

# Práctica 1: Gestión de parámetros, números pseudoaleatorios y medición de tiempos

## 1. Objetivos de la práctica

El **objetivo** principal de esta práctica es conocer y utilizar algunas funcionalidades de **C++** que serán de utilidad en las siguientes prácticas de la asignatura. En concreto, se cubrirán los siguientes aspectos de programación:

1. **Gestión de parámetros suministrados desde la línea de órdenes** en la que se invoca la ejecución de un programa.
2. **Generación de números pseudoaleatorios** para realizar pruebas con ellos.
3. **Medición del tiempo** empleado en ejecutar un segmento de código **C++**.

## 2. Tecnología y herramientas

En este apartado se presentan algunos recursos tecnológicos del lenguaje **C++** cuyo conocimiento nos resultará de utilidad en el desarrollo de algunos de los trabajos que se van a proponer en las prácticas de esta asignatura. Además del enunciado de la práctica, en el curso Moodle de la asignatura están disponibles ficheros con el código de apoyo para varios de los ejemplos.

### 2.1. Transmisión de datos a través de la línea de órdenes

Al invocar la ejecución de un programa es posible transmitirle información a través de la línea que contiene la orden para ejecutarlo.

De este modo, en la invocación al programa **listarArgumentos**, que se presenta a continuación, las cadenas “**estamos**”, “**en**”, “**enero**”, “**de**” y “**2025**” son datos que se transmiten al programa **listarArgumentos** en la misma línea que contiene la orden de ejecutar el programa.

```
$ ./listarArgumentos estamos en enero de 2025
```

Un posible diseño para este programa se presenta a continuación. El código fuente **listarArgumentos.cpp** se limita únicamente a presentar por pantalla un listado numerado de los datos dados en la línea de órdenes.

```
#include <iostream>
using namespace std;

// Pre: ---
// Post: Presenta por la salida estándar un listado numerado
//       de los argumentos dados por la línea de órdenes
int main(int argc, char* argv[]) {
    for (int i = 0; i < argc; ++i) {
        cout << i + 1 << ". " << argv[i] << endl;
    }
    return 0;
}
```

Al ser ejecutada la orden `./listarArgumentos estamos en enero de 2025`, el argumento `argc` (*argument counter*) de la función `main` toma como valor el número de cadenas (o secuencias de caracteres) dados por la línea de órdenes y separadas por espacios (por defecto), incluyendo también la cadena relativa al nombre del programa, `./listarArgumentos`. Es decir, en este caso `argc` tomará el valor 6.

El segundo argumento, `argv` (*argument vector*), es una referencia a un vector de cadenas de caracteres. En cada posición de este vector se almacena cada una de las cadenas dadas por la línea de órdenes. Así, en este ejemplo `argv[0]` almacena la cadena `./listarArgumentos`, `argv[1]` almacena la cadena `estamos`, `argv[2]` almacena la cadena `en`, `argv[3]` almacena la cadena `enero`, `argv[4]` almacena la cadena `de` y, finalmente, `argv[5]` almacena la cadena `2025`.

Por lo tanto, al ejecutar la orden `./listarArgumentos estamos en enero de 2025`, por pantalla se mostrará lo siguiente:

```
$ ./listarArgumentos estamos en enero de 2025
1. ./listarArgumentos
2. estamos
3. en
4. enero
5. de
6. 2025
```

## 2.2. Generación de números pseudoaleatorios

Los recursos que vamos a utilizar para generar números pseudoaleatorios se encuentran definidos en la biblioteca `cstdlib`. Únicamente vamos a utilizar dos funciones y una constante:

- La función `srand`(semilla) permite definir la semilla a partir de la cual inicializar la generación de números pseudoaleatorios. Para un valor de semilla determinado, se genera una misma secuencia de números pseudoaleatorios. Por ello, es conveniente

utilizar una semilla diferente como inicialización de dicha secuencia. Una forma práctica de resolver el problema es definir una semilla que dependa del instante en que se ejecuta la instrucción, por ejemplo, `srand(time(nullptr))`<sup>1</sup>. La función `time(nullptr)`, que está definida en la biblioteca predefinida `ctime`, devuelve un valor que representa el instante actual (para conocer los detalles conviene consultar el manual de las bibliotecas estándar predefinidas en C++).

- La función `rand()` devuelve un entero de tipo `int` generado de forma pseudoaleatoria y comprendido entre los valores 0 y `RAND_MAX`.
- La constante `RAND_MAX` es igual al máximo dato de tipo `int` que puede ser representado y depende de cada implementación del lenguaje.

