplo: Si los permisos de un fichero son **rwxr–xr––**, significa que el propietario tiene permiso de lectura, escritura y ejecución (**rwx**), los usuarios que pertenecen al mismo grupo al que pertenece el fichero tienen permiso de lectura y ejecución pero no de escritura (**r–x**), y el resto de los usuarios sólo tienen permiso de lectura (**r––**).

2.3. Formato general de las órdenes

El procesador de órdenes (también llamado *shell* o *intérprete de comandos*) es el programa del sistema operativo que se encarga de analizar las órdenes que se dan en una sesión y ejecutarlas. La línea donde se introducen las órdenes se denomina normalmente *línea de órdenes* o *línea de comandos*.

El formato general de las órdenes es:

nombredeorden [opciones] [argumentos]

Los corchetes (**[** y **]**) no forman parte de la orden, sino que indican que lo que va dentro es opcional. Se recuerda que hay que pulsar la tecla **<INTRO>** al final de la orden para que el intérprete de comandos comience a ejecutarla. Las opciones y los argumentos dependerán de cada orden. La orden, las opciones y los argumentos van separados unos de otros por uno o más espacios en blanco; las opciones van precedidas normalmente de un guión: **-**. Algunos ejemplos de órdenes (que se verán más adelante) son **ls**, **passwd** o **who**.

Las órdenes en linux (como en la mayoría de los sistemas operativos de tipo UNIX) se ejecutan de forma *silenciosa*. Esto quiere decir que en la mayoría de los casos, si la orden se ejecuta correctamente no recibiremos ninguna indicación del intérprete de comandos; sólo obtendremos alguna salida por el terminal si se ha producido algún error. Esta regla general no se aplica, lógicamente, a aquellas órdenes cuyo objetivo es presentar alguna información en el terminal.

2.3.1. Ayuda en línea

Dentro de una sesión se puede consultar el modo de utilización de cualquier comando mediante la orden **man**. Su formato es:

man nombredelaorden

Ejemplo: Para conocer cómo se utiliza el comando **pwd**, escribiremos:

man pwd

Para saber cómo se usa **man** se puede hacer:

man man

Para finalizar la presentación de la ayuda, basta con pulsar la tecla **q**. También le son de aplicación todas las acciones válidas para el comando **less**, que se verá más adelante.

2.3.2. Operaciones sobre el sistema de ficheros

Orden ls

Muestra el contenido de un directorio. El formato de la orden es:

ls [opciones] [nombre]

Si no se le da ningún argumento, **ls** muestra los nombres de los ficheros del directorio de trabajo. Si se le da como argumento un nombre de fichero, muestra por pantalla el nombre del fichero si existe en el directorio de trabajo. Si se le da como argumento un nombre de directorio, muestra el nombre de todos los ficheros de ese directorio.

Opciones:

-l formato largo. Muestra el modo de acceso (permisos), número de enlaces (que no se estudian en estas prácticas), propietario, tamaño, fecha de modificación y nombre de cada fichero.

-a muestra también los ficheros que empiezan por punto, que sin esta opción no se muestran.

-t los ficheros mostrados aparecen ordenados por la fecha de última modificación.

-d muestra el nombre del directorio pero no su contenido. Sirve para ver información acerca del directorio.

-r invierte el orden de salida.

-R muestra recursivamente los directorios. Muestra los ficheros del directorio y también los de todos sus subdirectorios.

-F indica con un carácter al lado del nombre del fichero su naturaleza, con **/** indica los que son directorios y con ***** los que son ejecutables.

Ejemplo: La siguiente secuencia de comandos realiza las operaciones descritas:

ls	muestra los ficheros del directorio de trabajo
ls /dev	muestra los ficheros del directorio **/dev**
ls -l /usr/lib	muestra en formato largo los ficheros del directorio **/usr/lib**
ls -a	muestra los ficheros del directorio de trabajo, incluso los que empiezan por un punto
ls -R /etc	muestra todos los ficheros y todos los subdirectorios que están en el directorio **/etc**
ls -ld /usr/lib	muestra las propiedades del directorio **/usr/lib**, no su contenido.
ls -F	muestra el contenido del directorio de trabajo y da información acerca e la naturaleza de los ficheros
ls -lt	muestra los ficheros en formato largo, ordenados por fecha de última modificación
ls -ltr	igual que el ejemplo anterior pero con el orden de salida invertido

Con la orden **ls -l** se muestra al comienzo de cada línea un conjunto de diez caracteres asociado a cada fichero. El primer carácter indica el tipo de fichero: **-** si se trata de un fichero normal, **d** si se trata de un directorio, y **l** si es un enlace. Los restantes 9 caracteres representan los permisos del fichero o directorio.

Orden chmod

Cambia los permisos de un fichero o directorio. El formato de este comando es:

chmod [opciones] permisos nombre

Los permisos puede proporcionarse de dos formas:

- Indicando los cambios sobre los permisos actuales. Para ello especificameros **u** (usuario o dueño), **g** (grupo) u **o** (otros, no es el usuario ni tampoco pertenece a su grupo), un signo **+** o **-** para dar o quitar permiso y la letra **r**, **w** o **x**:

  u+x da al usuario permiso de ejecución si es un fichero, o de búsqueda si es un directorio.

  g+r da al grupo permiso de lectura.

  Con este método, los permisos a los que no se hace referencia no se modifican.

- Un número de 3 dígitos en octal. El primer dígito indica los permisos del usuario, el segundo dígito los permisos del grupo y el tercer dígito los permisos para los demás. Cada dígito se calcula sumando 4 si se quiere dar permiso de lectura, 2 si se quiere dar permiso de escritura y 1 si se quiere dar permiso de ejecución:

  700 da todos los permisos al usuario (4+2+1) y ningún permiso al grupo ni a los demás (recomendable para el directorio de inicio de sesión).

  644 da permiso de lectura y escritura al usuario (4+2+0), y de lectura al grupo y a los demás (4+0+0).

  Con este método, todos los permisos se ven afectados.

Ejemplo: La siguiente secuencia de comandos realiza las operaciones descritas:

chmod g-x bin	quita el permiso de búsqueda al grupo para el subdirectorio **bin**
chmod g+x bin	da al grupo el permiso quitado con el ejemplo anterior

Orden cp

Copia ficheros y directorios. Presenta dos formatos:

cp nombrefichero nombreficherocopia

cp nombre1 nombre2 ... directoriodestino

En el primer caso, se crea un nuevo fichero que es una copia del original.

En el segundo, el comando permite una serie de argumentos (no sólo dos como en el primer caso), debiendo ser el último de ellos un directorio. El resultado de la ejecución del comando es que todos los ficheros especificados en los argumentos (menos el último) se copian en el directorio especificado en el último argumento.

Opciones:

-r Copia directorios. Se copia en el directorio de destino el directorio especificado como origen, incluyendo toda la estructura de directorios que esté por debajo y los ficheros que contienen. Las copias tienen los mismos nombres que los originales. Si se va a copiar un directorio en otro directorio, el de destino puede no existir, y en ese caso no se copia el directorio sino lo que contiene. Cuidado con esta opción ya que no se debe dar como directorio de destino el nombre de un subdirectorio del directorio a copiar.

Ejemplo: Suponiendo que en el directorio de trabajo existen el fichero **p1** y los directorios **prac1** y **prac2**, la siguiente secuencia de comandos realiza las operaciones descritas:

cp p1 p2	copia el fichero **p1** en **p2** (se crea el fichero **p2** que es una copia de **p1**)
cp p1 p3	copia el fichero **p1** en **p3**
cp p1 p2 p3 prac1	copia los ficheros **p1**, **p2** y **p3** en el subdirectorio **prac1** (se crean los ficheros **prac1/p1**, **prac1/p2** y **prac1/p3**)
cp -r prac1 prac2	como **prac2** ya existe, copia el directorio **prac1** y todo lo que está por debajo en **prac2**
cp -r prac1 prac3	como **prac3** no existe, copia el contenido del directorio **prac1** (pero no el propio directorio) con todo lo que está por debajo en **prac3** (se crea **prac3** ya que no existía)

Para poder aplicar el comando, es necesario que el directorio de destino disponga de permiso de escritura.

Orden mv

Cambia de nombre o ubicación un fichero. El formato del comando es:

mv nomfichero nuevonomfichero

mv nomfichero1 nomfichero2 ... directorio

No se crea ningún fichero, simplemente cambia de nombre. El nuevo nombre puede hacer que el fichero cambie de directorio. Si el último argumento es un directorio, se cambian los ficheros a ese directorio (es como si se 'movieran').

Ejemplo: Suponiendo que en el directorio de trabajo existen el fichero **p1** y el directorio **prac1**, la siguiente secuencia de comandos realiza las operaciones descritas:

mv p1 p2	El fichero **p1** se llama ahora **p2**
mv p2 prac1	El fichero **p2** está ahora en el subdirectorio **prac1** (su nuevo nombre es **prac1/p2**)

Para poder aplicar el comando, es necesario que tanto el directorio de origen como el de destino dispongan de permiso de escritura.

Orden rm

Borra ficheros (o directorios, si se utiliza la opción **-r**). El formato del comando es:

rm [opciones] nombreficheroodirectorio

Opciones:

-i pregunta si se quiere borrar o no. Esta opción está activa por defecto. Si se quiere borrar el fichero se debe responder a la pregunta con **yes** o **y**.

-r borra recursivamente los ficheros del directorio y el propio directorio.

Ejemplo: Suponiendo que en el directorio de trabajo existen el fichero **p1** y el directorio **prac1**, la siguiente secuencia de comandos realiza las operaciones descritas:

rm p1	borra el fichero **p1**
rm prac1	da error, puesto que **prac1** es un directorio
rm -r prac1	borra el directorio **prac1**, su contenido y todo lo que está por debajo

Para poder aplicar el comando, es necesario que el directorio padre del fichero o directorio disponga de permiso de escritura.

2.3.3. Operaciones sobre directorios

Orden pwd

Muestra el directorio de trabajo. Su formato es:

pwd

Orden cd

Cambia el directorio de trabajo. Su formato es:

cd [directorio]

Si se utiliza **cd** sin nombre de directorio, vuelve al directorio de inicio de sesión.

El directorio al que se desea cambiar puede especificarse con cualquiera de los tres métodos vistos anteriormente:

- Mediante su ruta absoluta desde el directorio raiz.
- Mediante su ruta relativa al directorio de inicio de sesión.
- Mediante su ruta relativa al directorio de trabajo.

Ejemplo: La siguiente secuencia de comandos realiza las operaciones descritas:

cd /usr	cambia al directorio **/usr**
pwd	muestra el directorio de trabajo
cd	cambia al directorio de inicio de sesión
cd ../..	cambia al directorio que se encuentra en el directorio que está dos niveles por encima del directorio de trabajo dentro del árbol de directorios. Suponiendo que el directorio de trabajo es **/home/salas/prog**, con el primer **..** nos referimos al directorio **salas** (padre del directorio de trabajo) y con el siguiente **..** nos referimos a **home** (padre del directorio **salas**)
cd	nos sitúa en el directorio de inicio de sesión
cd bin	cambia al directorio **bin** que está dentro del directorio de inicio de sesión. Si no existe da un mensaje de error.

Orden mkdir

Crea un nuevo directorio. El formato de la orden es:

mkdir [opciones] nuevodirectorio

Aunque a esta orden se le puedan dar opciones, no se explican en la práctica ya que el uso más común de la orden **mkdir** es sin opciones. Para información acerca de las opciones se puede ver:

man mkdir

Ejemplo: La siguiente secuencia de comandos realiza las operaciones descritas:

mkdir bin	crea el directorio **bin** dentro del directorio de trabajo
mkdir bin/dirnuevo	crea el directorio **dirnuevo** dentro del subdirectorio **bin**
mkdir ../otrobin	crea el directorio **otrobin** dentro del directorio padre
mkdir bin	intentamos crear un directorio que ya existe. El sistema nos da un error

Para poder crear un directorio, es necesario disponer de permiso de escitura en el directorio en el que se va a crear.

Orden rmdir

Borra un directorio. Su formato es:

rmdir directorio

El directorio a borrar no puede contener ningún fichero ni ningún otro directorio, es decir, tiene que estar vacío. El directorio a borrar no puede ser el directorio de trabajo.

Ejemplo: La siguiente secuencia de comandos realiza las operaciones descritas:

rmdir bin	no borra el directorio **dirnuevo** debido a que no está vacío
rmdir bin/dirnuevo	borra **dirnuevo** dentro de **bin**
rmdir ../otrobin	borra el subdirectorio **otrobin** del directorio padre

Para poder borrar un directorio, es necesario disponer de permiso de escitura en el directorio en el que se va a borrar.

2.3.4. Comandos para visualización de ficheros

Orden cat

Muestra el contenido de uno o varios ficheros:

cat [opciones] fichero1 [fichero2 ..]

Opciones:

-n precede cada línea con un número de línea

Ejemplo: La siguiente secuencia de comandos realiza las operaciones descritas:

cat p1	presenta por pantalla el contenido del fichero **p1**
cat p1 prac/p4	presenta por pantalla el contenido de los ficheros **p1** y **prac/p4**, sin pausa entre ellos
cat -n p1	presenta por pantalla el contenido del fichero **p1**, anteponiendo en cada línea el número de secuencia de la misma

Orden less

Es un filtro para ver el contenido de un fichero fácilmente. La diferencia con respecto al anterior consiste en que sólo presenta lo que cabe en la ventana, y es necesaria una acción por parte del usuario para continuar. Su formato es:

less [opciones] fichero

Una vez ejecutado el comando, es posible realizar las siguientes acciones:

<espacio>	avanza una página
<b>	retrocede una página
<INTRO>	avanza una línea
<q>	termina
<h>	ayuda

Ejemplo: La siguiente secuencia de comandos realiza las operaciones descritas:

less p1	presenta por pantalla el contenido del fichero **p1**, una página cada vez
man cp	la salida del comando **man** utiliza el comando **less**, por lo que son de aplicación las acciones de dicho comando

Orden tail

Muestra parte del contenido de un fichero. El fomato del comando es:

tail [opciones] fichero

Opciones:

+n muestra a partir de la línea **n**.

−n muestra las últimas **n** líneas.

Si no se pone opción muestra las últimas 10 líneas.

