Cuaderno de

Introducción a la Programación sobre UNIX

Francisco Rosales García

Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Facultad de Informática
Universidad Politécnica de Madrid

5 de febrero de 2008

Índice general

Programación Estructurada

Capítulo 1

¿Qué es programar?

En esta sesión se pretende que el alumno entienda qué sabe hacer un ordenador, y cómo el programador será capaz de instruirle para que haga lo que el desea.

1.1. ¿Cómo es un ordenador por dentro?

Los computadores de hoy día están construidos sobre la base de la arquitectura ideada por John Von Newmann (Fig.1.1).

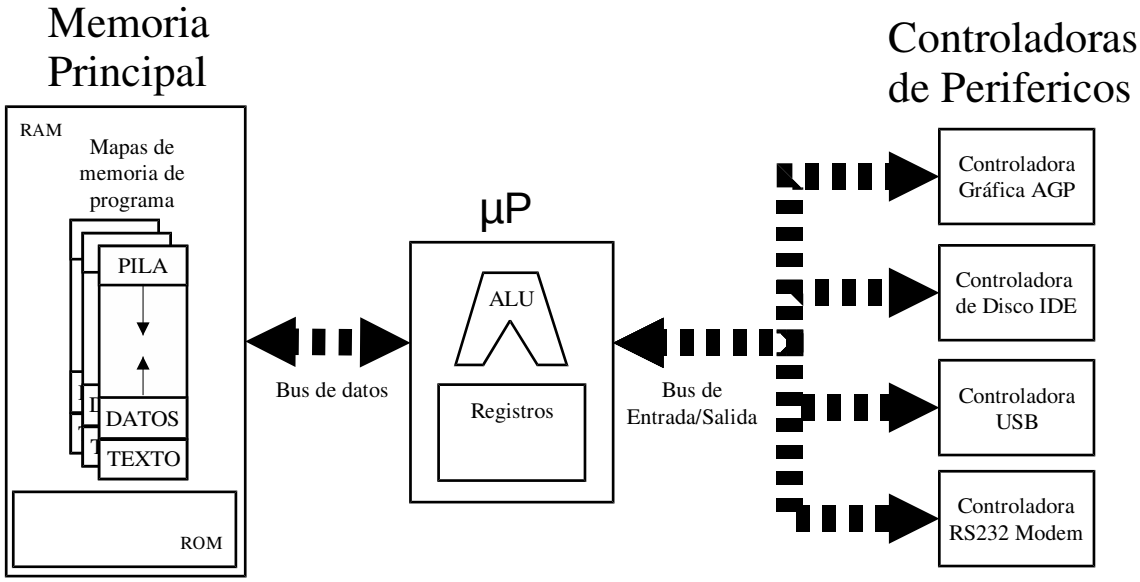


Figura 1.1: Arquitectura Von Newmann.

Alrededor de una unidad con capacidad de cálculo, denominada microprocesador, se organizan por un lado los dispositivos periféricos, y por otro una zona de almacenamiento de información, la memoria principal.

El microprocesador esconde dentro de sí una unidad capaz de realizar operaciones sobre un conjunto de registros. La naturaleza y orden de estas operaciones está dictada por las

instrucciones que están almacenadas en memoria. Existen instrucciones que gobiernan la transferencia de datos entre la memoria y los registros. También existen instrucciones destinadas a operar sobre los dispositivos conectados al denominado bus de entrada/salida.

1.2. ¿Qué sabe hacer un ordenador?

Muy poco. Muy rápido.

- Ejecuta instrucciones paso a paso.
  - Existe un único flujo o hilo de ejecución que avanzará linealmente si no le indicamos otra cosa.
- Sólo entiende instrucciones muy muy básicas.
  - Aritméticas (suma, resta, multiplicación, división).
  - Lógicas (igual, mayor, menor).
  - Salto (condicional, incondicional, llamada a subrutina).
- Las ejecuta muy muy rápido, una increíble velocidad.
  - Actualmente más de 100 MIPS y subiendo.
  - Cuenta cada habitante de la tierra en 1 minuto!
- Sin pausa.
  - 8.640.000.000.000 al día.
  - 3.155.760.000.000.000 al año.

1.3. ¿Para qué se usan los ordenadores?

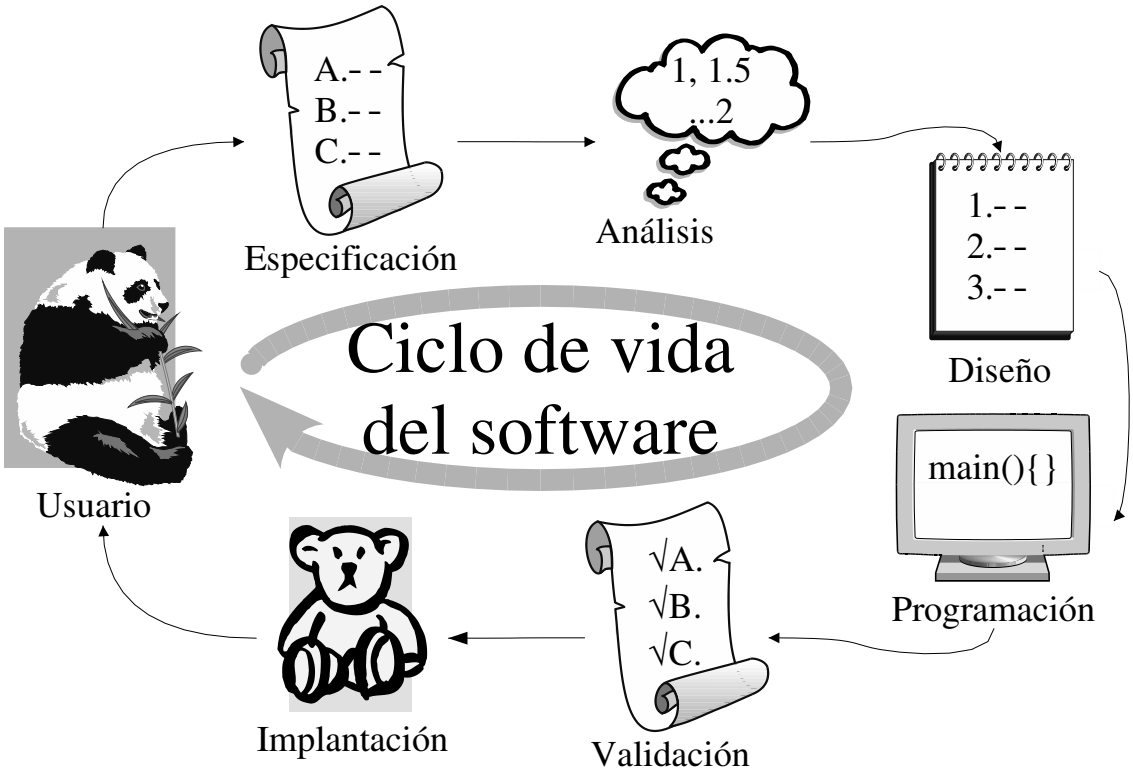
El ordenador sólo hará lo que alguien le haya dicho cómo hacer.

- Ejecutan programas/aplicaciones que son útiles para alguien.
- Programas que resuelven cómoda y rápidamente problemas/tareas que costaría mucho resolver a mano.
- Sólo aquellos problemas para las que existe solución "a mano".

1.4. Ciclo de vida del software

El desarrollo de software implica numerosas etapas que pueden solaparse en el tiempo, subdividirse a su vez, y que en numerosos casos implicarán la necesidad de volver a una etapa anterior (Fig.1.4).

Especificación Se le plantea realizar un determinado programa o aplicación. Se delimita con precisión qué se pide y qué no, para ello se habrá de consultar al usuario final.



Análisis Se averigua cómo resolver el problema, dividiéndolo en subproblemas más delimitados y fáciles de abordar.

Diseño Se esboza cada parte de la aplicación (módulo) y su interconexión.

Programación Esta etapa es la correspondiente al programador.

Integración Se supervisa la calidad de cada módulo y su integración con el resto.

Validación Se comprueba el correcto funcionamiento de la aplicación y el cumplimiento de las especificaciones originales.

Implantación Se supervisa la puesta en marcha y el correcto funcionamiento del programa en su entorno final y la aceptación por parte del usuario final.

1.5. Ciclo de programación

También el trabajo del programador será cíclico. En muchas ocasiones deberá volver a etapas anteriores, al corregir errores o para modificar o añadir funcionalidades.

Codifica Escribe código que implementa con precisión cada módulo.

Comenta Comenta suficientemente el código fuente para ayudar a su lectura y comprensión.

Compila Traduce este código a instrucciones que el ordenador sabe realizar. Para esto hace uso del compilador.

Verifica Verifica su funcionamiento correcto ante todo tipo de casos posibles. Para esto crea programas de prueba que someten a cada módulo a todos los casos de prueba posibles y hace uso de herramientas de cobertura de código.

Depura Localiza y corrige los errores que encuentre. Para esto hace uso del depurador.

Documenta Ayuda a generar la documentación que el usuario final necesitará para utilizar correctamente el programa.

Esta labor está normalmente infravalorada, pero es vital. Por muy bueno que sea un programa o aplicación, no valdrá de nada si el usuario no sabe como usarlo.

1.6. ¿Qué conocimientos tiene un programador?

El programador es también usuario "aventajado"del sistema sobre el que desarrolla.

- Conoce y utiliza el sistema. Sabe cómo sacarle el mejor partido.
  - Lee manuales incesantemente.
- Conoce y utiliza muchas herramientas que le ayudan en su tarea.
  - Editores, compiladores-intérpretes, depuradores, bibliotecas.
  - Consulta la ayuda, lee los manuales.
- Conoce y utiliza varias técnicas de programación y lenguajes.
  - Programación estructurada, C.
  - Programación orientación a objetos, C++.

1.7. Resumen

- El ordenador es un tonto muy rápido, preciso e infatigable.
- El programador es inteligente y sabe como instruirle de forma precisa. Su labor es creativa y gratificante, pero exige muchos conocimientos.
- El usuario es... impredecible y exigente.

Capítulo 2

Programación estructurada

En un principio, la labor de desarrollo de software fue artesana pero a medida que los proyectos se hicieron más grandes, ambiciosos y multitudinarios, se observaron problemas de baja productividad, muy difícil mantenimiento, redundancia de código, etc., etc. A esto se le denominó "la crisis del software".

Como respuesta se crearon **metodologías de diseño** y **técnicas de programación** que permiten abordar el desarrollo de software de una forma más ordenada y sistemática y obtener un software de mayor calidad.

Software de calidad

Puede ser definido como aquél que es:

Correcto. Hace exactamente lo que dice, con precisión.

Eficiente. Si existen varias formas de hacer algo, se escogerá la menos costosa.

Reusable. Evita reinventar la rueda. Utiliza las librerías existentes. Programa con generalidad para poder reaprovechar el código.

Transportable. Evita que su código dependa de una determinada arquitectura, sistema, etc.

Estándar. Hace que su programa se comporte según la gente espera, evitando originalidades innecesarias.

Robusto. Debe tolerar en lo posible los errores que cometan los usuarios u otros programas, y no errar él.

Legible. El código debe estar escrito para facilitar su comprensión a otros programadores.

Mantenible. El software está destinado a evolucionar, a ser corregido y mejorado. Su diseño y estructura deben facilitar su mantenimiento.

2.1. Conceptos

$$\mathsf{Algoritmo} + \mathsf{Datos} = \mathsf{Programa}$$

Algoritmo

Es un método para resolver un problema sin ambigüedades y en un número finito de pasos.

El algoritmo ha de ser la solución general a todos los problemas del mismo tipo. Deberán considerarse todas las posibles situaciones que se puedan dar.

Es posible que existan varios algoritmos para un mismo problema. Habrá que saber escoger el más adecuado, eficiente, elegante, etc.

Valga el siguiente ejemplo:

```
Algoritmo de la burbuja:
        Ordena N elementos (A,B,C,D,...) si y sólo si
        para cualesquiera dos elementos X e Y existe:
        * un criterio numérico de distancia(X,Y)
        * un mecanismo para intercambiar
          (...X...Y...) en (...Y...X...)

Ordenar 2 elementos X y Y es:
        Si distancia(X,Y) es negativa
        entonces intercambiar X y Y.

Ordenar N elementos es:
        Ordenar cada elemento X respecto a cada uno de los
        demás elementos Y.
```

Datos

Es la información que el programa recibe, maneja o devuelve.

Un programa autista, que no tomara ni produjese datos sería de nula utilidad.

Los datos se manejarán agrupando los conjuntos de información relacionada en estructuras de datos que faciliten su uso.

Abordaremos las "estructuras de datos" en una sección aparte pero por ahora valga como ejemplo:

```
        altura es: un número real
        TA es: una tabla de alturas
        NTA es: el número de elementos en TA
        TA[i]: es el elemento número i de TA
```

Programa

Es el resultado de expresar un algoritmo en un lenguaje, como un conjunto de instrucciones cuya secuencia de ejecución aplicada a los datos de entrada, resolverá el problema planteado produciendo el resultado deseado.

