Práctica 2: Diseño de programas recursivos

1. Objetivo de esta práctica

El objetivo principal de esta práctica es diseñar algoritmos sin bucles. En concreto, se implementarán varias funciones recursivas que trabajan con números enteros.

2. Funciones recursivas que trabajan con enteros

Se debe diseñar un módulo de biblioteca denominado **`calculos`** que implemente las funciones especificadas en **`calculos.hpp`** y que se listan a continuación:

```
#ifndef CALCULOS_HPP
#define CALCULOS_HPP

// Pre: 0 ≤ n ∧ 2 ≤ b ≤ 10
// Post: (n = 0 → numCifras(n, b) = 1) ∧
//       (n > 0 → numCifras(n, b) = NC ∧ b^(NC−1) ≤ n < b^NC)
int numCifras(const int n, const int b = 10);

// Pre: 0 ≤ n ∧ 1 ≤ i ∧ 2 ≤ b ≤ 10
// Post: cifra(n, i, b) = (n / b^(i−1)) % b
int cifra(const int n, const int i, const int b = 10);

// Pre: 0 ≤ n ∧ 2 ≤ b ≤ 10
// Post: cifraMayor(n, b) = Max α ∈ [1, ∞].cifra(n, α, b)
int cifraMayor(const int n, const int b = 10);

// Pre: 0 ≤ n ∧ 2 ≤ b ≤ 10
// Post: cifraMasSignificativa(n) = n / b^(NC−1) ∧ b^(NC−1) ≤ n < b^NC
int cifraMasSignificativa(const int n, const int b = 10);

// Pre: 0 ≤ n ∧ 2 ≤ b ≤ 10
// Post: sumaCifras(n, b) = ∑ α ∈ [1, ∞].cifra(n, α, b)
int sumaCifras(const int n, const int b = 10);

#endif
```

Listado 1: Fichero de interfaz del módulo **calculos** (fichero **`calculos.hpp`**).

Todas las funciones del módulo tienen un parámetro con un valor definido por defecto, ver Subsección 2.1. Esta circunstancia no debe condicionar sus diseños.

En el código de apoyo disponible en Moodle se encuentran:

- el fichero **calculos.hpp** de especificación del módulo **calculos**,
- el fichero **pruebasCal.cpp** con un programa de pruebas parcialmente desarrollado,
- el fichero **Make_pruebasCal** que facilita la compilación, ver Subsección 2.3.

El diseño del módulo de biblioteca **calculos** comprende las siguientes subtareas:

1. Escribir el fichero de implementación **calculos.cpp** del módulo **calculos** de forma que no se programe ningún bucle.
2. Realizar el diseño de un programa, **pruebasCal.cpp**, que permita hacer cuantas pruebas de las funciones definidas en el módulo **calculos** sean necesarias para confiar en su buen comportamiento.

2.1. Parámetros de una función cuyo valor puede estar definido por defecto

Los parámetros de una función pueden tomar valores definidos por defecto en la propia lista de parámetros de la función. Observa la definición de la siguiente función **numCifras**(...):

```
// Devuelve el número de cifras del número natural n cuando se expresa
// en base b, siendo b un entero comprendido entre 2 y 10
int numCifras(const int n, const int b = 10);
```

En esta función se está asignando un valor 10 de forma predeterminada a su segundo argumento. Los parámetros con valores definidos por defecto pueden ser uno o más y han de situarse obligatoriamente al final de la lista de parámetros de la función.

Cualquier invocación de la función anterior requiere definir el valor de su primer parámetro, mientras que la definición del valor del segundo es opcional. Si se omite este valor, el segundo parámetro toma el valor por defecto, es decir, el valor 10. Tras cada una de las invocaciones que se presentan a continuación, se explica el valor que toman los parámetros **n** y **b** de la función **numCifras**(n,b).

```
numCifras(9708725, 2);        // n = 9708725 y b = 2
numCifras(9708725);           // n = 9708725 y b = 10
numCifras(9708725, 6);        // n = 9708725 y b = 6
numCifras(9708725, 10);       // n = 9708725 y b = 10
```

2.2. Compilación y ejecución del programa anterior

Al igual que en la primera práctica, la compilación y ejecución del programa desarrollado puede realizarse mediante los siguientes pasos. Tras situarnos en el directorio **programacion2/practica2**, compilaremos en primer lugar el fichero de implementación del módulo de biblioteca **calculos**, **calculos.cpp**. Como resultado de la compilación se genera el fichero objeto **calculos.o**.

```
$ cd $HOME/programacion2/practica2
$ g++ calculos.cpp -c -std=c++11
```

A continuación se compila el módulo principal del programa, el fichero **pruebasCal.cpp**. En la orden se debe incluir el nombre del fichero con el código objeto del módulo calculos, es decir, **calculos.o**.

```
$ g++ pruebasCal.cpp calculos.o -o pruebasCal -std=c++11
```

Como resultado se obtiene el ejecutable **pruebasCal**.

2.3. Compilación mediante la herramienta `make`

La compilación de programas puede requerir varios pasos, sobre todo si hay varios ficheros fuente. La herramienta `make` simplifica y facilita dicha compilación. 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

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

define que el fichero `calculos.o` depende de los ficheros `calculos.cpp` y `calculos.hpp`, y que se obtiene mediante la ejecución del comando `g++ -c calculos.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 `calculos.cpp` ni `calculos.hpp`, y ya existe un `calculos.o` de una compilación anterior, la instrucción `g++ -c calculos.cpp -std=c++11` no se vuelve a ejecutar, por lo que se gana en tiempo de compilación.

Como material de apoyo, se proporciona un fichero de tipo **`Makefile`** con nombre **`Make_pruebasCal`** que permite compilar el programa **`pruebasCal`** mediante la orden:

```
$ make -f Make_pruebasCal
```

Puedes comprobar que, si dicho fichero tuviera por nombre **`Makefile`**, bastaría la siguiente orden para compilar:

```
$ make
```

3. Resultados del trabajo desarrollado en las dos primeras prácticas

Como resultado de las dos primeras prácticas, cada alumno dispondrá en su cuenta de un directorio (carpeta) denominado **`programacion2`** dentro del cual se encontrarán los directorios (carpetas) y ficheros que se detallan a continuación.

- Carpeta **`programacion2/practica1`**, con los siguientes ficheros: **`tiempoReaccion.cpp`**, **`generarTabla.cpp`**, **`genNum.hpp`**, **`genNum.cpp`** y **`aproxPI.cpp`**.
- Carpeta **`programacion2/practica2`** con los siguientes ficheros:
  - Ficheros de interfaz y de implementación, **`calculos.hpp`** y **`calculos.cpp`**.
  - Ficheros con los programa de prueba (**`pruebasCal.cpp`**, etc.) que se hayan puesto a punto para realizar pruebas de los desarrollos anteriores.
  - Fichero **`Make_pruebasCal`** para compilar los programas de prueba.