Ejemplo: La siguiente secuencia de comandos realiza las operaciones descritas:

tail p1	muestra las 10 últimas líneas del fichero **p1**
tail +2 p1	muestra desde la segunda línea del fichero **p1**
tail -3 p1	muestra las tres últimas líneas del fichero **p1**

2.3.5. Comandos informativos

Orden wc

Cuenta líneas, palabras y caracteres de los ficheros. El primer número indica el número de líneas, el segundo el de palabras y el tercero el de caracteres:

wc [opciones] fichero

Opciones:

−c muestra sólo el número de caracteres

−l (letra ele) muestra sólo el número de líneas

−w muestra sólo el número de palabras

Ejemplo: La siguiente secuencia de comandos realiza las operaciones descritas:

wc p1	Muestra el número de líneas, palabras y caracteres del fichero **p1**
wc -l p1	Muestra sólo el número de líneas del fichero **p1**

Orden du

Muestra el espacio en disco (normalmente en KB) ocupado por los ficheros y directorios dados como parámetros. Se considera el espacio ocupado por un directorio al espacio ocupado por todos los ficheros y, recursivamente, directorios dentro de este. El formato del comando es:

du [opciones] [nombre]

Si no se especifica **nombre** se supone **.** (el directorio de trabajo).

Opciones:

−a muestra el tamaño de cada fichero

−s sólo muestra el resumen

Si no se especifica ninguna opción, sólo hace un resumen por directorios, sin especificar lo que ocupa cada fichero.

Ejemplo: La siguiente secuencia de comandos realiza las operaciones descritas:

du	muestra información del directorio de trabajo y de todos los subdirectorios
du -s prac	muestra información del directorio **prac**
du -a prac	muestra información del directorio **prac** y de sus subdirectorios (especificando lo que ocupa cada fichero)

2.3.6. Otros comandos

Orden sort

Muestra las líneas de un fichero ordenadas alfabéticamente:

sort [opciones] fichero

Opciones:

-r muestra en orden inverso.

Ejemplo: La siguiente secuencia de comandos realiza las operaciones descritas:

sort p1	muestra por pantalla el fichero **p1** ordenado alfabéticamente

Orden echo

Escribe sus argumentos en la salida estándar. Es útil para comprobar el uso correcto de los metacaracteres del *shell* (se verá en el siguiente apartado de esta práctica).

Ejemplo: El siguiente comando realiza la operación descrita:

echo hola a todos	escribe en pantalla **hola a todos**

Orden diff

Busca diferencias entre dos ficheros. Su formato es:

diff ficheroorigen destino

Si **destino** es un fichero, compara los dos ficheros indicados. Si es un directorio, busca en **destino** el fichero de nombre **ficheroorigen** y los compara.

La salida de la orden muestra las diferencias encontradas entre ambos ficheros. Cuando el comando encuentra diferencias en líneas consecutivas del fichero, las agrupa.

Al presentar las diferencias, las referencias al primer fichero aparecen precedidas por **<** (fichero de la izquierda en la línea de comandos), y las correspondientes al segundo por **>** (fichero de la derecha).

A continuación se presentan los tres casos que pueden aparecer al ejecutar este comando:

- Si la línea (o líneas) sólo aparece en el primero de los ficheros, sólo aparecerán las líneas precedidas del símbolo **<**.
- Si la línea (o líneas) sólo aparece en el segundo fichero, sólo aparecerán líneas precedidas del símbolo **>**.
- Si las líneas aparecen en los dos ficheros, pero son diferentes, aparecerán primero las líneas del primer fichero, precedidas del símbolo **<**, a continuación una línea formada por tres guiones, y por último las líneas del segundo fichero precedidas del símbolo **>**.

Los números y letras que anteceden a cada cambio especifican qué comandos habría que ejecutar en el editor de línea **sed** para obtener el segundo fichero a partir del primero. Puesto que dicho editor no se utiliza en la asignatura, pueden ignorarse, sirviendo únicamente para separar cada grupo de cambios.

Ejemplo: La siguiente secuencia de comandos muestra el comportamiento que se indica:

```
salas@318CDCr12:~$ cat dato1
Primera linea del primer fichero
Segunda linea del primer fichero
Esta linea esta en los dos
Linea de los dos que cambia
Otro linea modificada
Esta linea esta en los dos
salas@318CDCr12:~$ cat dato2
Esta linea esta en los dos
Linea de los 2 que cambia
otra linea modificada
Esta linea esta en los dos
Ultima linea del segundo fichero
salas@318CDCr12:~$ diff dato1 dato2
1,2d0
< Primera linea del primer fichero
< Segunda linea del primer fichero
4,5c2,3
< Linea de los dos que cambia
< Otro linea modificada
---
> Linea de los 2 que cambia
> otra linea modificada
6a5
> Ultima linea del segundo fichero
salas@318CDCr12:~$
```

2.4. Funciones especiales del intérprete de comandos

El intérprete de comandos, o *shell*, es el procesador de órdenes. Es un programa que lee la orden que se le da por teclado, la analiza y llama al programa del sistema operativo que realmente la ejecuta. El *shell* por defecto utilizado en los ordenadores del centro de cálculo es **bash** (*Bourne-Again Shell*). Para más información sobre sus características (además de las vistas en esta práctica) ejecutar:

man bash

2.4.1. Caracteres ordinarios y caracteres de control

Los caracteres que se escriben en el teclado son enviados al sistema, el cual normalmente los devuelve al terminal para que aparezcan en pantalla. Este proceso se llama eco y funciona usualmente con los caracteres ordinarios excepto cuando se escribe la palabra clave.

Existen algunos caracteres con un significado especial: son los caracteres de control. El más importante es el asociado a la tecla **<INTRO>** que señala el fin de una línea de entrada, y el sistema le hace eco moviendo el cursor al principio de la siguiente línea de la pantalla. La tecla **<INTRO>** debe ser pulsada para que el sistema interprete los caracteres que se acaban de teclear.

Otros caracteres de control no tienen una tecla propia sino que deben ser escritos manteniendo pulsada la tecla **<CONTROL>** y pulsando simultáneamente otra tecla, por lo general una letra. Por ejemplo, pulsar la tecla **<CONTROL>** y a la vez la tecla **<S>** se indica con **^S**. Algunos de estos caracteres son los siguientes:

^S Detiene el scroll de la pantalla (desplazamiento hacia arriba de la información cuando no cabe en una pantalla).

^Q Reanuda el scroll de la pantalla.

^D Envía una marca de fin de fichero desde el teclado. Si se da cuando el sistema espera una orden es como si se diera la orden de fin de sesión.

2.4.2. Metacaracteres

Hay caracteres que para el *shell* tienen un significado especial, los metacaracteres. Con los metacaracteres se pueden construir expresiones que el *shell* sustituye por nombres de ficheros que ya existan y que estén de acuerdo con la expresión. Los metacaracteres que se van a estudiar son:

`*` sustituye cualquier carácter o grupo de caracteres (excepto un punto inicial).

`?` sustituye un único carácter cualquiera.

`[]` sustituye cualquier carácter situado entre los corchetes. También se pueden utilizar rangos, separando el carácter de inicio y el carácter de final del rango con un guión.

Para que el *shell* no haga estas sustituciones con los metacaracteres se pueden poner entre comillas simples (se encuentran sobre la tecla a la derecha de la tecla del cero) o anteponiendo el carácter **** si sólo es un carácter.

Ejemplo: La siguiente secuencia de comandos muestra el comportamiento que se indica:

`echo d*`	muestra en pantalla el nombre de todos los ficheros del directorio actual que empiezan por **`d`**
`echo dat0[12345].*`	muestra en pantalla el nombre de todos los ficheros del directorio actual que se llaman **`dat0`** seguido de **`1`**, **`2`**, **`3`**, **`4`** ó **`5`**, seguido de un punto y con cualquier extensión
`echo *`	muestra el nombre de todos los ficheros del directorio actual excepto los que empiezan por punto
`echo \*`	muestra un **`*`** en pantalla
`echo dat0?.?`	muestra el nombre de todos los ficheros del directorio actual que se llaman **`dat0`** seguido de un carácter cualquiera, un punto y otro carácter cualquiera
`echo 'dat0?.?'`	muestra en pantalla **`dat0?.?`**
`echo dat[01][1-37-9].*`	muestra el nombre de todos los ficheros del directorio actual que se llaman **`dat`** seguido de un **`0`** ó **`1`**, seguido de **`1`**, **`2`**, **`3`**, **`7`**, **`8`** ó **`9`**, seguido de punto y con cualquier extensión

Aunque en los ejemplos sólo se utiliza la orden **`echo`**, los metacaracteres del *shell* se utilizan con cualquier orden. Puede comprobarlo utilizando la orden **`ls`** en lugar de **`echo`**.

2.4.3. Redireccionamiento de la entrada/salida

Hay muchas órdenes que toman los datos de entrada de la entrada estándar (teclado) y muestran sus datos de salida por la salida estándar (pantalla). También pueden dar mensajes de error o diagnóstico, que van a la salida de errores (normalmente también la pantalla).

El *shell* puede redirigir la E/S estándar a ficheros. De esta forma se puede conseguir que una orden tome la entrada de un fichero en vez de teclado y que la salida se lleve a un fichero en vez de a la pantalla.

Este redireccionamiento lo hace el *shell* sin que la orden tenga que detectar lo que está pasando. Los comandos de redirección pueden variar según el *shell* que estemos usando. En el caso de **`bash`**, tendríamos:

`orden > nomf`	Se crea un fichero de nombre **`nomf`**, y a él va la salida estándar. Si **`nomf`** ya existía se pierde.
`orden >> nomf`	Se añade la salida estándar al final del fichero **`nomf`**. Si no existe, se crea.
`orden 2> nomf`	Se crea un fichero de nombre **`nomf`**, y a él va la salida de errores. Si **`nomf`** ya existía se pierde.
`orden 2>> nomf`	Se añade la salida de errores al final del fichero **`nomf`**. Si no existe, se crea.
`orden < nomf`	La entrada estándar es leída del fichero **`nomf`**

Si se desea redireccionar las salidas estándar y de errores al mismo fichero, deberá utilizarse el doble símbolo para que una salida no oculte a la otra:

orden >>nomf 2>> nomf

Ejemplo: La siguiente secuencia de comandos muestra el comportamiento que se indica:

ls > temp	guarda en el fichero **temp** el listado de todos los ficheros del directorio actual
wc < temp	cuenta el número de líneas, de palabras y caracteres del fichero **temp**
sort < temp > ordenados	guarda en el fichero **ordenados** el contenido de **temp** una vez ordenado
du .. 2> errores	guarda en el fichero **errores** los mensajes de error generados por la ejecución del comando

2.5. Procesos

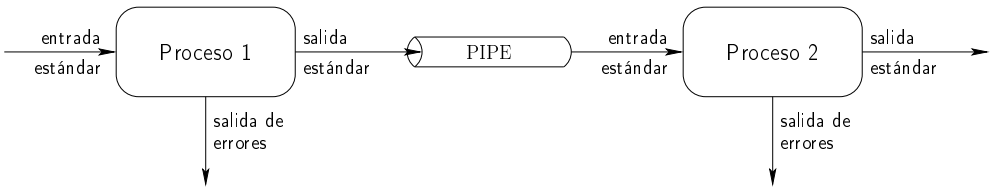
Un proceso (o tarea) es la ejecución de un programa. Un programa se puede ejecutar simultáneamente por uno o varios usuarios, existiendo así varios procesos y un solo programa.

Cada proceso tiene una serie de atributos que el sistema operativo mantiene en una tabla. El más importante es el **PID** (*Process IDentification*) que es un número entero que identifica de forma única al proceso.

2.5.1. Intercomunicación entre procesos

Una forma de comunicación entre procesos es mediante un PIPE o tubería, que conecta la salida de un proceso con la entrada de otro, es decir, hace que la salida estándar de un proceso sea la entrada estándar del que le sigue. Para representar una tubería se utiliza el símbolo **|**.

La siguiente figura recoge este concepto:



Los programas se ejecutan a la vez, y es el sistema operativo el que se encarga de pasar la información de uno a otro. Se puede usar para cualquier programa que use la E/S estándar, por ejemplo las órdenes vistas anteriormente.

Si se quiere conectar la salida de errores y no la estándar, se usa la secuencia **|&**.

Ejemplo: La siguiente secuencia de comandos muestra el comportamiento que se indica:

ls | wc -w	la salida de la orden **ls** se utiliza como entrada para la orden **wc**. Como consecuencia, el resultado de la operación es el número de ficheros y directorios del directorio actual
du .. |& wc -l	cuenta el número de líneas de error que genera la orden **du**; el resultado del comando **du** sigue apareciendo por el terminal.

2.5.2. Control de trabajos

Un trabajo es el resultado de una petición al intérprete de comandos. El trabajo puede estar formado por un único proceso, o por un conjunto de ellos. Por ejemplo, el trabajo siguiente:

ls -lR sólo involucra un proceso, mientras que este otro:

ls -lR /usr/lib | wc -l (que da una idea aproximada del número de ficheros del directorio **/usr/lib** y todos sus subdirectorios) se ha creado un trabajo con dos procesos.

Un trabajo se puede encontrar en uno de tres modos de ejecución:

- en *foreground* (o modo interactivo): es el estado normal de un proceso. El terminal está controlado por el proceso. No se puede hacer otra cosa hasta que no termine de ejecutarse.
- en *background* (o segundo plano): el terminal queda libre para seguir trabajando en otras cosas (es útil si se está ejecutando un proceso que tarda bastante). El trabajo sigue ejecutándose.
- parado: el trabajo deja de ejecutarse, a la espera de que se le especifique un nuevo cambio del modo de ejecución, pasándolo a *background* o a *foreground*, o se finalice.

Por defecto, todos los trabajos se ejecutan en modo interactivo. Para ejecutar una orden en segundo plano se finaliza la orden con el símbolo **&**. Existen comandos, que se verán más adelante, para cambiar el estado de ejecución de los trabajos.

Un trabajo queda totalmente determinado por su identificador, que es un número entero, asignado por el intérprete de comandos cuando se lanzó la orden. No debe confundirse el identificador de un trabajo con los PIDs de los diferentes procesos que componen dicho trabajo. Para hacer referencia a un trabajo se antepone el símbolo **%** al identificador de dicho trabajo.

Ejemplo: Para referirnos al trabajo de identificador 3, escribiremos:

```
%3
```

Si un proceso que está en segundo plano intenta leer de la entrada estándar, queda parado. Sin embargo sí puede escribir en la salida estándar, pudiendo interferir con la salida del proceso que está ejecutándose en primer plano.

Cuando un trabajo en segundo plano termina, el sistema envía un mensaje al terminal indicándolo ('**Done**'). En el mensaje aparece entre corchetes el identificador del trabajo terminado y la orden que lo puso en marcha.