El programa incluye la descripción de las estructuras de datos que se manipularán, y las instrucciones del lenguaje que manipulan dichos datos.

En el caso del ejemplo, el programa implementaría el algoritmo de la burbuja, aplicado a la ordenación de las NTA alturas contenidas en una tabla de nombre TA. Además definiría un criterio de comparación de alturas y un mecanismo de intercambio de elementos de la tabla.

2.2. Técnicas de programación

Vamos a ver unas cuantas técnicas cuya aplicación nos facilitará abordar el desarrollo de programas de cierta entidad.

Términos abstractos

La solución natural, no informática, de un problema es independiente de la herramienta a emplear para su ejecución.

Así pues, deberemos concebir la lógica del programa en los términos o vocabulario naturales del problema, olvidando hasta el último momento el hecho de que finalmente se usará un ordenador para resolverlo.

En una primera instancia es primordial identificar cuáles son estos términos y qué relaciones se establecen entre ellos.

☑ *Enumere y describa brevemente al menos 10 términos en los que expresaría a un amigo cómo se juega al mus.* T 2.1

☑ *Imagine un programa capaz de simular colisiones de turismos contra un muro, destinado a estudiar la seguridad de los ocupantes.* T 2.2

Enumere al menos 20 términos del vocabulario natural de este problema y organicelos en un gráfico que exprese su interrelación.

? *¿Cree que deberían aparecer en el gráfico términos relativos a la representación gráfica de las colisiones?* T 2.3

Con esta técnica tan elemental estamos identificando los datos que deberá manejar nuestro programa. Además, esta forma de razonar es fundamental en programación orientada a objetos.

Razonamiento descendente (*Top-Down*)

Muchas veces, la naturaleza o magnitud del problema no nos permite visualizarlo en toda su extensión. En estos casos es muy posible que sí seamos capaces de identificar etapas o fases en el mismo.

Se trata pues, de ir subdividiendo el problema original en subproblemas de menor tamaño, yendo de lo general a lo específico, siempre, pensando en *términos abstractos*. A medida que avancemos en la descomposición del problema original iremos refinando una solución.

El objetivo es ir evolucionando hacia una descomposición con un grado de detalle que permita su expresión con las estructuras básicas de la programación estructurada (que veremos más adelante).

☑ *Descomponga el problema "**desarrollo de una partida de mus**", en una jerarquía que contenga todas las fases del juego.* T 2.4

T 2.5 ☑ *Descomponga jerárquicamente el enunciado "**proyecto y construcción de 150 viviendas VPO**" en sus diferentes fases.*

El razonamiento *top-down* es vital para abordar problemas de cierta entidad. Con él estamos descubriendo lo que será el "hilo argumental" de nuestro programa.

Modularización

A medida que se perfilan los componentes fundamentales de un programa, deberemos pensar en si son suficientemente "genéricos" y/o están suficientemente "delimitados" como para merecer ser tratados como una pieza independiente, posiblemente reutilizable. A esta pieza le denominaremos módulo.

Un módulo estará definido por el conjunto de las funcionalidades que ofrece (**interfaz**), y a su vez estarán concebidos sobre las interfases de otros más primitivos, formando finalmente una jerarquía.

El concepto de módulo es indispensable para el desarrollo de medianos y grandes proyectos ya que podrá ser: analizado, codificado y verificado de forma independiente.

Cada funcionalidad de cada módulo deberá ser programada estructuradamente.

2.3. Estructuras de programación

La programación estructurada define una serie limitada de estructuras básicas de programación a utilizar y unas reglas de estilo de codificación. Esto minimiza la probabilidad de error humano y hace que los programas sean más fiables, eficientes y adaptables a otros lenguajes o sistemas.

Secuencia

Sucesivos pasos o acciones que se ejecutarán en estricto orden.

En la figura 2.1 se muestra una representación gráfica de este tipo de estructura, denominada **ordinograma**. Haremos uso de este tipo de representación cuando queramos realizar una representación gráfica de nuestro algoritmo antes de programarlo.

A continuación se muestran dos ejemplos equivalentes, escritos en **pseudocódigo** y en lenguaje C. Usaremos pseudocódigo para expresar textualmente el algoritmo o programa que tenemos en mente antes de programarlo.

Secuencia de pseudocódigo

```
# Esto es un comentario.
# Los comentarios no ejecutan.
PRINT "Introduzca un número "
INPUT valor
PRINT "Introdujo el ", valor, NL
```

Secuencia de código C

```
/* Esto es un comentario. */
/* Los comentarios no ejecutan. */
printf("Introduzca un número >= 0 ");
scanf("%u", &valor);
printf("Introdujo el %u\n", valor);
```

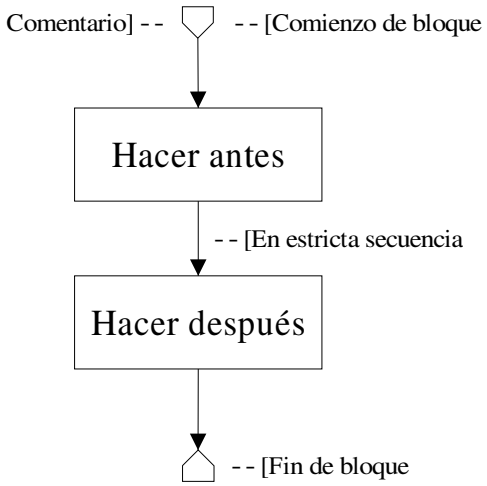


Figura 2.1: Estructura en secuencia.

El pseudocódigo puede ser tan libre como queramos, siempre y cuando seamos sistemáticos en su uso. Por el contrario, como se puede observar en la figura, los lenguajes de programación (en este caso C) precisan que nos ajustemos a una sintaxis muy concreta.

Selección

Permiten dirigir el flujo de ejecución a una de entre varias alternativas en función de cierta condición o condiciones establecidas sobre los datos.

Cada condición se evalúa como una expresión lógica, esto es, finalmente tomará un valor CIERTO o un valor FALSO.

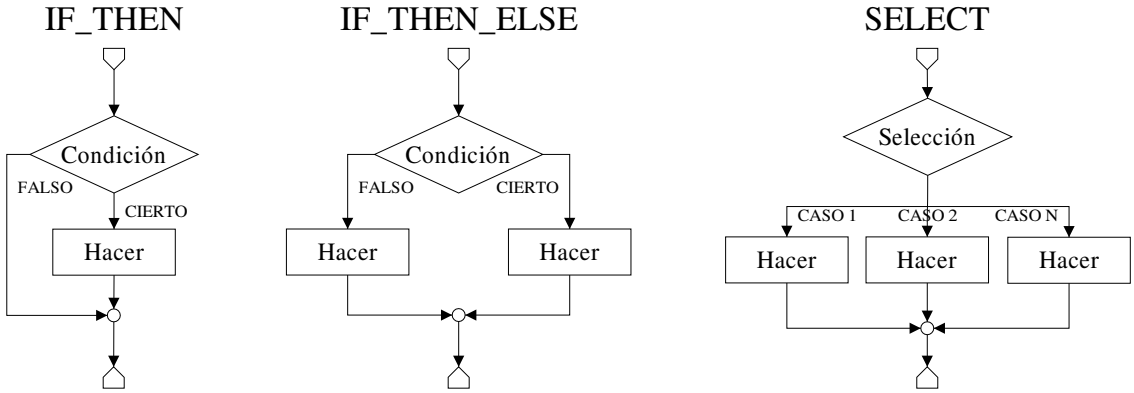


Figura 2.2: Estructuras de selección.

La figura 2.2 muestra el ordinograma de los diferentes tipos de selección. A continuación se muestran unos ejemplos de cada caso en pseudocódigo y en C.

IF_THEN

```
# Acción condicional
IF (num es distinto de 0)
    duplicar num
END_IF
```

`if` en C

```
if (num != 0)
{
    num = num * 2;
}
```

IF_THEN_ELSE

```
# Acciones alternativas
IF (num es igual a 0)
    incrementar num
ELSE
    decrementar num
END_IF
```

`if else` en C

```
if (num == 0)
{
    num = num + 1;
} else {
    num = num - 1;
}
```

SELECT

```
# Selección múltiple
# dependiendo del valor de expr.
SELECT (expr)
CASE -1
    Elevar num al cuadrado
CASE 0
    # no hacer nada
ELSE
    dividir num por 2
END_SELECT
```

`switch` en C

```
switch(num + 5)
{
case -1:
    num = num * num;
    break;
case 0:
    break;
default:
    num = num / 2;
}
```

Iteración

Son las estructuras de programación denominadas *bucles*, que permiten ejecutar ninguna, una o varias veces el conjunto de acciones indicadas en el cuerpo del bucle. La iteración del bucle estará controlada por una determinada condición establecida sobre los datos. Esta condición tiene que poder cambiar de estado (de CIERTO a FALSO o viceversa) en el cuerpo del bucle para que este pueda terminar. De otro modo, se trataría de un bucle infinito (o que nunca sucediera), lo cual sería sintomático (las más de las veces) de un error en nuestro programa.

La figura 2.3 presenta el ordinograma de los tres bucles más clásicos. A continuación se muestran unos ejemplos de cada caso en pseudocódigo y en C.

El cuerpo del bucle WHILE se ejecuta mientras la condición se evalúa a cierto, luego puede no ser ejecutado ni una sola vez.

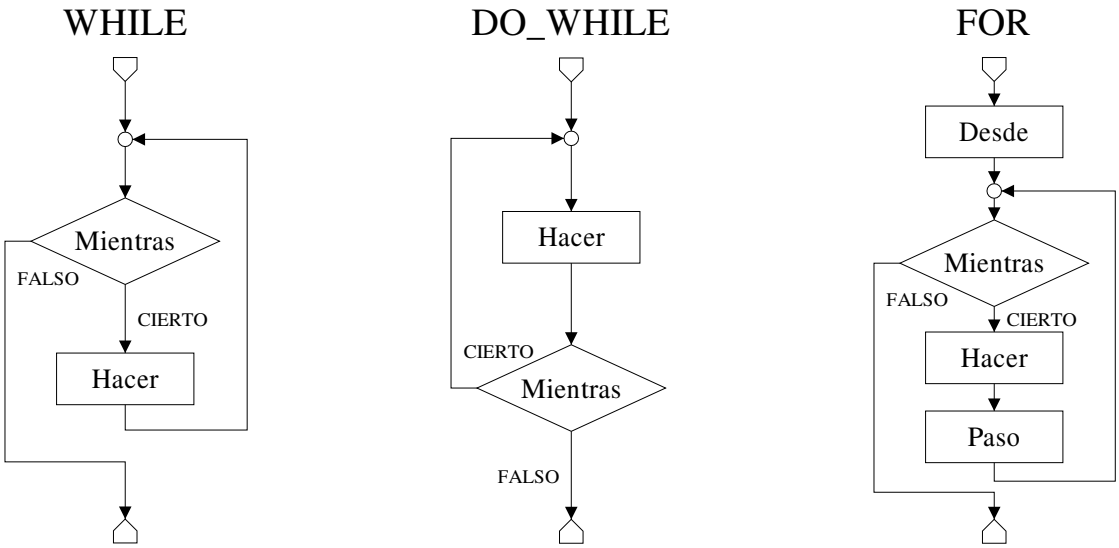


Figura 2.3: Estructuras de iteración.

WHILE

```
WHILE (num sea negativo)
    # Si num es mayor o igual
    # que 0 no se ejecutará.
    incrementar num
DO
```

`while` en C

```
while (num < 0)
{
    num = num + 1;
}
```

El cuerpo del bucle DO_WHILE es ejecutado al menos una vez y se iterará cuando la condición se evalúe a CIERTO.

DO_WHILE

```
DO
    decrementar num
WHILE (num mayor que 0)
```

`do while` en C

```
do
{
    num = num - 1;
} while (num > 0);
```

Si para iterar la condición debiera resolverse a FALSO, estaremos hablando del bucle REPEAT_UNTIL.

REPEAT_UNTIL

```
REPEAT
    decrementar num
UNTIL (num menor o igual a 0)
```

En C no existe este bucle pero siempre se puede realizar negando la condición:
`do {...}while(!(cond));`

Como se puede observar en la figura 2.3, el bucle FOR es básicamente de un bucle WHILE al que se la añade una etapa de inicialización de la variable de control del bucle y otra etapa de incremento (o decremento) de esta variable de control.

