

Presentación del laboratorio de PRO2

El laboratorio de la asignatura *Programació 2* (abreviadamente PRO2) tiene como objetivo ejercitar los conocimientos sobre diseño de programas impartidos en dicha asignatura y la precedente, *Programació 1*, y adquirir experiencia en la codificación y puesta a punto de programas de tamaño y dificultad medios.

Un cuatrimestre de este laboratorio se divide en dos partes. La primera consta de una serie de sesiones en las que se presentan las herramientas de programación necesarias para realizar la práctica de la asignatura y se resuelven pequeños ejercicios. La segunda parte consiste en el desarrollo de la práctica propiamente dicha.

Las mencionadas sesiones se describen en este documento, que es al mismo tiempo una guía de referencia y un manual de ejercicios. Salvo en la sesión 1, que se basa mayoritariamente en contenidos aparecidos en *Programació 1*, los elementos que presentaremos estarán muy relacionados con los conceptos que paralelamente se muestran en las clases de teoría. Terminadas las sesiones, el alumno estará en condiciones de abordar la práctica. El tiempo de clase se dedicará a la supervisión de la práctica por parte del profesor.

El entorno de trabajo sobre el que realizaremos las sesiones consta del sistema operativo Linux y el lenguaje de programación C++. Supondremos que el alumno ya está familiarizado con los conceptos básicos del lenguaje C++ (instrucciones de control, tipos simples, el tipo `vector`, etc.) y posee un mínimo dominio del sistema operativo Linux (por ejemplo, con editores y con la gestión de ficheros).

De todas formas, se puede encontrar un resumen de los comandos más importantes de Linux en el Racó de la FIB

http://www.fib.upc.edu/fib/serveis/guies/entorn_linux.html

Sobre el compilador de C++, recomendamos el g++ (<http://gcc.gnu.org>), a ser posible la versión 4.8.3 como en el linux de la FIB. Viene incorporado en todas las distribuciones normales de Linux y es actualizable via web.

Toda la información sobre la asignatura se encuentra en la página

<http://www.cs.upc.edu/pro2>

Dicha página incluye: un enlace a la guía docente, los apuntes de teoría, este manual, el enunciado de la práctica, etc.

También es importante mirar periódicamente los avisos del Racó de la FIB.

Sesión 1

Entorno de trabajo y repaso de C++

- Este documento contiene ejercicios que hay que resolver en el Jutge (en la lista correspondiente del curso actual) y que aquí están señalados con la palabra *Jutge*.
- Recomendamos resolver los ejercicios en el orden en el que aparecen en este documento. No se supervisarán los problemas del Jutge si antes no se han resuelto los ejercicios previos.

En esta primera sesión realizaremos algunos ejercicios para repasar la codificación de algoritmos en C++. Para empezar, copiad todo el subdirectorio `sessi01`:

```
cp -r /assig/pro2/sessi01 sessi01
```

1.1. Compilación y enlazado de programas en C++

Comenzaremos repasando el proceso de obtención de un fichero ejecutable de C++ (extensión `.exe`) a partir de un fichero fuente (extensión `.cc`).

Consideremos el fichero `PR02.cc`, contenido en `sessi01`. Para obtener el correspondiente `PR02.exe` aplicamos el comando

```
g++ -o PR02.exe PR02.cc
```

Si el fichero fuente no contiene errores, e incluye un método `main`, se generará el correspondiente ejecutable; la opción `-o` es la que nos permite elegir el nombre del mismo. El proceso implica generar el código máquina de nuestro programa y combinarlo con todos los elementos standard del lenguaje que necesita para funcionar.

Si un programa se compone de varios ficheros fuente y todos se encuentran en la carpeta en la que estamos trabajando, se pueden compilar todos juntos de la misma manera. Si el fichero `f1.cc` es el que contiene el método `main` y los demás ficheros implicados son `f2.cc ... fn.cc`, podemos hacer

```
g++ -o f1.exe f1.cc f2.cc ... fn.cc
```

Sin embargo, muchas veces resulta conveniente que cada uno de esos ficheros se compile de forma separada, con lo que el ejecutable no se obtendrá mediante una única instrucción como la anterior. Esto es particularmente relevante en programación modular, como la que introducimos en esta asignatura, pues cada módulo o clase dará lugar a ficheros que se habrán de compilar separadamente.