Ejemplo: La siguiente secuencia de comandos muestra el comportamiento que se indica:

ls -lR /usr > b0.dat &	se crea un trabajo en segundo plano con un único proceso. Se crea el fichero **b0.dat** con el listado del directorio **/usr** y todos sus subdirectorios
ls -lR /usr | wc -l &	se crea un trabajo en segundo plano con dos procesos. Cuenta el número de ficheros y directorios que hay dentro del directorio **/usr**

2.5.3. Órdenes de información sobre procesos y trabajos

Orden ps

Muestra información de los procesos:

ps [opciones]

Opciones:

- **a** muestra los procesos de todos los usuarios
- **x** muestra todos los procesos aunque no estén asociados a un terminal
- **u** muestra más información más detallada

La información que ofrece este comando es la siguiente:

PID	número de identificación del proceso
TT	terminal lógico asociado
S	estado del proceso. Puede ser **0** (en ejecución), **R** (a la espera del procesador), **S** (inactivo a la espera de que se produzca un evento) o **T** (parado).
TIME	tiempo de CPU consumido por el proceso
COMMAND	nombre de la orden.

Orden `jobs`

Muestra los trabajos que están en *background* o parados:

`jobs`

La diferencia con la anterior es que esta orden especifica trabajos, mientras que la anterior especifica procesos; y un trabajo involucra al menos un proceso, pero puede involucrar cualquier número de ellos.

Ejemplo: La orden

`ls -lR /usr/lib | wc -l & ps ; jobs` que da una idea aproximada del número de ficheros del directorio y todos sus subdirectorios, crea un trabajo con dos procesos. Si utilizamos la orden **`jobs`** obtendremos una única entrada (puesto que es un único trabajo) mientras que si utilizamos la orden **`ps`** obtendremos dos entradas, una por cada proceso involucrado en el trabajo.

2.5.4. Órdenes de cambio de estado de un trabajo

Orden `^C`

Destruye el trabajo que se encontraba en modo interactivo. El comando se ejecuta pulsando simultáneamente las teclas **`<Control>`** y **`<C>`**.

Orden `^Z`

Pasa un trabajo del modo interactivo al estado parado. El comando se ejecuta pulsando simultáneamente las teclas **`<Control>`** y **`<Z>`**.

Orden `bg`

Pasa un trabajo de parado a segundo plano. Su formato es:

`bg [identificacióndetrabajo]`

Si no se indica ninguna identificación de trabajo, por defecto se actúa sobre el último trabajo parado.

Orden `fg`

Pasa un trabajo de parado o en segundo plano a modo interactivo:

`fg identificacióndetrabajo`

Orden `kill`

Envía la señal de terminación a un proceso o trabajo del usuario (lo destruye). El formato de la orden es:

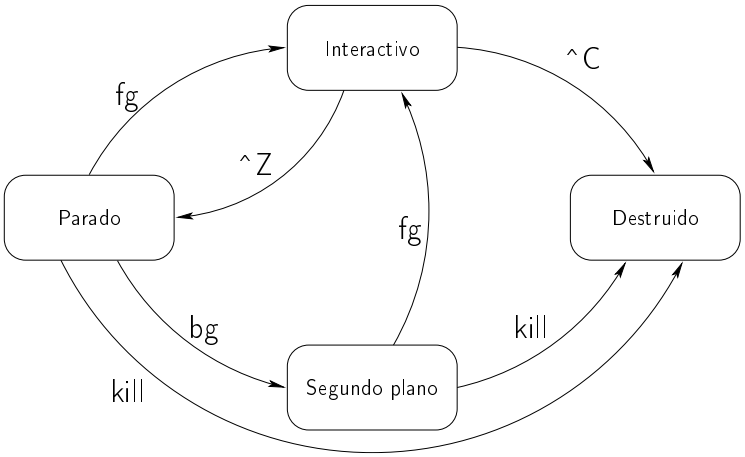
`kill [opciones] identificacióndeprocesootrabajo`

Opciones:

-9 Para procesos que ignoran la señal de terminación de trabajo.

Resumen

La siguiente figura resume los diferentes estados en los que se puede encontrar un trabajo, y las órdenes necesarias para pasarlo de un estado a otro.


Interactivo
^C
fg
^Z
Parado
fg
Destruido
bg
kill
kill
Segundo plano

Anexo 3

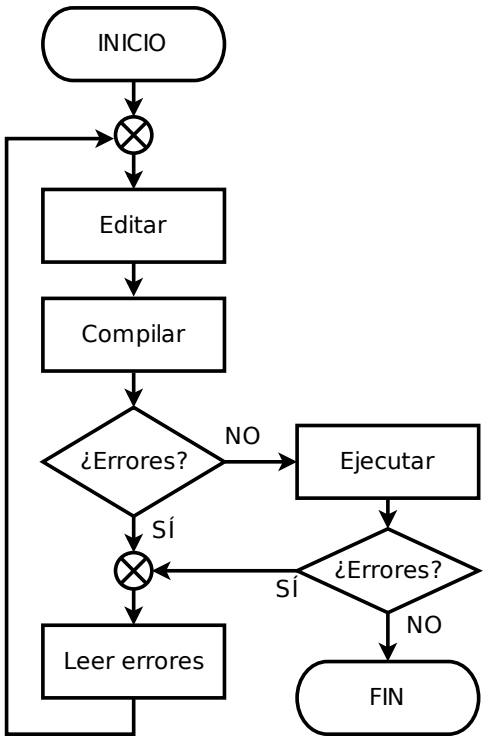
El compilador `gcc`

Índice

3.1. Introducción **3-1**
3.1.1. Etapas en la generación de un ejecutable 3-2
3.2. El comando `gcc` **3-4**
3.2.1. Uso de la opciones **`-W`** y **`-Wall`** 3-4
3.2.2. Uso de la opción -c 3-5

3.1. Introducción

Las fases del desarrollo de un programa responden generalmente al siguiente esquema:



Como se puede apreciar en el diagrama de flujo anterior, debemos comenzar utilizando un editor para escribir nuestro programa fuente. Los ficheros que contienen texto fuente en C deberán tener la extensión **.c** (o **.h** si se trata de un fichero de preprocesado).

Una vez escrito, debemos compilarlo y eliminar los errores de codificación que indique el propio compilador, que incluso indica la línea de código donde se ha producido cada error. Para ello debemos leer atentamente los mensajes de error y, posiblemente, buscar ayuda en el manual (**man**).

Cuando el programa no tenga errores de compilación debemos ejecutarlo y comprobar que realmente hace lo que queríamos. Si no fuera así, deberíamos volver a la fase de edición para corregir los errores detectados.

Sólo cuando el programa funciones como debe podremos decir que hemos terminado el programa con éxito.

Todo programa fuente debe ser compilado antes de poder ser ejecutado, por ello, el objetivo de esta sección es el aprendizaje de las herramientas de compilación, que permiten obtener ejecutables a partir del código fuente.

3.1.1. Etapas en la generación de un ejecutable

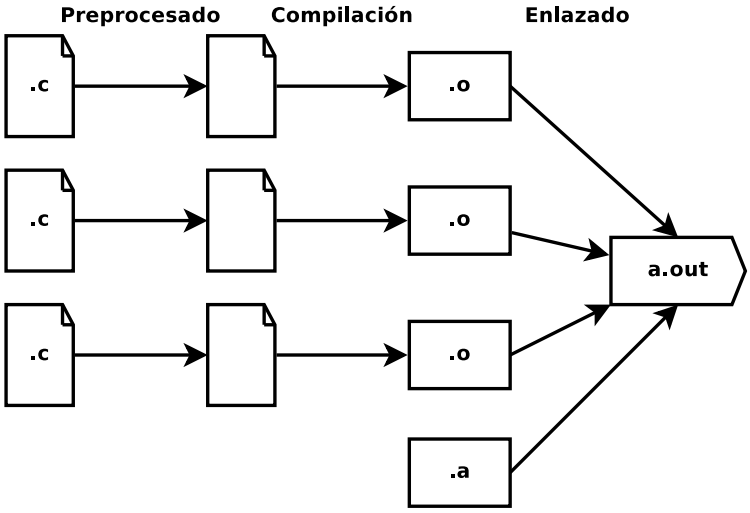
Una vez que se han editado los ficheros fuente, la generación de un ejecutable se realiza en tres estapas sucesivas:

- Preprocesado: en esta etapa se ejecutan las órdenes del preprocesador (**#include**, **#define**, **#ifndef**, etc.). Al realizar esta operación sobre un fichero fuente se obtiene como resultado una unidad de compilación. La unidad de compilación no se almacena en un fichero intermedio, pero se puede ver en pantalla con la opción **-E** de **gcc**. Esta etapa la realiza de forma automática el compilador antes de realizar la compilación.

- Compilación: obtiene el código objeto a partir del código fuente. El código fuente es la unidad de compilación resultante del preprocesado. El código objeto (o código intermedio) se almacena en un fichero con el mismo nombre que el fuente pero terminado en **.o**. Este fichero contiene código máquina pero que no se puede ejecutar porque contiene referencias no resueltas (por ejemplo llamadas a funciones definidas en otros ficheros). Además, durante la compilación se analiza la sintaxis del fichero fuente y si se detectan errores se informa al usuario y no se obtiene código intermedio. Si el compilador detecta sentencias sintácticamente correctas, pero que pueden ser fruto de una equivocación del programador muestra el correspondiente mensaje de aviso (o *warning*) pero sí genera el código intermedio. Se debe corregir el programa para que no aparezca ningún aviso.

- Enlazado o linkado: una vez compiladas con éxito todas las unidades de compilación obtenidas en el preprocesado, todos los ficheros con código intermedio (los que terminan en **.o**) deben ser enlazados conjuntamente para obtener un único programa ejecutable (por defecto **a.out**). Esta operación se conoce como enlazado o linkado del programa y la realiza el enlazador o linker. En el enlazado también se resuelven las referencias a funciones de biblioteca (que normalmente se hace de forma automática).

La siguiente figura recoge el proceso de generación de un ejecutable a partir de varios ficheros de código fuente:



En las prácticas de programación, todas estas operaciones se realizan con el programa **gcc**. Por defecto, si **gcc** recibe como argumentos ficheros **.c**, intenta realizar estas tres operaciones, y si obtiene un programa ejecutable elimina los ficheros **.o** intermedios generados. Si **gcc** recibe como argumentos ficheros **.o** los intenta enlazar directamente.

Ejemplo: El siguiente código pinta un abeto en la salida estándar:

```
#include <stdio.h>

int main()
{
  int n = 7;
  int i = 1;
  int j;

  while (i <= n)
    {
      for ( j = i; j < n ; j++)
        printf(" ");
      for ( j = 1; j <= 2*i-1; j++)
        printf("*");
      i++;
      printf ("\n");
    }

  for ( i = 1; i < n; i++)
    printf(" ");
  printf("#\n");

  return 0;
}
```

Si compilamos el programa **abeto.c** con la orden **gcc**:

```
salas@318CDCr12:~$ gcc abeto.c
salas@318CDCr12:~$
```

Aparecerá una versión ejecutable del programa en un fichero de nombre **a.out**. Si lo ejecutamos:

```
salas@318CDCr12:~$ a.out
      *
     ***
    *****
   *******
  *********
 ***********
*************
      #
salas@318CDCr12:~$
```

3.2. El comando gcc

El formato de la orden **gcc** es el siguiente:

gcc [opciones] ficheros

Las opciones del compilador son innumerables, y permiten controlar de forma exhaustiva cómo queremos que se realice la compilación. Sin embargo, durante las prácticas utilizaremos un número muy reducido de ellas, que se exponen a continuación:

-E Sólo realiza el preprocesado del fichero, enviando el resultado a la salida estándar.

-Wall Avisa de todos los tipos de errores posibles (sintácticos y de construcción).

-W Muestra mensajes de aviso adicionales.

-c realiza la compilación y por tanto crea el fichero objeto pero no llama al enlazador y por tanto no crea el fichero ejecutable.

-g Produce información necesaria para el depurador (se verá en una práctica posterior).

-o *<nombrefichero>* El programa ejecutable se guarda en el fichero ***<nombrefichero>*** en vez de en **a.out**. Si previamente ya existía el fichero **a.out**, éste no se modifica.

-l*<nombre>* Busca una biblioteca llamada **lib*<nombre>*.a**. Por ejemplo, **-lm** busca la biblioteca llamada **libm.a**. En esta biblioteca se encuentran las funciones matemáticas (seno, coseno ...). Es importante el orden en que se coloca esta opción; se pone normalmente al final, después de los ficheros a compilar.

3.2.1. Uso de la opciones -W y -Wall

En C se distinguen dos tipos de incidencias a la hora de compilar un programa en C:

- Los errores: son errores sintácticos, que hacen que el compilador no sepa qué hacer, y como consecuencia se detiene la compilación y no se genera ningún fichero objeto ni ningún ejecutable.
- Los avisos: son alteraciones que hacen que el programa no se corresponda de forma estricta con lo especificado por el estándar C, pero aun así el compilador puede tomar una decisión y generar el objeto o ejecutable. Realmente lo que hace el compilador con los avisos es indicarnos precisamente la opción que ha tomado, por si no fuera esa la intención del programador.

Ejemplo: El siguiente código comprueba si el valor introducido por teclado vale 1:

```
#include <stdio.h>

int main()
{
  int numero;

  printf("Introduzca un numero: ");
  scanf("%d", &numero);
  if (numero = 1)
    printf("Vale uno\n");
  else
    printf("No vale uno\n");

  return 0;
}
```

Si se genera el ejecutable sin utilizar las opciones **-W** ni **-Wall**, y se realizan algunas ejecuciones, se obtienen los siguientes resultados:

```
salas@318CDCr12:~$ gcc avisos.c
salas@318CDCr12:~$ a.out
Introduzca un numero: 1
Vale uno
salas@318CDCr12:~$ a.out
Introduzca un numero: 8
Vale uno
salas@318CDCr12:~$
```

Como puede observarse, el funcionamiento del programa es incorrecto. Existe un error en la codificación que no ha sido detectado. Sin embargo, si utilizamos las opciones **-W** y **-Wall**, el propio compilador nos habría puesto sobre aviso:

```
salas@318CDCr12:~$ gcc -W -Wall avisos.c
avisos.c: In function 'main':
avisos.c:9: warning: suggest parentheses around assignment used as truth value
salas@318CDCr12:~$
```

El ejecutable se ha generado igualmente, pero el compilador nos avisa de que en la línea 9 hay algo que puede no estar bien. Efectivamente, si observamos dicha línea podemos comprobar que estamos utilizando el operador **=** (de asignación) en lugar del operador **==** (de comparación), que es el que queríamos utilizar.

