11 PILAS

11.1 Fundamentos

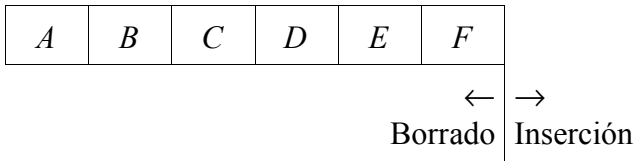
En este tema y en el siguiente se analizarán las estructuras de datos lineales pilas y colas. Las pilas y las colas son dos de las estructuras de datos más utilizadas. Se trata de dos casos particulares de las estructuras lineales generales (secuencias o listas) que, debido a su amplio ámbito de aplicación, conviene ser estudiadas de manera independiente. En este tema se estudiarán las pilas, su utilización y su implementación más habitual en C++.

La pila es una lista de elementos caracterizada porque las operaciones de inserción y eliminación de elementos se realizan solamente en un extremo de la estructura. El extremo donde se realizan estas operaciones se denomina habitualmente cima (*top* en la nomenclatura inglesa).

Dada una pila *P*, formada por los elementos *a*, *b*, *c*, ..., *k* (*P=(a,b,c,...,k)*), se dice que *a*, que es el elemento más inaccesible de la pila, está en el fondo de la pila (*bottom*) y que k, por el contrario, el más accesible, está en la cima (*top*).

Las restricciones definidas para la pila implican que si una serie de *elementos A, B, C, D, E, F* se añaden, en este orden, a una pila entonces el primer elemento que se elimine (borre) de la estructura deberá ser *E*. Por tanto, resulta que el último elemento que se inserta en una pila es el primero que se borra. Por esa razón, se dice que una pila es una lista o estructura lineal de tipo LIFO (*Last In First Out*, el último que entra es el primero que sale).

Estructura Pila



Un ejemplo típico de pila lo constituye un montón de platos: Cuando se quiere introducir un nuevo plato, éste se coloca en la posición más accesible, encima del último plato. Cuando se coge un plato, éste se extrae, igualmente, del punto más accesible, el último que se ha introducido. O, si somos más estrictos, otro ejemplo sería una caja llena de libros. Las paredes de la caja impiden acceder libremente a su contenido y sólo se puede acceder al libro que está más arriba en la caja. Al almacenar o extraer un libro, sólo es posible actuar sobre ese primer libro. No es posible siquiera saber el número total de libros guardados en la pila. Sólo se conocerá el número de elementos de la pila de libros si previamente los sacamos hasta vaciar la caja.

Otro ejemplo natural de la aplicación de la estructura pila aparece durante la ejecución de un programa de ordenador, en la forma en que la máquina procesa las llamadas a las funciones. Cada llamada a una función hace que el sistema almacene toda la información asociada con esa función (parámetros, variables, constantes, dirección de retorno, etc...) de forma independiente a otras funciones y permitiendo que unas funciones puedan invocar a otras distintas (o a si mismas) y que toda esa información almacenada pueda ser recuperada convenientemente cuando corresponda. Como durante la ejecución de un programa sólo se

puede estar ejecutando una función (asumiendo que no existe ejecución concurrente dentro del programa), esto quiere decir que sólo es necesario que sean accesibles los datos de la función activa (la que está en la cima de llamadas). De ahí que una estructura muy apropiada para este fin sea la estructura pila.

En general, una pila tiene su utilidad cuando interesa recuperar la última información generada (el estado inmediatamente anterior).

Asociadas con la estructura pila existen una serie de operaciones necesarias para su manipulación. Éstas son:

Iniciación de la estructura:

- Crear la pila (`CrearPila`): La operación de creación de la pila inicia la pila como vacía.

Operaciones para añadir y eliminar información:

- Añadir elementos en la cima (`Apilar`): pondrá un nuevo elemento en la parte superior de la pila.
- Eliminar elementos de la cima (`Desapilar`): lo que hará será eliminar el elemento superior de la pila.

Operaciones para comprobar tanto la información contenida en la pila, como el propio estado de la cima:

- Comprobar si la pila está vacía (`PilaVacia`): Esta operación es necesaria para verificar la existencia de elementos de la pila.
- Acceder al elemento situado en la cima (`CimaPila`): Devuelve el valor del elemento situado en la parte superior de la pila.

Como en el caso de cualquier contenedor de datos las operaciones relevantes tienen que ver con el almacenamiento (`Apilar`), eliminación (`Desapilar`) o consulta (`CimaPila`) de información.