FOR

```
# Iterar para los valores
# indicados de la variable
FOR num=1 hasta MAX de 1 en 1
    sumatorio de num
DO
```

`for` en C

```
for (num = 1; num <= MAX; num++)
{
    sum = sum + num;
}
```

Función-Procedimiento

Los lenguajes de programación soportan un concepto semejante al de función matemática: aquella operación o método que, para unos argumentos de invocación fijos, devuelve a su salida un único valor posible.

$$f(x) = ax^2 + bx + c$$

Esta definición nos habla de conceptos importantes como: invocación, argumentos y valor devuelto. Consideraremos siempre un único punto de entrada a la función alcanzado al invocarla, y un único punto de salida allá donde se devuelva el valor resultante.

En programación estructurada deberíamos ceñirnos lo más posible a esta definición de función, pero la realidad nos llevará a utilizar el término función para referirnos al más genérico, funcionalidad, que en muchos casos devolverá más de un valor, modificando los argumentos con que la invocamos. Llamaremos procedimiento a la función tal que no devuelve valor, si bien modificará los argumentos con que se le invoca.

Usaremos funciones para agrupar bajo el nombre de la misma, cada bien delimitada unidad de operación, esto es, cada funcionalidad o método concreto de cada módulo de nuestro programa.

Función principal

De hecho, desde el punto de vista de su ejecución, los programas no serán más que una enorme colección de funciones que se llaman unas a otras, siendo una concreta la primera en ser invocada. Esta es la función principal o punto de entrada al programa, en C es siempre la función `main`.

minimo.c

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

Este es el programa C "correcto" más pequeño que se puede escribir. Una función principal que termina devolviendo un 0.

Argumentos, parámetros y variables locales

A continuación se presenta la definición matemática de la función $factorial(n)$, siendo n un número natural (no negativo). Así mismo se muestra una implementación en C la misma y de una función principal que la utiliza.

$$factorial(n) = n! = \begin{cases} 1 & \text{sii } n \text{ es } 0 \\ \prod_{i=1}^{n} i & \text{sii } n > 0 \end{cases}$$

factorial.c

```
unsigned factorial(unsigned n)
{
    unsigned value = 1;
    /* Recorre de n a 2 */
    for (; n > 1; n--)
        value = value * n;
    return value;
}
```

main.c

```
int main(void)
{
    printf("3! = %d\n", factorial(3));
    printf("7! = %d\n", factorial(7));
    return 0;
}
```

Observe que no realizamos una iteración de `1` *a* `n`*, sino de* `n` *a* `2`*. Dado que el resultado que finalmente se obtiene será idéntico al de la definición matemática, este tipo de mejoras es lícito, pero exigirían cierta explicación.*

En este ejemplo se aprecian los detalles importantes:

`unsigned n` Las funciones declaran que reciben ciertos parámetros (llamados parámetros formales), que determinarán su comportamiento ya que toman el valor indicado en la invocación a la función.

`unsigned value = 1;` Las funciones pueden declarar variables locales que sirvan de soporte a la implementación del algoritmo. Estas variables deberán ser convenientemente inicializadas en cada invocación antes de ser usadas ya que de otra manera su valor será indeterminado (basura).

`factorial(3)` Cuando una función sea invocada, los parámetros formales declarados tomarán el valor indicado en dicha invocación.

Así, en este caso el parámetro `n` tomará el valor 3 durante la primera invocación y durante la segunda tomará el valor 7 (`factorial(7)`).

Cada invocación a una función o procedimiento es independiente de las demás, en el sentido de que tanto los parámetros formales, como las variables locales son "otras distintas" en cada invocación. No se guarda su "estado" de una invocación a otra.

☑ *Programe en pseudocódigo o en C, la función principal de un programa, que muestre el factorial de los números del 0 al 20 inclusives. Use un bucle.* T 2.6

☑ *Programe la función binomial(m, n) definida sobre números m y n naturales como:* T 2.7

$$binomial(m, n) = \binom{m}{n} = \frac{m!}{n!\ (m-n)!}$$

☑ *Programe el siguiente desarrollo en serie de Taylor que aproxima la función* e^x, *definida para valores x reales (*`float` *en C).* T 2.8

$$e^x = \sum_{i=1}^{10} \frac{x^i}{i!}$$

Recursividad

Es la facultad de las funciones (o procedimientos) de invocarse a sí mismas, directa, o indirectamente a través de otras.

Es una forma elegante y natural de expresar la solución a ciertos problemas que son "autocontenidos". El ejemplo típico es la función matemática $factorial(n)$ que ya hemos visto, pero que también puede ser definida recursivamente de la siguiente manera.

$$factorial(n) = n! = \begin{cases} 1 & \text{sii } n \text{ es } 0 \\ n * (n-1)! & \text{sii } n > 0 \end{cases}$$

factorial.c

```
unsigned factorial(unsigned n)
{
    if (n == 0)
        n = 1;
    else
        n = n * factorial(n - 1);
    return n;
}
```

main.c

```
int main(void)
{
    unsigned num = 5;
    unsigned res;
    res = factorial(num);
    printf("%d! = %d\n", num, res);
    return 0;
}
```

Observe que en este caso la implementación no contiene ningún bucle iterativo. No obstante esta implementación es perfectamente equivalente a la anterior, aunque es sin duda más difícil de comprender.

En este ejemplo se debe apreciar que:

`n = 1;` En la implementación realizada se ha decidido prescindir de usar una variable auxiliar y se ha reutilizado el parámetro `n` cambiando su valor como si de una variable local se tratara.

`factorial(num)` En el ejemplo, la variable `num`, local al programa principal, vale 5 en el momento de invocar la función `factorial`.

El paso de parámetros es por valor. A la función se le pasa el valor de la variable, no la variable en sí. Esto será siempre así en C.

Dado que los parámetros y variables locales son exclusivas de cada invocación, **modificar su valor no implica modificación del argumento con que se invocó**.

`printf ... num` La variable `num` no cambia de valor aunque lo haga el parámetro `n` al que da valor. Seguirá valiendo 5 cuando el programa principal la imprima.

Una función recursiva debe controlar con precisión su condición de terminación. Si no lo hiciese así, la recursión sería infinita y esto provocaría irremisiblemente la terminación anómala del programa.

El uso extensivo de la recursividad se desaconseja, dado que hacen que el código sea más difícil de entender. Es más, toda solución recursiva siempre puede ser transformada en una iterativa.

2.4. Estilo de codificación

Las reglas de estilo tienen por objeto mejorar la legibilidad del código y hacerlo más comprensible e independiente del autor. Su aplicación es importantísima. Como veremos a continuación, su uso no implica cambio alguno a la estructura de nuestros programas no obstante, marcan la diferencia entre un buen y un mal programador.

Nombrado

Es muy importante que en la fase de implementación se mantengan explícitos los términos abstractos que se hayan venido manejando en la descomposición del problema, para que esta quede nítidamente reflejada en el código.

Los nombres de variable y/o función que usemos, deberán ser suficientemente claros y específicos. Evitaremos usar nombres que cueste identificar, o que sean ambiguos.

MAL

```
void bar(m R)
{
    int c = R.c;
    int nc = 10;
    int pc = rand() % c;
    ...
}
```

BIEN

```
void barajar(mazo_naipes restantes)
{
    int cuantos = restantes.cantidad;
    int num_cortes = 10;
    int primer_corte = rand() % cuantos;
    ...
}
```

Endentación

Es igualmente importante que el "dibujo" de las líneas de nuestro código refleje la estructura del mismo, para que su visualización ayude a su comprensión.

Esto lo conseguiremos haciendo un buen uso de los "espacios en blanco", tabulando correctamente las líneas indicando su "profundidad" y situando los delimitadores de bloque ({ y }) de manera uniforme en el código que escribimos.

MAL

```
if (n == 0)
{
    n = 1;
    if (n == 1) {
        n = 2;
        }
else {
            n = 3;
}n = n + 1;}
```

BIEN

```
if (n == 0)
{
    n = 1;
    if (n == 1)
    {
        n = 2;
    }
    else
    {
        n = 3;
    }
    n = n + 1;
}
```

Comentarios

Independientemente de los dos puntos anteriores, el hecho de "codificar la idea" que tenemos en mente implica necesariamente una pérdida de información. El programador sabe qué quiere conseguir, pero al codificar indica estrictamente cómo conseguirlo.

Continuamente al programar, tras un razonamiento no trivial, tomamos decisiones sobre cómo y porqué codificar las cosas. Aunque el código fuese perfecto en su cometido, nombrado y estilo, alguien que lo leyera tendría que hacer un gran esfuerzo para entender porqué el autor tomó las decisiones que tomó.

Un buen programador debe comentar convenientemente el código que escribe, para "iluminar" a sus lectores sobre las razones que determinaron las decisiones tomadas. Si se hizo un buen nombrado se podrán evitar muchos comentarios inútiles.

MAL

```
/* Resto 1 */
H = H - 1;
...
/* Elevo al cuadrado */
r = r * r;
/* Multiplico por PI */
s = r * 3.14159;
```

BIEN

```
desfase_horario = -1;
hora = hora + desfase_horario;
...
const float PI = 3.141592654
/* Es una circunferencia */
superf = r * r * PI;
```

Restricciones

En muchos lenguajes de programación "estructurados" existen cláusulas que permiten "romper" la estructura del programa, ya que implican saltos bruscos del hilo de ejecución. Su abuso puede tentar, pero hemos de evitarlo a toda costa, salvo en ciertos casos donde por el contrario resulta procedente.

Citamos a continuación las que aparecen en el lenguaje C, indicando las situaciones en que se pueden usar.

`goto` Salta directamente a otro punto del código. No se deberá usar **jamás**.

`break` Sale bruscamente fuera del bucle o `switch` más interno. Su único uso lícito es en las sentencias `switch` de C.

MAL

```
do {
    ...
    ...
    if (salir)
        break;
    ...
    ...
} while (1); /* Siempre */
```

BIEN

```
switch (respuesta) {
case 'y':
case 'Y':
    salir = 1;
    break;
default:
    salir = 0;
}
```

`continue` Salta bruscamente a la siguiente iteración del bucle más interno. Su único uso lícito es como cuerpo de bucles que de otra manera deberían estar vacíos.

MAL

```
while (time() < limite);
```

BIEN

```
while (time() < limite)
    continue;
```

`return` Sale bruscamente de una función devolviendo un valor (o de un procedimiento sin devolver nada). Sólo debe existir un único `return` como última sentencia de cada función de nuestro programa. El final de cada procedimiento hay un `return` implícito.

MAL

```
int F(...
    if (...
        if (...
        for (...
            return 0;
        } else {
        switch(...
        default: return 2;
        }
    }
    return -1;
}
```

BIEN

```
int F(...
    int ret = -1;
    if (...
        if (...
        for (...
            ret = 0;
        } else {
        switch(...
        default: ret = 2;
        }
    }
    return ret;
}
```

`exit` Da por terminado el programa. Solo deberíamos usar un `exit` por programa, y debería aparecer como última sentencia de la función principal del mismo.

Capítulo 3

Tipos y estructuras de datos

Todo programa utiliza variables y constantes para representar cantidades y/o cualidades del problema que implementa. Antes de poder usar una variable en nuestro programa deberemos haberla declarado, para que el compilador (o intérprete) del lenguaje sepa a qué atenerse. La declaración de una variable le indica al compilador el nombre con el cual nos referiremos a ella, su ámbito de vida y visibilidad y el tipo de datos asociado a la misma.

Nombrado

Puede que el lenguaje restrinja el nombrado de las variables que, por otro lado y como ya hemos visto, habrá de ser suficientemente explícito y reflejar los términos abstractos del problema. Concretamente, los compiladores no permitirán declarar variables cuyo nombre coincida con alguna de las palabras reservadas del lenguaje.

Ámbito de vida

El ámbito de vida o de una variable determina durante cuánto tiempo existe. Una variable declarada como global a un módulo o al programa completo existirá durante todo el tiempo de ejecución del mismo. Una variable local a (o un parámetro formal de) una función existe sólo durante la ejecución de dicha función.

Visibilidad

La visibilidad de una variable determina desde qué puntos de nuestro código podemos hacer referencia a ella.

Las variables globales al programa pueden ser referidas desde cualquier sitio. Las variables globales a un módulo del programa pueden ser referidas sólo desde dicho módulo.

Las variables locales a (o parámetros formales de) una función sólo pueden ser referidas desde el código de dicha función.

Si declaramos una variable local con el mismo nombre que una global, la global quedará oculta durante la ejecución de dicha función.

Tipo

El programador escogerá el tipo de datos de la variable en función de la naturaleza del concepto que representa y del conjunto de posibles valores que queremos que la variable pueda tomar. El tipo de una variable podrá ser uno de los tipos básicos que el lenguaje conozca de antemano, o bien de un tipo derivado de estos.

Los tipos numéricos básicos determinan posibles valores dentro de un rango. Por ejemplo, una variable entera no podrá contener (y por lo tanto no podrá valer) el valor decimal 3.5. Una variable declarada como siempre positiva, no podrá contener un valor negativo.