Sólo los programadores expertos pueden interpretar adecuadamente los informes de aviso del compilador, y estar completamente seguros de que el programa funciona como se desea a pesar de producirse avisos. Por ello, es imprescindible que los programas de los alumnos carezcan tanto de errores (por supuesto), como de avisos durante la fase de compilación. Y para asegurarse de ello, siempre habrá que incluir las opciones **-W -Wall** en todas las compilaciones que se realicen durante el curso (y se aconseja que se continúe haciendo lo mismo para futuros programas).

3.2.2. Uso de la opción -c

Como se ha indicado anteriormente, si sólo queremos alcanzar la segunda fase del proceso de generación de los ejecutables, la compilación, debemos indicarlo con la opción **-c**. Cuando se utiliza esta opción, no se genera el fichero **a.out**, y en su lugar se generan los ficheros objeto (**.o**) correspondientes a los fuentes especificados. Si queremos obtener además el ejecutable, deberemos a continuación invocar el comando **gcc** para que realice el enlazado, pasándole como parámetros los ficheros objeto (**.o**) y no los ficheros fuente (**.c**). Puede comprobarlo ejecutando la siguiente serie de comandos:

```
salas@318CDCr12:~$ ls a.out *.o
ls: No match.
```

```
salas@318CDCr12:~$ gcc -W -Wall -c abeto.c
salas@318CDCr12:~$ ls a.out *.o
a.out not found
abeto.o
salas@318CDCr12:~$ gcc abeto.o
salas@318CDCr12:~$ ls a.out *.o
a.out abeto.o
```

Uso de la opción -o

Si se desea que el nombre del fichero ejecutable tenga un nombre diferente, deberá indicárselo al compilador con la opción **-o**, seguida por el nombre que se desea dar al ejecutable.

Ejemplo: El siguiente ejemplo genera el ejecutable correspondiente al dibujo del abeto utilizando **abeto** como nombre del ejecutable:

```
salas@318CDCr12:~$ gcc -Wall -W -o abeto abeto.o
salas@318CDCr12:~$ abeto
      *
     ***
    *****
   *******
  *********
 ***********
*************
      #
salas@318CDCr12:~$
```

Puesto que si no se utiliza esta opción cada vez que generemos un nuevo ejecutable borraremos el ejecutable anterior que pudiera existir en el directorio en el que estamos trabajando, se aconseja utilizarla siempre, y ponerle un nombre al ejecutable que indique qué es lo que hace.

Anexo 4

El procesador de órdenes **make**

Índice

4.1. Introducción **4-1**
4.2. Descripción **4-1**
4.3. Sintaxis **4-2**
4.4. El Comportamiento de make **4-3**
4.4.1. Objetivos sin Dependencias 4-5
4.4.2. Forzar la reconstrucción completa 4-5
4.5. Macros en makefiles **4-5**

4.1. Introducción

Para conseguir modularidad en la programación es conveniente (y necesario) organizar un programa en distintos ficheros. Si un programa es complejo, podría estar estructurado en un gran número de ficheros. La gestión de múltiples ficheros es una tarea que puede ser abrumadora sin herramientas que la faciliten.

El procesador de órdenes permite automatizar procesos repetitivos de forma automática, como por ejemplo generar el ejecutable de un programa a partir de sus diferentes componentes.

El procesador de órdenes que se analizará en este capítulo es el **make**.

La herramienta **make** es una utilidad orientada al mantenimiento de los programas, que simplifica y automatiza el proceso de compilación y enlace de un programa aplicando reglas predefinidas.

La herramienta **make** se diseñó originalmente para mantener programas, y hablaremos de ella en este contexto. Esta herramienta especifica y controla el proceso mediante el cual se genera un ejecutable que puede constar de múltiples ficheros. La ventaja de utilizar **make** es que ahorra tiempo. Debido a que puede interpretar las dependencias de un programa, puede compilar de manera selectiva sólo aquellos ficheros que hayan sido modificados. La alternativa sería recompilar todo, que da lugar a una pérdida de tiempo considerable en los programas extensos.

4.2. Descripción

La herramienta **make** lee de un fichero de descripción de dependencias la especificación de cómo los componentes de un programa están relacionados entre sí y cómo procesarlos para crear una versión actualizada del programa. Revisa las fechas en las que los distintos componentes fueron modificados por última vez,

encuentra la cantidad mínima de recompilación que debe hacerse para tener una nueva versión, y entonces compila y enlaza.

La herramienta **make** construye un programa (o varios) de acuerdo con un conjunto de criterios contenidos en el fichero de descripción de dependencias, al que, por defecto, llamaremos **makefile**. Un fichero **makefile** contiene tres tipos de información que **make** examina. Básicamente se puede dividir en el *qué*, el *por qué* y el *cómo*.

- El *qué*, es(son) el(los) nombre(s) de lo que se pretende construir. En la terminología **make**, el *qué* se llama **objetivo**.
- El *por qué* es la parte más curiosa del uso de **make**. Se informa de la razón por la que un objetivo determinado debería construirse a partir de un conjunto específico de componentes. En general, se reconstruye un objetivo a partir de sus componentes cuando el objetivo tiene una fecha anterior con respecto a los componentes de los que depende. Por ello a la información *por qué* se llama **prerrequisitos** o **dependencias**.
- El *cómo* son las órdenes del sistema que **make** debería invocar para construir los objetivos (normalmente a partir de los prerrequisitos). En terminología **make**, el *cómo* se llama **regla**.

El formato general de un fichero de descripción de dependencias es el siguiente:

```
# Linea de comentario
objetivo_1: dependencias_1
      regla_1_1
      regla_1_2
      ...
      regla_1_n

# Otra linea de comentario
objetivo_2: dependencias_2
      regla_2_1
      regla_2_2
      ...
      regla_2_m
```

Para especificar un objetivo el formato que se sigue es escribir el objetivo al principio de una línea, y separarlo de los prerrequisitos por el carácter **:**. Las reglas correspondientes a un objetivo deben estar en las líneas siguientes y deben comenzar cada una de ellas por el carácter tabulador. Las reglas a aplicar finalizan cuando aparece una línea en blanco o un nuevo objetivo.

En el fichero de descripción de dependencias pueden aparecer líneas de comentario, que comienzan por el carácter **#**. La herramienta **make** ignora dichas líneas.

4.3. Sintaxis

La sintaxis de la orden **make** es la siguiente:

make [-f dependencias] [objetivo]

Las opciones tienen el siguiente significado:

-f dependencias Define el nombre del fichero en el que están definidas las normas para construir el ejecutable. Si se invoca a **make** sin la opción **-f**, utilizará como fichero de descripción de dependencias un fichero llamado **makefile** o **Makefile** (en este orden). Es práctica común llamar al fichero **Makefile**, con una **M** mayúscula, porque así aparece al principio de la lista de ficheros en un listado del directorio, puesto que los nombres de los ficheros fuente suelen comenzar con minúsculas, que alfabéticamente van detrás.

objetivo Es el nombre del objetivo a construir, y que será necesario buscar en el fichero de descripción de dependencias. Si no se indica ningún objetivo, **make** interpreta que se quiere obtener el primer objetivo que aparezca en el fichero.

Ejemplo: Si tenemos en el directorio de trabajo los ficheros **Makefile**:

```
# Esto es un fichero makefile de ejemplo
# Makefile de arbol

arbol: arbol.o pinta.o
    gcc -o arbol arbol.o pinta.o

arbol.o: arbol.c pinta.h
    gcc -W -Wall -c arbol.c

pinta.o: pinta.c pinta.h
    gcc -W -Wall -c pinta.c

borra:
    rm *.o
```

y **trabajo**:

```
# Esto es un fichero makefile de ejemplo
# Makefile de arbol

programa:    principal.o libreria.o
             gcc -o programa principal.o libreria.o

principal.o: principal.c libreria.h
             gcc -W -Wall -c principal.c

libreria.o:  libreria.c libreria.h
             gcc -W -Wall -c libreria.c
```

las siguientes órdenes tendrán el siguiente resultado:

make	genera el objetivo **arbol** (que es el que primero aparece) definido en el fichero **Makefile**
make -f trabajo	genera el objetivo **programa** (que es el que primero aparece) definido en el fichero **trabajo**
make borra	genera el objetivo **borra** definido en el fichero **Makefile**
make -f trabajo programa	genera el objetivo **programa** definido en el fichero **trabajo**
make programa	Produciría un error, puesto que el objetivo **programa** no está definido en el fichero **Makefile**

4.4. El Comportamiento de make

El comportamiento de **make** es el siguiente.

Si hay que construir **objetivo**, que siempre será un fichero, primero busca dicho fichero en el directorio de trabajo:

- Si no existe, directamente se aplican las reglas asociadas a dicho objetivo, secuencialmente, desde la primera a la última.
- Si el fichero **objetivo** existe, entonces compara la fecha de dicho fichero con la de todos los ficheros que componen sus dependencias:
  - Si **ALGUNA** de las dependencias es más nueva que **objetivo**, entonces se aplican las reglas asociadas a dicho objetivo, secuencialmente, desde la primera a la última.

- Si **TODAS** las dependencias tienen fecha anterior a **objetivo**, no se hace nada.

Este proceso se aplica de forma recursiva, de forma que si alguna de las dependencias no existe, se busca una descripción en la que dicha dependencia aparezca como objetivo, y se aplica el algoritmo anterior.

El siguiente listado recoge un ejemplo de un fichero **makefile**:

```
# Esto es un fichero makefile de ejemplo
# Makefile de arbol

arbol: arbol.o pinta.o
    gcc -o arbol arbol.o pinta.o

arbol.o: arbol.c pinta.h
    gcc -W -Wall -c arbol.c

pinta.o: pinta.c pinta.h
    gcc -W -Wall -c pinta.c

borra:
    rm *.o
```

En este ejemplo hay cuatro objetivos (**arbol**, **arbol.o**, **pinta.o** y **borra**). Nótese que los prerrequisitos de un cierto objetivo pueden ser a su vez objetivos que haya que construir.

Imaginemos que en el directorio de trabajo sólo existen los ficheros fuente (**.c** y **.h**), y que se invoca el siguiente comando:

make

En ese caso, el objetivo a construir será el primero que se encuentre en el fichero, **arbol** en el ejemplo. La secuencia que sigue **make** es la siguiente:

- busca **arbol** en el directorio de trabajo y no lo encuentra, por lo que debe aplicar las reglas secuencialmente.
- busca la línea en la que aparece **arbol** como objetivo, que en el ejemplo es la cuarta. Intenta aplicar su única regla:

  gcc -W -Wall -o arbol arbol.o pinta.o

  para lo cual necesita **arbol.o** y **pinta.o**.
- busca **arbol.o** en el directorio de trabajo y no lo encuentra, por lo que debe aplicar las reglas secuencialmente.
- busca la línea en la que aparece **arbol.o** como objetivo, que es la séptima del ejemplo. Intenta aplicar su regla:

  gcc -W -Wall -c arbol.c

  y lo consigue con éxito, generando **arbol.o**
- busca la línea en la que aparece **pinta.o** como objetivo, que es la décima del ejemplo. Intenta aplicar su regla:

  gcc -W -Wall -c pinta.c

  y lo consigue con éxito, generando **pinta.o**
- una vez que ha obtenido **arbol.o** y **pinta.o**, aplica la regla de obtención de **arbol**:

  gcc -W -Wall -o arbol.o pinta.o

  y lo consigue con éxito, generando **arbol** y dando por finalizado el proceso.

Si inmediatamente después volvemos a invocar **make**, sin modificar fichero alguno, el resultado que se obtendrá será:

```
salas@318CDCr12:~$ make
make: 'arbol' is up to date.
salas@318CDCr12:~$
```

y **make** no hace nada.

4.4.1. Objetivos sin Dependencias

Si existe algún objetivo que no tenga dependencias, en ese caso las reglas se aplicarán siempre que sea necesario obtener el objetivo, sin comprobación de ninguna clase. En el ejemplo anterior tenemos el objetivo **borra**, que se utiliza para borrar todos los ficheros **.o**. Para ello lo único que habría que hacer es:

```
salas@318CDCr12:~$ make borra
rm *.o
salas@318CDCr12:~$
```

Este caso es el único en el que el nombre del objetivo no tiene que corresponderse con el nombre de un fichero.

4.4.2. Forzar la reconstrucción completa

Se puede forzar a **make** para que reconstruya todo utilizando la orden **touch**:

touch *.[ch]

touch simplemente actualiza la fecha de modificación de cada uno de los ficheros en la lista de argumentos. En el ejemplo se actualiza la fecha de modificación de todos los ficheros terminados en **.c** y en **.h**. Una invocación subsiguiente a **make** reconstruirá todo lo que haya definido en el **makefile**.

Se puede forzar la reconstrucción total creando en el **Makefile** la siguiente dependencia:

```
all:
  touch *.[ch]
  $(MAKE)
```

y ejecutar:

make all

4.5. Macros en makefiles

Para evitar escribir repeticiones en el fichero Makefile, o para permitir su uso en distintos entornos, se usan las macros. Para definir una macro se utiliza un nombre (normalmente en mayúsculas), a continuación el carácter **<=>**, y a continuación el valor.

Ejemplo: El siguiente código define la macro **OBJETOS** y le asigna el valor `arbol.o pinta.o`:

```
OBJETOS = arbol.o pinta.o
```

Una macro se referencia haciendo preceder su nombre con un signo **<$>**, encerrando el nombre entre paréntesis (como en el ejemplo) o entre llaves.

Ejemplo: El siguiente fragmento de código utiliza la macro anterior para especificar una regla:

```
arbol:    $(OBJETOS)
      gcc -W -Wall -o arbol $(OBJETOS)
```

La utilización de las macros permite que si se añaden o eliminan ficheros para generar un ejecutable, es suficiente con modificar la definición de la macro que define los ficheros necesarios. También es de utilidad el uso de las macros para especificar parámetros globales del desarrollo, como serían el nombre del compilador a utilizar o las opciones con las que debe funcionar el compilador.