Las siguientes líneas de código muestran cómo se puede generar un entero pseudoaleatorio en el intervalo  $[a, b]$  donde  $a$  y  $b$  son los argumentos de la línea de instrucciones:

---

```
#include <iostream>
#include <ctime>
using namespace std;

// Pre:  $a \leq b \wedge b - a \leq \text{RAND\_MAX}$  donde  $a$  y  $b$  son los dos primeros argumentos
// Post: Escribe un entero pseudoaleatorio en el intervalo  $[a, b]$ 
int main(int argc, char* argv[]) {
    if (argc == 3) {
        srand(time(nullptr));
        int a = atoi(argv[1]);
        int b = atoi(argv[2]);
        cout << a + rand() % (b-a+1) << endl;
    } else {
        cout << "La invocacion debe tener dos parametros numericos" << endl;
    }
    return 0;
}
```

---

De manera *similar*, el siguiente programa muestra cómo se puede generar un real pseudoaleatorio en el intervalo  $[a, b]$  donde  $a$  y  $b$  son los argumentos de la línea de instrucciones:

---

<sup>1</sup>En algunos tutoriales de Internet u otras implementaciones puedes encontrar la llamada a la función `time()` con el parámetro `NULL`. El valor `NULL` viene del lenguaje de programación C, y aunque también es válido, puede resultar algo confuso, dado que sirve para representar un valor nulo (valor 0) como un puntero nulo. Con C++ se introdujo `nullptr` para representar un puntero nulo mediante un tipo de dato específico. Si te interesa, puedes leer más acerca de esta decisión en <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2431.pdf>.

```
#include <iostream>
#include <ctime>
using namespace std;

// Pre:  $a \leq b \wedge b - a \leq RAND\_MAX$  donde  $a$  y  $b$  son los dos primeros argumentos
// Post: Escribe un real pseudoaleatorio en el intervalo  $[a,b]$ 
int main(int argc, char* argv[]) {
    if (argc == 3) {
        srand(time(nullptr));
        double a = atof(argv[1]);
        double b = atof(argv[2]);
        cout << a + ((double)rand()/RAND_MAX)*(b-a) << endl;
    } else {
        cout << "La invocacion debe tener dos parametros numericos" << endl;
    }
    return 0;
}
```

### 2.3. Medida del coste en tiempo

A la hora de medir el tiempo de ejecución de un código, vamos a distinguir entre dos conceptos:

- *Tiempo real (Wall time, clock time, wall-clock time)*: Tiempo total que transcurre desde que comienza la ejecución hasta que termina. Este concepto incluye, por ejemplo, el tiempo que un usuario tarda en introducir un dato.
- *Tiempo de CPU (CPU time)*: Tiempo que la CPU ha estado ocupada. Se excluyen los tiempos de espera de las operaciones de entrada/salida.

En Linux, es muy sencillo medir ambos tiempos para un ejecutable cualquiera mediante la instrucción *time*. Por ejemplo, la siguiente instrucción calcula los tiempos del ejecutable *mTiempo*:

```
$ time ./mTiempo
Escribe un numero entero: 3
El doble es: 6

real    0m2,048s
user    0m0,637s
sys     0m0,004s
```

donde *real* se refiere al tiempo real, *user* al tiempo de CPU, y *sys* al tiempo del sistema.

Si se desea medir el tiempo de un fragmento concreto de código, entonces se deben incluir en el código instrucciones que permitan realizar esa medición. De las diversas

maneras que existen para medir tiempos, en estas prácticas nos centraremos en el uso de `clock_gettime()`<sup>2</sup>.

El siguiente ejemplo, muestra cómo se puede medir el tiempo real de un fragmento de código utilizando `clock_gettime()`:

---

```
#include <iostream>
#include <time.h>
using namespace std;

int main () {

    ...
    struct timespec inicio, fin;
    clock_gettime(CLOCK_REALTIME, &inicio);

    ... // Código cuyo tiempo se desea medir

    clock_gettime(CLOCK_REALTIME, &fin);
    long segs = fin.tv_sec - inicio.tv_sec;
    long nanosegs = fin.tv_nsec - inicio.tv_nsec;
    double totalSegs = segs + nanosegs*1e-9;

    cout << "Tiempo real:" << totalSegs << " segundos" << endl;
    ...
}
```

---


Para medir el tiempo de CPU, simplemente debe sustituirse `CLOCK_REALTIME` por `CLOCK_PROCESS_CPUTIME_ID`.

### 3. Trabajo a desarrollar en esta práctica

Para consolidar las ideas y el conocimiento sobre las herramientas presentadas en esta práctica, cada alumno debe desarrollar y poner a punto los tres programas que se detallan a continuación. Para la compilación de los ficheros fuentes se debe hacer uso de las instrucciones descritas en el anexo sobre los primeros pasos en un sistema operativo Linux.

El código de los ficheros de estos programas se ubicará en tu directorio `programacion2/practica1`.

#### 3.1. Tiempo de reacción

El primer programa ejecutable a desarrollar, **tiempoReaccion**, pedirá al usuario cierto número de veces que pulse la tecla de fin de línea (tecla ) , informando cada

---

<sup>2</sup>Solo disponible en Linux

una de las veces del tiempo transcurrido (en segundos) entre la aparición del mensaje de la petición y el momento en que el usuario ha pulsado la tecla pedida.

El número de veces que el programa pide al usuario que pulse la tecla de fin de línea se ha de especificar como argumento en la línea de órdenes. Este argumento es opcional. Si en la línea de órdenes se omite el argumento numérico relativo al número de veces, el programa se limitará a pedir al usuario que pulse la tecla de fin de línea una sola vez.