La especificación correcta de todas estas operaciones permitirá definir adecuadamente el tipo pila.

Una declaración más formal de las operaciones definidas sobre la estructura de datos pila, y los axiomas que las relacionan es la siguiente:

Estructura

`Pila ( Valor )` /* `Valor` *constituye el dominio sobre el que se define el tipo de datos y hace referencia a los datos que se almacenan en la pila* */

Operaciones

```
CrearPila () → Pila
Apilar (Pila , Valor) → Pila
Desapilar (Pila) → Pila
CimaPila (Pila) → Valor
PilaVacia (Pila) → Lógico
```

Axiomas

∀ s ∈ Pila, x ∈ Valor se cumple que:

```
PilaVacia (CrearPila ()) → cierto
PilaVacia (Apilar (s, x)) → falso
```

```
Desapilar (CrearPila ()) → error
Desapilar (Apilar (s, x)) → s
CimaPila (CrearPila ()) → error
CimaPila (Apilar (s, x)) → x
```

Estos axiomas explican de manera precisa el comportamiento de las operaciones que caracterizan las pilas y constituyen la referencia en la que se debe basar cualquier implementación de este tipo. Son el plano sobre el que basarse a la hora de la construcción.

11.2 Representación de Pilas en C++

Los lenguajes de programación de alto nivel no suelen disponer de un tipo de datos predefinido pila. Por lo tanto, es necesario representar la estructura pila a partir de otros tipos de datos existentes en el lenguaje.

En C++, lo primero que se plantea al construir una clase que represente el tipo pila son los métodos públicos a través de los que se podrá manipular la estructura. Las operaciones serán básicamente las definidas en el tipo abstracto de datos, aunque será preciso adaptarse a las características del lenguaje para establecer los detalles finales de la representación. Así, se puede establecer el siguiente interfaz para la clase pila:

```
class Pila
{
   public:
      Pila ();
      bool Apilar (Valor);
      bool Desapilar ();
      bool CimaPila (Valor &);
      bool PilaVacia ();
   private:
      //Todavía por definir
};
```

Los criterios establecidos para esta definición son los siguientes:

- Aquellas operaciones que, según el TAD, puedan generar un error (`Desapilar`, `CimaPila`) devolverán un valor lógico que indique si la operación se ha realizado correctamente o no.
- La operación `CimaPila` devuelve un valor lógico para informar de un posible error, pero también debe retornar el valor que se ubica en la cima de la pila. Como no es posible que una función devuelva dos valores diferentes, se plantea que tenga un argumento por referencia que permita almacenar como salida de la operación el valor almacenado en la cima de la pila.
- La operación `Apilar` permitirá añadir nuevos elementos a la pila. No obstante, es posible que el espacio en memoria para los elementos de la pila esté limitado (en principio, el límite puede ser la memoria disponible en el ordenador) y es posible que la pila se "llene" y no admita nuevos elementos. Ésto debería generar un error similar al generado cuando se desea eliminar un dato y la pila está vacía. Por ese motivo, se plantea que la operación

devuelva un valor lógico que informe sobre si la operación se ha podido realizar correctamente.

- Recordar que en C++, si la implementación se realiza mediante clases, no es necesario pasar como parámetro el objeto receptor de la petición de operación. Ese es un argumento implícito en todas las operaciones.
- Finalmente, el método de iniciación de la estructura `CrearPila` toma la forma de lo que se conoce en C++ como un constructor de objetos de la clase. El constructor es un método especial que es llamado siempre que se instancia (declara) un nuevo objeto de una clase. Si no existe el constructor por defecto, la instanciación se limita a reservar el espacio necesario para guardar la información del objeto. Si existe el constructor por defecto, éste es llamado inmediatamente después de la reserva de memoria y permite establecer un estado inicial correcto para el objeto. El constructor es un método que tiene el mismo nombre que la clase, no devuelve ningún valor (ni siquiera **`void`**) y puede tener los parámetros que se consideren oportunos, en nuestro caso ninguno (constructor por defecto).

11.2.1. Implementación mediante estructuras estáticas

La forma más simple, y habitual, de representar una pila es mediante un vector unidimensional. Este tipo de datos permite definir una secuencia de elementos (de cualquier tipo) y posee un eficiente mecanismo de acceso a la información contenida en ella.

