Práctica 2: Diseño de programas recursivos

Programación-II

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

1. Objetivos de esta práctica

Esta práctica persigue un doble objetivo:

- **Diseño de estructuras de datos.** Repaso del diseño de estructuras de datos junto con una colección de funciones básicas para trabajar con ellas.

  Lo aprendido se aplicará al diseño de una pila de números enteros.

- **Diseño de algoritmos sin bucles**. Programar varios algoritmos sin bucles, aplicando un diseño recursivo cuando sea necesario. Para diseñar estos algoritmos sin bucles deberá aplicarse la metodología de diseño recursivo estudiada en la asignatura.

  Se programarán de este modo varias funciones que trabajan con números naturales, así como varias funciones para gestionar la información de pilas de datos.

2. Implementación de funciones recursivas que trabajan con enteros

Se debe hacer un diseño de un módulo de biblioteca denominado `calculos` que incluya en su fichero de especificación, `calculos.hpp`, las funciones que se especifican formalmente en el Listado 1. Los ficheros de interfaz e implementación del módulo se ubicarán en el directorio (carpeta) `programacion2/practica2`.

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.

```hpp
#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, b, i) = (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) = Σ 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**).

En el código de apoyo disponible en Moodle se encuentran:

- el fichero **calculos.hpp** de especificación del módulo **calculos** y
- el fichero **pruebas01.cpp** con un programa de pruebas desarrollado parcialmente (cada alumno debería completarlo).

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 en el código de ninguna de las funciones.
2. Realizar el diseño de un programa, **pruebas01.cpp**, que permita hacer cuantas pruebas de las funciones definidas en el módulo **calculos** sean necesarias para confiar en su buen comportamiento.
3. Escribir un fichero **Makefile** para compilar el programa desarrollado.

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 concreto, 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 al final de la lista de parámetros de la función obligatoriamente.

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
```

2.1.1. Compilación y ejecución del programa anterior

La compilación y ejecución del programa desarrollado pueden 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 **pruebas01.cpp**. En la orden se debe incluir el nombre del fichero con el código objeto del módulo calculos, es decir, **calculos.o**.

```
$ g++ pruebas01.cpp calculos.o -o pruebas01 -std=c++11
```

Como resultado se obtiene el fichero compilado **pruebas01** que puede ser ejecutado cuantas veces se desee.

```
$ ./pruebas01
  ...mostrará los resultados de ejecutar el programa anterior
```

Para facilitar la introducción de las órdenes de compilación, se pide que, haciendo uso de lo explicado en el enunciado de la práctica 1, se escriba un fichero de tipo **Makefile** con nombre **Make_pruebas01** que permita compilar el programa **pruebas01** mediante la orden:

```
$ make -f Make_pruebas01
```

3. Implementación del tipo de dato PilaEnt

3.1. El tipo de datos PilaEnt y sus funciones básicas

Asumamos que hemos de desarrollar diversos programas que trabajan con pilas de números enteros. Para ello, podemos definir un tipo de dato denominado **PilaEnt**, que almacene datos de tipo **int**, y una reducida colección de funciones básicas que permitan definir el contenido inicial de una pila, añadir y eliminar datos de ella y consultar su estado. La implementación de estas funciones no requerirá ni bucles ni recursividad. Estas funciones se resumen a continuación:

- **vaciar**(p): tras ser ejecutada, la pila **p** no almacena ningún dato. Esta función permite empezar a trabajar con una pila **p** vacía.
- **apilar**(p,nuevo): modifica la pila **p** añadiéndole el dato entero **nuevo**, que se sitúa en su cima, como su dato más reciente.
- **desapilar**(p): modifica la pila **p**, retirando de **p** su dato cima, es decir, su dato más reciente.
- **cima**(p): devuelve un valor igual al dato más reciente de la pila **p**. Esta función no altera el contenido de la pila.
- **estaVacia**(p): devuelve un valor cierto si y sólo si la pila **p** no contiene en ese momento ningún dato. Esta función no altera el contenido de la pila.
- **estaLlena**(p): devuelve un valor cierto si y sólo si la pila **p** está llena por almacenar un número de datos igual a su capacidad máxima. Esta función no altera el contenido de la pila.