### Ejemplo de interacción deseada:

```
$ ./tiempoReaccion 3
1) Pulse la tecla de fin de linea, por favor ...
Su tiempo de reaccion ha sido de 2.19326 segundos

2) Pulse la tecla de fin de linea, por favor ...
Su tiempo de reaccion ha sido de 4.88755 segundos

3) Pulse la tecla de fin de linea, por favor ...
Su tiempo de reaccion ha sido de 1.0562 segundos
```

Sustituye ahora en tu programa `CLOCK_REALTIME` por `CLOCK_PROCESS_CPUTIME_ID`. Ejecuta el programa resultante e interpreta los resultados.

## 3.2. Generación de números pseudoaleatorios

El segundo programa a desarrollar, **generarTabla**, presentará por la salida estándar (la pantalla), en 10 columnas, una secuencia de números generados de forma pseudoaleatoria. El programa recibirá cuatro parámetros a través de la línea de órdenes:

- El primer parámetro será el carácter 'E' o 'R'. Si es 'E', entonces se generarán números enteros, si no, se generarán números reales.
- El segundo parámetro será un entero que representa el número de números que hay que generar.
- El tercer y cuarto parámetro serán valores numéricos, enteros si el primer parámetro es 'E' y reales si es 'R', que especifican el límite inferior y superior del intervalo en el que se generarán los números.

La generación de números aleatorios la realizarán las funciones:

---

```
// Pre: a <= b
// Post: randInt(a,b) es un entero generado aleatoriamente en el intervalo [a,b]
int randInt(const int a, const int b);

// Pre: a <= b
// Post: randDouble(a,b) es un real generado aleatoriamente en el intervalo [a,b]
double randDouble(const double a, const double b);
```

---

cuyas cabeceras se facilitan en el fichero `genNum.hpp`, disponible en código de apoyo, y que deberán ser implementadas en el fichero `genNum.cpp`. El programa `generarTabla` deberá llamar a dichas funciones implementadas en `genNum.cpp`.

**Ejemplos de ejecución:** (con 250 datos en el intervalo  $[1, 100]$ )

```
$ ./generarTabla E 22 7 177
165    14    58    139    48    20    82    47    155    57
 25    18    156    100    169    62    67    91    69    91
155    103
$ ./generarTabla R 31 -1.5 2.7
1.847  -0.663  -1.343  0.497  2.168  2.189  1.992  1.828  2.316  2.600
1.200   0.861   1.537  1.076  2.055  0.511  -0.569  -1.455  1.204  -0.194
1.754   2.093  -0.660  2.601  2.405  0.970  0.011  0.903  2.235  -0.309
1.417
```

### 3.3. Aproximando $\pi$

El tercer programa a desarrollar, **aproxPI**, debe aproximar el número  $\pi$  mediante la generación de puntos aleatorios en un cuadrado con coordenadas  $[0, 0]$ ,  $[1, 0]$ ,  $[1, 1]$ ,  $[0, 1]$ , ver Figura 1. Nótese que si los puntos se generan de manera aleatoria (uniforme), entonces la probabilidad de que un punto esté dentro del cuarto de circunferencia dibujado es igual a  $\frac{\pi}{4}$ . Se debe utilizar este hecho a la hora de diseñar el programa **aproxPI**.

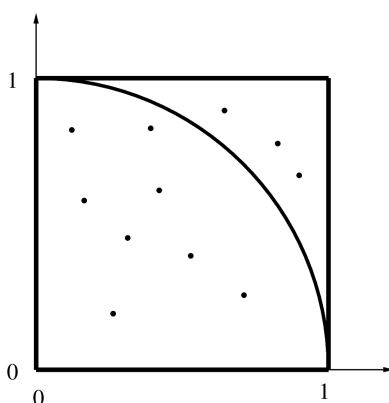


Figura 1: Esquema para la aproximación del número  $\pi$ .

El programa recibirá tres parámetros de tipo entero, *minN*, *paso* y *nIter*, de la línea de órdenes donde:

- *minN* es el número de puntos aleatorios a generar inicialmente.
- *paso* es el incremento de número de puntos a generar en cada iteración.
- *nIter* es el número de iteraciones, es decir, el número de veces que se va a aproximar  $\pi$ .

Por ejemplo, los parámetros  $minN = 20$ ,  $paso = 10$  y  $nIter = 3$ , querían decir que se van a realizar tres aproximaciones de  $\pi$ : una generando 20 puntos aleatorios, otra generando 30, y una tercera generando 40.

Para cada iteración, se debe mostrar por pantalla el valor aproximado calculado, el error relativo en tanto por ciento con respecto al valor "verdadero" de  $\pi$  y el tiempo de CPU requerido.

