Sesión 6

Uso de pilas, colas y árboles

En nuestros programas manejaremos un tipo particular de clases predefinidas: aquellas que contienen la traducción a C++ de los tipos abstractos de datos presentados en la teoría: pilas, colas y árboles binarios. De ellas sólo conoceremos algunos detalles, como su nombre y las cabeceras de sus métodos públicos. Todo ello consta en los respectivos ficheros de documentación `.doc`. Los correspondientes ficheros `Pila.hpp`, `Cola.hpp` y `Arbol.hpp` se encuentran en `$INCLUDES_CPP`. Las pilas y las colas están implementadas sobre las clases estándar `stack` y `queue`, por lo que no es necesario linkar su correspondiente fichero `.o`. Los árboles están implementados sobre la clase `Impl_Arbol`, que no es estándar. Su fichero `Impl_Arbol.o` está en `$OBJETOS_CPP`.

6.1 Restricciones para el uso de los TADs en C++

Los TADs que usaremos se basan en las especificaciones de los módulos correspondientes mostrados en teoría. Ahora bien, debido a las particularidades de la orientación a objetos y del lenguaje C++, vistos en una sesión anterior, y a la genericidad, se han tomado algunas decisiones de implementación que influyen en su uso.

- En primer lugar, notad que al igual que en los ejemplos de sesiones anteriores, se han modificado las **cabeceras** de las operaciones adaptándolas a los convenios de C++, de forma que toda operación se aplica sobre un parámetro implícito, con una sintaxis de la forma

  ```
  <objeto>.<método>(<otros parámetros>)
  ```

  Por ejemplo, la instrucción $p :=$ apilar(x, p) de la notación algorítmica se traduce como `p.apilar(x)`.

- Para cada instrucción que pueda modificar un objeto de manera no deseada (típicamente, las llamadas a métodos), hay que plantearse la conveniencia de hacer una **copia** del objeto involucrado. Por ejemplo, si deseamos escribir una pila tendremos que consultar su cima,

escribirla, desapilar, etc. Este proceso irá destruyendo la pila, de forma que antes de comenzar la copiaremos y aplicaremos el proceso sobre la copia. Si somos nosotros los que implementamos el método invocado, siempre podemos realizar la copia dentro del código del mismo.

Las tres clases que presentamos tienen la asignación redefinida para que funcione como una copia, por lo que podremos aplicar dicho operador cuando lo necesitemos.

- Se ha mantenido la **genericidad** de los tipos: en un objeto de la clase `Pila`, `Cola` o `Arbol` pueden almacenarse directamente objetos de cualquier clase. Esto se logra mediante el uso de `templates`. Una clase definida como template permite instanciar el tipo de sus componentes en el momento de declarar un objeto. Por ejemplo, declararíamos una pila de enteros así:

```
Pila<int> p;
```

- No puede ser genérico ningún método que dependa de la clase de los componentes de los TADs. En particular, ningún método genérico puede contener en su código ni lecturas, escrituras o copias de los componentes, ya que no se conoce la clase a la que pertenecerán éstos en cada momento. En particular, eso obliga a crear código separado para los correspondientes **métodos de lectura y escritura**: métodos para leer o escribir pilas de enteros, otros para las pilas de pares de enteros, otros para las pilas de `Estudiante`, etc. Podéis estudiar el ejemplo de pilas de enteros en el fichero `PilaIOint.hpp`

- La asignación/copia genérica de TADs no evita algunos efectos secundarios. Lo que garantiza es que el original y la copia son independientes respecto a su estructura (se puede borrar un elemento de una sin afectar a la otra), pero si se modifica un componente del original, también podría modificarse el de la copia y viceversa (el famoso aliasing). El resultado, en estas condiciones, se denomina *copia superficial* (en inglés, "shallow copy").

  Para que eso no ocurra hay que garantizar que la asignación funcione como una copia en la clase de los componentes, como pasa en esta asignatura, ya sea porque los campos de la clase lo admiten de forma natural o bien porque la asignación se ha redefinido.

  Incluso, si usamos mal la asignación no evitaremos los problemas de alias. Por ejemplo, si queremos aplicar algún tratamiento a la cima de una pila, pero sin afectar a ésta, podríamos hacer

```
Estudiante e;
e=p.cima();
tratar(e);
```

  o bien

```
Estudiante e(p.cima());
tratar(e);
```

pero no

```
Estudiante e=p.cima();
tratar(e); // podría modificar la cima de p y la de sus copias
```

Esta situación aparece en el ejercicio de modificar una pila de Estudiante.