Valor y formato

Es preciso que quede clara la diferencia entre el valor que una variable tiene (o contiene) y el formato o manera en que dicho valor puede ser representado.

Recuerde siempre.

> Un Kilo de oro tiene un determinado valor, independiente de su formato: lingote, polvo,...

Durante la explicación que sigue veremos ejemplos de un mismo valor numérico representado en diferentes formatos (bases de representación).

Tipos básicos

En este apartado presentaremos brevemente los tipos básicos de datos para poder comprender mejor las denominadas estructuras dinámicas de datos que veremos a continuación.

Como parte de los ejercicios prácticos que realizaremos más avanzado el curso, se manejarán estos tipos de datos y las estructuras dinámicas de datos, por lo cual aquí no abundaremos mucho en el tema.

Recuerde que el tipo de una variable o contante determina la naturaleza y cantidad de valores distintos que puede tomar.

Normalmente, para cada tipo básico, el lenguaje ofrece una forma de expresar contantes de dicho tipo.

Sin tipo

Si queremos declarar una función que no devuelve nada, esto es, un procedimiento, usaremos el pseudotipo `void`.

Por lo demás, no pueden declararse variables de este pseudotipo.

BIEN

```
void esperar(unsigned segundos);
```

MAL

```
void variable_nula;
```

Lógico

En muchos lenguajes existe un tipo de datos lógico (*boolean*) capaz de representar los valores lógicos CIERTO o FALSO. El resultado de las operaciones lógicas (comparación)

y de su combinación mediante operadores lógicos (AND, OR, NOT, etc.) es un valor de tipo lógico.

En C y en C++ no existe un tipo de datos lógico, sino que cualquier expresión puede ser evaluada como una expresión lógica. El valor numérico cero equivale al valor lógico FALSO, y como negación, cualquier valor numérico distinto de cero es equivalente a un valor lógico CIERTO.

Carácter

Representan caracteres ASCII o ASCII extendido (Latin1).

- ASCII: `'\0', 'a', 'B', '?', '\n'`
- Latin1: `'á', 'Ü'`

```
char letra;
```

Entero

Son números positivos o negativos o sólo positivos, de naturaleza entera, esto es, que no pueden dividirse en trozos (sin decimales o con parte decimal implícitamente 0). Sirven para numerar cosas o contar cantidades.

Los valores enteros pueden ser representados en diferentes bases o formatos.

- En decimal (base 10): `0, -1, 4095, 65184`
- En octal (base 8): `00, -01, 07777, 0177240`
- En hexadecimal (base 16): `0x0, -0x1, 0xFFF, 0xFea0`

```
int saldo_bancario; /* Nºs negros o rojos */
```

```
unsigned numero_de_dedos; /* Jugando a chinos */
```

Real

Son números positivos o negativos con o sin decimales. Son los denominados "coma flotante".

- Notación punto: `3.14159265358979323846`
- Notación Exponente: `42E-11` que vale $42 * 10^{-11}$

```
float distancia, angulo;
```

```
double PI;
```

Enumerado

Para variables que pueden tomar cualquiera de los valores simbólicos que se enumeran. Internamente se representarán como un valor entero.

```
enum {
        pulgar,
        indice,
        corazon,
        anular,
        menique
} dedos;
enum {varon, hembra} sexo;
dedo = anular;
sexo = hembra;
```

Puntero

Son variables que, en vez de contener un dato, contienen la dirección de otra variable del tipo indicado.

Se declaran con el carácter `*`. Con el operador `&` se puede obtener la dirección de una variable. Para *de-referenciar* un puntero y así acceder al valor de la variable a la que apunta se utiliza el operador `*`.

```
int variable = 7;
int * puntero_a_entero;
puntero_a_entero = & variable;
*puntero_a_entero = 8;
/* Ahora variable vale 8 */
```

Como veremos dentro de poco, las variables de tipo puntero son la base para la creación de estructuras dinámicas de datos.

La dirección de valor 0 (cero) es tratada como una marca especial. El símbolo NULL asignado a una variable de tipo puntero, le asigna un valor cero, que como dirección es inválida (no puede ser de-referenciada), pero que nos sirve para conocer que un puntero "no apunta" de momento a ningún sitio.

Agrupaciones de datos

Vectores

Son agrupaciones de información homogénea. Se declaran indicando el tipo de cada componente, su nombre y, entre corchetes (`[]`) su dimensión o tamaño. Pueden ser de una dimensión o de varias, en cuyo caso hablamos de matrices.

Los contenidos de un vector o matriz se almacenan de forma contigua en memoria. Sus contenidos son accedidos indexando (con `[`*indice*`]`) con otro valor entero como índice.

En C la indexación comienza por cero.

```
char buffer[80];
float tabla[100];
unsigned char screen[1024][768];
unsigned char icono[16][16];
buffer[0] = 'a';
tabla[100] = 1.0; /* FUERA */
```

Las tiras de caracteres son también vectores. Una tira de caracteres constante se expresa entre comillas dobles. Su último carácter es implícitamente un carácter nulo (`'\0'`).

```
char Hello[12] = "Hola Mundo!";
Hello[0] = 'M';
Hello[6] = 'a';
Hello[11] == '\0'; /* Fin */
```

Estructuras

Agrupan en una única entidad un conjunto heterogéneo de campos relacionados que conforman una información concreta.

Los campos se acceden con la notación *.nombre_de_campo*.

En C se pueden definir nuevos tipos de datos con la sentencia `typedef`.

alumno.h

```
typedef struct Alumno {
    char * Nombre, * Apellido;
    struct Fecha {
        int Anyo, Mes, Dia;
    } Nacimiento;
    float Edad;
} Alumno_t;
```

alumno.c

```
#include "alumno.h"
Alumno_t yo;
Alumno_t Clase[22], *actual;
yo.Edad = 33;
actual = &Clase[0];
*actual.Nombre = "Ramón";
actual->Apellido = "Ramírez";
```

Estructuras dinámicas de datos

Cuando nuestros programas necesitan manejar agrupaciones homogéneas de datos cuya cantidad desconocemos de antemano, organizaremos estos como una estructura dinámica de datos.

Ejemplo: Imagine un programa para ordenar alfabeticamente las palabras contenidas en un fichero. No sabemos a priori el número de palabras que puede haber: 100, 1000, cienmil o un millón. El programa debería ser capaz de adaptarse en cada caso, y ocupar tan sólo la memoria necesaria.

Una estructura dinámica de datos es una estructura de datos (algo capaz de contener datos) que puede crecer y encoger en tiempo de ejecución. Para ello, se construyen mediante otras estructuras de datos que incorporan campos "puntero" para interconectarse entre sí. El valor nulo de estos punteros servirá para marcar los *extremos* de la estructura.

Las estructuras dinámicas, como su nombre indica, se ubican dinámicamente en un espacio de memoria especial denominado, así mismo, memoria dinámica. Esta memoria se gestiona mediante primitivas básicas, cuyo nombre cambia según el lenguaje, pero que coinciden en su funcionalidad, a saber:

- **Ubicación de un número de bytes:** Reservan la cantidad de memoria dinámica solicitada. Devuelven la dirección de dicha memoria. `malloc` (en C) o `new` (en C++).
- **Liberación de espacio ubicado:** Liberan la zona de memoria dinámica indicada, que debe haber sido previamente devuelta por la primitiva de ubicación. `free`, (en C) o `delete` (en C++).

Si queremos usar una estructura dinámica de datos en un programa, deberemos disponer de (o programar nosotros), un conjunto de funciones de utilidad que sean capaces de realizar las operaciones básicas sobre dicho tipo: insertar y extraer un elemento, buscar por contenido, ordenar, etc., etc.

Entrar en detalles sobre las estructuras dinámicas de datos daría para un libro entero. En los capítulos posteriores se tendrá oportunidad de hacer uso de ellas y conocerlas más a fondo. En este capítulo nos limitaremos a comentar ciertas características básicas de las más comunes, que son:

Listas Son una estructura lineal de elementos. Como los eslabones de una cadena.

Según como cada elemento esté vinculado al anterior o/y al posterior hablaremos de:

- **Lista simplemente encadenada:** Cada elemento apunta al siguiente. El puntero "siguiente" del último elemento será nulo. Solo permite el recorrido de la lista en un sentido y limita las operaciones de inserción y extracción.

  Como referencia a la lista mantendremos un puntero al primero de sus elementos. Estará vacia si este puntero es nulo.
- **Lista doblemente encadenada:** Cada elemento apunta tanto al siguiente como al anterior. El puntero "siguiente" del último elemento y el "anterior" del primero, serán nulos. Permite el recorrido de la lista en ambos sentidos y operaciones de inserción/extracción de cualquier posición/elemento de la misma.

  Para manejar la lista mantendremos un puntero al primero de sus elementos, o bien otra estructura que guarde punteros a los dos extremos. Estará vacia si este puntero (o ambos) es nulo.

Según el orden de inserción/extracción de elementos hablaremos de:

- **Cola FIFO:** El primero en entrar será el primero en salir.
- **Pila LIFO:** El último en entrar será el primero en salir.

Árboles Son una estructura arborescente de **nodos**. Todo árbol tendrá un **nodo raíz**, y los nodos que no tengan descendientes son denominados **hojas**.

Según el número de hijos de cada nodo hablaremos de:

- **Árbol binario:** Cada nodo tiene dos hijos.

- **Árbol N-ario:** Múltiples hijos por nodo.

Tablas Hash Son estructuras mixtas que, normalmente, se componen de un vector (de tamaño fijo) de punteros a una estructura dinámica. Su uso fundamental es la optimización de la búsqueda de datos, cuando estos son muy numerosos.

Toda tabla *hash* utiliza una función de barajado (función *hash*) que, en función de un campo clave de cada elemento, determina la entrada del vector en la que debería encontrarse dicho elemento.

Entorno UNIX y Herramientas

Capítulo 4

El Entorno UNIX

¿Qué es UNIX?

Es sin lugar a dudas uno de los sistemas operativos más extensos y potentes que hay. Sus principales virtudes son:

Multiusuario En una máquina UNIX pueden estar trabajando simultáneamente muchos usuarios. Cada usuario tendrá la impresión de que el sistema es suyo en exclusiva.

Multiproceso Cada usuario puede estar ejecutando simultáneamente muchos procesos. Cada proceso se ejecuta independientemente de los demás como si fuese el único en el sistema. Todos los procesos de todos los usuarios son ejecutados concurrentemente sobre la misma máquina, de forma que sus ejecuciones avanzan de forma independiente. Pero también podrán comunicarse entre si.

Multiplataforma El núcleo del sistema operativo UNIX está escrito en más de un 95 % en lenguaje C. Gracias a ello ha sido portado a máquinas de todos los tamaños y arquitecturas. Los programas desarrollados para una máquina UNIX pueden ser llevados a cualquier otra fácilmente.

¡Cómo es UNIX!

Sensible al tipo de letra. Distingue entre mayúsculas y minúsculas . No es lo mismo `unix` que `Unix` que `UNIX`.

Para usuarios NO torpes. Otros Sistemas Operativos someten cada acción "peligrosa" (ej. reescribir un fichero) al consentimiento del usuario.

En UNIX, por defecto, se entiende que el usuario sabe lo que quiere, y lo que el usuario pide, se hace, sea peligroso o no.

Lo borrado es irrecuperable. Es un ejemplo del punto anterior. Hay que tener cuidado. Como veremos más adelante existe la posibilidad de que el usuario se proteja de sí mismo en este tipo de mandatos.

¡Cómo es Linux!

POSIX. Es una versión de UNIX que cumple el estandard *Portable Operating System Interface*.

Libre. Se distribuye bajo licencia GNU, lo que implica que se debe distribuir con todo su código fuente, para que si alguien lo desea, lo pueda modificar a su antojo o necesidad.

Evoluciona. Por ser libre, está en permanente desarrollo y mejora, por programadores voluntarios de todo el mundo.

Gratis. Al increíble precio de 0 mil pesetas.

Contenidos

En este capítulo se presenta el entorno UNIX desde el punto de vista de un usuario del mismo. El objetivo es que el usuario se sienta mínimamente confortable en UNIX.

4.1. Usuarios y Grupos

Para que una persona pueda utilizar un determinado sistema UNIX debe haber sido previamente dado de alta como usuario del mismo, es decir, debe tener abierta una cuenta en el sistema.

Usuario

Todo usuario registrado en el sistema se identifica ante él con un nombre de usuario (*login name*) que es único. Cada cuenta está protegida por una contraseña (*password*) que el usuario ha de introducir para acceder a su cuenta. Internamente el sistema identifica a cada usuario con un número único denominado UID (*User Identifier*).

Grupo

Para que la administración de los usuarios sea más cómoda estos son organizados en grupos. Al igual que los usuarios, los grupos están identificados en el sistema por un nombre y por un número (GID) únicos. Cada usuario pertenece al menos a un grupo.