### Ejemplo de ejecución:

```
$ ./aproxPi 10 200000 12
n=10; PI (estimada)=3.2; Error relativo(%)=1.85916; Tiempo CPU(s)=1.5878e-05
n=200010; PI (estimada)=3.14018; Error relativo(%)=0.044871; Tiempo CPU(s)=0.0259823
n=400010; PI (estimada)=3.14504; Error relativo(%)=0.109776; Tiempo CPU(s)=0.0200913
n=600010; PI (estimada)=3.14071; Error relativo(%)=0.0279582; Tiempo CPU(s)=0.0276937
n=800010; PI (estimada)=3.14388; Error relativo(%)=0.0726717; Tiempo CPU(s)=0.0328496
n=1000010; PI (estimada)=3.14214; Error relativo(%)=0.017441; Tiempo CPU(s)=0.0409514
n=1200010; PI (estimada)=3.1394; Error relativo(%)=0.0697783; Tiempo CPU(s)=0.0501653
n=1400010; PI (estimada)=3.1424; Error relativo(%)=0.0258027; Tiempo CPU(s)=0.0538063
n=1600010; PI (estimada)=3.14298; Error relativo(%)=0.044172; Tiempo CPU(s)=0.0655207
n=1800010; PI (estimada)=3.14161; Error relativo(%)=0.000456376; Tiempo CPU(s)=0.0767599
n=2000010; PI (estimada)=3.14315; Error relativo(%)=0.0497082; Tiempo CPU(s)=0.0782784
n=2200010; PI (estimada)=3.14246; Error relativo(%)=0.0277326; Tiempo CPU(s)=0.0864525
```

Como valor "verdadero" de  $\pi$  se puede tomar el de la constante  $M\_PI$ , el cual requiere la inclusión de las siguientes líneas en el fichero fuente:

```
#define _USE_MATH_DEFINES

#include <cmath>
```

## 3.4. Resultados del trabajo desarrollado en la primera práctica

Como resultado de esta primera práctica, cada alumno dispondrá en su cuenta de un directorio (carpeta) denominado **programacion2** dentro del cual se encontrarán el directorio y los ficheros que se detallan a continuación.


- Carpeta **programacion2/practica1** con los siguientes ficheros fuentes:
  - Fichero **tiempoReaccion.cpp** (primer programa).
  - Fichero **generarTabla.cpp** (segundo programa).
  - Fichero **genNum.hpp** (segundo programa).
  - Fichero **genNum.cpp** (segundo programa).
  - Fichero **aproxPI.cpp** (tercer programa).

## 4. Anexo. Primeros pasos en un sistema operativo Unix o Linux

Los siguientes párrafos presentan una introducción al manejo del sistema operativo **Unix** mediante línea de comandos. Se recomienda leer el texto a la vez que se ejecutan las órdenes que se describen en él. Lo que aquí se presenta es también de aplicación



para un sistema operativo **Linux**. Al margen de otras diferencias, **Unix** es un sistema de pago, mientras que **Linux** es gratuito.

Para ejecutar una orden **Unix** o **Linux**, basta escribir la orden en la terminal o línea de comandos y pulsar la tecla de finalización de línea (tecla ).

#### 4.1. Cómo iniciar y finalizar una sesión de trabajo

Se puede trabajar en uno de los ordenadores del laboratorio con **Linux** (distribución CentOS) o desde tu ordenador personal (si tiene **Linux** o **Unix**).

Una vez hayas accedido a tu cuenta personal de usuario en el sistema operativo CentOS del laboratorio de prácticas (o en tu ordenador personal), tienes que abrir una ventana de emulador de terminal o línea de comandos.

Una línea de comandos (también llamada *shell* en inglés), se refiere a un programa que recoge los comandos introducidos por teclado por el usuario y se los manda al sistema operativo para que realice las operaciones que sean pertinentes. La mayoría de los sistemas Linux vienen con una shell por defecto llamada “sh”, desarrollada para sistemas Unix por Steve Bourne.

Una terminal (o emulador de terminal) es una ventana gráfica que nos permite interactuar con la shell. Existen muchos emuladores de terminales en Linux. Por ejemplo, KDE usa **konsole** mientras que GNOME usa **gnome-terminal**. Procede ahora a abrir una ventana de emulador de terminal. Para ello, localiza el icono de la aplicación **Terminal**, situado en la parte superior de la ventana. Una vez pulsado, se abrirá una nueva ventana con el nombre **Terminal**.

Tras abrir esta aplicación aparecerá en pantalla una secuencia de caracteres de aviso, que dependerá de la máquina en la que se esté trabajando y de cómo se haya configurado. Se mostrará algo similar a:

```
XXXXX : ~ $
```

Esto es lo que se conoce como el indicador de comandos de shell (*shell prompt*, en inglés). Su estructura es similar, independientemente del sistema operativo. Suele ser algo tipo **nombre-usuario@nombre-máquina** y el signo del dólar (símbolo \$). En el caso de que el último símbolo fuera el signo “#”, indica que esa sesión de terminal tiene permisos de superusuario. Esto quiere decir que todos los comandos que se realicen se hacen con el máximo nivel de privilegios. El superusuario, en un sistema Linux, es equivalente a un usuario administrador en un sistema Windows: es decir, tiene el poder de “hacer y deshacer” a su antojo.

A partir de este momento el usuario puede ejecutar órdenes **Unix** desde su **cuenta de trabajo** hasta que finalice la **sesión de trabajo**.

Al concluir el trabajo, el usuario debe **cerrar la sesión** antes de abandonar el terminal para evitar que una persona no autorizada pueda acceder a su cuenta de trabajo. Ello se logra mediante la orden **Unix exit**.

```
$ exit      <<-- fin de la sesión de trabajo
```

Para volver a trabajar será necesario iniciar una nueva sesión de trabajo de la forma que se ha indicado anteriormente.