Ejemplo: El siguiente listado recoge el fichero **`Makefile`** original, modificado para hacer uso de las macros:

```
# Esto es un fichero makefile de ejemplo
# Makefile de arbol

CC = gcc
OPCIONES = -g -W -Wall
OBJETOS = arbol.o pinta.o

arbol: $(OBJETOS)
    $(CC) $(OPCIONES) -o arbol $(OBJETOS)

arbol.o: arbol.c pinta.h
    $(CC) $(OPCIONES) -c arbol.c

pinta.o: pinta.c pinta.h
    $(CC) $(OPCIONES) -c pinta.c

borra:
    rm *.o
```

Anexo 5

El depurador de programas **gdb**

Índice

5.1. Introducción **5-1**
5.2. Llamada al depurador gdb **5-2**
5.2.1. El intérprete de comandos de **gdb** 5-3
5.3. Órdenes de ayuda **5-4**
5.3.1. Orden **help** 5-4
5.3.2. Orden **info** 5-4
5.3.3. Orden **show** 5-4
5.3.4. Orden **shell** 5-4
5.4. Ejecución de programas bajo gdb: orden run **5-4**
5.5. Puntos de parada (*breakpoints*) **5-5**
5.5.1. Establecimiento de puntos de parada 5-5
5.5.2. Información sobre los puntos de parada 5-5
5.5.3. Deshabilitar un punto de parada 5-6
5.5.4. Habilitar un punto de parada 5-6
5.5.5. Borrado de un punto de parada 5-6
5.5.6. Ejecución automática en la parada 5-6
5.5.7. Continuación de la ejecución 5-7
5.6. Examinar los Ficheros Fuente **5-7**
5.7. Examinar la Pila **5-8**
5.8. Examinar los datos **5-9**

5.1. Introducción

El depurador de programas es una herramienta que permite encontrar y corregir errores de un programa mediante la ejecución paso a paso del mismo. Su propósito es permitir ver qué está ocurriendo dentro de un programa cuando se está ejecutando, o ver qué estaba haciendo un programa en el momento en que se interrumpió su ejecución de forma inesperada.

Cuando durante la ejecución de un programa se produce un error que impide que se pueda continuar con dicha ejecución, por ejemplo por acceder a una posición de memoria no autorizada, la *shell* (procesador de órdenes) envía un mensaje a la salida estándar de errores similar a:

```
Segmentation fault (core dumped)
```

En tales casos, el propio sistema operativo se encarga de generar un fichero llamado **core** en el que se facilitan datos para depurar el error producido.

El sistema operativo puede imponer límites al tamaño de dicho fichero, e incluso evitar que se cree. Para eliminar esos límites, debe ejecutarse la siguiente sentencia en el terminal en la que se va a realizar la ejecución:

ulimit -c unlimited

Para evitar tener que ejecutarlo cada vez que se abra un nuevo terminal, es conveniente incluirlo en el fichero **.bashrc**, que se encuentra en el directorio de inicio de sesión.

Para poder depurar un programa con el depurador es necesario haberlo compilado previamente con la opción **-g** del compilador:

gcc -g -W -Wall pNulo.c

Es necesario utilizar dicha opción en la compilación de cada uno de los ficheros fuentes que componen el ejecutable, así como en el enlazado de todos los objetos para obtener el ejecutable.

El depurador de programas que se va a utilizar en las prácticas es el **gdb**. Con *gdb* podemos realizar diferentes tipos de acciones que nos ayuden a averiguar dónde se está produciendo el fallo de nuestro programa:

- Hacer que nuestro programa se detenga bajo ciertas condiciones.
- Examinar qué ha ocurrido, una vez que nuestro programa se ha parado.
- Cambiar cosas del programa de forma que podamos corregir los efectos de los fallos para continuar examinando nuestro programa.
- Examinar los valores de las variables durante la ejecución de un programa.

5.2. Llamada al depurador gdb

Para arrancar el depurador se utiliza el siguiente comando:

gdb [-d *<dir>*] [*<prog>* [core]]

siendo:

- **-d *<dir>*** añade el directorio **dir** al conjunto de directorios que especifican dónde están los ficheros fuente. Por defecto, busca los ficheros en el directorio actual.
- ***<prog>*** es el nombre del programa ejecutable que queremos depurar
- **core** es el nombre del fichero que genera el sistema operativo cuando un programa tiene un error durante la ejecución, por ejemplo cuando se intenta acceder a una posición de memoria que no pertenece al programa.

A la vista del formato de la orden anterior, podemos distinguir tres formas de arrancar el depurador:

1. Sin indicar ni el programa ni el fichero de **core**. Este caso es de poca utilidad para esta asignatura, por lo que no entraremos en él.
2. Indicando el programa, pero no el fichero de **core**. En este caso se inicia una sesión de depuración para el ejecutable indicado, pero sin tener en cuenta ninguna ejecución anterior:

```
salas@318CDCr12:~$ gdb a.out
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
<---- Texto oculto ---->
```

```
This GDB was configured as "i486-linux-gnu"...
(gdb) run
Starting program: /home/salas/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0804838f in main () at pNulo.c:7
7               printf("%d", *pint);
(gdb)
```

3. Indicando tanto el programa como el fichero de **core**. Si el programa se ha ejecutado previamente y se ha producido el error de ejecución, generándose el correspondiente fichero **core**, éste se puede pasar como segundo argumento a **gdb**, para obtener información sobre el error:

```
salas@318CDCr12:~$ a.out
Fallo de segmentacion (core dumped)
salas@318CDCr12:~$ gdb a.out core
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
<---- Texto oculto ---->
This GDB was configured as "i486-linux-gnu"...

warning: Can't read pathname for load map: Input/output error.
Reading symbols from /lib/libc.so.6...done.
Loaded symbols for /lib/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Core was generated by `a.out'.
Program terminated with signal 11, Segmentation fault.
[New process 6176]
#0  0x0804838f in main () at pNulo.c:7
7               printf("%d", *pint);
(gdb)
```

Cuando invocamos al depurador, entramos en una sesión de depuración. Podemos comprobar que estamos dentro de una sesión de depuración porque el *prompt* es diferente. En lugar del habitual, tendremos uno como el siguiente:

(gdb)

lo que nos indica que quien va a interpretar las órdenes que introduzcamos por el teclado será el depurador, y no el intérprete de comandos.

Para salir del depurador, y volver al intérprete de comandos, debemos utilizar la orden **quit**:

(gdb) quit

5.2.1. El intérprete de comandos de gdb

El intérprete de comandos **gdb** presenta tres particularidades:

- La primera es que no necesita que las órdenes se escriban completamente, sino que es suficiente que no haya ambigüedad con ninguna otra orden. Por ejemplo, hay cinco órdenes que empiezan por **m**, pero sólo una (**monitor**) que empieza por **mo**. Por ello, si queremos ejecutar ese comando, bastará con que escribamos sus dos primeras letras.

  En otros casos, se priorizan las órdenes más habituales; así, por ejemplo, aunque hay 17 comandos que empiezan por **s**, si sólo se indica esa letra el intérprete asume que la orden que se desea ejecutar es **step**, por ser la más común de todas.

  En la explicación de los diferentes comandos que se van a utilizar, se indicará el nombre completo del comando, pero indicando la parte que se puede eliminar.

- La segunda es que el intérprete de comandos tiene memoria, por lo que podemos utilizar las teclas de cursor hacia arriba y hacia abajo para recuperar un comando ejecutado con anterioridad, y una vez en la línea, las teclas de cursos hacia la izquierda y hacia la derecha para editar la línea correspondiente.
- Por último, si se teclea **<Intro>** sin haber escrito nada en la línea de comandos, el intérprete de **gdb** repite el último comando ejecutado.

5.3. Órdenes de ayuda

5.3.1. Orden help

Nos proporciona información de ayuda sobre los comandos disponibles en **gdb**. Debido a la gran cantidad de comandos, se agrupan en clases, atendiendo a la funcionalidad de cada uno de ellos.

h[elp] [clase | orden]

Para conocer las clases en las que están organizadas las órdenes, utilizaremos el comando sin parámetros. Una vez conocida la clase, utilizaremos el nombre de la clase como parámetro, y obtendremos la relación de comandos que se agrupan en dicha clase.

Si utilizamos el nombre de un comando como parámetro, obtendremos la ayuda correspondiente a dicho comando. Por ejemplo:

```
(gdb) help print
(gdb) help set
```

5.3.2. Orden info

Nos proporciona información del estado del programa a depurar, como por ejemplo, puntos de parada (**info breakpoint**), variables disponibles (**info locals**), argumentos recibidos ...

Si sólo tecleamos la orden **info**, sin argumentos, obtendremos una lista de posibles informaciones que podemos obtener.

5.3.3. Orden show

Nos proporciona información del estado del propio depurador:

sho[w]

5.3.4. Orden shell

Si necesitamos ejecutar ocasionalmente comandos del procesador de órdenes (como **ls**, **mkdir**, **chmod**, **cp**, **mv**...) debemos invocar en la línea de comandos del *gdb* la orden **shell**, especificando el comando que se desea ejecutar.

she[ll] *<comando>*

siendo ***<comando>*** la orden que se desa ejecutar acompañada de sus correspondientes parámetros.

5.4. Ejecución de programas bajo gdb: orden run

Para comenzar a ejecutar el programa a depurar ejecutar la orden:

r[un] [*<argumentos>*]

siendo ***<argumentos>*** los parámetros que sea necesario trasladar al programa a ejecutar.

Se pueden comprobar los argumentos con los que se ejecutó el programa mediante **show args**.

5.5. Puntos de parada (*breakpoints*)

Los puntos de parada se utilizan para detener un programa en ejecución si se cumple una determinada condición, antes de la ejecución de una instrucción, en la que se ha puesto el punto de parada. Cuando el programa que se está ejecutando (con la orden **run**) alcanza un punto de parada, se detiene la ejecución (justo antes de ejecutar la instrucción correspondiente a este punto). También podemos establecer la parada de un programa en un punto de parada cuando el programa al llegar a dicho punto cumpla ciertas condiciones.

Los puntos de parada se pueden borrar, o bien deshabilitar para posteriormente poder utilizarlos.

5.5.1. Establecimiento de puntos de parada

Para establecer un punto de parada se utiliza la orden

b[reak] *<situación>* [if *<condicion>*]

donde:

<situacion> especifica dónde queremos poner el punto de parada; podemos utilizar cualquiera de las siguientes fórmulas:

- indicando el nombre de una función. El programa se detendrá al entrar en la función: **break *<función>***
- indicando un número de línea. La ejecución se detendrá antes de ejecutar la línea cuyo número hemos indicado del fichero fuente actual: **break *<númerodelínea>***

En ambos casos hacemos referencia al fichero actualmente activo (en el que se encuentra la sentencia en la que estamos parados). Si queremos hacer referencia a un fichero distinto, antepondremos a la ubicación el nombre del fichero seguido del carácter dos puntos:

break *<fichero>*:*<función>*

break *<fichero>*:*<númerodelínea>*

<condicion> especifica que sólo queremos que el programa se detenga cuando se cumpla la condición que hemos indicado. Si no se pone ninguna condición, el programa se detendrá siempre que llegue al punto de parada.

De esta forma, al llegar al punto de parada se evalúa la condición, y se detiene el programa sólo si es cierta: **break 8 if (i > 10)**

Si en lugar de **break** utilizamos **tbreak** el punto de parada será temporal: al llegar a él la primera vez se desactiva, no teniendo ningún efecto posterior.

Al detenerse en un punto de parada, **gdb** muestra la siguiente sentencia que se debe ejecutar.

5.5.2. Información sobre los puntos de parada

Permite obtener información sobre los puntos de parada establecidos en el programa.

i[nfo] breakpoint

5.5.3. Deshabilitar un punto de parada

Deshabilita los puntos de parada especificados. Si no se especifica ningún argumento afecta a todos los puntos de parada. Con esta orden no se eliminan los puntos de parada: siguen existiendo, pero en caso de alcanzar el punto indicado el programa no se parará.

dis[able] [*<númeropuntoparada>* [, *<númeropuntoparada>*]]

5.5.4. Habilitar un punto de parada

Habilita los puntos de parada especificados, que han debido ser deshabilitados previamente. Si no se especifica ningún argumento afecta a todos los puntos de parada que se hayan deshabilitado con anterioridad.

en[able] [*<númeropuntoparada>* [, *<númeropuntoparada>*]]

5.5.5. Borrado de un punto de parada

Hoy dos formas de borrar un punto de parada previamente establecido:

- Indicando la situación del punto de parada:

  clear [*<situación>*]

  donde ***<situación>*** puede ser cualquiera de las formas de especificar un punto de ruptura: número de línea, nombre de función, ... Si no se especifican argumentos, borra todos los puntos de parada.

- Indicando el número del punto de parada a borrar:

  delete [*<númeropuntoparada>* [, *<númeropuntoparada>*]]

  El número del punto de parada podemos obtenerlo con **info break**.

5.5.6. Ejecución automática en la parada

Se puede especificar una serie de órdenes para que se ejecuten cuando el programa se detenga porque haya alcanzado un punto de parada. Para ello empleamos la orden:

comm[ands] *<númeropuntoparada>*

Tras introducir dicha orden, escribiremos en líneas separadas las órdenes a ejecutar cuando se alcance dicho punto de parada. Cuando terminemos la lista de órdenes debemos escribir **end**.

Por ejemplo:

```
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>print n
>printf "x tiene el valor%d \n",x
>set x = x+4
>continue
>end
(gdb)
```

Este código en **gdb** realiza las siguientes acciones: en primer lugar imprime el valor de la variable **n**, luego imprime el valor de la variable **x**, después suma cuatro al valor de la variable **x**, y por último continúa con la ejecución del programa sin quedarse parado en este punto de parada.

5.5.7. Continuación de la ejecución

Una vez que nuestro programa se ha detenido en un punto de parada, podemos indicar que deseamos que continúe la ejecución del programa mediante la orden:

c[ontinue] [*<número>*]

donde el parámetro ***<número>*** especifica el número de veces que podemos volver a pasar por el punto de parada en el que se ha detenido nuestro programa sin que se vuelva a detener la ejecución del programa (ignorando dicho punto de parada) al llegar a él. Este parámetro resulta de utilidad cuando el punto de parada se encuentra dentro de un bucle y queremos pasar un cierto número de iteraciones sin detener el programa. Si no se especifica ningún número, la ejecución del programa continúa hasta alcanzar el siguiente punto de parada (o el final del programa).

Si queremos continuar la ejecución del programa en otro punto distinto utilizaremos la orden:

j[ump] *<númerodelínea>*

Podemos especificar que deseamos ejecutar la siguiente línea del código y luego retomar el control, es decir realizar una nueva parada en la siguiente instrucción. Esto se conoce como ejecución paso a paso, y se consigue con la orden:

s[tep]

Una técnica habitual consiste en establecer un punto de parada al comienzo de una función donde pensamos que podría estar el problema y luego ejecutarla paso a paso con **step**. Cada vez que se ejecuta una instrucción se muestra en pantalla la próxima instrucción que se va a ejecutar.

