

---

# Práctica 1: Introducción al desarrollo de programas C++

---

Programación-II

Dpto. de Informática e Ingeniería de Sistemas,  
Grado de Ingeniería Informática  
Escuela de Ingeniería y Arquitectura  
Universidad de Zaragoza

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

## 2. Preparación y desarrollo de la práctica

Antes de acudir a la sesión de laboratorio asociada a esta práctica conviene haber leído el texto de este enunciado.

Las prácticas se llevarán a cabo por parejas y serán supervisadas remotamente por los profesores de prácticas. Para su realización, se puede utilizar el entorno que se desee. Es decir, se puede utilizar el mismo que en Programación I u otro, por ejemplo diferente IDE y diferente sistema operativo. A modo informativo, en el anexo se detallan los pasos básicos para la realización en un sistema operativo **Linux**.

En la sesión de laboratorio correspondiente a esta práctica cada alumno deberá seguir el presente guión ejecutando las operaciones y tareas que en él se van describiendo,

---

comprendiendo su significado y preguntando al profesor o profesora de prácticas las dudas que le vayan surgiendo.

El desarrollo del trabajo de programación propuesto y la obtención de los resultados pedidos deberán completarse dentro de las dos semanas del mes de febrero en las que están programadas las sesiones de laboratorio asociadas a esta práctica.

### 3. 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.

#### 3.1. 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**”, “**febrero**”, “**de**” y “**2020**” son datos que se transmiten al programa **listarArgumentos** en la misma línea que contiene la orden de ejecutar el programa.

```
$ ./listarArgumentos estamos en febrero de 2020
```

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 febrero de 2020**, 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 **febrero**, `argv[4]` almacena la cadena **de** y, finalmente, `argv[5]` almacena la cadena **2020**.

Por lo tanto, al ejecutar la orden `./listarArgumentos estamos en febrero de 2020`, por pantalla se mostrará lo siguiente:

```
$ ./listarArgumentos estamos en febrero de 2020
1. ./listarArgumentos
2. estamos
3. en
4. febrero
5. de
6. 2020
```

### 3.2. Medida del tiempo desde un programa

A continuación, se va a explicar cómo medir desde un programa el **tiempo real** invertido en la ejecución de una secuencia de instrucciones y el **tiempo dedicado por el procesador** en la ejecución de una secuencia de instrucciones.

#### 3.2.1. Medida del tiempo invertido en ejecutar una secuencia de instrucciones

Los recursos a utilizar para medir el tiempo real empleado para ejecutar una secuencia de instrucciones de código se encuentran definidos en la biblioteca predefinida `ctime` y son los siguientes:

- Tipo de dato `time_t`. Un dato de este tipo representa una medida de tiempo con precisión de un segundo. Los datos de tipo `time_t` admiten operaciones aritméticas.
- La invocación `time(nullptr)` que devuelve el valor del tiempo actual como un dato de tipo `time_t`. La llamada a la función `time()` necesita un parámetro, de tipo puntero. En este caso, se está invocando con el puntero nulo (`nullptr`)<sup>1</sup>.

Para mayor información y detalle sobre estos recursos conviene consultar el manual de bibliotecas predefinidas en **C++**.

La forma de calcular el tiempo real invertido en ejecutar una secuencia de instrucciones `A1; ...; Ak;` se ilustra en el esquema mostrado a continuación.

```
// Declara la variable t1 y toma una primera medida del tiempo real
time_t t1 = time(nullptr);
A1; ...; Ak; // Sec. de instrucciones cuya duración se desea medir
// Declara la variable t2 y toma una segunda medida del tiempo real
```

---

<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>.

```
time_t t2 = time(nullptr);  
// Calcula la diferencia entre ambas medidas de tiempo y la transforma  
// en un int que expresa el tiempo transcurrido, en segundos  
int segundos = int(t2 - t1);
```

### 3.2.2. Medida del tiempo de procesador dedicado a ejecutar una secuencia de instrucciones

El tiempo real transcurrido durante la ejecución de una secuencia de instrucciones no coincide necesariamente con el tiempo que el procesador (CPU) del computador ha estado dedicado a ejecutarla. Esta diferencia se manifiesta especialmente en programas interactivos, cuando su duración depende de la rapidez con la que el operador facilita los datos solicitados por el programa.