Privilegios

Las operaciones que un usuario podrá realizar en el sistema estarán delimitadas en función de su identidad, esto es por la pareja UID-GID, así como por los permisos de acceso al fichero o recurso al que desee acceder.

Superusuario

De entre todos los usuarios, existe uno, denominado "superusuario" o *root* que es el encargado de administrar y configurar el sistema.

Es aquel que tiene como UID el número 0 (cero). El superusuario no tiene restricciones de ningún tipo. Puede hacer y deshacer lo que quiera, ver modificar o borrar a su antojo. Ser superusuario es una labor delicada y de responsabilidad.

La administración de un sistema UNIX es compleja y exige amplios conocimientos del mismo.

4.2. Sesión

Una *sesión* es el periodo de tiempo que un usuario está utilizando el sistema. Como sabemos, cada usuario tiene una cuenta que estará protegida por una contraseña o *password*.

Ahora siga los siguientes pasos para entrar y salir del sistema.

`login: myname_` T 4.1

Para entrar al sistema introduzca su nombre de usuario o login name.

`passwd: *******_` T 4.2

Si ya ha puesto una contraseña deberá usted introducirla ahora, para que el sistema verifique que usted es quien dice ser.

prompt`>_` T 4.3

Al entrar en el sistema se arranca un programa que nos servirá de intérprete de mandatos. Nos presenta un mensaje de apremio.

`logout` T 4.4

Para terminar una sesión deberá usar este mandato o bien `exit`.

4.3. Mandatos

Todos los sistemas operativos tienen mandatos que facilitan el uso del sistema. Su forma general es:

```
mandato [opciones] [argumentos...]
```

Los campos están separados por uno o más espacios. Algo entre corchetes `[ ]` es opcional. La notación `...` indica uno o varios. Las opciones generalmente comenzarán por un `-` (menos).

A lo largo de esta presentación se irá solicitando que usted realice numerosas prácticas, observe el comportamiento del sistema o conteste preguntas...

☑ *Ahora debe usted entrar en el sistema como se indicó en el apartado anterior.* T 4.5

!! *Observe que* UNIX *distingue entre mayúsculas y minúsculas.* T 4.6

? *Conteste, ¿sabría usted terminar la sesión?* T 4.7

...en muchos casos se sugerirá que ejecute ciertos mandatos...

`who am i` T 4.8

Obtenga información sobre usted mismo.

`date` T 4.9

Conozca la fecha y hora.

`cal` T 4.10

Visualice un calendario del mes actual.

...se darán más explicaciones para ampliar sus conocimientos...

La mayoría de los mandatos reconocen las opciones `-h` o `--help` como petición de ayuda sobre él mismo.

...y se solicitará que realice más practicas...

T 4.11 `date -h`

Aprenda que opciones admite este mandato.

T 4.12 ? *¿Qué día de la semana fue el día en que usted nació?*

...así mismo, durante esta presentación iremos introduciendo un resumen de algunos mandatos que se consideran más importantes.

Si desea información completa sobre algo, deberá usar el mandato `man`.

`man [what]` — *Manual Pages*

Visualiza una copia electrónica de los manuales del sistema. Es seguro que nos asaltará a menudo la duda de como usar correctamente algún mandato o cualquier otra cosa. Deberemos consultar el manual.

Los manuales se dividen en secciones:

1 Mandatos (ej. `sh`, `man`, `cat`).

2 Llamadas al sistema (ej. `open`, `umask`).

3 Funciones de librería (ej. `printf`, `fopen`).

4 Dispositivos (ej. `null`).

5 Formato de ficheros (ej. passwd).

6 Juegos.

7 Miscelánea.

8 Mandatos de Administración del Sistema.

T 4.13 `man -h`

Busque ayuda breve sobre el mandato `man`.

T 4.14 `man man`

Lea la documentación completa del mandato `man`.

T 4.15 `man [1-8] intro`

En caso de conflicto, si ejemplo existe en diferentes secciones la misma hoja de manual, deberá explicitar la sección en la que buscar.

La hoja intro *de cada sección informa de los contenidos de la misma.*

4.4. Procesos

Cando usted invoca un mandato, el **fichero ejecutable** del mismo nombre es ejecutado. Todo programa en ejecución es un *proceso*. Todo proceso de identifica por un número único en el sistema, su PID.

Concurrencia

En todo momento, cada proceso activo de cada usuario del sistema, esta compitiendo con los demás por ejecutar. El sistema operativo es el que marca a quién le toca el turno en cada momento.

Todo esto sucede muy muy deprisa, con el resultado de que cada proceso avanza en su ejecución sin notar la presencia de los demás, como si fuera el único proceso del sistema, pero en realidad todos están avanzando simultáneamente. A esta idea se le denomina *concurrencia*.

`ps` T 4.16

Permite ver una instantánea de los procesos en el sistema. Admite multitud de opciones.

Jerarquía

Todos los procesos del sistema se organizan en una jerarquía *padre–hijo(s)* que tiene como ancestro último al proceso denominado *init* cuyo PID es 1.

Todo proceso tiene también asociado un número que identifica a su proceso padre, es el PPID.

`pstree` T 4.17

Permite ver una instantánea de la jerarquía de procesos en el sistema. Admite multitud de opciones.

4.5. Árbol de Ficheros

Existe una única estructura jerárquica de nombres de fichero, es decir, **no existe el concepto de "unidad"**. En UNIX los dispositivos se "montan".

Directorio raíz

La raíz de la jerárquica de nombres es el denominado directorio raíz y se denota con el carácter **/** (dividido por) **no por el \ (*backslash*)**.

Directorio HOME

Nada más entrar en el sistema, el usuario es situado en el denominado directorio HOME de su cuenta.

Cada cuenta tiene su propio HOME, que es creado por el administrador del sistema al abrirle la cuenta al usuario.

Este directorio pertenece al usuario correspondiente. Por debajo de él podrá crear cuantos subdirectorios o ficheros quiera.

Tipos de objeto

Para averiguar qué es o qué hay contenido bajo un nombre de fichero puede usar el mandato `file`.

Existen (básicamente) 3 tipos de "objetos", a saber:

Directorios: Es un contenedor cuyas entradas se refieren a otros ficheros y/o directorios. El uso de los directorios da lugar a la jerarquía de nombres. Todo directorio contiene siempre las siguientes dos entradas:

- `.` *(punto)* Se refiere al mismo directorio que la contiene.
- `..` *(punto punto)* Se refiere al directorio padre de este, es decir, a su padre en la jerarquía.

Ficheros normales: Están contenidos en los directorios. Contienen secuencias de bytes que podrían ser códigos de programas, datos, texto, un fuente en C o cualquier otra cosa.

Ficheros especiales: Son como ficheros normales, pero no contienen datos, sino que son el interfaz de acceso a los dispositivos periféricos tales como impresoras, discos, el ratón, etc, etc.

Cada dispositivo, existente o posible, se haya representado por un fichero especial que se encuentra bajo el directorio `/dev`.

UNIX trata estos ficheros especiales (y por lo tanto a los dispositivos) exactamente igual que trata a los ficheros normales, de forma y manera que para los programas, los unos y los otros son **indistinguibles**.

`pwd` — *Print Working Directory*

Permite averiguar cuál es el directorio en que nos encontramos en cada momento, al cuál denominamos directorio actual de trabajo.

T 4.18 `pwd`

Visualice su directorio actual de trabajo.

T 4.19 ‼ *Observe que se muestra el camino absoluto (desde el directorio raíz) de su directorio* HOME.

T 4.20 ‼ *Observe que las componente de un camino se separan entre sí con el carácter* `/` *(dividido por) y no por el* `\` *(*backslash*).*

`cd [dir]` — *Change Directory*

Con este mandato cambiamos de un directorio a otro. Admite como argumento el nombre del directorio al que queremos cambiar.

Existen tres casos especiales que son de utilidad.

`cd`

Invocado sin argumento, nos devuelve directamente a nuestro HOME.

`cd .`

El directorio de nombre . (punto) hace referencia siempre al directorio actual de trabajo. Realmente no nos va a cambiar de directorio.

`pwd` T 4.21

Observe que permanecemos en el mismo directorio.

Como veremos más adelante, el directorio . resulta útil para referirnos a ficheros que "están aquí mismo".

`cd ..`

Todo directorio contiene siempre una entrada de nombre .. (punto punto) que hace referencia siempre al directorio padre del directorio actual.

`pwd` T 4.22

Observe que ascendemos al directorio padre.

`cd ..` T 4.23

Repita la operación hasta llegar al raíz.

`pwd` T 4.24

Observe que no se puede ir más allá del directorio raíz.

`cd` T 4.25

¿A qué directorio hemos ido?

`ls [-opt] [dirs...]` ——— *List Directory Contents*

Este mandato nos presenta información sobre los ficheros y directorios contenidos en el(os) directorio(s) especificado(s).

Si no se indica ningún directorio, informa sobre el directorio actual de trabajo.

Este mandato admite cantidad de opciones que configuran la información obtenida. Entre ellas las más usadas son, por este orden:

`-l` Formato "largo". Una línea por fichero o directorio con los campos: permisos, propietario, grupo, tamaño, fecha y nombre.

`-a` Informa también sobre los nombres que comienzan por . (punto), que normalmente no se muestran.

`-R` Desciende recursivamente por cada subdirectorio encontrado informando sobre sus contenidos.

Si necesita explorar más opciones, consulte el manual.

`ls` T 4.26

Observe que su directorio HOME *parece estar vacío.*

`ls -a` T 4.27

Pero no lo está. Al menos siempre aparecerá las entradas . y ...

!! *Es posible que aparezcan otros nombres que con* `ls` *no vio.* T 4.28

Este mandato evita mostrar aquellas entradas que comienzan por el carácter punto. Podríamos decir que los ficheros o directorios cuyo nombre comienza por un punto están "ocultos", pero se pueden ver si usamos la opción `-a`.

Suelen ser ficheros (o directorios) de configuración.

T 4.29 `ls -la`

Si queremos conocer los detalles del "objeto" asociado a cada entrada de un directorio, usaremos la opción `-l`.

T 4.30 ? *¿A quién pertenecen los ficheros que hay en su* HOME*?*

¿Qué tamaño tienen?

¿Cuándo fueron modificados por última vez?

T 4.31 `ls /home`

Observe el HOME *de otros usuarios de su sistema.*

Inspeccione el contenido de sus directorios.

T 4.32 `ls -la /`

Muestra el "tronco" del árbol de nombres.

T 4.33 `man 7 hier`

Lea una descripción completa de la jerarquía del sistema de ficheros UNIX *que está usando.*

T 4.34 `ls -???? /bin`

Obtenga el contenido del directorio `/bin` *ordenado de menor a mayor el tamaño de los ficheros.*

T 4.35 !! *Observe que aparece un fichero de nombre* `ls`*. Ese es el fichero que usted está ejecutando cada vez que usa el mandato* `ls`.

Esos ficheros son sólo una parte de los usted podría ejecutar, hay muchos más.

T 4.36 ? *¿A quién pertenece el fichero* `/bin/ls`*?*

T 4.37 !! *Sus permisos permiten que usted lo ejecute.*

4.6. Descriptores de fichero

Los procesos manejan ficheros a través de los denominados descriptores de fichero. Por ejemplo, el resultado de abrir un fichero es un descriptor. Las posteriores operaciones de manejo (lectura, escritura, etc.) de este fichero reciben como parámetro el descriptor.

Desde el punto de vista de nuestro programa, un descriptor no es más que un número entero positivo (0, 1, 2, ...).

Descriptores estándar

Los tres primeros descriptores de fichero son los denominados "estándar", y se considera que siempre están disponibles para ser usados.

Normalmente estarán asociados al terminal pero podrían estar asociados a un fichero o a cualquier otro objeto del sistema de ficheros, pero este es un detalle que no ha de preocuparnos a la hora de programar.

Los programas que se comporten de forma "estándar" los utilizarán adecuadamente.

Entrada estándar. Es la entrada principal al programa. Los programas que se comportan de forma estándar reciben los datos que han de procesar leyendo de su entrada estándar (descriptor de fichero número 0).

Salida estándar. Es la salida principal del programa. Los programas que se comportan de forma estándar emiten sus resultados escribiendo en su salida estándar (descriptor de fichero número 1).

Error estándar. Es la salida para los mensajes de error del programa. Si un programa que se comporta de forma estándar ha de emitir un mensaje de error deberá hacerlo a través de su error estándar (descriptor de fichero número 2).

4.7. Intérprete de mandatos

La interacción del usuario con el sistema podría ser gráfica, pero en UNIX es más común que sea textual, esto es, será a través del diálogo con un intérprete de mandatos.

En UNIX se le denomina *shell* (concha). No existe un único *shell* sino muchos.