Por último, la orden:

n[ext]

es parecida a la orden **step**, sólo que si la instrucción a ejecutar es una llamada a una función, no se detiene en la primera instrucción ejecutable de la función, sino que se detiene al terminar la ejecución de la función.

5.6. Examinar los Ficheros Fuente

Para ver las líneas de código de los ficheros fuente utilizaremos la orden:

l[ist] [- | *<situacion>*]

siendo:

- imprime las líneas anteriores a las últimas visualizadas.

<situacion> imprime líneas de código centrándose en el punto especificado, donde ***<situacion>*** se puede especificar de idéntica forma a lo visto para el establecimiento de puntos de ruptura (número de línea o función, del fichero actual o de otro fichero).

Por defecto se imprimen 10 líneas, para variar dicho valor se empleará la orden:

set listsize *<númerodelíneasaimprimir>*

Para buscar dónde se encuentra un cierto código o texto dentro del programa, podemos utilizar la orden:

sea[rch] *<expresión>*

que busca la expresión en las líneas de código que van detrás de la última línea que se mostró (con **list**) hasta el final del código. Para buscar hacia atrás (desde la última línea mostrada hasta el comienzo) podemos utilizar la orden:

rev[erse-search] *<expresión>*

Si no se especificaron los directorios en los que encontrar los ficheros fuentes en la invocación del depurador (con la opción **-d**), puede utilizarse en cualquier momento la orden:

dir[ectory] ***<nombrededirectorio>***

que añade un nuevo directorio a la lista de directorios donde buscarlos.

5.7. Examinar la Pila

Una vez que nuestro programa se detiene, necesitamos conocer dónde se ha detenido y cómo ha llegado hasta allí. Cada vez que nuestro programa realiza una llamada a una función, se genera una cierta información sobre dicha llamada. Dicha información se guarda en un bloque de datos denominado **trama**. Las tramas se van almacenando en una pila de llamadas cada vez que se llama a una función, y desaparecen de la pila cuando se retorna de la función.

Por tanto, la pila de llamadas está dividida en tramas que representan las llamadas a funciones. Estas tramas contienen los argumentos pasados a la función que ha sido llamada, variables locales, ...

Cuando comienza la ejecución de un programa, la pila sólo tiene una trama, la correspondiente a la función **main**, que se denomina **outermost frame** o **initial frame**. Cada vez que se llama a una función se crea una nueva trama y se guarda en la pila, y cuando la función finaliza se elimina dicha trama de la pila. La trama de la función que se estaba ejecutando cuando se detuvo la ejecución del programa (en un punto de parada) se denomina: **innermost frame**. *gdb* asigna un número a todas las tramas existentes en la pila, comenzando con **0** para la trama **innermost**.

Para examinar la pila utilizaremos el comando:

ba[cktrace]

que proporciona un resumen de cómo nuestro programa llegó al punto de parada actual. Muestra toda la pila de tramas, comenzando por la actual.

```
(gdb) backtrace
#0  pregunta (p_numero=0xbfb66084, mensaje=0xbfb660ab "Introduzca un numero")
    at pregunta.c:30
#1  0x080484b1 in inicializa (mensaje=0xbfb660ab "Introduzca un numero")
    at pregunta.c:22
#2  0x08048464 in main () at pregunta.c:11
(gdb)
```

Pueden observarse tres tramas:

#0 La función actual: correspondiente a **pregunta()**

#1 La función que la llamó: correspondiente a **inicializa()**

#2 La inicial: correspondiente a **main()**

Interpretándolo desde la trama final: la función **main** llamó a **inicializa** que llamó a **pregunta**, que es donde estamos detenidos actualmente.

La orden:

f[rame] ***<númerodetrama>***

nos permite movernos en la pila a la trama especificada en ***<númerodetrama>***.

Al movernos a una trama, se imprimirán dos líneas donde se muestra el nombre de la función correspondiente a esta trama con sus argumentos, y la línea que se está ejecutando en esta función. Si ahora invocamos la orden **list** sin argumentos, se nos muestran 10 líneas de código centradas en ese punto.

Con la orden:

i[nfo] frame

se nos muestra información de la trama seleccionada en la pila (sobre todo información relacionada con las direcciones de memoria que se están utilizando).

Con la orden:

i[nfo] args

podemos ver los argumentos de la trama seleccionada.

5.8. Examinar los datos

La forma usual de examinar datos es con la orden:

p[rint] *<expresión>*

donde ***<expresión>*** es una expresión del lenguaje del código fuente, en este caso 'C'. Cualquier tipo de variable, constante u operador definido en el lenguaje de programación, se acepta como expresión, incluyendo expresiones condicionales, llamadas a funciones, . . . , a excepción de los símbolos definidos en el preprocesador (con **#define**).

Cuando se imprime el valor de una variable, se entiende que es de la trama seleccionada. Si queremos referirnos a una variable en otra trama, debemos movernos a dicha trama (con la orden **frame**).

Por defecto **print** nos muestra una variable en su formato correcto. Puede forzarse a que el formato de salida sea otro diferente con los modificadores:

/d Entero decimal con signo

/u Entero decimal sin signo

/t Binario

/a Imprime una dirección, tanto en valor absoluto (hexadecimal) como en forma de desplazamiento respecto al símbolo que le precede de forma más cercana.

/c Constante de carácter

/f Número en coma flotante.

Si se quiere imprimir el valor de una expresión frecuentemente (por ejemplo, para ver cómo va cambiando), puede añadirse a una *lista de display* automática, que se imprime cada vez que *gdb* detiene la ejecución de un programa. Para añadir una expresión o variable a dicha lista, utilizaremos la orden:

disp[lay] /*<formato>* *<expresión>*

Esto añade la expresión a la lista con un cierto número, y se imprimirá el valor de la expresión cada vez que se detenga la ejecución del programa. Para quitar una expresión de la lista de expresiones a imprimir automáticamente, se utiliza:

und[isplay] *<númerodelaexpresión>*

Con:

i[nfo] display

obtendremos una lista de todas las expresiones que se encuentran en la lista, con su número de expresión correspondiente.

Todos los valores que obtenemos con la orden **print** son almacenados en el historial de valores de *gdb*, en variables auxiliares creadas por *gdb* automáticamente al mostrar el resultado ($1, $2, . . .). Se pueden ver con:

sho[w] values

Puede imprimirse el contenido de un array o tabla mediante el operador **@**. Por ejemplo. si tenemos un programa con una línea de código tal como:

```
int *array = (int *) malloc (elementos * sizeof (int));
```

podemos imprimir los valores de los elementos de la tabla, indicando detrás del carácter **@** el número de elementos:

print *array@elementos

Otra fórmula muy utilizada es usar variables auxiliares como contadores, y luego repetir las expresiones mediante **<Intro>** (ya que cuando se pulsa esta tecla se repite la última orden). Por ejemplo, si se tiene una tabla de punteros a registros denominada **dtab**. Para ver los valores del campo **fv** en cada una de los registros podríamos hacer lo siguiente:

```
(gdb) set $i = 0
(gdb) print dtab[$i++]-> fv
(gdb)
(gdb)
```

La primera línea declara una variable auxiliar **$i** (todas las variables auxiliares comienzan por **$**) con el valor **0**. Dicha variable es una variable local de *gdb*.

Para examinar la memoria utilizaremos la orden **x**:

x /nfu [*<direcciónininicial>*]

donde:

/u nos indica el tamaño de la unidad a imprimir:

- **b** 1 octeto
- **h** 2 octetos
- **w** 4 octetos (valor por defecto)
- **g** 8 octetos

/n nos indica el número de unidades a imprimir

/f el formato de la salida (igual que en **print**):

- **s** cadena de caracteres terminada en **\0**
- **i** instrucción en lenguaje máquina
- **x** hexadecimal (valor por defecto)
- **c** carácter

En el siguiente ejemplo se imprime la variable **mensaje**, que contiene la cadena "`Introduzca un numero`", utilizando diferentes formatos:

```
(gdb) print mensaje
$17 = "Introduzca un numero"
(gdb) x /6 mensaje
0xbfb660ab:     0x72746e49      0x7a75646f      0x75206163      0x756e206e
0xbfb660bb:     0x6f72656d      0x0a000000
(gdb) x /21c mensaje
0xbfb660ab:     73 'I'  110 'n' 116 't' 114 'r' 111 'o' 100 'd' 117 'u' 122 'z'
0xbfb660b3:     99 'c'  97 'a'  32 ' '  117 'u' 110 'n' 32 ' '  110 'n' 117 'u'
0xbfb660bb:     109 'm' 101 'e' 114 'r' 111 'o' 0 '\0'
(gdb)
```

Con la primera orden (**print mensaje**) se ve la cadena de caracteres que contiene la variable.

Con la segunda orden (**x /6 mensaje**) se muestra en hexadecimal 6 grupos de 4 octetos a partir de la dirección **mensaje**.

Con la tercera orden (**x /21c mensaje**) se muestra el carácter (y su valor correspondiente en decimal a su izquierda) de 21 octetos a partir de la dirección mensaje.

Anexo 6

Guía de Estilo. Consejos y normas de estilo para programar en C

Índice

6.1. Introducción **6-1**
6.2. El fichero fuente **6-2**
6.3. Sangrado y separaciones **6-3**
6.4. Comentarios **6-4**
6.5. Identificadores y Constantes **6-5**
6.6. Programación Modular y Estructurada **6-6**
6.7. Sentencias Compuestas **6-6**
6.7.1. Sentencia **`switch`** 6-7
6.7.2. Sentencia **`if-else`** 6-7
6.8. Otros **6-7**
6.9. Organización en ficheros **6-8**
6.9.1. Ficheros de cabeceras 6-8
6.10. La sentencia `exit` **6-9**
6.11. Orden de evaluación **6-9**
6.12. Los operadores `++`, `--` **6-9**
6.13. De nuevo los prototipos **6-10**

6.1. Introducción

Se ha insistido a lo largo del curso de lo importante de programar de forma estructurada (utilizando las estructuras de control vistas en clase), y de la utilidad de hacerlo de forma modular (módulos o subprogramas reutilizables, en los que resulta más fácil la detección de errores que considerando la totalidad).

Al mismo tiempo se hace hincapié en que el código de nuestros programas debe ser legible. Esto implica claridad en su escritura, para facilitar su posterior lectura, mantenimiento y modificación. Es también importante el uso de comentarios que permitan el entendimiento del programa a personas ajenas al mismo o al propio programador una vez pasado un cierto tiempo.

Para ello intentaremos aplicar una serie de normas de "estilo" a la hora de programar. Aquí daremos las principales, y a lo largo de las prácticas iremos insistiendo en ellas y añadiremos algunas otras.

6.2. El fichero fuente

Existe una estructura global que se debe respetar, que será la siguiente:

1. Todos los ficheros deben comenzar con un comentario, con el nombre del fichero, autor, fecha de realización, y una breve descripción de su contenido. Existen numerosos estilos para este tipo de introducciones. Se recomienda que no sea demasiado extenso.

2. A continuación se deben incluir los ficheros de cabecera (ficheros **.h**), mediante las directivas de prepocesado **include**. Primero los ficheros de cabecera del sistema (**stdio.h**, **stdlib.h**, etc.) y luego los propios de la aplicación.

3. Constantes simbólicas y tipos definidos por el usuario.

4. Variables estáticas si las hubiera (en esta asignatura no están permitidas, ya que pueden da lugar a errores difíciles de detectar).

5. Prototipos de funciones locales al fichero (que no se usan en otros ficheros).

6. Funciones. Primero **main** (si existe), y a continuación el resto, agrupadas por algún criterio. Si no existe criterio, al menos alfabéticamente. Se recuerda que la declararación o prototipo de una función debe aparacer antes de ser llamada.

Cada una de estas secciones debe tener un comentario que la preceda.

El siguiente fragmento es un ejemplo de lo anterior:

```
/*
**    Fichero:   demostraciones.c
**    Autor:     Fulanito Menganito Blas
**    Fecha:     21-12-99
**
**    Descripcion:   conjunto de funciones de demostraciones
**                   matematicas..
*/

/* Includes del sistema */
#include <stdio.h>

/* Includes de la aplicacion */
#include "ejemplos.h"

/* constantes */
#define MAXIMO  100
#define ERROR "Opcion no permitida"

/* tipos definidos por el usuario */


/* Prototipo de funciones locales */


int main()
{

  return 0;
}

/* Definiciones de funciones locales */
```

6.3. Sangrado y separaciones

Se debe usar el espaciado tanto vertical como horizontal "generosamente", para reflejar la estructura de bloques del código (a esto se le llama sangrado o indentación). Un programa bien indentado será mucho más legible que uno mal indentado o sin indentar.

Sólo debemos tener una sentencia por línea:

```
if (correcto)
  printf("no hubo problemas");
else
  printf("SI hubo problemas");
```

en vez de

```
if (correcto) printf("no hubo problemas");
else printf("SI hubo problemas");
```

Las distintas partes o secciones del fichero deben ir separadas por al menos una línea en blanco. Por ejemplo, la parte de include de la sección de constantes, etc.

De la misma manera, dentro de una función los distintos bloques deben ir separados por líneas en blanco:

```
#include <stdio.h>

int main()
{
  int num=0;                /* numero a leer de teclado */
  int contador=0;           /* contador total */
  int impares=0;            /* contador de impares */
  int positivos=0;          /* contador de positivos */
  int resp;                 /* respuesta del usuario */

  /* Separamos definiciones de codigo */

  do
    {
      /* Solicitamos el numero por teclado */
      printf("\nIntroduzca un numero: ");
      scanf("%d", &num);

      /* Contamos los numeros, los positivos y los impares */
      contador++;
      if (num > 0)
        positivos++;
      if (num % 2)
        impares++;

      while (getchar() != '\n')   /* leer hasta  */
        ;                         /* nueva_linea */
      printf("Terminar (S/N)?");
      resp = getchar();

    }
  while (resp!='S');

  printf("Numeros: %d \t Pares: %d \t Positivos: %d \n",
         contador, contador - impares, positivos);

  return 0;
}
```

Hay distintos estilos a la hora de utilizar las llaves de bloques, bucles, etc. Se recomienda utilizar siempre el mismo. El propuesto es:

```
control
  {
    sentencia;
    sentencia;
  }
```

ya que es visualmente sencillo comprobar los pares de llaves.