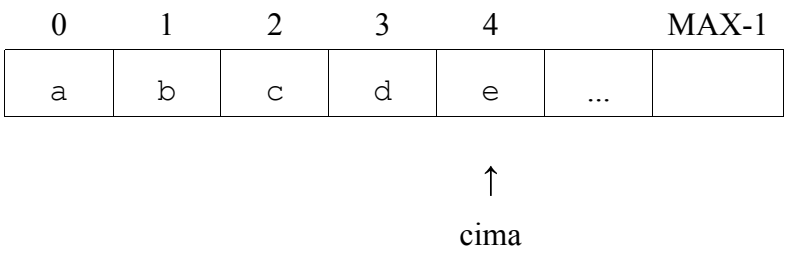
Al definir un *array* hay que determinar el número de índices válidos y, por lo tanto, el número de componentes definidos. Entonces, la estructura pila representada por un array tendrá limitado el número de posibles elementos.

La parte privada de la clase, que hace referencia a los detalles de implementación seleccionados, estará constituida por un vector para almacenar los elementos de la pila. El primer elemento se almacenará en la posición `0` y será el fondo de la pila, el segundo elemento en la posición `1` y así sucesivamente. En general, el elemento *i*-ésimo estará almacenado en la posición `i-1` del vector.

Como todas las operaciones se realizan sobre la cima de la pila, es necesario tener correctamente localizada en todo instante esta posición. Es necesaria una variable adicional, `cima`, que apunte al último elemento de la pila (no del array) o que indique cuántos elementos hay almacenados en ella.

Sea una pila `p` cuyos elementos son (`a`, `b`, `c`, `d`, `e`), siendo `e` su cima. La representación gráfica de la implementación de `p` mediante un array sería:

`p` = (`a`, `b`, `c`, `d`, `e`)



Si se reescribe el interfaz de la clase pila para reflejar esta forma de implementación se tiene que:

```
class Pila
{
   public:
      Pila ();
      bool Apilar (Valor);
      bool Desapilar ();
      bool CimaPila (Valor &);
      bool PilaVacia ();
   private:
      typedef Valor Vector[MAX];
      Vector datos;
      int cima;
};
```

Suponiendo `Valor` como el tipo de dato que se puede almacenar en la pila y `MAX` una constante que limita el tamaño máximo del array (y de la pila).

Con estas consideraciones prácticas, se puede pasar a construir las operaciones que definen la pila.

Operación `CrearPila`

La creación de la pila se realizará mediante el constructor por defecto. La tarea que deberá realizar será establecer un estado inicial en el que no existen elementos en la pila:

```
Pila::Pila ()
{
   cima = -1;
}
```

Como la primera posición válida del array es la identificada por el índice 0, con la asignación cima = -1 se intenta representar que, si no hay elementos en la pila, la cima se encuentra una posición antes de la primera posición del vector y, por tanto, en una posición no válida del mismo.

Esta operación permitirá que al declarar objetos de esta clase, en la forma habitual en C++, éstos se inicialicen adecuadamente de forma automática. Por ejemplo: `Pila p;`

Operación `PilaVacia`

Esta operación permitirá determinar si la estructura tiene o no elementos almacenados. Aunque el array empleado tiene un número de elementos fijo (MAX), no todos ellos representan valores almacenados en la pila. Sólo están almacenados en la pila los valores comprendidos entre los índices `0` (fondo de la pila) y `cima`. Por lo tanto, la pila estará vacía cuando la cima indique una posición por debajo del fondo:

```
bool Pila::PilaVacia ()
{
   return (cima == -1);
```

```
}
```

Operación `Apilar`

La operación de inserción normalmente se conoce por su nombre inglés *Push*, o Apilar. La operación, aplicada sobre un pila y un valor, almacena el valor en la cima de la pila. Esta operación está restringida por el tipo de representación escogido. En este caso, la utilización de un array implica que se tiene un número máximo de posibles elementos en la pila, por lo tanto, es necesario comprobar, previamente a la inserción, que realmente hay espacio en la estructura para almacenar un nuevo elemento. Con está consideración, el algoritmo de inserción sería:

```
bool Pila::Apilar (Valor x)
{
   bool ok;

   if (cima == MAX)
      ok = false;
   else
   {
      cima++;
      datos[cima] = x;
      ok = true;
   }
   return (ok);
}
```

Operación `Desapilar`

La operación de borrado elimina de la estructura el elemento situado en la cima. Normalmente recibe el nombre de *Pop* en la bibliografía inglesa. Eliminar un elemento de la pila consiste fundamentalmente en desplazar (decrementar) la cima. Puesto que no es posible eliminar físicamente los elementos de un array, lo que se hace es dejar fuera del rango válido de elementos al primero de ellos.

```
bool Pila::Desapilar (void)
{
   bool ok;

   if (cima == -1)
      ok = false;
   else
   {
      cima--;
      ok = true;
   }
   return (ok);
```

```
}
```

Operación `CimaPila`

La operación de consulta permite conocer el elemento almacenado en la cima de la pila, teniendo en cuenta que si la pila está vacía no es posible conocer este valor.