Se ha de crear un nuevo directorio, **programacion2/funciones**, que alojará diversos ficheros en los que se definan las estructuras de datos y el código de funciones para trabajar con dichas estructuras.

En esta carpeta se situarán:

- el fichero **pilaEnt.hpp**: tendrá que incluir la definición del tipo **PilaEnt** y de la interfaz de las funciones básicas que facilitan la gestión de una pila de enteros;
- el fichero **pilaEnt.cpp**: tendrá que incluir la implementación de las funciones básicas.

El esquema del fichero **pilaEnt.hpp**, mostrado en el Listado 2, se encuentra disponible como parte del código de apoyo proporcionado en Moodle.

Cada alumno debe completar la definición del tipo de dato **PilaEnt** en **pilaEnt.hpp**, e implementar en C++ las seis funciones básicas en el fichero **pilaEnt.cpp**, explicadas anteriormente.

```
#ifndef PILAENT_HPP
#define PILAENT_HPP
// Capacidad máxima de almacenamiento de datos en una pila
const int DIM = 120;          // Redefinir su valor en caso necesario

// Un dato de tipo PilaEnt representa una pila de datos enteros.
// Los datos son gestionados según una política LIFO,
// es decir el último en ser apilado será el primero en ser desapilado.
struct PilaEnt {
                            //completar la estructura de dato
};

// Notación empleada en la especificación de funciones:
//    p = [] la pila está vacía, es decir, almacena 0 datos
//    p = [d_1, d_2, ..., d_K] la pila almacena K datos. El más antiguo
//        es el dato d_1, el segundo más antiguo el dato d_2, etc.,
//        y el más reciente es el dato d_K

// Pre: ---
// Post: p = []
void vaciar(PilaEnt &p);

// Pre: p = [d_1, d_2, ..., d_K] ∧ 0 ≤ K < DIM
// Post: p = [d_1, d_2, ..., d_K, nuevo]
void apilar(PilaEnt &p, const int nuevo);

// Pre: p = [d_1, d_2, ..., d_(K-1), d_K] ∧ 0 < K
// Post: p = [d_1, d_2, ..., d_(K-1)]
void desapilar(PilaEnt &p);

// Pre: p = [d_1, d_2, ..., d_(K-1), d_K] ∧ 0 < K
// Post: cima(p) = d_K
int cima(const PilaEnt &p);

// Pre: ---
// Post: estaVacia(p) = (p = [])
bool estaVacia(const PilaEnt &p);

// Pre: p = [d_1, d_2, ..., d_K]
// Post: estaLlena(p) = (K = DIM)
bool estaLlena(const PilaEnt &p);
#endif
```

Listado 2: Definición del tipo **PilaEnt** y sus funciones básicas (fichero **pilaEnt.hpp**).

3.2. Diseño recursivo de funciones para trabajar con pilas de enteros

Se debe diseñar un módulo de biblioteca denominado **`funcionesPilaEnt`** que incluya en su fichero de especificación, **`funcionesPilaEnt.hpp`**, las funciones que se especifican formalmente en el Listado 3. Los ficheros de interfaz e implementación del módulo (**`funcionesPilaEnt.hpp`** y **`funcionesPilaEnt.cpp`**, respectivamente) se ubicarán en el directorio (carpeta) **`programacion2/practica2`**.

Estas funciones trabajan con pilas de datos enteros definidas a partir del tipo **`PilaEnt`** y resuelven problemas de un nivel de abstracción superior al de las funciones básicas dedicadas a gestionar directamente las pilas, desarrolladas en la primera tarea y contenidas en el fichero **`pilaEnt.hpp`**.