Cuando tengamos una expresión condicional de bastante longitud, se debe separar en varias líneas, tabulando el comienzo de cada una para que queden alineadas las condiciones:

```
/* Incorrecto: !No demasiado legible! */
while ( cond1 != "error"  && cond2 !=TRUE &&  cond3 >=
7.5 && cond4 <=10 && cond5 !="esto es verdad")

/* correcto */
while (cond1 != "error" && cond2 != TRUE &&
       cond3 >= 7.5 && cond4 <= 10 &&
       cond5 != "esto es verdad")
```

Debido a que es posible que nuestro fichero se utilice en distintas plataformas, y se edite con distintos editores de texto, se debe limitar la longitud de las líneas a 70 caracteres. Por ejemplo, si estamos realizando comentarios al código y sobrepasamos esta longitud se debe introducir un retorno de carro y continuar en la siguiente línea.

Por la misma razón, se deben evitar en los comentarios los caracteres acentuados, la eñe, y en general todo carácter ASCII superior al código 127.

Líneas de gran longitud debidas a una tabulación demasiado profunda a veces reflejan una pobre organización del código.

6.4. Comentarios

"Cuando el código y los comentarios no coinciden, probablemente ambos sean incorrectos". Norm Schryer.

Los comentarios breves se pueden realizar a la derecha del código, y con una cierta tabulación para que no se confunda con el código. Se debe procurar que todos los comentarios estén al mismo nivel.

Cuando se comenta un bloque de código se puede comentar previamente y al mismo nivel de sangrado.

```
/*
** bucle que mientras el deposito no esta lleno aumenta
** el caudal siguiendo las tablas de Leibniz-Kirchoff
*/
while (nolleno)
  {
     ...
  }
```

Tan malo es no comentar como comentar en exceso o innecesariamente. Hay que evitar comentarios que se puedan extraer o deducir fácilmente del código:

```
i++;  /* Se autoincrementa el valor de i en una unidad */
```

Cuando se define una función se debe preceder de un comentario con el funcionamiento básico de la función.

Se deben comentar brevemente los prototipos de las funciones en los ficheros de cabecera, explicando el significado de los parámetros y el valor devuelto por la función. Los detalles del funcionamiento interno estarían en los comentarios de la definición de la función.

6.5. Identificadores y Constantes

Se deben evitar los identificadores que comiencen o terminen con el carácter de subrayado (_), ya que suelen estar reservados para el sistema.

Recuerde que los identificadores deben ser representativos del valor que identifican. Además le ahorrarán comentarios:

```
velocidad = espacio / tiempo;
```

en vez de

```
resultado = valor / aux2;
```

Si se usan variables locales en distintas funciones, pero con el mismo significado, ayuda a la legibilidad mantener el identificador. En cambio, confunde utilizar el mismo identificador con diferentes significados.

A la hora de definir variables, se debe definir una por línea. Al mismo tiempo los tipos, identificadores y comentarios deben aparecer al mismo nivel para dar mayor claridad y facilidad de lectura.

```
int  trayecto[NUMPARADAS]; /* comentario sobre trayecto */
int  origen;               /* comentario sobre origen   */
char destino[LONDESTINO];  /* comentario sobre destino  */
```

Evitar nombres que difieran en una sola letra de otros, la confusión es muy probable.

Evitar el uso de identificadores que puedan coincidir con bibliotecas del sistema.

Para evitar la confusión entre asignaciones y comparaciones, es conveniente utilizar en las comparaciones la constante a la izquierda de la variable para obtener un error en compilación si relizamos una asignación (=) en lugar de una comparación (==): if (4 == a) en lugar de if (a == 4)

Las constantes simbólicas deben ir en MAYÚSCULAS.

Las funciones, nombres de variables, tipos definidos por el usuario, etiquetas de estructuras y uniones irán en minúsculas.

Las constantes de una enumeración van todas en mayúsculas, o al menos con la primera letra en mayúsculas.

En general, los valores numéricos no deben ser usados directamente en el código, dan lugar a lo que se conoce como "números mágicos", que hacen el código difícil de mantener y de leer. Para ello se usan las constantes simbólicas, o las enumeraciones.

Las constantes de tipo carácter deben ser definidas como literales y no como sus equivalentes numéricos:

```
#define PRIMERNUMERO    48      /* INCORRECTO */
#define PRIMERNUMERO    '0'     /* CORRECTO */
#define PRIMERALETRA    'A'
```

El calificador de puntero * debe ir con el nombre de la variable en vez de con el tipo.

```
int *pi;
```

y no se deben mezclar de la forma

```
int x, y, *z;
```

En una estructura o unión, los elementos de ésta deben ir cada uno en una línea, tabulados y con un comentario:

```
struct barco
{
  int   flotacion;   /* en metros */
  int   tipo;     .  /* segun clasif. internacional */
  float precio;      /* en dolares */
```

```
};
```

6.6. Programación Modular y Estructurada

Sólo se deben utilizar las estructuras que la programación estructurada recomienda.

No se deben usar las sentencias **goto** o **continue**. Con respecto a la sentencia **goto**:

- *Su uso está restringido en una programación estructurada, pues tiende a generar programas ilegibles de difícil entendimiento y comprensión, lo que dificulta, y en ocasiones impide, cualquier tipo de actualización o modificación sobre los mismos, obligando al programador a desarrollar un nuevo programa o aplicación.* Fundamentos de Programación, J. López Herranz, E. Quero Catalina, Ed. Paraninfo.

- *In the case of the goto statement, it has long been observed that unfettered use of goto's quickly leads to unmaintainable spaghetti code.* Indian Hill Recommended C Style and Coding Standards.

La sentencia **break** sólo tiene justificado su uso en la selectiva múltiple, es decir, con la sentencia compuesta **switch**.

Toda función tendrá un único punto de retorno (una única sentencia **return**), que además deberá ser la última línea de la función. Además, deberá devolver el control a la función que la invocó, y no a otra o al sistema.

Aunque no existen reglas estrictas, el tamaño de una función no debería superar una o dos páginas en pantalla, para permitir una visualización correcta de la misma. Si no es así, posiblemente no se está descomponiendo el programa adecuadamente.

Siempre se debe especificar el tipo de retorno de una función. Si no lo tiene, especificar **void**.

En ficheros que sean parte de un programa más grande, hay que procurar que las funciones y variables sean lo más "locales"posible. Para ello se recomienda usar el calificador **static**.

La división en funciones se debe hacer de forma que cada función tenga una tarea bien definida. Al menos deben codificarse en forma de función aquellas tareas con cierta entidad que se realizan dos o más veces en un programa.

6.7. Sentencias Compuestas

Las sentencias compuestas son aquellas que van encerradas entre llaves. Hay varias formas de formatear estas llaves, pero es importante seguir siempre una. La propuesta, como ya vimos en un apartado anterior es:

```
control
  {
    sentencia;
    sentencia;
  }
```

Si el cuerpo de un bucle o función es nulo debe estar sólo en una línea, y comentado para saber que no es un error u olvido, sino intencionado:

```
  for ( i = 0; NULL != tabla[i]; i++ )
    ;
```

El cuerpo de los bucles **do-while** siempre debe ir entre llaves.

6.7.1. Sentencia **switch**

Las etiquetas deben ir el líneas diferentes, y de igual modo el código asociado a cada una de ellas.

Se debe incluir siempre una sentencia **break** al final de cada uno de los casos, y si se desea un comportamiento "en cascada" se debe comentar muy claramente.

```
switch (expresion_entera)
  {
    case ETIQ_1:        /* igual que ETIQ_2 */
    case ETIQ_2:
      sentencia;
      break;
    case ETIQ_3:
      sentencia;
      break;
  }
```

6.7.2. Sentencia **if-else**

Si cualquiera de los brazos tiene una sentencia compuesta, se recomienda que ambos brazos sean encerrados con llaves. Estos además son necesarios en el caso de sentencias **if-if-else** (si se quiere asociar el **else** al primer **if**).

La sentencia **if-else if-else if ...** se debe codificar con los **else if** alineados a la izquierda, para que refleje un comportamiento selectivo más que anidado:

```
if (expresion)
  {
    ...
  }
else if (expr2)
  {
    ...
  }
else if (expr3)
  {
    ...
  }
else
  {
    ...
  }
```

6.8. Otros

Es preferible realizar la comparación con 0 que con 1, ya que la mayoría de las funciones garantizan el cero si es falso, y no cero si es verdadero. Por lo tanto:

```
if (VERDADERO == func())
```

se debe codificar como

```
if (FALSO != func())
```

(suponiendo que `FALSO` y `VERDADERO` son 0 y 1 respectivamente).

Las asignaciones embebidas deben evitarse ya que hacen el código más difícil de leer, como en:

```
d = (a = b + c) + r;        /* Incorrecto */
```

En cambio, cuando se trate de funciones de la biblioteca estándar que devuelvan un resultado que se utilice en la condición de una expresión condicional, puede resultar útil la llamada en la propia expresión, como en:

```
while (EOF != (c = getchar()))
  {
    /* procesa el caracter */
  }
```

Utilizar con precaución los operadores **++** y **--**, ya que pueden dan lugar a efectos laterales, código poco legible, etc. (ver un ejemplo en un apartado específico más abajo).

6.9. Organización en ficheros

El primer carácter del nombre del fichero debe ser letra, y el resto letras, números o el carácter de subrayado (_). Debe procurarse usar siempre minúsculas, reservando las mayúsculas como separación entre palabras.

El nombre de los ficheros fuente en lenguaje C deben tener como sufijo **.c** o **.h** (en minúsculas). Los ficheros con las definiciones de las funciones deben ser **.c** (si no es así el compilador no los acepta).

Se deben evitar ficheros de más de 1000 líneas.

Los ficheros **.c** **NO** deben ser incluidos en otros ficheros.

6.9.1. Ficheros de cabeceras

Contienen las definiciones de constantes, tipos definidos por el usuario y los prototipos de las funciones utilizadas en nuestro programa.

Tienen la extensión **.h**. Los ficheros de cabecera del sistema (p.e. **stdio.h**, **string.h**, etc..) se encuentran en un directorio predeterminado que el compilador conoce, por ello para incluirlos en nuestro programa no es necesario especificar la ruta de los ficheros.

Cada fichero de definición de funciones debe llevar asociado un fichero de cabecera, con los prototipos de las funciones que vayan a ser utilizadas en otros ficheros. Ambos ficheros deben tener el mismo nombre, pero el de definición terminado en .c y el de prototipos terminado en .h .

No usar los nombres de ficheros de cabecera del sistema para los de la aplicación.

No usar caminos absolutos para la inclusión de los ficheros de cabecera definidos por el usuario. Usar caminos relativos al directorio de trabajo entrecomillados.

Los ficheros de cabecera deben ser incluidos en el fichero que define la función o variable. De esta forma se hace comprobación de tipos, ya que el compilador comprueba que la declaración coincide con la definición (cada fichero .c incluye al menos a su correspondiente .h).

No se debe incluir en ellos definiciones de variables.

No se deben anidar ficheros de cabecera. Para evitar doble inclusión se deben utilizar las órdenes de preprocesado que hacen que se defina una etiqueta en el momento de la primera inclusión del fichero de cabecera. Por ejemplo, si se tuviera un fichero de cabecera de nombre **cuentaPalabras.h** con el prototipo de la función **cuentaPalabras**, debería quedar así:

```
#ifndef CUENTAPALABRAS_H
#define CUENTAPALABRAS_H

void cuentaPalabras(int argc,char **argv,
                    int *palabras,int *total);

#endif
```

De esta manera, si se intentara incluir de nuevo el mismo fichero de cabecera, al comprobar que la etiqueta (CUENTAPALABRAS_H) ya está definida, el preprocesador se saltaría todo lo que hay hasta el final del fichero (donde está **#endif**).

Se deben declarar todas las funciones que se utilicen. Para usar funciones que están definidas en otros ficheros, se deben incluir los ficheros de cabecera correspondientes. Si existen funciones locales al fichero (sólo se usan en ese fichero) sus prototipos no irán en el correspondiente **.h**, sino que irán identificados claramente en el propio fichero **.c**.

6.10. La sentencia **exit**

Esta sentencia no se ha explicado en las clases teóricas. Y a pesar de ello, en algún momento, es posible sentirse tentado de usarla, posiblemente sin conocer muy bien su funcionamiento.

Y aunque no se ha prohibido su uso expresamente, sí se ha realizado de forma indirecta. Recordemos una de las bases de la programación estructurada.

Toda función tendrá un único punto de retorno, que además deberá ser la última línea de la función. Además, deberá devolver el control a la función que la invocó, y no a otra o al sistema.

¿En qué consiste la sentencia exit? Esta sentencia provoca la interrupción del programa que se está ejecutando en el punto donde se encuentre, y permite devolver un valor al entorno desde el que se ejecutó el programa. Ya podemos ver que esta sentencia rompe el flujo normal del programa, y hace que nuestro programa no sea estructurado.

Conclusión: **no se puede utilizar la sentencia exit**.

6.11. Orden de evaluación

En la siguiente expresión:

```
f() + g() * h()
```

se puede asegurar que la multiplicación se realiza antes que la suma, pero no en qué orden se llama a las funciones.

Pero, ¿qué ocurre con los operadores lógicos && y ||? Son una excepción, y se asegura la evaluación de izquierda a derecha. Esto permite sentencias como:

```
while( (c = getchar()) != EOF  && c != '\n')
```

Además, en la evaluación de una expresión con operadores lógicos, se deja de evaluar cuando se puede asegurar el resultado de la expresión. Por ejemplo, en la siguiente condición, no se llama a la función comprueba si p vale NULL (ya que si p vale NULL la condición completa es falsa independientemente del valor que devolviera la función comprueba):

```
if (p != NULL && comprueba(*p))
  {
    ...
  }
```

6.12. Los operadores ++, --

```
Fichero: opera.c
/*
```

```
** Operador de incremento
*/
#include <stdio.h>

int main()
{
  int i = 7;

  printf("%d \n", i++ * i++);
  printf("%d \n", i);

  return 0;
}
```

Hay que evitar este tipo de expresiones. Se asegura que **++** provoca el incremento de la variable después de su uso. Este "después" es ambiguo, y el estándar lo deja indefinido. Sí aclara que se incrementa antes de ejecutar la siguiente sentencia. Luego el compilador puede optar por multiplicar 7*7, o 7*8. El conocer cómo realiza esta operación "nuestro" compilador no debe servirnos de referencia, ya que no conocemos cómo se realiza en otros compiladores.