6.2 Uso de la clase `Pila`

Podemos traducir cualquier ejercicio de pilas sin más que aplicar cuidadosamente las ideas anteriores y tener en cuenta el contenido del fichero `Pila.doc`.

6.2.1 Ejemplo: búsqueda en una pila de enteros

El fichero `pertpila.cpp` contiene un programa que comprueba si un cierto número está en una pila de enteros. El programa usa una función que realiza la búsqueda. Por su parte, el método `main` se encarga de la entrada/salida y de invocar correctamente la función anterior. Dicha función se basa en el siguiente algoritmo.

funcion f_pert (p : pila; x : entero) **retorna** b : booleano
  {Pre: cierto}
  si es_nula(p) $\rightarrow$ $b := $ falso
  [] $\neg$es_nula(p) $\rightarrow$
    si $x =$ cima(p) $\rightarrow$ $b :=$ cierto
    [] $x \neq$ cima(p) $\rightarrow$ $b :=$ f_pert (desapilar(p), x);
      {HI: b indica si x está en desapilar(p)}
    fsi
  fsi;
  {Decr: alt(p)}
  {Post: b indica si x está en p}

Probad varios juegos de pruebas con distintas situaciones (que el elemento buscado esté en la cima de pila, que esté abajo del todo, que no esté, etc.)

6.2.2 Ejercicio: búsqueda en una pila de pares de enteros

Repetid el programa anterior considerando una pila de pares de enteros. Dado un número, comprobad si aparece como primer elemento de algún par de la pila y, en caso de éxito, escribid el par completo.

Por ejemplo, si la pila contiene el par (3, 6) y buscamos el elemento 3, el programa ha de retornar algo como

```
El compañero del 3 en la pila es el 6.
```

Definid previamente en fichero aparte una struct `parint` (“par de enteros”), como envoltura para apilar los números, y otro fichero `PilaIOparint.hpp` con operaciones para leer y escribir las pilas de `parint`.

6.2.3 Ejercicio: búsqueda en una pila de estudiantes

Repetid el programa anterior con una pila de estudiantes. Dado un entero, buscad si hay algun estudiante en la pila que lo tenga como DNI. En caso de éxito hay que informar sobre la nota del estudiante. El resultado puede ser uno de éstos

```
El estudiante no esta en la pila

El estudiante esta en la pila, pero no tiene nota

El estudiante esta en la pila y su nota es <la que sea>
```

Necesitaréis un módulo `PilaIOEstudiante`.

6.2.4 Ejercicio: modificar los elementos de una pila

1. Con la misma estructura del ejemplo anterior, escribid un programa que dada una pila de enteros le sume un número k a todos sus elementos.

2. Repetid el ejercicio, pero usando esta vez pilas de pares de enteros y sumando k al segundo elemento de cada par.

3. Por último, aplicad las ideas de los problemas anteriores para redondear las notas de una pila de estudiantes.

Tomad como base el siguiente algoritmo. Podéis traducirlo como acción o función. En el segundo caso, poned especial cuidado en lograr que las pilas originales no se vean modificadas.

funcion f_incrementar $(p$: pila; k : entero$)$ **retorna** q : pila
  {Pre: cierto}
  si es_nula(p) $\rightarrow$ $q :=$ p_nula
  [] $\neg$es_nula(p) $\rightarrow$ $x :=$ cima$(p) + k$;
    $q :=$ f_incrementar(desapilar$(p), k$);
    {HI: q es la pila resultante de incrementar todo
    elemento de desapilar(p) en k unidades}
    $q :=$ apilar(x, q)
  fsi
  {Decr: alt(p)}
  {Post: q es la pila resultante de incrementar todo elemento de p en k unidades}

6.3 Uso de la clase Cola

Para probar la clase Cola se pueden utilizar los mismos ejemplos y ejercicios que con la clase Pila: búsqueda en una cola de números, de pares o de estudiantes y modificar los elementos de una cola de números, pares o estudiantes. Nuevamente, deberéis introducir los ficheros ColaIOparint y ColaIOEstudiante cuando corresponda.

6.3.1 Ejemplo: búsqueda en una cola de enteros

La búsqueda en una cola de enteros está en el fichero pertcola.cpp y se basa en el siguiente algoritmo, muy similar al de pilas:

funcion f_pert $(c$: cola$; x$: entero$)$ **retorna** b : booleano
  {Pre: cierto}
  si es_nula(c) $\rightarrow$ $b :=$ falso
  [] $\neg$es_nula$(c) \rightarrow$
    $b :=$ f_pert (avanzar$(c), x$);
    {HI: b indica si x está en avanzar(c)}
    $b := b \vee x =$ primero(c)
  fsi;
  {Decr: tam(c)}
  {Post: b indica si x está en c}

Probad varias situaciones distintas de búsquedas (que el elemento buscado sea el primero de la cola, que sea el último, que no esté, etc.)