En el diseño de estas seis nuevas funciones se ha de tener en cuenta que el trabajo con una pila ha de ser independiente de la representación de ésta y debe realizarse exclusivamente invocando las funciones definidas en el fichero **`pilaEnt.hpp`** (véase Listado 2).

Por razones *didácticas*:

- el diseño de estas funciones debe ser recursivo, y
- no se debe hacer uso de estructuras de datos auxiliares en las que almacenar temporalmente el contenido de las pilas de datos con las que se trabaje. Es decir, no se pueden definir otras pilas, vectores, ficheros, etc. auxiliares.

Los ficheros **`funcionesPilaEnt.hpp`** y **`funcionesPilaEnt.cpp`** se ubicarán en la carpeta **`programación2/practica2`**. El fichero de interfaz se encuentra disponible como parte del código de apoyo proporcionado en Moodle.

```cpp
#ifndef FUNCIONES_HPP
#define FUNCIONES_HPP

#include <iostream>
#include <iomanip>

// Tipo PilaEnt y operaciones básicas para el trabajo con pilas de datos
#include "pilaEnt.hpp"

// * Notación empleada en las especificaciones:
// p = [] La pila p está vacía, es decir, almacena 0 datos
// p = [D_1, D_2, ..., D_K] la pila p almacena K datos. El más antiguo
//     es el dato D_1, el segundo más antiguo el dato D_2, etc.,
//     y el más reciente es el dato D_K

// * Pre: p = [D_1, D_2, ..., D_K] ∧ 0 ≤ K
// * Post: p = [D_1, D_2, ..., D_K] ∧ numDatos(p) = K
int numDatos(PilaEnt &p);

// Pre: p = [D_1, D_2, ..., D_K] ∧ 0 ≤ K ∧ anchura ≥ 1
// Post: p = [D_1, D_2, ..., D_K] y presenta por pantalla un listado de
//       los datos apilados en la pila comenzando por la cima, D_K, y acabando
//       por el del fondo de la pila, D_1. Cada dato lo escribe en una línea,
//       empleando anchura caracteres y alineado a la derecha. Cada dato es
//       precedido por el carácter '|' y es seguido por los caracteres ' ' y
//       '|', tal como se ilustra a continuación. Tras el último dato se
//       presenta una linea de la forma "+--...--+", seguida por una línea
//       en blanco:
//        | D_K |
//        | .. |
//        | D_2 |
//        | D_1 |
//        +--...--+
void mostrar(PilaEnt &p, const int anchura=3);

// Pre: p = [D_1, D_2, ..., D_K] ∧ 0 ≤ K ∧ 1 ≤ anchura
// Post: p = [D_1, D_2, ..., D_K] y presenta por pantalla un listado de los
//       datos apilados en la pila comenzando por el del fondo de la pila, D_1,
//       y acabando por el de la cima de la pila, D_1. Cada dato lo escribe
//       en una línea, empleando anchura caracteres y alineado a la derecha.
//       Cada dato es precedido por el carácter '|' y es seguido por los
//       caracteres ' ' y '|', tal como se ilustra a continuación. Antes
//       del primer dato se presenta una linea de la forma "+--...--+":
//        +--...--+
//        | D_1 |
//        | D_2 |
//        | .. |
//        | D_K |
void mostrarInvertida(PilaEnt &p, const int anchura=3);

// * Pre: p = [D_1, D_2, ..., D_K] ∧ 0 ≤ K
// * Post: p = [D_2, ..., D_K]
void eliminarFondo(PilaEnt &p);
```

```
// * Pre: p = [D_1, ..., D_(k-i), D_(k-i+1), D_(k-i+2), ..., D_k] ∧ 0 ≤ K ∧ 1 ≤ i ≤ K
// * Post: p = [D_1, ..., D_(k-i), D_(k-i+2), ..., D_k]
void eliminar(PilaEnt &p, const int i);

// * Pre: p = [D_1, D_2, ..., D_K] ∧ 0 ≤ K
// * Post: p = [nuevo, D_1, D_2, ..., D_K]
void insertarEnFondo(PilaEnt &pila, const int nuevo);

#endif
```

Listado 3: Fichero de interfaz **funcionesPilaEnt.hpp**.