6.13. De nuevo los prototipos

A pesar de todo lo comentado hasta ahora, puede que no se haya convencido de la necesidad de declarar una función antes de utilizarla, es decir, que aparezca su prototipo antes de la llamada.

Para convencerle de su necesidad, se plantea el siguiente ejemplo:

```
/*
**  Fichero: proto.c
**
**  Demostracion de error debido a falta de prototipo de
**  funcion. En este caso, al no existir el prototipo, el
**  compilador supone que la funcion multip devuelve un
**  valor de tipo int.
*/

#include <stdio.h>

int main()
{
  double prod;
  int    res;

  prod = multip(7.3, 3.5);

  printf("\n Prod= %f \n", prod);

  return 0;
}
```

```
/*
** Fichero: proto_aux.c
**
** Fichero de demostracion del problema de no
** utilizar prototipos de las funciones
*/

double multip(double a, double b)
{
  return a * b;
}
```

Si los compilamos y ejecutamos:

```
%> gcc proto.c proto_aux.c

%> a.out
```

se puede observar que el resultado de la operación no es el esperado. Pruebe a compilar como:

```
%> gcc -Wall proto.c proto_aux.c
```

Se observa que ahora el compilador sí informa de estos problemas.

Parte III

Ejercicios

Ejercicios tema 3

Codificación de la Información

1. Determinar la representación binaria de 128.
2. Obtener la representación binaria de 1254674, utilizando como paso intermedio la codificación octal y la hexadecimal.
3. Obtener la representación en complemento a dos utilizando 2 octetos de las siguientes cantidades:
   - -135
   - -13000
   - 13000
4. Obtener la representación decimal de los siguientes números, sabiendo que están codificados en complemento a dos:
   - 11100000_2
   - 00100001_2
   - 1000000110001110_2
5. Determinar el mayor valor que puede representarse con 16 bits si:
   - se utiliza la codificación binaria.
   - se utiliza la codificación en complemento a dos.
6. Dada la siguiente secuencia: 1001001110101001_2, encontrar su representación decimal si:
   - está codificada en binario.
   - está codificada en complemento a dos.

Ejercicios tema 4

Tipos, Constantes y Variables

1. ¿Qué tipo permite representar el menor entero en una máquina?. ¿Y el mayor entero?.
2. Determinar el menor y mayor valor que puede representarse para cada uno de los tipos según el tamaño indicado en 4.1.2.
3. Repetir el ejercicio anterior para cada uno de los tipos según el tamaño indicado en 4.1.3.
4. Escriba la constante entera 12345 en cada uno de los tres sistemas numéricos utilizados en C.
5. Utilizando la tabla de caracteres que aparece en la página 4-4, exprese la constante de carácter que representa el carácter punto y coma en los tres formatos especificados en el apartado 4.2.3.
6. Utilizando la tabla de caracteres que aparece en la página 4-4, exprese la constante entera que representa el carácter punto y coma en los tres sistemas numéricos.

Ejercicios tema 5

Expresiones

1. Determinar el orden de ejecución de los operadores en la siguientes expresiones:

```
i + f <= 10
i >= 6 && c == 'w'
c != 'p' || i + f <= 10
error >.0001 || cont <100
```

2. Si **i** y **j** son variables de tipo **int**, ambas con valor **4**, determinar el resultado de las siguientes expresiones:

```
(double) (i / 3 + j / 3)
(double) i / 3 + j / 3
(double) i / 3 + (double) j / 3
```

3. ¿Cuál es el valor numérico de las siguientes expresiones?

```
5 > 2
3 + 4 > 2 && 3 < 2
x >= y || y > x
d = 5 + (6 > 2)
'X' > 'T' ? 10 : 5
x > y ? y > x : x > y
```

4. Construya una expresión para indicar las siguientes condiciones:

   - **numero** es igual o mayor que **1** y menor que **9**.
   - **ch** (variable de tipo carácter) no es una **q** ni una **k**.
   - **numero** está entre **1** y **9**, pero no es un **5**.
   - **numero** no está entre **1** y **9**.

5. Indique cuáles de las siguientes proposiciones son ciertas y cuáles son falsas:

```
100 > 3
'a' > 'c'
100 > 3 && 'a' > 'c'
100 > 3 || 'a' > 'c'
!(100 > 3)
```

Ejercicios tema 6

Estructuras de Control

1. Repita el programa del año bisiesto, partiendo de la suposición de que el año sí es bisiesto.

2. Modifique el programa **whileBien.c** del texto si el bloque de la sentencia **while** es:

```
  suma += num;
  num++;
```

3. Implememte mediante sentencia **for**, **while** y **do while** un programa que calcule el factorial de un número que se lee desde el teclado.

4. El siguiente programa imprime los 10 primeros enteros utilizando una sentencia **for**:

```
#include <stdio.h>

#define MAX 10

/* Representacion de un bucle desde con mientras y
   hacer-mientras. */

int main()
{
  int i = 0;      /* Variable indice o de control */

  printf("\n Bucle DESDE con for\n");
  for (i = 1; i <= MAX; i++)
    printf("%d\t", i);

  return 0;
}
```

   Escriba un programa que realice la misma función utilizando un bucle **while**. Repita el ejercicio utilizando un bucle **do-while**.

5. El siguiente código determina si un número es par o impar:

```
#include <stdio.h>

/* Determina si un numero cualquiera (incluso negativo)
   es par o impar. */

int main()
{
  int num = 0;
```

```
  printf("\nIntroduzca el numero: ");
  scanf("%d", &num);
  switch (num % 2)
    {
    case 0:
      printf("\n El numero %d es par\n", num);
      break;
    case 1:
    case -1:
      printf("\n El numero %d es impar\n", num);
      break;
    default:
      printf("\n El modulo es distinto de 0, 1, -1\n");
      break;
    }

  return 0;
}
```

Modificarlo sustituyendo el bloque **`switch`** por una simple sentencia selectiva **`if`**.

6. Realizar un programa que pida un número positivo y escriba el cuadrado del mismo si dicho número se encuentra comprendido dentro del rango de 0 a 100, en caso contrario dar un mensaje.

7. Programa que calcule el valor medio de los números del 100 al 1000.

8. El siguiente programa (que denominaremos **`cuentaLineas.c`**) lee el número de líneas de texto que se escriben por teclado:

```
#include <stdio.h>

/* Programa que lee el numero de lineas a la entrada.
   El texto de entrada es una secuencia de lineas, cada una de
   ellas terminada con un '\n' (caracter nueva linea).
   El caracter que marca el final del texto es EOF, que en los
   terminales se consigue con Ctrl-D. */

int main()
{
  int c  = 0;      /* Almacena caracteres */
  int nl = 0;      /* Cuenta lineas */

  c = getchar();
  while (c != EOF)
    {
      if (c == '\n')
        ++nl;
      c = getchar();
    }
  printf("El numero de lineas es %d\n", nl);

  return 0;
}
```

Modificar el código para que cuente todos los caracteres.

9. Modificar **`cuentaLineas`** para que cuente palabras.

10. Escriba un programa que pida un número y un exponente, ambos enteros, y calcule el valor del número elevado al exponente, utilizando la propiedad de que elevar un número a un exponente equivale a multiplicar el número tantas veces como indique el exponente.

11. Escribir un programa que lea números enteros del teclado y los sume. El programa terminará cuando se introduzca un número negativo, imprimiéndose la suma.

12. Realizar un programa que multiplique, sume o reste dos números en función de la opción elegida.

Ejercicios tema 7

Funciones

1. Función que devuelva la suma de 100 números dados por teclado.
2. Función que devuelva verdadero o falso si el número dado por teclado es o no menor que el parámetro que se le pase al módulo.
3. Función que recibe un número positivo y calcula la suma de los números desde 1 a dicho número
4. Función que recibe un número positivo y calcula la suma y el producto de los números desde 1 a dicho número.
5. Función que imprima por pantalla la sucesión de Fibonacci hasta el término **n**, que se pasará como parámetro. Se define esta sucesión de la siguiente forma:

$$\begin{aligned} F(0) &= 0 \\ F(1) &= 1 \\ F(n) &= F(n-2) + F(n-1) \end{aligned}$$

```
int fibo (int n)
{
  int r;

  if (n == 0)
    r = 0;
  else if (n==1)
    r = 1;
  else
    r = fibo(n - 1) + fibo(n - 2);

  return (r);
}
```

6. El siguiente programa calcula la pontencia entera de un número aplicando la multiplicación de forma recursiva.

```
/*
** Ejemplo de funcion potencia con recursividad
*/
#include <stdio.h>

double potencia(double, int);

int main()
```

```
{
  double x;
  int    n;

  printf("Teclee base: ");
  scanf("%lf", &x);
  printf("Teclee exponente: ");
  scanf("%d", &n);
  printf("Potencia %d de %f es : %f\n",
         n, x, potencia(x, n));

  return 0;
}

double potencia(double x, int n)
{
  double local;

  if (n == 0)
    local = 1;
  else
    local = (x * potencia(x, n - 1));

  return local;
}
```

7. En el siguiente ejemplo, se obtiene la división entera de dos números aplicando la propiedad de que el cociente es el número de veces que el dividendo contiene al divisor.

```
#include <stdio.h>

int div_entera(int a, int b);

int main()
{
  int dividendo;
  int divisor;
  int cociente;

  printf("Introducir dividendo: ");
  scanf("%d", &dividendo);
  printf("Introducir divisor: ");
  scanf("%d", &divisor);

  cociente = div_entera(dividendo, divisor);

  printf("La division entera de %d y %d es %d\n",
         dividendo, divisor, cociente);

  return 0;
}


int div_entera(int a, int b)
{
  int coc;

  if (a < b)
    coc = 0;
  else
    coc = 1 + div_entera(a - b, b);

  return coc;
}
```

8. Función que calcule el factorial de un número de forma recursiva y no recursiva.

```
/*
** Factorial de forma iterativa
*/
#include <stdio.h>

int factorial(int n);

int main()
{
  int numero, resultado;

  printf("Introducir numero \n");
  scanf(" %d",&numero);
  resultado = factorial(numero);
  printf("El factorial del %d es = %d", numero, resultado);

  return 0;
}


int factorial(int n)
{
  int facto = 1;

  while (n > 0)
    {
      facto = facto * n ;
      n--;
    }

  return facto;
}
```

```
/*** Factorial de forma recursiva  ***/

#include <stdio.h>

int factorial(int n);

int main()
{
  int numero, resultado;

  printf("Introducir numero: ");
  scanf(" %d", &numero);
  resultado = factorial(numero);
  printf("El factorial de %d es %d\n", numero, resultado);

  return 0;
}

int factorial(int n)
{
  int x;

  if (n == 0)
    /* Caso BASE */
    x = 1;
  else
    x = n * factorial(n - 1);

  return x;
}
```

9. Este ejemplo permite resolver el problema de las torres de Hanoi de forma recuriva:

   Hay 3 varillas, en la primera de ellas existen n discos concéntricos, cada uno de un tamaño diferente, y están ordenados de mayor (el de más abajo) a menor (el de más arriba; inicialmente las otras dos varillas están vacías. Consiste en mover todos los discos de una varilla a otra, sabiendo que un disco no puede situarse encima de otro si el tamaño del disco que hay debajo es menor que el tamaño del disco de arriba.

```c
#include <stdio.h>

void Torres(int n, int i, int j);

int main()
{
  int discos;
  enum {primera = 1, segunda, tercera};

  printf("Escribir el numero de discos en varilla origen: ");
  scanf("%d", &discos);
  Torres( discos, primera, tercera);

  return 0;
}

void Torres(int discos, int origen, int destino)
{
  int paso;

  if (discos == 1)
    printf("Mueve un disco de la varilla %d a la %d\n",
           origen, destino);
  else
   {
     /* Si las varillas van del 1 al 3, suman 6 */
     paso = 6 - origen - destino;
     Torres( discos - 1 , origen , paso);
     Torres(          1 , origen , destino);
     Torres( discos - 1 ,   paso , destino);
   }
}
```

   Un ejemplo de ejecución sería el que se muestra:

```
$> gcc -W -Wall -o hanoi hanoi.c
$> ./hanoi
Escribir el numero de discos en varilla origen: 4
Mueve un disco de la varilla 1 a la 2
Mueve un disco de la varilla 1 a la 3
Mueve un disco de la varilla 2 a la 3
Mueve un disco de la varilla 1 a la 2
Mueve un disco de la varilla 3 a la 1
Mueve un disco de la varilla 3 a la 2
Mueve un disco de la varilla 1 a la 2
Mueve un disco de la varilla 1 a la 3
Mueve un disco de la varilla 2 a la 3
Mueve un disco de la varilla 2 a la 1
Mueve un disco de la varilla 3 a la 1
Mueve un disco de la varilla 2 a la 3
Mueve un disco de la varilla 1 a la 2
Mueve un disco de la varilla 1 a la 3
Mueve un disco de la varilla 2 a la 3
$>
$
```

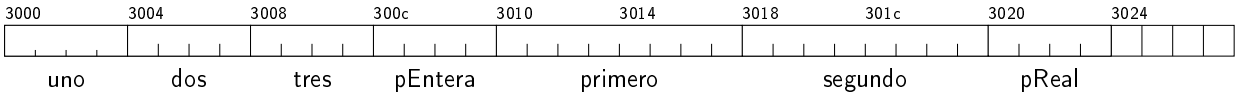
Ejercicios tema 8

Punteros

1. En una máquina con tipo **int** y tamaño de punteros de cuatro octetos, y **float** de ocho octetos, se realizan las siguientes declaraciones:

```
int     uno     = 1;
int     dos     = 2;
int     tres    = 3;
int   * pEntero = &uno;
float   primero = 1.1;
float   segundo = 2.2;
float * pReal   = &segundo;
```

con las que se tiene la siguiente distribución de variables en memoria:



Indique el valor de cada una de las variables tras cada una de las sentencias que siguen:

```
*(pEntero + 1) = 3
pEntero ++;
*pReal = uno;
dos = *pEntero + 1;
(*pReal--)++;
(*++pReal)++;
```

Ejercicios tema 9

Tipos Agregados

1. Indicar cómo quedaría inicializada la tabla **q** con la siguiente sentencia:

```
short q[4][3][2] = {
  {
        { 1 },
  },
  {
        { 2, 3 },
  },
  {
        { 4, 5 },
        { 6 },
  }
};
```

2. Ídem con:

```
short q[4][3][2] = {
  { 1 },
  { 2, 3 },
  { 4, 5, 6 }
};
```