En esos casos, se trabaja en dos fases: primero se obtiene el código máquina de cada fichero fuente, dando lugar a los correspondientes ficheros *objeto* (extensión `.o`). Cada fichero objeto contendrá también la información necesaria para que su código máquina pueda usar o ser usado por el de otros ficheros objeto. Por último se combinan, o *enlazan*, todos esos ficheros objeto para formar el ejecutable.

En el ejemplo de `PRO2.cc` las dos fases serían:

```
g++ -c PRO2.cc --> compilación: produce el fichero objeto PRO2.o
g++ -o PRO2.exe PRO2.o --> linkado: enlaza PRO2.o con los elementos que le faltan
```

En un caso general haríamos

```
g++ -c f1.cc --> produce f1.o, y así sucesivamente
g++ -c f2.cc
...
g++ -c fn.cc
g++ -o f1.exe f1.o f2.o ... fn.o
```

En la próxima sesión iremos más allá: veremos ejemplos de programas que usan código contenido en ficheros fuente a los que no tendremos acceso (por tanto, no los podremos compilar nosotros mismos), pero sí a los correspondientes ficheros objeto.

Para garantizar una compilación en las mismas condiciones que el Jutge de PRO2, se puede definir un alias con las opciones correspondientes. Incluid esta definición en el fichero `.tcshrc` de vuestro directorio principal (ha de estar en una sola línea).

```
alias p2++ "g++ -D_GLIBCXX_DEBUG -ansi -O2 -Wall -Wextra -Werror -Wno-uninitialized
-Wno-sign-compare -std=c++0x"
```

Para usar dicha definición hay que hacer

```
p2++ -c PRO2.cc
```

Para conocer las opciones de compilación más importantes de g++ puede visitarse

<https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html>

La opción `-D_GLIBCXX_DEBUG` será comentada en un momento posterior de la sesión.

1.2. Ejecución de programas en C++

Los programas que codificaremos en la asignatura gestionarán su entrada/salida mediante el canal estándar, en ocasiones redirigido a ficheros de texto.

Las ejecuciones se realizarán desde la línea de comandos. Por ejemplo, si queremos ejecutar el programa contenido en el fichero `PRO2.exe`, recibiendo sus datos por teclado y escribiendo sus resultados en pantalla, tecleamos

```
./PRO2.exe
```

y a continuación escribimos los datos del programa respetando el formato esperado por éste. Tras la ejecución, veremos escrita la salida del mismo.

Si deseamos ejecutar el mismo programa sobre los datos almacenados en el fichero `PRO2.dat`, pero queremos ver los resultados por pantalla, redireccionaremos el canal estándar de entrada hacia ese fichero:

```
./PRO2.exe <PRO2.dat
```

Si además queremos que los resultados no se escriban en pantalla sino en otro fichero `PRO2.sal` ejecutaremos

```
./PRO2.exe <PRO2.dat >PRO2.sal
```

Éstos son los mecanismos de redirección de los canales estándar de entrada y salida en Linux. Puede usarse cualquier combinación de ellos cuando sea preciso.

Tenéis un segundo ejemplo, puramente numérico, en el fichero `suma.cc`. Probad con él todo el proceso de compilación, linkado y ejecución.

1.2.1. Ejercicio: suma de una secuencia de enteros

Modificad el programa `suma.cc` para que sume una secuencia de enteros terminada en 0. Leed y sumad los números en el mismo método `main`, pero de forma que cada número se pierda una vez sumado (no vale almacenar primero los números en un vector y luego sumarlos). Probadlo de forma interactiva y mediante redirección de la entrada. En el fichero `sumasec-ejemplos.txt` tenéis algunas posibles entradas con sus correspondientes salidas.

1.3. Estructura de un programa en C++

Un programa en C++ puede consistir en un único fichero (normalmente con extensión `.cc` o `.cpp`; en esta asignatura optamos por la extensión `.cc`). En ese caso, el fichero ha de tener la siguiente estructura.