## 4.2. Directorios y ficheros en Unix (o en Linux)

La información se almacena en **ficheros** y estos se organizan en **directorios**. El concepto de directorio es análogo al de carpeta en un sistema operativo basado en iconos. Cuando accedes a tu cuenta, el sistema te sitúa en tu directorio personal.

Ese directorio se conoce normalmente como tu **HOME**. En cualquier momento, escribiendo la orden **pwd** (**p**rint **w**orking **d**irectory) en la terminal puedes conocer el nombre completo del directorio en que te encuentras.

```
$ pwd <-- muestra la ruta desde la raiz al directorio actual
```

Un directorio puede contener otros directorios, así como ficheros, de forma similar a como una carpeta puede almacenar otras carpetas junto a una colección de ficheros. La manera natural de imaginarse esta organización es considerarla como un árbol de directorios y ficheros (véase el diagrama de la Figura 2).

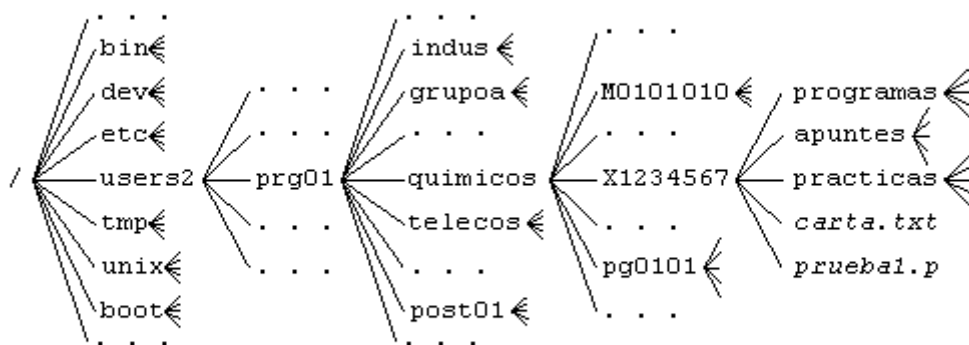


Figura 2: Árbol de directorios y ficheros.

El nombre completo de un directorio consta de la ruta o camino desde el directorio raíz / hasta el propio directorio. El nombre completo de un fichero es el resultado de concatenar la ruta desde el directorio raíz / hasta el directorio que lo aloja, seguida del nombre del fichero.

Para ilustrarlo, se presentan a continuación los nombres de algunos de los directorios y ficheros similares a los mostrados en la figura anterior.

```

/                                     <- directorio raiz
/users2
  
```

```
/users2/prg02
/users2/prg02/infor
/users2/prg02/infor/X1234567          <- HOME de X1234567
/users2/prg02/infor/X1234567/practicas <- directorio de prácticas
/users2/prg02/infor/X1234567/carta.txt <- fichero carta.txt
/users2/prg02/infor/X1234567/prueba1.cpp <- fichero prueba1.cpp
```

Para inspeccionar el contenido de un directorio (en este caso, el de tu **HOME**), se utiliza la instrucción **ls** (**list**). Si listas el contenido de tu directorio y no aparece nada, eso significa que, en ese momento, tu directorio aún está vacío.

```
$ ls          <-- lista el contenido del directorio actual
```

### 4.3. Órdenes Unix o Linux

Las órdenes que se van a presentar en los apartados que siguen son válidas tanto en un sistema **Unix** como también en un sistema **Linux**.

Antes de presentar nuevas órdenes conviene advertir que tanto el sistema operativo **Unix** como **Linux distinguen entre mayúsculas y minúsculas**. Es decir, una orden de terminal escrita con letras mayúsculas será diferente de la misma orden escrita con letras minúsculas. Prueba a ejecutar la orden **PWD** en la terminal y verás como la terminal te informa de que la orden no se ha reconocido (no existe ningún programa asociado a esa orden).

Existen multitud de órdenes o instrucciones en **Unix**. Para ver cómo funcionan vamos a experimentar con algunas de ellas.

Hay órdenes que no requieren ningún parámetro. Ejecuta, por ejemplo, la orden **date** que informa sobre la fecha y hora actuales.

```
$ date          <-- presenta la fecha y hora actual
Mon Feb 13 11:51:14 CET 2023
```

Sin embargo, la mayoría de las órdenes **Unix** requieren de algún parámetro para trabajar. Estos parámetros se le pasan a la instrucción como argumentos en la misma línea de la instrucción, separados por espacios. Por ejemplo, la instrucción **ls** permite conocer el contenido de la colección de directorios que se enumeren en la propia orden:

```
$ ls /home /etc
```

La salida estándar de cualquier instrucción **Unix** es, por defecto, la pantalla del terminal. No obstante, la salida de resultados puede dirigirse hacia un fichero de texto de nueva creación (para almacenarla en él) mediante el operador **>** seguido del nombre del fichero, de la siguiente forma:

```
$ ls /etc > resultados
```

Si ahora listas el contenido de tu directorio (orden **ls**), verás un fichero de texto con el nombre **resultados**.

```
$ ls
```