`ls -l /bin/*sh` T 4.38

Los ficheros que aparecen son los diferentes shell *disponibles. Quizás aparezca algún otro fichero. Quizás aparezcan enlaces simbólicos (otro tipo de objeto del sistema de ficheros).*

Se distinguen fundamentalmente en la sintaxis que reconocen y en las facilidades que proporcionan para su uso interactivo y para usarlos como lenguaje de programación de "guiones" de mandatos (*scripts*).

Los más comunes para uso interactivo son el `tcsh` y el `bash`.

Ya hemos estado usando el *shell*, de hecho **usted está ejecutando uno de ellos en este momento**. !!

Vamos a ver algunas características básicas del *shell* que nos serán de mucha utilidad.

Uso interactivo

Cuando el usuario entra en su cuenta, termina por aparecer en su terminal un *prompt* o mensaje de apremio que le informa de que un *shell* está a su servicio.

En este documento se muestran los diálogos con el *shell* de la siguiente manera:

```
$
```

El prompt *será el símbolo* `$`.

Un buen *shell* para uso interactivo ofrece servicios que le facilitan la trarea al usuario. Esto significa que existen múltiples combinaciones de teclas asociadas con alguna funcionalidad especial. Para conocerlas completamente deberá consultar el manual.

Edición de línea

Podemos mover el cursor (␣) sobre la línea de mandatos que estamos tecleando con [←] [→], borrar antes y sobre el cursor con [Backspace] y [Del], ir al inicio y al final con [Inicio] y [Fin], etc.

T 4.39 ⚠ *Observe que pasa si intenta editar una línea más larga que la pantalla.*

Histórico de mandatos

Si hace poco que tecleamos un mandato no es preciso volverlo a teclear. Basta con buscarlo haciendo uso de las flechas del teclado [↑] y [↓].

El mandato interno `history` nos informa de todos los mandatos que hemos usado últimamente.

T 4.40 (`history`)

Visualice los mandatos que hasta ahora ha realizado en su shell.

T 4.41 ⚠ *Observe que* `history` *guarda los mandatos anteriores de una sesión otra.*

T 4.42 [`!!`]

Repite el último mandato.

T 4.43 [`!5`]

Repite el mandato que ocupa la posición 5 en la historia.

Fin de datos

La combinación de teclas [Ctrl][D] en una línea vacía indica "fin de datos".

Si la pulsamos y hay algún programa leyendo del terminal, entenderá que no hay más datos y terminará normalmente.

T 4.44 [`cat`]

Es un mandato que lo que lee de su entrada lo escribe en su salida. Escriba varias líneas de datos para observar su comportamiento.

Como está leyendo del terminal, para que termine normalmente, deberá indicarle "fin de datos".

T 4.45 ($ [Ctrl][D])

El shell *también es un programa que lee de su entrada estandard. Indíquele "fin de datos".*

T 4.46 ⚠ *El* shell *entenderá que no hay más datos, terminará normalmente y por lo tanto nos sacará de la cuenta.*

Si ha sido así y este comportamiento no le agrada, hay una forma de configurar el shell *para evitar que suceda. Más adelante la veremos.*

Control de trabajos

En UNIX podemos arrancar mandatos para que se ejecuten en segundo plano (*background*) poniendo el caracter `&` al final de la línea.

Tambien, desde su *shell* puede suspender el trabajo que está en primer plano (*foreground*), rearrancar trabajos suspendidos, dejarlos en segundo plano, matarlos, etc.

Ctrl C Pulsando esta combinación de teclas cuando tenemos un programa corriendo en primer plano, le mandaremos una señal que "lo matará".

Ctrl Z El programa en primer plano quedará suspendido y volveremos a ver el *prompt* del *shell*.

`jobs` Este mandato interno nos informa de los trabajos en curso y de su estado. Podremos referinos a uno concreto por la posición que ocupa en esta lista.

`fg` Ejecutándolo, el último proceso suspendido volverá a primer plano. Para referirnos al segundo trabajo de la lista de trabajos usaremos `fg %2`.

`bg` Ejecutándolo, el último proceso suspendido volverá a ejecutar, pero en segundo plano! Es como si lo hubiéramos arrancado con un `&` detrás.

`kill` Este mandato interno permite enviarle una señal a un proceso, con lo que, normalmente, lo mataremos.

☑ *Arranque varios procesos en* background*, por ejemplo varios* `sleep 60 &`*, y a continuación consulte los trabajos activos.* T 4.47

☑ *Arranque un proceso en primer plano, por ejemplo* `cat` *y suspéndalo. Consulte los trabajos activos. Vuelva a ponerlo en primer plano y luego mándele una señal que lo mate.* T 4.48

⍰ *¿Cómo lo mataría si se encuentra en segundo plano?* T 4.49

Completar nombres de fichero

Si tenemos un nombre de fichero escrito "a medias" y pulsamos el tabulador (↦), el `tcsh` intentará completar dicho nombre. Si lo que pulsamos es la combinación Ctrl D, nos mostrará la lista de nombres que empiezan como él.

También funciona para nombres de mandato.

☑ *Escriba* `ls .cs↦` *y luego intente* `ls .cs` Ctrl D*, observe la diferencia.* T 4.50

Control de *scroll* de terminal

Denominamos terminal al conjunto pantalla/teclado o a la ventana a través de la cual dialogamos con el *shell*.

Ctrl S Congela el terminal. No mostrará nada de lo que nuestros programas escriban, ni atenderá a lo que tecleemos.

Si por descuido pulsamos Ctrl S podríamos pensar que el sistema se ha quedado colgado, pero no es así.

Ctrl Q Descongela el terminal. Lo que los programas hubiesen escrito mientras las pantalla estubo congelada, así como lo que hubiesemos tecleado, aparecerá en este momento.

El uso común de la pareja Ctrl S Ctrl Q es para controlar la salida de programas que emiten muchas líneas.

T 4.51 Ctrl S `ls -la`

Observe que, de momento, nada de lo que hemos tecleado aparece en el terminal.

T 4.52 Ctrl Q

Verá como el mandato que tecleamos aparece y se ejecuta con normalidad.

T 4.53 `ls -laR /`

El resultado de este mandato son demasiadas líneas de texto. Juege con Ctrl S Ctrl Q *a parar y dejar correr la pantalla.*

T 4.54 Ctrl C

Cuando se canse, mate el trabajo.

Redirección

En UNIX podemos redirigir la entrada estandard, salida estandard y/o error estandard de los mandatos.

Esta es una caracteristica muy muy util, ya que, sin necesidad de modificar en absoluto un mandato podemos usarlo para que tome su entrada de un fichero en vez de tener que teclearla.

De igual mamera en vez de que la salida del mandato aparezca en la pantalla, podríamos enviarla a un fichero para usarla con posterioridad.

La sintaxis correcta para realizar redirecciones es:

mandato < *fichero*

En su ejecución, el mandato usará `fichero` como entrada, leerá de él y no del teclado.

mandato > *fichero*

En su ejecución, la salida del mandato será enviada a `fichero` y no a la pantalla. Si `fichero` no existiera con anterioridad, será creado automáticamente, pero si existe, será previamente truncado a tamaño cero.

mandato >> *fichero*

Anexa la salida del mandato al final del fichero indicado.

Los mensajes de error que un mandato pueda generar también pueden ser redirigidos a un fichero, pero normalmente preferimos que se visualicen por la pantalla.

T 4.55 ☑ *Obtenga en un fichero de nombre* `todos_los_ficheros` *un listado en formato largo de todo el árbol de ficheros recorriendolo recursivamente desde el directorio raiz.*

☑ *Si lo arrancó en primer plano, suspendalo y páselo a segundo plano. Al estar en segundo plano sigue corriendo, pero mientras podemos seguir haciendo otras cosas.* T 4.56

`ls -l todos_los_ficheros` T 4.57

Compruebe que el fichero está creciendo de tamaño.

`tail -f todos_los_ficheros` T 4.58

Examine como va creciendo el contenido del fichero. Para salir de mátelo con Ctrl C.

!! *Aparecerán en pantalla sólo los mensajes de error que el mandato produce al no poder acceder a ciertos directorios.* T 4.59

`grep` *login_name* `<todos_los_ficheros >mis_ficheros` T 4.60

Use el mandato `grep` *para localizar las líneas del fichero de entrada que contienn el texto indicado, en este cado su nombre de usuario. Deje el resultado en un fichero de nombre* `mis_ficheros`.

Secuencia

En UNIX podemos redirigir la salida estandard de un mandato directamente a la entrada estandard de otro. Con ello contruimos secuencias de mandatos que procesan el resultado del anterior y emiten su resultado para que sea procesado por el siguinte mandato de la secuencia.

Esto lo conseguimos separanto los mandatos con el caracter "`|`" (*pipe*).

Esta es es una facilidad muy poderosa de UNIX, ya que, sin necesidad de ficheros intermedios de gran tamaño podemos procesar grandes cantidades de información.

`ls -la /home | grep` *login_name* T 4.61

Obtiene un listado en formato largo de todos los directorios de cuentas y filtra aquellas líneas que contienen nuestro nombre de usuario.

!! *No se crea ningún fichero intermedio.* T 4.62

☑ *Obtenga el listado "todos sus ficheros" de la sección anterior, conectando el resultado al mandato* `grep` *para quedarse sólo con sus ficheros. Mande el resultado a un nuevo fichero de nombre* `MIS_ficheros`. *Pongalo así, con mayúsculas, para no machacar el fichero nombrado así pero en minúsculas.* T 4.63

Recuerde invocar el mandato con `&` *para poder seguir haciendo otras cosas.*

☑ *Cuando hayan terminado de ejecutar los mandatos, compruebe las differencias entre los ficheros de nombre* `mis_ficheros` *y* `MIS_ficheros`. *Para ello utilice el mandato* `diff`, *consultando el manual si es necesario.* T 4.64

`less [files...]` — *On Screen File Browser*

Filtro para la visualización por pantalla de ficheros. Es una versión mejorada del clásico mandato `more`.

Si lo usamos indicando un sonjunto de ficheros, podremos pasar al siguiente y al anterior con `:n` y `:p` respectivamente.

Con este mandato sólo podremos ver los ficheros, pero no modificarlos, más adelante veremos algún editor.

T 4.65 `less *_ficheros`

Visualice el contenido de los ficheros que existen ya en su HOME *que terminan por "_ficheros".*

T 4.66 `ls *`

Observe que el metacaracter `*` *no alude a los ficheros cuyo nombre empieza por punto.*

Metacaracteres

Los metacaracteres (*wildcards*) son caracteres comodín. Con su uso podemos hacer referencia a conjuntos de ficheros y/o directorios, indicando la expresión regular que casa con sus nombres.

`?` Comodín de caracter individual.

`*` Comodín de tira de cero o más caracteres.

`~` Abreviatura del directorio *home*.

`~`*`user`* Abreviatura del directorio *home* de *user*.

`[`*`abc`*`]` Casa con un caracter del conjunto especificado entre los corchetes.

`[`*`x-y`*`]` Casa con un caracter en el rango especificado entre los corchetes.

`[^...]` Niega la selección indicada entre los corchetes.

{*str,...*} Agrupación. Casa sucesivamente cada tira.

T 4.67 ☑ *Obtenga un listado en formato largo, de los ficheros de los directórios* `/bin` *y* `/usr/bin/` *que contengan en su nombre el texto* `sh`.

T 4.68 ☑ *Obtenga un listado en formato largo, de los ficheros del directorio* `/dev` *que comiencen por* `tty` *y* **no** *terminen en dígito.*

`passwd` — *Change User Password*

UNIX protege el acceso a las cuentas de los usuarios mediante una palabra clave.

Las cuentas del sistema, sus contraseñas y resto de información, están registradas en el fichero `/etc/passwd`.

T 4.69 `less /etc/passwd`

Visualice el contenido del fichero de contraseñas.

Este fichero contiene la información sobre las cuentas de usuario que hay en su sistema.

Compruebe que existe una cuenta con su nombre de usuario.

`man 5 passwd` T 4.70

Estudie cuál es el contenido del fichero de contraseñas.

Un usuario puede cambiar su palabra clave haciendo uso del mandato `passwd`. Es muy importante que la contraseña no sea trivial, ni una fecha, ni una matrícula de coche, teléfono, nombre própio, etc.

`passwd` T 4.71

¡Si no tiene un buen password*, cámbielo ahora mismo!*

`cp orig dest` — *Copy Files*

Obtiene una copia de un fichero o bien copia un conjunto de ficheros al directorio indicado.

Admite variedad de opciones, que no vamos a describir en este documento (si tiene dudas, consulte el manual).

Interesa destacar que este **es un mandato peligroso**, dado que (por defecto) hace lo que le pedimos sin consultar, lo cuál puede suponer que machaquemos ficheros que ya existan. Para que nos consulte deberemos usarlo con la opción `-i`.