- inclusiones y espacio de nombres
- definiciones de constantes o tipos

- procedimientos
- programa principal (main)

Las inclusiones permiten emplear en el programa elementos definidos en otro lugar. Las más típicas proceden de clases o librerías del propio C++:

```
#include <iostream>: canales standard de entrada y salida
```

```
#include <vector>: clase vector
```

```
#include <string>: clase string
```

```
#include <cmath>: funciones matemáticas
```

Sin embargo, en cuanto un programa adquiere un cierto tamaño seguramente requerirá una cierta descomposición modular. C++ ofrece un mecanismo esencial para obtener programas modulares: la **clase**. En PRO2 usaremos clases para traducir módulos de datos. Para cada programa C++ nos tocará obtener un fichero `.cc` que contendrá el correspondiente `main` y varios ficheros más que contendrán las diversas clases que necesitemos.

En un programa codificado en C++ pueden aparecer una o varias clases, que clasificamos en los siguientes grupos:

- las clases estándar del lenguaje (`iostream`, `vector`, etc)
- las clases definidas en el directorio en el que estemos
- las clases definidas en otros directorios

Para poder usar una clase que no sea estándar y que no tengamos definida en nuestro directorio, hemos de informar al compilador del lugar donde se encuentra. Veremos cómo se hace en próximos capítulos.

Otra posibilidad para descomponer y reutilizar código consiste en distribuirlo en varios ficheros, compilar éstos separadamente y usar solo los componentes que nos interesen en cada momento. Veremos un ejemplo en esta misma sesión.

1.4. Acciones y funciones; paso de parámetros

Un requisito de la asignatura es que todas las variables que intervengan en las acciones y funciones que diseñemos han de aparecer en la cabecera de éstas o bien ser locales (no se permiten variables globales). Mantendremos esta línea al codificar nuestros algoritmos en C++, si bien deberemos adaptarnos a las particularidades de dicho lenguaje.

Independientemente del lenguaje de programación empleado, podemos clasificar los parámetros de una acción de la siguiente manera (en una función todos los parámetros son de *entrada*):

- De *entrada*: su valor inicial permanece inalterado tras la ejecución de la acción, aunque durante ésta se haya modificado
- De *entrada/salida*: su valor inicial es relevante para la ejecución de la acción pero puede quedar modificado tras la misma
- De *salida*: su valor inicial es irrelevante para la ejecución de la acción (muchas veces ni siquiera se conoce) y su valor final es creado por la propia acción

En C++ existen las siguientes opciones, que permiten traducir la clasificación anterior con bastante claridad:

1. *Parámetros por valor*. La acción recibe una copia del parámetro y trabaja con ella. Al acabar, dicha copia se destruye. El valor del parámetro no cambia, por lo que este mecanismo trata a los parámetros como de *entrada* según la clasificación anterior.

Para indicar que un parámetro se pasa por valor no hay que hacer nada.

2. *Parámetros por referencia*. La acción recibe la dirección de memoria del parámetro y toda modificación del mismo es permanente. Respecto a la clasificación anterior, este mecanismo trata a los parámetros como de *salida* si al entrar en la acción no tienen valor (por ejemplo, si no han sido inicializados) o su valor es irrelevante. En caso contrario, tendríamos un parámetro de *entrada/salida*.

Para indicar que un parámetro se pasa por referencia, se precede su nombre de un signo &.

3. *Parámetros por referencia constante*. Al igual que con los parámetros por referencia, la acción recibe la dirección de memoria del parámetro, pero si intenta modificar su valor el compilador da un error, por lo que de hecho funcionan como parámetros por valor. Este mecanismo es especialmente indicado para usar vectores y otras estructuras complejas como parámetros de *entrada*.

Para indicar que un parámetro se pasa por referencia constante, se precede su nombre de un signo & y su tipo de la palabra `const`.