6.3.2 Ejercicio: modificar los elementos de una cola

Este problema varía significativamente respecto al de pilas, ya que no es tan sencillo colocar un elemento en la primera posición de una cola ya creada. Una solución recursiva, que emplea una operación auxiliar, sería la siguiente

funcion f_incrementar $(c$: cola$; k$: entero$)$ **retorna** q : cola
  {Pre: cierto}
  si es_nula(c) $\rightarrow$ $q :=$ c_nula
  [] $\neg$es_nula(c) $\rightarrow$ $x :=$ primero$(c) + k$;
    $q :=$ f_incrementar(avanzar$(c), k$);
    {HI: q es la cola resultante de incrementar todo
    elemento de avanzar(c) en k unidades}
    $q' :=$ pedir_turno$(x,$ c_nula$())$;
    $q :=$ concat(q', q);
  fsi
  {Decr: tam(c)}
  {Post: q es la cola resultante de incrementar todo elemento de c en k unidades}

La operación `concat(c1, c2)` crea una nueva cola, introduciendo primero los elementos de `c1` y después los de `c2`

funcion f_concat (c_1, c_2 : cola) **retorna** q : cola
{Pre: cierto}
si es_nula(c_2) $\rightarrow$ $q := c_1$
[] $\neg$es_nula(c_2) $\rightarrow$
$c_1 :=$ pedir_turno(primero(c_2), c_1);
{A: c_1 es la cola formada por todos los elementos de la
c_1 original seguidos por el primero de c_2}
$q :=$ concat(c_1, avanzar(c_2));
{HI: q es la cola formada por todos los elementos de la
c_1 original seguidos por el primero de c_2 seguidos de todos los
elementos de avanzar(c_2) en el orden original} $\equiv$ Post
fsi
{Decr: tam(c_2)}
{Post: q es la cola formada por todos los elementos de
c_1 seguidos de todos los elementos de c_2 en el orden original}

Existen otras soluciones, ya sean introduciendo parámetros adicionales o bien recurriendo a un algoritmo iterativo, pero entonces la dificultad radica en justificar su corrección. Por ejemplo, una versión iterativa sería

funcion f_incrementar (c : cola; k : entero) **retorna** q : cola
{Pre: $c = C$}
{Inv: q es la cola resultante de incrementar todo elemento de la parte visitada de C en k
unidades y c es la parte no visitada de C}
$q :=$ c_nula;
mientras $\neg$es_nula(c) **hacer**
$x :=$ primero(c) + k;
$q :=$ pedir_turno(x, q);
$c :=$ avanzar(c);
fmientras
{Decr: tam(c)}
{Post: q es la cola resultante de incrementar todo elemento de C en k unidades}

Por último, obtened una versión de ambas soluciones en forma de *acción*.

6.3.3 Ejercicio: convertir una cola en una pila

Escribid una función que convierta una cola en una pila con los mismos elementos y en el mismo orden. El primer elemento de la cola ha de convertirse en la cima de la pila y así sucesivamente.

6.4 Uso de la clase `Arbol`

En el fichero `ArbolIOint.hpp` están definidos dos posibles métodos de lectura y escritura de árboles binarios de enteros, que a su vez utilizan la clase `Arbol` y sus métodos. Para probarlos, escribid programas que calculen operaciones sencillas sobre árboles como la altura, el tamaño u obtener el árbol resultante de sumar un valor k a todos los nodos de un árbol dado. Recordad que los programas de árboles han de linkarse con el fichero `Impl_Arbol.o`.

Por ejemplo, comenzad probando la siguiente función, situada en el fichero `tamarb.cpp` que calcula el número de elementos de un árbol:

funcion f_tam $(a$: arbol$)$ **retorna** n : nat
  $\{$Pre: cierto$\}$
  si es_nulo(a) $\rightarrow$ $n := 0$
  $[]$ $\neg$es_nulo(a) $\rightarrow$ $n_1 :=$ f_tam $(\text{hi}(a))$;
    $n :=$ f_tam $(\text{hd}(a))$;
    $\{\text{HI: } n_1 = \text{tamaño } (\text{hi}(a)) \wedge n = \text{tamaño } (\text{hd}(a))\}$
    $n := 1 + n_1 + n$
  fsi
  $\{$Decr: tamaño$(a)\}$
  $\{\text{Post: } n = \text{tamaño } (a)\}$

Resolved también algún problema con árboles de `parint` y de estudiantes (buscar, redondear, sumar un valor k, etc.)