Los recursos a utilizar para medir el tiempo que el procesador dedica a ejecutar una secuencia de instrucciones se encuentran también definidos en la biblioteca predefinida `ctime` y son los siguientes:

- Tipo de dato `clock_t`. Un dato de este tipo representa una medida del tiempo más precisa que la anterior función explicada. En concreto, `clock_t` expresa el tiempo en unas unidades denominadas **tics**.
- La constante `CLOCKS_PER_SEC`, cuyo valor es igual al número de **tics** que hay en un segundo. Este valor no es fijo, ya que depende de la implementación de cada compilador.
- La función `clock()` devuelve el valor del tiempo de procesador invertido en de ejecución del programa desde su inicio. El valor devuelto es un dato de tipo `clock_t`.

Para mayor información y detalle sobre estos recursos conviene consultar el manual de bibliotecas predefinidas en **C++**.

La forma de calcular el tiempo de procesador invertido en ejecutar una secuencia de instrucciones `A1; ...; Ak`; se ilustra en el esquema mostrado a continuación.

```
// Declara la variable t1 y toma el tiempo actual  
// del procesador al programa  
clock_t t1 = clock();  
A1; ...; Ak; // Sec. de instrucciones cuya duración se desea medir  
// Toma el tiempo actual en t2  
clock_t t2 = clock();  
// Tiempo transcurrido desde t1 hasta t2, en segundos (como float)  
float segundos = float(t2 - t1) / CLOCKS_PER_SEC;
```

### 3.3. Generación de datos pseudoaleatorios

Los recursos que vamos a utilizar para generar datos numéricos pseudoaleatorios se encuentran definidos en la biblioteca predefinida `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)). Tal como se ha mencionado anteriormente, la función **time**(nullptr) está definida en la biblioteca predefinida **ctime**. Devuelve un valor que representa el tiempo 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**, cuyo valor es igual al del máximo dato de tipo **int** que puede ser representado y depende de cada implementación del lenguaje.

El segmento de código C++ mostrado a continuación ilustra cómo es posible generar una secuencia de números enteros pseudoaleatorios.


```
// Pre: 1 <= n AND d <= h AND (h - d) <= RAND_MAX
// Post: El vector v[0,n-1] almacena n datos del intervalo [d,h]
//       generados de modo pseudoaleatorio
void generar(int v[], const int n, const int d, const int h) {
    time_t semilla = time(nullptr);           // tiempo actual como semilla
    srand(semilla);                           // Inicializa el generador de
                                              // núms. pseudoaleatorios

    for (int i = 0; i < n; ++i) {
        // Asigna a v[i] un pseudoaleatorio del intervalo [d,h]
        v[i] = d + rand() % (h - d + 1);
    }
}
```

## 4. 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 cuatro programas que se detallan a continuación. Para la compilación de los ficheros fuentes se debe hacer uso de las instrucciones y del fichero **Makefile** descritos en la sección 7. El código de los ficheros de estos programas se ubicará en el directorio **programacion2/practica1**.

### 4.1. Especificación del primer programa a desarrollar

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 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 línea, por favor ...
Su tiempo de reacción ha sido de 2 segundos

2) Pulse la tecla de fin de línea, por favor ...
Su tiempo de reacción ha sido de 6 segundos

3) Pulse la tecla de fin de línea, por favor ...
Su tiempo de reacción ha sido de 0 segundos
```

## 4.2. Especificación del segundo programa a desarrollar

El segundo programa ejecutable a desarrollar, **generarTabla01**, presentará por la salida estándar (la pantalla), en 10 columnas, una secuencia de datos enteros generados de forma pseudoaleatoria. En la propia línea de órdenes de invocación al programa se especificará el número de datos a generar y los valores enteros mínimo y máximo de los datos pseudoaleatorios generados.

**Ejemplo de interacción deseada:** (con 250 datos en el intervalo  $[1, 100]$ )

```
$ ./generarTabla01 250 1 100
 37   85    3   12   88   43   37   21   10   62
 54   33   13   52   61   11   93    7   26   48
    .    .    .
 28   83   19   74   62   50   49   92   35   66
```

Para realizar este programa, se deberá implementar y utilizar una función con la siguiente cabecera que devuelve un número aleatorio entero en el intervalo  $[a, b]$ :