`cp /etc/passwd ~` T 4.72

Copie en su HOME *el fichero* `/etc/passwd`*.*

☐ *Visualice que su contenido es realmente una cópia del original.* T 4.73

☑ *Cópie el fichero* `.cshrc` *sobre el que ahora tiene en su cuenta con el nombre* `passwd`*.* T 4.74

☐ *¿Le ha consultado* `cp` *antes de reescribir? ¿Se ha modificado el fichero?* T 4.75

☑ *Intente las mismas operaciones usando la opción* `-i`*.* T 4.76

`rm files...` — *Remove Files*

Borrar ficheros. Para ver sus opciones consulte el manual.

También **es un mandato peligroso**. En este caso, `rm` no sólo hace lo que le pedimos sin consultar, sino que en UNIX **no existe manera de recuperar** un fichero (o directorio) que haya sido borrado. Por lo tanto se hace imprescindible usarlo con la opción `-i`.

☑ *Intente borrar el fichero* `passwd` *de su cuenta con* `rm -i passwd`*. Conteste que no.* T 4.77

☑ *Borre el fichero* `passwd` *que ahora tiene en su cuenta.* T 4.78

☐! *Observe que efectivamente ha sido eliminado.* T 4.79

☑ *Intente borrar el fichero* `/etc/passwd`. T 4.80

T 4.81 ☐? *¿Puede hacerlo? ¿Porqué?*

`mv orig dest` — *Move (Rename) Files*

Cambia el nombre de un fichero o bien mueve un fichero o conjunto de ficheros a un directorio.

También **es un mandato peligroso**, porque podría "machacar" el(os) fichero de destino sin pedirnos confirmación. Se debe usar con la opción `-i`.

T 4.82 ☑ *Intente mover el fichero* `/etc/passwd` *a su* HOME.

T 4.83 ☑ *Intente mover algún fichero de su cuenta (uno que no le valga) al directorio* `/etc`.

T 4.84 ☐? *¿Puede hacer estas operaciones? ¿Porqué?*

T 4.85 ☑ *Cambie de nombre al fichero* `mis_ficheros` *a* `MIS_ficheros`.

T 4.86 ☐? *¿Pudo hacerlo? ¿Le consultó?*

`mkdir dir` — *Make Directory*

Crea un nuevo directorio. Por supuesto, si ya existe un fichero o directorio con dicho nombre dará un error.

Los directorios que usted cree serán suyos. Sólo podrá crear ficheros o directorios en aquellos directorios donde usted tenga permiso de escritura.

T 4.87 ☑ *Cree en su cuenta un subdirectorio de nombre* `subarbol`.

T 4.88 ☑ *Visualice su contenido con* `ls -la`.

T 4.89 ☑ *Cree otros subdirectorios más profundos, pero sin hacer uso del* `cd`.

T 4.90 `ls -laR ~`

Visualice el contenido actual de su HOME *recursivamente.*

`rmdir dir` — *Remove Directory*

Borra un directorio. Por supuesto el directorio debe estar vacio.

T 4.91 ☑ *Intente borrar un directorio* **no** *vacio, ej.* `subarbol`.

T 4.92 ☐? *¿Puede hacerlo?*

T 4.93 ☑ *Visualice el contenido de* `subarbol`.

T 4.94 ☑ *Elimine los directorios más profundos, y vaya ascendiendo, hasta conseguir eliminar el directorio* `subarbol`.

4.8. Configuración

Cada ususario puede configurar el comportamiento de su cuenta editando convenientemente ciertos ficheros presentes en su HOME que dependen de qué *shell* use usted.

El fichero de configuración que es de nuestro interés en este momento es:

`.bashrc` Se ejecuta siempre que el usuario arranca un nuevo `bash`.

`.cshrc` Se ejecuta siempre que el usuario arranca un nuevo `csh` o `tcsh`.

Dado que estos ficheros sólo se leen al arrancar el *shell*, para que los cambios que hagamos en ellos tengan efecto, deberemos bien salir de la cuenta y volver a entrar o bien hacer uso del mandato interno `source` para que su *shell* lea e interprete el contenido del fichero.

Mandatos peligrosos

En UNIX, **si se borra un fichero o un directorio no hay forma de recuperarlo**. Además, los mandatos obedecen ciegamente a quien los usa, y no consultan. Para evitar meter la pata es conveniente usar ciertos mandatos de forma que consulten antes de actuar. Para ello, vamos a usar el mandato interno `alias`, para protegernos de los mandatos problemáticos.

Fin de datos

Evitaremos que el *shell* termine a la pulsación Ctrl D pidiendole que lo ignore.

`alias` T 4.95

Ejecutando este mandato observará que alias *están en funcionamiento.*

Los alias *modifican "al vuelo" los mandatos que queremos ejecutar, justo antes de ser finalmente ejecutados.*

`pico` *`file`* T 4.96

Es un editor de uso muy sencillo. Uselo para realizar la siguiente modificación en el fichero de confguración de su shell.

Escoja entre la versión de la izquierda, si usted usa el `tcsh` *o el* `csh`*, o la de la derecha si usa el* `bash`*. Añada al final del fichero de configuración indicado las siguientes líneas:*

.cshrc

```
alias cp 'cp -i'
alias mv 'mv -i'
alias rm 'rm -i'
set ignoreeof
```

.bashrc

```
alias cp='cp -i'
alias mv='mv -i'
alias rm='rm -i'
IGNOREEOF=10
```

`source` T 4.97

Recuerde que debe pedirle a su shell *que lea e interprete el contenido del fichero de configuración, para que tenga constancia de la modificación que acaba de hacer.*

`alias` T 4.98

Observará si los alias *que ha establecido están en funcionamiento.*

T 4.99 Ctrl D

Compruebe si esta secuencia hace terminar su shell.

4.9. Xwindows

Hasta ahora hemos experimentado el uso de UNIX desde un interfaz textual, esto es, desde un terminal. Existe un entorno gráfico muy potente llamado *X windows* o simplemente *X*.

Para arrancarlo podemos usar el mandato `startx`.

Capítulo 5

Herramientas de desarrollo

Nos acercaremos al uso de diversas herramientas útiles para el programador.

Existen entornos integrados para el desarrollo de software pero no son de uso común. Clásicamente en UNIX se hace uso de diversas herramientas que cumplen con las diversas etapas del desarrollo.

Esta parte del curso es eminentemente práctica. El alumno deberá consultar la ayuda disponible. Como método rápido de consulta, se dispondrá de trípticos con resúmenes útiles de varias herramientas.

5.1. Editor

Vamos a ver tres editores tipo. El alumno deberá escoger aquella opción que más se adecue a sus necesidades.

En primera instancia, se recomienda que use el `pico`.

`pico` — *Simple Text Editor*

Es el editor que internamente usa la herramienta para correo electrónico `pine`. Es muy sencillo de usar pero poco potente.

A modo de introducción a su uso realice las siguientes tareas:

(`pico`) T 5.1

Arranca el editor.

Ctrl G T 5.2

Accede a la ayuda para aprender los pocos controles que tiene.

Ctrl X T 5.3

Abandona el editor.

`vi` — *VIsual Editor*

El `vi` es el editor más clásico y lo encontraremos en cualquier máquina UNIX. Es por eso que, a pesar de ser poco amigable, hay mucha gente que lo prefiere a otros más sofisticados. Existe una versión más moderna y potente llamada `vim`.

T 5.4 `vim`

Arranca el editor.

T 5.5 Esc

Cambia a "modo mandato".

T 5.6 *:help*

Accede a la ayuda navegable.

T 5.7 *:q*

Abandona la edición actual.

T 5.8 `vimtutor`

Arranca el editor en la realización de un tutorial del mismo.

`emacs` — *GNU project Editor*

El `emacs`, desarrollado por FSF bajo licencia GNU, es mucho más que un simple editor. Es un entorno sobre el que se integran múltiples aplicaciones, correo electrónico, navegador web, news, etc., etc., etc. Está escrito sobre un intérprete de *lisp*.

Existe una versión llamada `Xemacs` más adaptada al entorno gráfico X, y totalmente compatible con el anterior.

T 5.9 `emacs`

Arranca el editor.

T 5.10 Ctrl G

Aborta la acción o mandato a medio especificar.

T 5.11 Ctrl H T

Accede a la realización de un tutorial.

T 5.12 Ctrl X Ctrl C

Abandona la edición actual.

Ejercicios

Para el editor de su elección, o para cada uno de ellos, realice los siguientes ejercicios prácticos:

T 5.13 ☑ *Arranque el editor para crear un nuevo fichero.*

T 5.14 ☑ *Lea el fichero de ejemplo "estilos.c" desde el editor.*

T 5.15 ☑ *Sin modificar su funcionalidad, formatee el texto con el estilo que considere más legible.*

T 5.16 ☑ *Guarde el texto editado con el nombre "mi_estilo.c".*

T 5.17 ☑ *Abandone el editor.*

5.2. Compilador

Compilar un lenguaje de alto nivel, lease C, es traducir el fichero o ficheros fuente en ficheros objeto, que contienen una aproximación al lenguaje máquina, más una lista de los símbolos externos referenciados pero no resueltos.

La etapa de montaje es capaz de enlazar unos ficheros objeto con otros, y a su vez con las bibliotecas necesarias, resolviendo todas las referencias a símbolos externos. Esta etapa produce finalmente un fichero ejecutable.

Un fichero ejecutable tiene un formato interno que el sistema operativos es capaz de entender, que describe como situar el proceso en memoria para su ejecución.

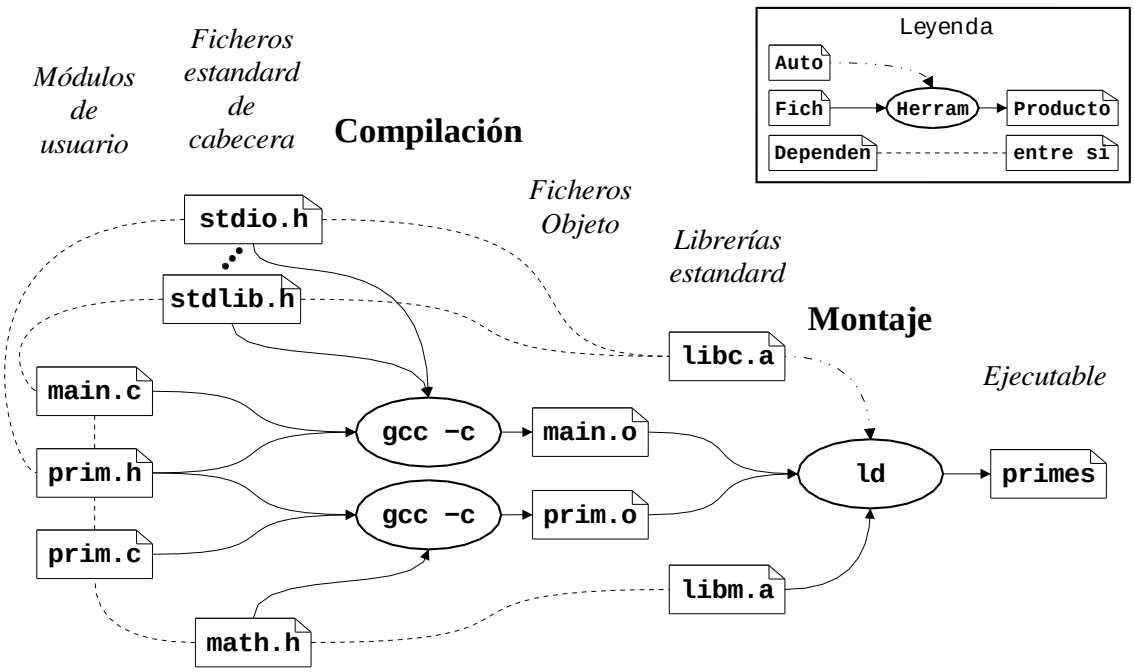


Figura 5.1: Fases de compilación y montaje.

En la figura 5.1 se observa la compilación separada de dos módulos `main.c` y `prim.c` y el montaje de los dos ficheros objeto resultantes para la obtención del fichero ejecutable `primes`.

Extensiones

Ciertos sufijos o extensiones de los nombres de fichero, le indican al compilador los contenidos del mismo, y por lo tanto el tipo de tratamiento que debe realizarse:

`.c` Fuente de C. Debe ser preprocesado, compilado y ensamblado.

`.h` Fichero de cabecera de C. Contiene declaraciones de tipos de datos y prototipos de funciones. Debe ser preprocesado.

`.o` Fichero objeto. Es el resultado de la compilación de un fichero fuente. Debe ser montado.

`.a` Biblioteca de ficheros objeto. Se utiliza para resolver los símbolos de las funciones estandard que hemos usado en nuestro programa y que por lo tanto aún quedan sin resolver en los ficheros objeto.

Errores y *Warnings*

Como ya habrá constatado, los lenguajes de programación, y entre ellos el C, presenta una sintaxis bastante estricta. Una de las misiones básicas del compilador es indicarnos los errores que hayamos cometido al codificar, dónde están y en qué consisten.