Para visualizar el contenido de un fichero de texto, podemos utilizar la instrucción **cat** (**concatenate**):

```
$ cat resultados <<-- presenta el texto seleccionado
```

Verás que el texto almacenado en el fichero es presentado por el terminal, pero como tiene más líneas de las que caben en la pantalla, en el listado final quedan ocultas una parte de las líneas. Para ver el fichero, página a página, podemos ejecutar la orden **more**:

```
$ more resultados <<-- presenta el texto seleccionado
```

Comprueba que la salida se detiene tras presentar una página, esperando que pulses la barra espaciadora para mostrar la siguiente página. Si quieres cancelar la visualización, pulsa la tecla **Q** o pulsa simultáneamente las teclas **Ctrl** y **C**, es decir, pulsa **Ctrl**+**C**.

La combinación de teclas **Ctrl**+**C** se utiliza frecuentemente cuando se desea **poner fin (abortar)** una orden o un programa en ejecución.

Hasta ahora hemos visto órdenes, con y sin argumentos, que implican la ejecución de una operación determinada. El comportamiento de una orden **Unix** puede ser modificado o matizado mediante lo que se denominan **opciones**.

Las opciones se le pasan a una instrucción como si fueran argumentos, viniendo siempre precedidas por el carácter guión (carácter '-').

Hemos visto la orden **ls** para listar el contenido de un directorio. Probemos ahora a ejecutar la siguiente orden.

```
$ ls -l <<-- presenta el contenido del directorio
```

El listado mostrado es ligeramente distinto ya que presenta más información que la anterior.

Para conocer todas las opciones disponibles y los argumentos necesarios de una orden se puede utilizar la ayuda de **Unix** mediante la orden **man** (**manual**) o acudir a Internet.

```
$ man ls <<-- manual de ayuda para explicar la orden "ls"
```

#### 4.4. Trabajo con ficheros y directorios en Unix o en Linux

Para copiar ficheros se utiliza la orden **cp** (**copy**):

```
$ cp resultados copiaResultados  
$ ls -l
```

La orden **rm** (**remove**) borra o elimina ficheros. Debe de extremarse el cuidado al ejecutar una orden de borrado, ya que no es posible la vuelta atrás en caso de error. Es decir, los ficheros borrados están definitivamente perdidos.

```
$ rm copiaResultados <<-- elimina el fichero
```

Si necesitas nuevos directorios para organizar tus ficheros, los puedes crear dentro de tu directorio con la orden **mkdir** (**make directory**). Crearemos el directorio *pruebas*.

```
$ mkdir pruebas <<-- crea un nuevo directorio
$ mkdir pruebas1 <<-- crea otro nuevo directorio
```

La orden **rmdir** (**remove directory**) borra o elimina directorios. Esta orden permite borrar directorios siempre y cuando se encuentren sin contenido (que no contengan ficheros u otros directorios en su interior).

```
$ rm pruebas1 <<-- elimina el directorio pruebas1
```

La orden **cd** (**change directory**) permite navegar por los directorios. Vamos a navegar por el directorio *pruebas* (**cd pruebas**) y otros más.

```
$ cd pruebas <<-- se sitúa en pruebas
$ cd / <<-- se sitúa en el directorio raíz /
```

La instrucción **cd** sin argumentos nos devuelve al directorio base de nuestra cuenta (al directorio **HOME**).

```
$ cd <<-- vuelve al directorio base de nuestra cuenta
```

La misma instrucción se puede expresar haciendo uso de la dirección simbólica **\$HOME** que hace referencia al directorio base de nuestra cuenta. El símbolo especial **~** también sirve para hacer referencia al directorio base de nuestra cuenta.

```
$ cd $HOME <<-- vuelve al directorio base de nuestra cuenta
```

La instrucción **cd ..** permite ir hacia atrás, volviendo al directorio padre del actual.

```
$ cd .. <<-- vuelve al directorio padre
```

El directorio que contiene al directorio en que estás (su directorio “padre”) se representa con **..** (dos puntos seguidos). El directorio en que estás se representa con **.** (un punto). Compruébalo ejecutando las siguientes órdenes:

```
$ ls -l . <<-- listado del directorio actual
$ ls -l .. <<-- listado del directorio padre del actual
```

Para trasladar un fichero de su directorio a otro directorio se utiliza la instrucción **mv** (**move**) expresando el nombre del fichero a trasladar como primer parámetro y el directorio de destino como segundo parámetro. Sitúate en tu directorio **HOME** y crea un fichero con un texto mediante la orden **cat** sin argumentos:

```
$ cat > nombreTexto
```

Ahora puedes ir escribiendo varias líneas de texto y la instrucción anterior las irá escribiendo en el fichero “**nombreTexto**”. Cuando decidas no introducir más texto, teclea **Ctrl** + **D**. Puedes comprobar con **ls** la existencia del fichero “**nombreTexto**”. Comprueba su contenido ejecutando la orden **cat** sobre el fichero.

Esta es una forma sencilla, pero muy limitada, de generar ficheros de texto. Lo normal es utilizar un editor de textos como por ejemplo **gedit**, que se estudiará un poco más adelante.