```
int randInt(const int a, const int b);
```

## 4.3. Especificación del tercer programa a desarrollar

El tercer programa ejecutable a desarrollar, **generarTabla02**, presentará por la salida estándar (la pantalla), en 10 columnas, una secuencia de datos reales generados de forma pseudoaleatoria. En la propia línea de órdenes de invocación al programa se especificará el número de datos a generar y los valores reales mínimo y máximo de los datos pseudoaleatorios generados.

---

**Ejemplo de interacción deseada:** (con 500 datos en el intervalo  $[0.0, 10.0]$ )

```
$ ./generarTabla02 500 0.0 10.0
  3.165   3.298   9.483   . . .   7.577   3.682   6.387
  6.257   2.449   5.876   . . .   7.370   3.961   8.703
      . . .
  1.014   3.009   8.039   . . .   9.203   2.231   5.319
$ ...
```

Para realizar este programa, se deberá implementar y utilizar una función con la siguiente cabecera que devuelve un número aleatorio real en el intervalo  $[r, s]$ :

```
double randDouble(const double r, const double s);
```

#### 4.4. Especificación del cuarto programa a desarrollar

El cuarto programa a desarrollar, **medirCoste**, al ser invocado deberán aportarse tres parámetros en la línea de órdenes: el número de datos a generar, los valores mínimo y máximo de dichos datos.

Al lanzar su ejecución, el programa generará una secuencia del número de datos a generar dado, comprendidos en el intervalo dado. Después de generar estos datos, mostrará por pantalla los 20 primeros y los 20 últimos datos de dicha secuencia e informará que está procediendo a su ordenación (de menor a mayor valor). Tras haber concluido la ordenación de la secuencia, mostrará de nuevo los 20 primeros y los 20 últimos datos (que deberían de salir ahora ordenados). Por último, informará del tiempo de CPU (procesador) invertido en dicha ordenación.

**Ejemplo de interacción deseada:** (con 20000 datos en el intervalo  $[1, 10000]$ )

```
$ ./medirCoste 20000 1 10000

Datos a ordenar:
  181   2382   2675   4096   4352   7153   5827   6079   6656   8802
 7847   5045   2292   8447   510   1169   7722   2165   2609   9946
      . . .
 8145   3888   548   1177   4850   99   727   4320   3446   5041
 4865   2129   7316   3578   9252   8179   8151   2608   624   2925

Ordenando 20000 datos enteros ...

Datos ordenados:
   2    3    4    4    5    5    5    6    7    8
   8    9    9    9    9   10   11   12   12   13
      . . .
 9991   9991   9992   9992   9992   9993   9993   9994   9994   9995
 9995   9996   9996   9997   9997   9998 10000 10000 10000 10000

Tiempo de CPU para ordenar 20000 enteros: 11.060 segundos
```

---

En el diseño del programa **medirCoste** se hará uso de la función `ordenar(v,n)` facilitada en el código de apoyo de la práctica para ordenar de menor a mayor valor los datos de un vector de datos de tipo `int`.

#### 4.5. Resultados del trabajo desarrollado en la primera prácticas

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.

1. Directorio (carpeta) **programacion2/practica1** con los siguientes ficheros fuentes:
  - Fichero **tiempoReaccion.cpp** (primer programa).
  - Fichero **generarTabla01.cpp** (segundo programa).
  - Fichero **generarTabla02.cpp** (tercer programa).
  - Fichero **medirCoste.cpp** (cuarto programa).

### 5. ¿Qué hay que saber hacer con soltura tras realizar esta práctica?

A continuación, se enumeran los conocimientos prácticos que cada alumno debe adquirir como resultado del trabajo desarrollado en esta práctica. Este aprendizaje no puede aplazarse para más adelante ya que lo aprendido va a ser aplicado de forma continuada a partir de este momento.


- Cada alumno debe ser capaz de realizar las siguientes operaciones que intervienen en el desarrollo de un programa **C++**:
  - Editar ficheros que almacenen código **C++**.
  - Compilar un programa **C++**, analizar los posibles errores detectados por el compilador y, en su caso, corregirlos.
  - Ejecutar el programa, una vez haya sido compilado de forma satisfactoria.
  - Identificar y corregir los errores en el comportamiento del programa puestos de manifiesto al ejecutarlo, así como someterlo a un conjunto de pruebas suficientemente amplio y significativo, hasta tener la convicción de su correcto funcionamiento.
- Finalmente, cada alumno debe ser capaz de lo siguiente:
  - Diseñar programas que reciban y lean datos a través de la línea de órdenes.
  - Generar secuencias de datos enteros o reales pseudoaleatorios.
  - Diseñar programas capaces de deducir experimentalmente el coste de la ejecución - en tiempo real o en tiempo de procesador - de determinadas acciones algorítmicas.



---

## 6. 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 )

### 6.1. 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 un montón de 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:

```
XXXXXX : ~ $
```

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.

## 6.2. 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 raíz 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 1).

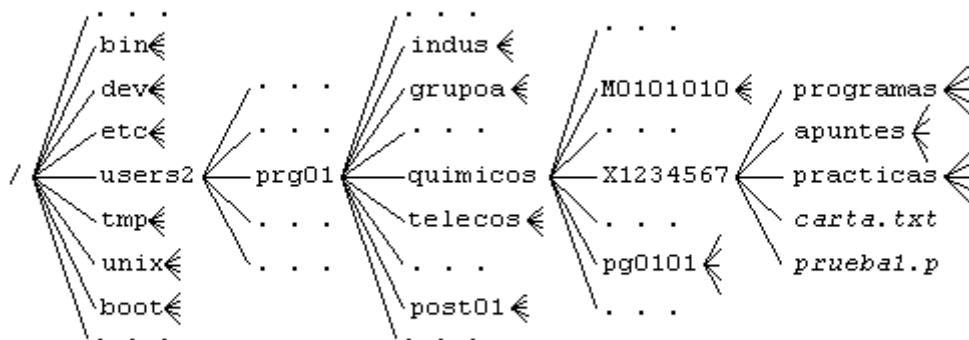


Figura 1: Árbol de directorios y ficheros.

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 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
```

### 6.3. 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 2019
```

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 /export /export/home
```

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 /export /proc > resultados
```

Si ahora listas el contenido de tu directorio (órden **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** (**man**ual) o acudir a Internet.

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

## 6.4. 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 /
$ cd /export/home <-- se sitúa en /export/home
$ pwd
/export/home/
```

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** (**m**ove) 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**.

---

## 6.5. 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 los siguientes nuevos directorios:

```
$ cd $HOME/programacion2
$ mkdir practica1
$ mkdir practica2
$ mkdir practica3
$ mkdir practica4
$ mkdir practica5
$ mkdir datos
```

- **practica1, practica2, ..., practica5** para almacenar el código fuente de los programas **C++** que desarrollemos en las diferentes prácticas de la asignaturas.
- **datos** para almacenar los ficheros de datos que utilicen o generen nuestros programas.

Los ficheros con código **C++** o con datos que se faciliten para la realización de estas prácticas se pueden encontrar en la página web de **Programación 2** (<http://webdiis.unizar.es/asignaturas/PROG2/>).

## 6.6. Edición de programas fuente C++

Para crear ficheros fuente con código escrito en **C++** y poder modificarlos posteriormente, utilizaremos el editor de textos **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 desde la carpeta **practica1**, accesible desde la web de **Programación 2**, y pegar 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 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
```



---

## 6.7. Aprendiendo más de la terminal

En la página web 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.

## 7. 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**.

### 7.1. 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**.

#### 7.1.1. 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 **saludo1**, ejecutando la orden `g++ saludo.o -o saludo1`.

```
$ cd $HOME/programacion2/practica1
$ g++ saludo.cpp -c -std=c++11
$ g++ saludo.o -o saludo1 -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>2</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 **saludo1**, creados al ejecutar las dos órdenes anteriores.

Ya estamos en condiciones de ejecutar cuantas veces lo deseemos el programa ejecutable **saludo1** mediante la orden `./saludo1`:

```
$ ./saludo1
Un saludo, amigos
```

---

<sup>2</sup>Puedes consultar más información sobre este estándar en <https://es.wikipedia.org/wiki/C++11>.

```
$ ./saludo1
Un saludo, amigos
$ ./saludo1
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++ ordenar.cpp -c -std=c++11
$ g++ medirCoste.cpp -c -std=c++11
$ g++ medirCoste.o ordenar.o -o medirCoste -std=c++11
```

### 7.1.2. 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 siguiente 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++ ordenar.cpp -c -std=c++11
$ g++ medirCoste.cpp ordenar.o -o medirCoste -std=c++11
```

o, alternativamente,