Los avisos que un compilador es capaz de notificar son de dos tipos:

Errores. Son problemas graves, tanto que evitan que el proceso de compilación pueda concluir. No obstante el compilador seguirá procesando el fichero o ficheros, en un intento de detectar todos los errores que haya.

Es imprescindible corregir estos errores para que el compilador puede terminar de hacer su trabajo.

Warnings. Son problemas leves o alertas sobre posibles problemas en nuestro código.

Aunque el compilador podrá terminar de hacer su trabajo, **es imprescindible corregir los problemas que originan estos warnings**, porque seguramente serán fuente de futuros problemas más graves.

La compilación de un programa correctamente realizado no debería dar ningún tipo de mensaje, y terminar limpiamente.

`gcc` — *GNU C and C++ Compiler*

El nombre que suele recibir el compilador de lenguaje C en UNIX es `cc`. Nosotros usaremos el `gcc` que es un compilador de C y C++, desarrollado por FSF bajo licencia GNU.

Es un compilador muy bueno, rápido y eficiente. Cubre las etapas de compilación y montaje. Parte de uno o más fuentes en C, objetos y/o bibliotecas y finalmente genera un ejecutable autónomo.

Admite multitud de opciones, entre las cuales cabe destacar:

`-c` *file.c* Realiza tan solo el paso de compilación del fichero indicado, pero no el de montaje.

`-o` *name* Solicita que el resultado de la compilación sea un fichero con este nombre.

`-Wall` Activa la detección de todo tipo de posibles errores. El compilador se convierte en más estricto.

`-g` Añade al fichero objeto o ejecutable información que (como veremos más adelante) permitirá la depuración simbólica del mismo.

`-O` Activa mecanismos de optimización que conseguirán un ejecutable más rápido, a costa de una compilación más larga.

`-l`*library* Solicita el montaje contra la biblioteca especificada.

`-L`*directory* Indica un directorio donde buscar las bibliotecas indicadas con la opción `-l`.

Tiene muchísimas más opciones. Consulte el manual si lo precisa.

`minimo.c` T 5.18

Escriba el fichero con el editor de su preferencia y los contenidos indicados en el capítulo anterior 2.3.

`gcc minimo.c` T 5.19

Usamos el compilador para obtener el ejecutable resultante de este fuente.

`ls` T 5.20

Observe que si no le indicamos al compilador otra cosa, el ejecutable resultante tomará el nombre por defecto `a.out`.

`./a.out` T 5.21

Ejecutamos el fichero, indicando que se encuentra en el directorio actual. Esto será necesario si no tenemos el directorio "." en la variable de entorno PATH.

Ahora vamos a trabajar un programa más complejo.

`factorial.c` T 5.22

Escriba el fichero con el editor de su preferencia y los contenidos indicados en el capítulo anterior. El objetivo será mostrar el factorial de los números de 0 al 20 usando un bucle.

`gcc -g -c factorial.c` T 5.23

Con este paso compilaremos, para obtener el fichero objeto `factorial.o`. *Este fichero contendrá información simbólica.*

`gcc factorial.o -o factorial` T 5.24

Cuando compile correctamente, montaremos para obtener el ejecutable.

`./factorial` T 5.25

Ejecute el mandato y compruebe su correcto funcionamiento.

5.3. Depurador

La compilación sin mácula de un programa no implica que sea correcto. Es muy posible que contenga errores que sólo se harán visibles en el momento de su ejecución. Para corregir estos errores necesitamos una herramienta que nos permita supevisar la ejecución de nuestro programa y acorralar el error hasta localizarlo en el código.

Existen depuradores de bajo y de alto nivel. Los de bajo nivel nos permiten inspeccionar el código máquina o ensamblador, y su ejecución paso a paso. Los alto nivel, también llamados depuradores simbólicos, permiten visualizar y controlar la ejecución línea a línea del código fuente que hemos escrito, así como inspeccionar el valor de las variables de nuestro programa..

Para que un programa pueda ser depurado correctamente, su ejecutable deberá contener información que lo asocie al código fuente del que derivó. Para ello deberá haber sido compilado con la opción `-g`.

Algunas de las funciones del depurador son:

- Establecer puntos de parada en la ejecución del programa (*breakpoints*).
- Examinar el valor de variables
- Ejecutar el programa línea a línea

Nosotros usaremos el `gdb` desarrollado por FSF bajo licencia GNU.
También veremos en `ddd` que es un *frontend* gráfico para el `gdb`.

Fichero `core`

Trabajando en UNIX nos sucederá a menudo, que al ejecutar un programa este termine bruscamente y apareciendo algún mensaje de error como por ejemplo:
`segmentation fault: core dumped`.
Este mensaje nos informa de que nuestro programa en ejecución, esto es, el proceso, ha realizado un acceso ilegal a memoria y el sistema operativo lo ha matado. Así mismo nos indica que se ha generado un fichero de nombre `core`. El fichero `core` es un volcado de la memoria del proceso en el preciso instante en que cometío el error, esto es una instantánea. Este fichero nos servirá para poder estudiar con todo detalle la situación que dio lugar al error.

`gdb prog [core]` — *GNU debugger*

Invocaremos al `gdb` indicádole el programa ejecutable que queremos depurar y opcionalmente indicaremos en fichero `core` que generó su anómala ejecución.
Una vez arrancado el depurador, proporciona una serie de mandatos propios para controlar la depuración:

`help` Da acceso a un menú de ayuda.

`run` Arranca la ejecución del programa.

`break` Establece un *breakpoint* (un número de línea o el nombre de una función).

`list` Imprime las líneas de código especificadas.

`print` Imprime el valor de una variable.

`continue` Continúa la ejecución del programa después de un *breakpoint*.

`next` Ejecuta la siguiente línea. Si se trata de una llamada a función, la ejecuta completa.

`step` Ejecuta la siguiente línea. Si se trata de una llamada a función, ejecuta sólo la llamada y se para al principio de la misma.

`quit` Termina la ejecución del depurador

Para más información sobre el depurador consulte su ayuda interactiva, el manual o el tríptico.
Más adelante tendremos oportunidad de hacer un uso extenso del depurador.

`ddd prog [core]` _____ *Data Display Debugger*

Es un *frontend* gráfico para el `gdb`, esto es, nos ofrece un interfaz gráfico bastante intuitivo para realizar la depuración, pero en definitiva estaremos usando el `gdb`. Como característica añadida, nos ofrece facilidades para la visualización gráfica de las estructuras de datos de nuestros programas, lo cuál será muy útil. Para usar el `ddd` es preciso tener arrancadas las X-Windows.

5.4. Bibliotecas

Una regla básica para realizar software correcto es utilizar código que ya esté probado, y no estar reinventando sistemáticamente la rueda. Si existe una función de biblioteca que resuelve un problema deberemos usarla y no reescribir tal funcionalidad.

Existen muchas muchas bibliotecas con facilidades para realizar todo tipo de cosas: aritmética de múltiple precisión, gráficos, sonidos, etc., etc.

Para poder hacer uso correcto de esas funciones de biblioteca debemos consultar el manual que nos indicará la necesidad de incluir en nuestro fichero fuente cierto fichero de cabecera (ej. `#include <string.h>`), que contiene las declaraciones de tipos de datos y los prototipos de las funciones de biblioteca.

Se recomienda que consulte el tríptico de ANSI C.

Destacaremos aquí dos bibliotecas fundamentales:

`libc.a` Es la biblioteca estandard de C. Contiene funciones para: manejo de tiras de caracteres, entrada y salida estandard, etc.

Se corresponde con las funciones definidas en los ficheros de cabecera:

```
<assert.h>  <ctype.h>  <errno.h>   <float.h>   <limits.h>
<locale.h>  <math.h>   <setjmp.h>  <signal.h>  <stdarg.h>
<stddef.h>  <stdio.h>  <stdlib.h>  <string.h>  <time.h>
```

El montaje contra esta biblioteca se realiza de forma automática. No es preciso indicarlo, pero lo haríamos invocando al compilador con la opción `-lc`.

`libm.a` Es la biblioteca que contiene funciones de cálculo matemático y trigonométrico: `sqrt`, `pow`, `hypot`, `cos`, `atan`, etc. . Para hacer uso de las funciones de esta biblioteca habría que incluir en nuestro fichero `<math.h>`.

El montaje contra esta biblioteca se realiza invocando al compilador con la opción `-lm`.

`ar -opts [member] arch files...` _____ *Manage Archive Files*

Cuando desee realizar medianos o grandes proyectos, podrá realizar sus propias bibliotecas utilizando esta herramienta. Es una utilidad para la creación y mantenimiento de bibliotecas de ficheros.

Para usar una biblioteca, se especifica en la línea de compilación la biblioteca (por convención `libnombre.a`) en vez de los objetos.

Algunas opciones frecuentemente usadas son:

`-d` Elimina de la biblioteca los ficheros especificados

`-r` Añade (o reemplaza si existe) a la biblioteca los ficheros especificados. Si no existe la biblioteca se crea

`-ru` Igual que -r pero sólo reemplaza si el fichero es más nuevo

`-t` Muestra una lista de los ficheros contenidos en la biblioteca

`-v` *Verbose*

`-x` Extrae de la biblioteca los ficheros especificados

A continuación se muestran algunos ejemplos del uso de este mandato.

Para obtener la lista de objetos contenidos en la biblioteca estándar de C, se ejecutaría:

```
ar -tv /usr/lib/libc.a
```

El siguiente mandato crea una biblioteca con objetos que manejan distintas estructuras de datos:

```
ar -rv $HOME/lib/libest.a pila.o lista.o
```

```
ar -rv $HOME/lib/libest.a arbol.o hash.o
```

Una vez creada la biblioteca, habría dos formas de compilar un programa que use esta biblioteca y además la matemática:

```
cc -o pr pr.c -lm $HOME/lib/libest.a
```

```
cc -o pr pr.c -lm -L$HOME/lib -lest
```

5.5. Constructor

Si el programa o aplicación que estamos desarrollando está convenientemente descompuesto en múltiples módulos, el proceso global de compilación puede ser automatizado haciendo uso de la herramienta `make`.

`make` ——— *Aplication Mantainer*

Esta utilidad facilita el proceso de generación y actualización de un programa. Determina automáticamente qué partes de un programa deben recompilarse ante una actualización de algunos módulos y las recompila. Para realizar este proceso, `make` debe conocer las dependencias entre los ficheros: un fichero debe actualizarse si alguno de los que depende es más nuevo.

La herramienta `make` consulta un fichero (`Makefile`) que contiene las reglas que especifican las dependencias de cada fichero objetivo y los mandatos para actualizarlo. A continuación, se muestra un ejemplo:

Makefile

```
# Esto es un comentario
CC=gcc                  # Esto son macros
CFLAGS=-g
OBJS2=test.o prim.o

all: primes test        # Esta es la primera regla

primes: main.o prim.o   # Esta es otra regla
       gcc -g -o primes main.o prim.o -lm
       # Este es el mandato asociado

test: $(OBJS2)         # Aquí usamos las macros
       ${CC} ${CFLAGS} -o $@ ${OBJS2}

main.o prim.o test.o : prim.h # Esta es una dependencia.

clean:  # Esta no depende de nada, es obligatoria.
       rm -f main.o ${OBJS2}
```

Observe que las líneas que contienen los mandatos deben estar convenientemente tabuladas haciendo uso del tabulador, no de espacios. De no hacerlo así, la herramienta nos indicará que el formato del fichero es erróneo.

Ejecutaremos `make` para que se dispare la primera regla, o `make clean` explicitando la regla que queremos disparar. Entonces lo que sucederá es:

1. Se localiza la regla correspondiente al objetivo indicado.

2. Se tratan sus dependencias como objetivos y se disparan recursivamente.

3. Si el fichero objetivo es menos actual que alguno de los ficheros de los que depende, se realizan los mandatos asociados para actualizar el fichero objetivo.

Existen macros especiales. Por ejemplo, $@ corresponde con el nombre del objetivo actual. Asimismo, se pueden especificar reglas basadas en la extensión de un fichero. Algunas de ellas están predefinidas (p.ej. la que genera el .o a partir del .c).

Ejercicio 5.1.

5.6. Otras herramientas

Existen variedad de otras herramientas que podrán serle de utilidad cuando desarrolle código en un sistema UNIX Citaremos tres:

`gprof`
: Es una herramienta que nos permite realizar un perfil de la ejecución de nuestros programas, indicando dónde se pierde el tiempo, de manera que tendremos criterio para optimizar si es conveniente estas zonas de nuestro código.

`gcov`

Es semejante a la anterior, pero su objetivo es distinto. En la fase de verificación del programa, permite cubrir el código, es decir, asegurar que todas las líneas de código de nuestro programa han sido probadas con suficiencia.

`indent`

Permite endentar el ficheros fuente en C. Es muy parametrizable, para que el resultado final se corresponda con el estilo del propio autor.