Ejemplos:

1. Cabecera de una acción que intercambia los valores de las variables enteras *m* y *n*. Como deseamos que ambos parámetros cambien de valor, los definimos como de *entrada/salida*. Al realizar la codificación en C++, han de pasarse por referencia.

```
void intercambiar (int &m, int &n)
/* Pre: cierto */
/* Post: m y n tienen sus valores intercambiados respecto a los iniciales */
```

2. Cabecera de una acción que intercambia los valores de las posiciones *i* y *j* de un vector *v*. Claramente el vector es un parámetro de *entrada/salida*, pues deseamos modificarlo, mientras que las posiciones son sólo de *entrada*. La codificación en C++ sería así

```
void intercambiar_vect (vector<int> &v, int i, int j)
/* Pre: 0 <= i, j < v.size() */
/* Post: v[i] y v[j] tienen sus valores intercambiados respecto a los iniciales */
```

Por la mencionada característica de C++, aseguramos que el vector será modificado de forma permanente y los enteros no.

3. Cabecera de una función que busca un elemento x en un vector v y devuelve el resultado en un booleano. Notad que, al tratarse de una función, los parámetros son de entrada y podrían traducirse a C++ como parámetros por valor. Sin embargo, en el caso del vector, es conveniente pasarlo por referencia constante, para ahorrarnos la copia del mismo.

```
bool busqueda_lin (const vector<int> &v, int x)
/* Pre: cierto */
/* Post: el resultado indica si x está en v */
```

4. Cabecera de una función que obtiene los valores máximo y mínimo de un vector v . Para poder devolver dos resultados hemos de usar un `struct` o un `pair`.

```
struct parint {
    int prim, seg;
};

parint max_min (const vector<int> &v)
/* Pre: v.size()>0 */
/* Post: el primer componente del resultado es el valor máximo de v;
        el segundo componente del resultado es el valor mínimo de v */

pair<int,int> max_min (const vector<int> &v)
/* Pre: v.size()>0 */
/* Post: el primer componente del resultado es el valor máximo de v;
        el segundo componente del resultado es el valor mínimo de v */
```

Otra opción consiste en convertir la función en acción y obtener los resultados como parámetros por referencia:

```
void max_min (const vector<int> &v, int& x, int& y)
/* Pre: v.size()>0 */
/* Post: x es el valor máximo de v; y es el valor mínimo de v */
```

Por último, hay dos soluciones más que no consideramos adecuadas para este ejercicio. Una es escribir los resultados dentro de la operación de búsqueda, mediante `cout`. La otra consiste en tratar uno de los resultados como tal y el otro como parámetro de salida. Lo que queda es un híbrido entre acción y función al que no deseamos recurrir por ahora.

Observemos que, si bien en la asignatura se establece claramente la diferencia entre acciones y funciones, en ocasiones necesitaremos transformar alguna función en acción, ya sea por conveniencia o por obligación. Para ello, sustituimos los resultados de la misma por parámetros por referencia, como en el ejemplo anterior.

1.4.1. Ejercicio: Intercambio de valores de dos variables enteras

Escribid un programa para intercambiar los valores de dos variables enteras, mediante una acción como la especificada en el apartado anterior. Comprobad qué ocurre si los parámetros no se pasan por referencia.

1.4.2. Ejercicio: Búsqueda lineal en un vector de enteros

En el fichero `busquedalin.cc` tenéis un programa basado en el algoritmo de búsqueda lineal sobre un vector de enteros, con una cabecera como la mostrada antes. Dicho programa usa la operación `leerVector_int`, cuya cabecera está definida en el fichero `vectorIOint.hh` (notad el `#include` en `busquedalin.cc`) y su código se encuentra en `vectorIOint.cc`. Las operaciones de estos ficheros se pueden reutilizar en otros programas tal como lo hacemos en éste.

Como vimos en la sección 1.1, para usar dicha operación se deben compilar los dos ficheros `.cc` y linkar ambos para formar el ejecutable.

```
p2++ -c busquedalin.cc
p2++ -c vectorIOint.cc
g++ -o busquedalin.exe busquedalin.o vectorIOint.o
```

Una vez probado que todo funciona correctamente, modificad la función *lo mínimo posible* para que devuelva un resultado entero que diga si el elemento buscado está en el vector. En caso de éxito, el entero ha de contener una posición en la que se encuentre dicho elemento. Reescribid la postcondición para que el valor del resultado quede exactamente definido en el caso de que el elemento buscado no esté en el vector. Notad que si no se modifica `vectorIOint.cc`, no hace falta volver a compilarlo.