Para trasladar el fichero texto **nombreTexto** al directorio **pruebas**, escribe y ejecuta:

```
$ mv nombreTexto pruebas
```

Sitúate en el directorio **pruebas** y comprueba que el fichero está allí:

```
$ cd pruebas
$ ls -l
```

La instrucción **mv** tiene una segunda función, diferente de la anterior, ya que permite cambiar el nombre de un fichero, definiendo como primer parámetro el nombre del fichero a cambiar y como segundo parámetro su nuevo nombre. Cambia el nombre del fichero **nombreTexto** a **texto.txt**, comprueba que el cambio se ha efectuado y finalmente bórralo:

```
$ mv nombreTexto texto.txt
$ ls -l
$ rm texto.txt
```

Queremos señalar finalmente que cabe la posibilidad de utilizar el carácter ‘**\***’ en una orden **Unix** a modo de *comodín*.

```
$ rm p*0.txt <--
```

Esta orden eliminará todos los ficheros almacenados en el directorio cuyo sufijo sea **txt** y cuyo nombre comience por la letra **p** y acabe con el dígito **0**.

```
$ cp *.txt pruebas
```

La orden anterior hace una copia en el directorio **pruebas** de todos los ficheros almacenados en el directorio actual cuyo sufijo sea **txt**.

## 4.5. Organización de nuestro directorio de trabajo

En primer lugar vamos a crear en nuestro directorio de trabajo un subdirectorio denominado **programacion2** para almacenar en él y en sus subdirectorios todos los ficheros relacionados con la asignatura.

```
$ cd <-- nos situamos en nuestro directorio de trabajo
$ mkdir programacion2
```

A su vez, en el subdirectorio `programacion2` creamos el siguiente directorio:

```
$ cd $HOME/programacion2
$ mkdir practica1
```

Los ficheros con código **C++** o con datos que se faciliten para la realización de estas prácticas se pueden encontrar en la página del curso Moodle de la asignatura.

#### 4.6. Edición de programas fuente C++

Para crear ficheros fuente con código escrito en **C++** y poder modificarlos posteriormente, utilizaremos un editor de textos, por ejemplo `gedit`. Podemos invocarlo por primera vez para editar el código del fichero `saludo.cpp`.

```
$ cd $HOME/programacion2/practica1
$ gedit saludo.cpp &
```

Al invocar el editor `gedit` conviene no olvidar el último carácter escrito, `&`, ya que facilita que se abra una nueva ventana de edición, manteniendo activa la ventana de órdenes **Unix**. Si se omite dicho carácter, la ventana de órdenes **Unix** queda desactivada hasta que no se sale del programa editor `gedit`.

El código del fichero `saludo.cpp` lo podemos copiar del código de apoyo para esta práctica, disponible en el curso Moodle, y editar en la ventana del editor `gedit`.

```
#include <iostream>

using namespace std;

//Pre: ---
//Post: Escribe por la salida estándar "Un saludo, amigos"
int main() {
    cout << "Un saludo, amigos" << endl;
    return 0;
}
```

A menos que se especifique otra cosa, cuando se invoca a `gedit` con el nombre de un nuevo fichero, éste se creará en el directorio actual. Si allí ya existe un fichero con dicho nombre, lo abrirá para su edición. Si se quiere acceder o crear un fichero en un directorio distinto al actual, basta con anteponer al nombre del fichero su “camino” o ruta (**path**). El camino representa el nombre completo de una ruta, desde la raíz (o desde el directorio en que estamos) hasta un fichero particular, a través del árbol de directorios.

Supón que estamos en nuestro directorio **HOME**, en el que tenemos un directorio `programacion2`, y dentro de él otro llamado `practica1` en el que hay un fichero `saludo.cpp`, y que queremos hacer una listado por pantalla de este último fichero. Esto se podría realizar con los siguientes comandos:

```
$ cd
$ cd programacion2
$ cd practica1
$ cat saludo.cpp
```

O bien:

```
$ cd
$ cd programacion2/practica1
$ cat saludo.cpp
```

E incluso, sin movernos de nuestro directorio **HOME**:

```
$ cd
$ cat programacion2/practica1/saludo.cpp
```

Es decir, en la orden correspondiente se ha detallado el nombre del fichero, **saludo.cpp**, precedido por su ruta, **programacion2/practica1/**. Del mismo modo, también sería válido:

```
$ cat $HOME/programacion2/practica1/saludo.cpp
```

O también, otro equivalente:

```
$ cat ~/programacion2/practica1/saludo.cpp
```

## 4.7. Aprendiendo más de la terminal

En el curso Moodle de la asignatura encontrarás una guía más extensa, titulada “Manejo de terminal de Linux”, que te servirá para aprender más comandos y a sentirte más cómodo frente a una terminal de comandos de un sistema operativo tipo Linux.

## 5. Puesta a punto de un programa escrito en C++

Lo que se describe en los siguientes apartados se puede aplicar al trabajo de puesta a punto de programas en un computador con sistema operativo **Unix** o en un computador con sistema operativo **Linux**.

### 5.1. Compilación y ejecución de un programa escrito en C++

Vamos a comenzar editando, compilando y ejecutando el programa **C++** almacenado en el fichero **saludo.cpp**.