3.2.1. Desarrollo de programas de pruebas de las funciones anteriores

Cada alumno debe ejecutar algún programa para probar el correcto funcionamiento de las diferentes funciones definidas en los apartados anteriores. Estos programas se ubicarán en el directorio (carpeta) **programacion2/practica2**.

Para ilustrar una forma sencilla de diseñar programas de prueba, se proporciona el programa **pruebas02.cpp** como parte del código de apoyo en Moodle.

3.2.2. Compilación y ejecución del programa de pruebas

El programa de pruebas que se acaba de mencionar consta únicamente de un módulo principal cuyo código se encuentra en el fichero **pruebas02.cpp**. Este programa depende de las funciones definidas en el módulo **funcionesPilaEnt**, es decir en el código de **pruebas02.cpp** se ha incluido una cláusula **#include** para la inserción del código del fichero **funcionesPilaEnt.hpp**. De forma similar, el fichero de interfaz **funcionesPilaEnt.hpp** depende de las funciones definidas en el módulo **pilaEnt**.

Tras situarnos en el directorio **programacion2/practica2**, generamos un ejecutable del programa principal **pruebas02.cpp** considerando las dependencias mencionadas anteriormente.

Es decir, generamos primero los códigos objeto de **pilaEnt.cpp** y **funcionesPilaEnt.cpp** y luego compilamos el programa de prueba y enlazamos los códigos objeto:

```
$ cd $HOME/programacion2/practica2
$ g++ ../funciones/pilaEnt.cpp -I ../funciones -c -o pilaEnt.o -std=c++11
$ g++ funcionesPilaEnt.cpp -I ../funciones -c -o funcionesPilaEnt.o -std=c++11
$ g++ pruebas02.cpp funcionesPilaEnt.o pilaEnt.o -I ../funciones -o pruebas02 -std=c++11
```

Como resultado se obtiene el fichero **pruebas02** que puede ser ejecutado cuantas veces se desee.

```
$ ./pruebas02
  ...mostrará los resultados de ejecutar el pograma anterior
```

Al igual que en las funciones recursivas que trabajan con enteros, se solicita que se escriba un fichero de tipo **Makefile** con nombre **Make_pruebas02** que permita compilar el programa **pruebas02** mediante la orden:

```
$ make -f Make_pruebas02
```

4. 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 podrán encontrar los directorios (carpetas) y ficheros que se detallan a continuación.

1. Carpeta **programacion2/funciones** con los siguientes ficheros:
   - Ficheros de interfaz y de implementación, **pilaEnt.hpp** y **pilaEnt.cpp**
2. Carpeta **programacion2/practica1**, con los siguientes ficheros fuentes:
   - Fichero **tiempoReaccion.cpp**.
   - Fichero **generarTabla01.cpp**.
   - Fichero **generarTabla02.cpp**.
   - Fichero **medirCoste.cpp**.
3. Carpeta **programacion2/practica2** con los siguientes ficheros:
   - Ficheros de interfaz y de implementación, **calculos.hpp** y **calculos.cpp**.
   - Ficheros de interfaz y de implementación, **funcionesPilaEnt.hpp** y **funcionesPilaEnt.cpp**.
   - Ficheros con los programas de prueba (**pruebas01.cpp**, **pruebas02.cpp**, etc.) que se hayan puesto a punto para realizar pruebas de los desarrollos anteriores.
   - Ficheros **Make_pruebas01** y **Make_pruebas02** para compilar los programas de prueba.

La duración de esta práctica es de dos sesiones. Por tanto, estos ficheros deberán estar listos antes de la cuarta sesión de prácticas.