El resultado esperado del programa solo puede ser uno de los siguientes (solo pueden usarse `cout` en el `main`):

```
El elemento no está en el vector
El elemento está en el vector en la posición <la que sea>
```

Aprovecharemos este ejercicio para ilustrar el efecto de la opción `-D_GLIBCXX_DEBUG` al compilar un programa. En general, dicha opción detecta y avisa de los accesos prohibidos a memoria realizados por operaciones de clases tales como los vectores y otros contenedores standard. Si se da el caso, el programa se interrumpe y se genera un mensaje con información sobre el problema.

Por ejemplo, en la función de búsqueda sustituid la instrucción `int i = 0;` por `int i = -1;` y analizar cuidadosamente el error obtenido en la ejecución.

1.4.3. Ejercicio: Intercambio de dos posiciones de un vector

Escribid un programa para intercambiar los valores de dos posiciones de un vector de enteros. Emplead una acción con la cabecera vista anteriormente y reutilizad las operaciones de los ficheros `vectorIOint.hh` y `.cc`. Completad el programa con un método `main` que lea un vector y dos posiciones, realice el intercambio y escriba el nuevo vector, todo ello empleando las operaciones mencionadas. Tened en cuenta que si dimensionamos un vector con un valor n , las posiciones disponibles son las del intervalo $[0..n-1]$. Comprobad qué ocurre si intentamos intercambiar otras.

1.4.4. Ejercicio: Máximo y mínimo de un vector de enteros (*Jutge*)

Escribid un programa para obtener los valores máximo y mínimo de un vector de enteros, recorriendo éste sólo una vez. Producid tres versiones de la función `max_min` basadas en las especificaciones del apartado anterior. Notad que el *Jutge* espera una entrega con un único fichero, por tanto no se podrá aplicar la reutilización de código empleada en los ejercicios anteriores.

Recordad que sólo pueden usarse `cout` en el `main`.

1.4.5. Ejemplo: suma de matrices

En C++ las matrices se representan como vectores de dos dimensiones y su parametrización es igual que la de los vectores de una dimensión.

Por ejemplo, una función que sume matrices de enteros se puede escribir así

```
vector <vector<int> > suma (const vector <vector<int> > &m1,
                           const vector <vector<int> > &m2)
/* Pre: m1 y m2 son de la misma dimensión */
{
    vector <vector<int> > s;
    ...
    ...
    return s;
}
/* Post: el resultado es la suma de m1 y m2 */
```

Se puede abreviar un poco la escritura empleando la palabra clave `typedef`, que permite renombrar el tipo `vector <vector<int> >`.

```
typedef vector <vector<int> > Matriz;

Matriz suma (const Matriz &m1, const Matriz &m2)
/* Pre: m1 y m2 son de la misma dimensión */
{
    Matriz s;
    ...
```

```

...
return s;
}
/* Post: el resultado es la suma de m1 y m2 */

```

En el fichero `suma_mat.cc` tenéis una implementación de esta función, así como operaciones de lectura y escritura de matrices. En los ficheros `matrizIOint.hh` y `matrizIOint.cc` tenéis operaciones reutilizables de lectura y escritura de matrices. En el fichero `suma_mat.dat` tenéis un ejemplo de datos de entrada. En el fichero `suma_mat.sal` tenéis el resultado correspondiente.

Como ejercicio, modificad la cabecera de la operación para que la matriz `suma` se obtenga como parámetro de salida. Comprobad que el programa resultante sigue siendo correcto.

1.4.6. Ejercicio: producto de matrices

Usad los elementos anteriores para obtener y probar la función del producto de matrices. También tenéis ficheros con ejemplos de entradas y salidas para esta operación.

```

Matriz producto (const Matriz& m1, const Matriz& m2)
/* Pre: el número de columnas de m1 es igual al número de filas de m2 */
/* Post: el resultado es el producto de m1 y m2*/
{
    Matriz prod;
    ...
    ...
    return prod;
}

```

Ayuda: el resultado de multiplicar una matriz a de dimensión x,y por otra matriz b de dimensión y,z es una matriz m de dimensión x,z . Cada posición (i,j) de m contendrá el resultado de multiplicar la fila i de a por la columna j de b , es decir,

$$\forall i, j, 1 \leq i \leq x, 1 \leq j \leq z, m(i, j) = \sum_{1 \leq k \leq y} a(i, k) b(k, j)$$