### 5.1.1. Compilación paso a paso de un programa escrito en C++

Tras situarnos en el directorio *programacion2/practica1*, compilaremos el fichero **saludo.cpp** ejecutando la orden `g++ saludo.cpp -c`. Esta orden compila el código fuente del programa y crea el fichero **saludo.o** que almacena el código objeto (binario) resultado de la compilación.

A continuación generaremos un programa ejecutable, al que vamos a dar el nombre **saludo**, ejecutando la orden `g++ saludo.o -o saludo`.

```
$ cd $HOME/programacion2/practica1
$ g++ saludo.cpp -c -std=c++11
$ g++ saludo.o -o saludo -std=c++11
```

**Recuerda que en las órdenes de compilación, ya sea de librerías o del programa principal, siempre vamos a añadir la opción de compilación `-std=c++11`.** En concreto, esta opción le indica al compilador de C++ que nuestros códigos fuente siguen la versión 11 del estándar de C++<sup>3</sup>. El uso de este estándar garantiza que se use la misma versión por el compilador de los laboratorios y por el compilador que tengas en tu ordenador personal. Adicionalmente, recuerda que el código que desarrolles durante el examen de prácticas ha de funcionar también con este estándar.

Se puede comprobar que, junto al fichero **saludo.cpp**, se encuentran los ficheros **saludo.o** y **saludo**, creados al ejecutar las dos órdenes anteriores.

Ya estamos en condiciones de ejecutar cuantas veces lo deseemos el programa ejecutable **saludo** mediante la orden `./saludo`:

```
$ ./saludo
Un saludo, amigos
$ ./saludo
Un saludo, amigos
$ ./saludo
Un saludo, amigos
$
```

Fíjate que para ejecutar el programa compilado será necesario añadir al principio de su nombre los caracteres `./`.

Si el programa que se desea compilar realiza llamadas a funciones definidas en un fichero con extensión **.hpp** e implementadas en el correspondiente fichero **.cpp**, se debe compilar primero los fuentes **.cpp** (se generan dos ficheros objetos) y posteriormente se genera el ejecutable. Por ejemplo:

```
$ g++ generarTabla.cpp -c -std=c++11
$ g++ genNum.cpp -c -std=c++11
$ g++ generarTabla.o genNum.o -o generarTabla -std=c++11
```

<sup>3</sup>Puedes consultar más información sobre este estándar en <https://es.wikipedia.org/wiki/C++11>.

### 5.1.2. Cómo abreviar la compilación de un programa escrito en C++

El proceso anterior se puede simplificar mediante una única orden que compila el programa fuente y crea un fichero que almacena el correspondiente programa ejecutable, borrando el objeto binario (fichero con extensión “.o”) resultante de la compilación. En el ejemplo que nos ocupa:

```
$ g++ saludo.cpp -o saludo2 -std=c++11
$ ./saludo2
Un saludo, amigos
$ ./saludo2
Un saludo, amigos
```

También es posible no especificar ningún nombre en la orden de invocación. Si no se especifica, el compilador `g++` por defecto creará el fichero compilado con el nombre “a.out”. Prueba a ejecutar las siguientes órdenes:

```
$ g++ saludo.cpp -std=c++11
$ ./a.out
Un saludo, amigos
$ ./a.out
Un saludo, amigos
```

La compilación de un programa que llama a funciones definidas en un fichero `.hpp` puede abreviarse de la siguiente manera:

```
$ g++ genNum.cpp -c -std=c++11
$ g++ generarTabla.cpp genNum.o -o generarTabla -std=c++11
```

o, alternativamente,

```
$ g++ genNum.cpp generarTabla.cpp -o generarTabla -std=c++11
```

## 5.2. Corrección de errores de un programa C++

Vamos a utilizar de nuevo el editor de textos `gedit` para provocar un par de errores en el código almacenado en el fichero `saludo.cpp`.

```
$ gedit saludo.cpp &
```

Recuerda al invocar el editor `gedit` no olvidar de nuevo escribir como último carácter ‘&’, ya que facilita que se abra una nueva ventana de edición, manteniendo activa la ventana de órdenes **Unix** (se dice que el editor se ejecuta en *background*).

Elimina del código fuente la línea la declaración `using namespace std;`.

Al compilar el programa `saludo.cpp`, el propio compilador informará de dos errores detectados, informando de la línea en la que se sitúan y dando un diagnóstico de cada uno de ellos.

```
$ g++ saludo.cpp -o saludo -std=c++11
saludo.cpp: In function 'int main()':
saludo.cpp:7:5: error: 'cout' was not declared in this scope
    cout << "Un saludo, amigos" << endl;
    ^~~~

...
saludo.cpp:7:36: error: 'endl' was not declared in this scope
    cout << "Un saludo, amigos" << endl;
                                ^~~~

...
```

Como se puede observar, el compilador avisa de que `cout` y `endl` no se encuentran declarados en el ámbito actual. Recuerda que este error se puede solucionar bien añadiendo “`std::`” como prefijo a `cout` y `endl`, o bien añadiendo la línea `using namespace std;` eliminada anteriormente.