```
$ g++ ordenar.cpp medirCoste.cpp -o medirCoste -std=c++11
```

### 7.1.3. Compilación usando la herramienta make

Sin embargo, cuando en el programa hay que trabajar con múltiples ficheros, es mejor utilizar algún gestor de proyectos. Uno muy utilizado en Unix (también se puede usar en linux o Windows si se instala) es la herramienta **make**. La herramienta requiere un fichero de especificación que describa, mediante una serie de reglas, qué depende de qué y cómo se obtiene. Las reglas tienen la estructura

```
<objetivo>: <requisitos>
    <instrucciones para generar objetivo>
```

Por ejemplo, la siguiente regla

```
ordenar.o: ordenar.cpp ordenar.hpp
    g++ -c ordenar.cpp -std=c++11
```

define que el fichero **ordenar.o** depende de los ficheros **ordenar.cpp** y **ordenar.hpp**, y que se obtiene mediante la ejecución del comando **g++ -c ordenar.cpp -std=c++11**. El conjunto de reglas del proyecto se agrupan en un fichero, por ejemplo **miFicheroMake** y se ejecuta desde la línea de comandos mediante **make -f miFicheroMake**.

Tradicionalmente, si no hay conflictos, el fichero de reglas suele llamarse **Makefile**, de manera que la simple invocación **make**, sin parámetros:

```
$ make
```

es suficiente para ejecutar la primera regla que haya definida. Esta invocación es equivalente a ejecutar **make -f Makefile**). Si se desea ejecutar una regla en particular, por ejemplo **clean**, basta con escribir:

```
$ make clean
```

Para simplificar la escritura de las reglas del proyecto, la herramienta permite también definir variables, usar caracteres comodín, etc.

Una de las ventajas de haber definido las reglas y establecido qué depende de qué es que el proceso de generación del ejecutable es más rápido. La herramienta es capaz de construir un grafo de dependencias entre los ficheros involucrados de manera que, en caso de recompilación tras una modificación de uno de los ficheros fuente, solo las reglas involucradas por las dependencias se ejecutarán. Por ejemplo, si se ha modificado algún fuente del proyecto (se ha añadido una nueva función por ejemplo), pero no se ha modificado ni **ordenar.cpp** ni **ordenar.hpp**, y ya existe un **ordenar.o** de una compilación anterior, la instrucción **g++ -c ordenar.cpp -std=c++11** no se vuelve a ejecutar, por lo que se gana en tiempo de compilación. Esto es fundamental cuando se trabaja con proyectos grandes, con decenas y decenas de ficheros **cpp** y **hpp** y procesos de compilación muy largos.

---

Adjuntamos en el material un ejemplo de fichero **Makefile** bastante rudimentario, pero que cubre las necesidades de esta práctica. Leed la cabecera del propio fichero, donde se describe su uso.

#### 7.1.4. Cómo optimizar el código generado por un compilador de C++

Los compiladores pueden intentar realizar determinadas optimizaciones del código que generan. Ello redundará en una mayor eficiencia de los programas ejecutables, a costa de un mayor esfuerzo en la compilación.

Al ejecutar una orden de compilación se puede seleccionar el nivel de optimización del código generado mediante la opción **-O<sub>n</sub>**, donde **n** es un natural que expresa el nivel de optimización (a mayor nivel, mayor grado de optimización).

Hay cinco niveles posibles de optimización. Por defecto, no se realiza ninguna optimización en la compilación.

Prueba a compilar el fichero “saludo.cpp” con los diferentes niveles de optimización, generando diferentes ficheros binarios cada vez:

- Nivel 0 de optimización (optimización por defecto).

```
$ g++ saludo.cpp -O0 -o saludo0 -std=c++11
```

O bien, omitiendo la opción **-O0**:

```
$ g++ saludo.cpp -o saludo0 -std=c++11
```

- Nivel 1 de optimización.

```
$ g++ saludo.cpp -O1 -o saludo1 -std=c++11
```

- Nivel 2 de optimización.

```
$ g++ saludo.cpp -O2 -o saludo2 -std=c++11
```

- Nivel 3 de optimización.

```
$ g++ saludo.cpp -O3 -o saludo3 -std=c++11
```

- Nivel 4 de optimización.

```
$ g++ saludo.cpp -O4 -o saludo4 -std=c++11
```

Comprueba ahora el tamaño de los ficheros binarios generados (usando en la terminal la orden **ls -l**). Observarás algunas diferencias entre ellos, provocados por los algoritmos de optimización que ha aplicado el compilador **g++**.

---

## 7.2. 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.