Ejemplo: si a es de dimensión $2,3$ y b es de dimensión $3,1$, su producto es una matriz de dimensión $2,1$.

$$a = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \end{pmatrix} \quad b = \begin{pmatrix} 3 \\ 3 \\ 3 \end{pmatrix} \quad a \cdot b = \begin{pmatrix} 9 \\ 18 \end{pmatrix}$$

En el fichero `prod_mat.dat` tenéis un ejemplo de datos de entrada. En el fichero `prod_mat.sal` tenéis el resultado correspondiente. Obtened y probad una segunda versión en la que la matriz `prod` se obtenga como parámetro de salida.

1.4.7. Ejercicio: clasificación de la liga (*Jutge*)

Consideremos una matriz cuadrada m de pares de naturales, que representa los resultados de una competición deportiva a doble vuelta de N equipos. En la posición $m(i, j)$ se encuentra el resultado del partido de ida del equipo i contra el equipo j , mientras que el resultado del correspondiente partido de vuelta se encuentra en $m(j, i)$. El primer número del par alojado en $m(i, j)$ son los goles del equipo i y el segundo los del equipo j (al revés en el partido de vuelta). La información de la diagonal (es decir, las posiciones $m(i, j)$ tales que $i = j$) no es relevante.

Supongamos que, para cada enfrentamiento, el equipo ganador se anota 3 puntos y el perdedor 0. En caso de empate, se llevan 1 cada uno. Programad una operación que, a partir de una matriz de estas características, obtenga la clasificación de la competición, ordenada decrecientemente por puntos. En caso de empate a puntos, se ha de usar el orden decreciente respecto a la diferencia de goles a favor y goles en contra de cada equipo. Si persiste el empate, los equipos implicados han de aparecer en orden creciente respecto a su índice. Para cada equipo han de obtenerse los siguientes datos: su identificador, sus puntos, sus goles a favor y sus goles en contra.

Ejemplo, con $N = 4$. Donde pone $x\ x$ ha de entenderse que cualquier par de valores es válido, ya que no será relevante para el programa.

$$m = \begin{pmatrix} x\ x & 1\ 0 & 2\ 1 & 0\ 2 \\ 2\ 2 & x\ x & 3\ 3 & 1\ 3 \\ 1\ 1 & 1\ 2 & x\ x & 3\ 2 \\ 1\ 0 & 0\ 1 & 2\ 3 & x\ x \end{pmatrix} \quad \text{Clasificación} = \begin{matrix} 4 & 9 & 10 & 8 \\ 3 & 8 & 12 & 12 \\ 1 & 8 & 6 & 7 \\ 2 & 8 & 9 & 10 \end{matrix}$$

En el fichero `clasif.dat` tenéis un ejemplo de datos de entrada. En el fichero `clasif.res` tenéis el resultado correspondiente.

Para ordenar la clasificación podéis usar `sort` (ver “Normes de programació de P1”), definiendo adecuadamente la relación de orden entre los equipos.

1.4.8. Ejercicio: clasificación de la liga (versión ampliada)

Supongamos ahora que no disponemos de la matriz de pares con los goles de los partidos sino de un listado con los resultados de toda la competición hasta una jornada determinada. Escribid una nueva operación que transforme dicho listado en una matriz válida para el ejercicio anterior.

En el fichero `jornadas1-24_1314.dat` tenéis los resultados de las primeras 24 jornadas de la temporada 13-14 de la Liga BBVA, que como sabéis consta de 20 equipos. Para que los resultados resulten más inteligibles, primero aparecen los de la primera jornada, después los de la segunda, etc., aunque esto no es estrictamente necesario. En el fichero `jornadas1-24_1314.res` tenéis la matriz resultante. En el fichero `clasif_j24_1314.res` tenéis la clasificación correspondiente. En el fichero `equipos_1314.txt` tenéis la relación entre índices y nombres de equipos, por si queréis verificar los datos por vuestra cuenta, así como otra información útil para interpretar los ficheros `jornadas1-24_1314.dat` y `jornadas1-24_1314.res`.