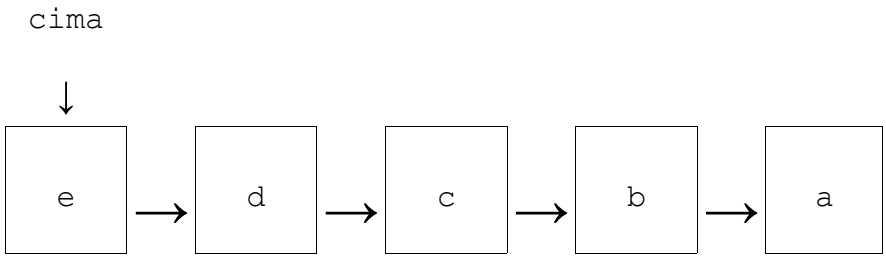
```
bool Pila::CimaPila (Valor & x)
{
   bool ok;

   if (cima == -1)
      ok = false;
   else
   {
      x = datos[cima];
      ok = true;
   }
   return (ok);
}
```

11.2.2. Implementación mediante estructuras dinámicas

Uno de los mayores problemas en la utilización de estructuras estáticas (arrays), estriba en el hecho de tener que determinar, en el momento de la realización del programa, el valor máximo de elementos que va a poder contener la estructura. Esto implica una estimación que no siempre es posible o que no siempre es muy ajustada a la realidad de una ejecución en particular. Esto implica un desaprovechamiento de la memoria y coste espacial ineficiente de esta forma de representación. Por ejemplo, si tamaño el array se define con un atamaño 100 y sólo hay 20 elementos en la pila, el 80% del espacio reservado no se utiliza.

Una posible solución a este problema es la utilización de estructuras dinámicas enlazadas (punteros) tal y como se explicó en el primer cuatrimestre del curso. En este caso, los elementos que constituirán la pila no estarán almacenados de manera consecutiva en memoria, como en un array, sino que se irán creando dinámicamente (durante la ejecución del programa) y se irán distribuyendo "aleatoriamente" en la memoria. Para que estos elementos dispersos puedan formar una secuencia es preciso que estén relacionados o enlazados entre sí. Lo que permite el tipo puntero es establecer estos enlaces entre los elementos. De manera que un elemento (o nodo) de la pila no estará formado exclusivamente por un elemento del tipo Valor, sino que se tratará de un dato compuesto que incluirá adicionalmente un puntero para enlazar con el siguiente elemento de la secuencia. Gráficamente, una pila `p` cuyos elementos son (`a`, `b`, `c`, `d`, `e`), siendo `e` su cima, se representaría de manera enlazada de la siguiente forma:

`p` = (`a`, `b`, `c`, `d`, `e`)



Con esta forma de implementación, el interfaz de la clase Pila quedaría como sigue:

```
class Pila
{
   public:
      Pila ();
      bool Apilar (Valor);
      bool Desapilar ();
      bool CimaPila (Valor &);
      bool PilaVacia ();
   private:
      struct Nodo;
      typedef Nodo* Puntero;
      struct Nodo
      {
            Valor info;
            Puntero sig;
      };
      Puntero cima;
};
```

Como se puede observar, se ha modificado la parte privado de la clase, pero sus elementos públicos. La idea es que la forma de utilizar una pila es la misma siempre, con independencia de como esté construida.

Ahora el único dato definido en la clase es un puntero (`cima`) que permite acceder al elemento situado en la cima de la pila y a partir de él, siguiendo el enlace `sig` de cada nodo, es posible acceder al resto de los elementos.

Los elementos de la pila no están declarados en la clase puesto que se irán creando a medida que se vayan añadiendo a la misma.

Ahora la implementación de cada una de las operaciones queda como sigue:

Operación `CrearPila`

Sólo es preciso indicar que inicialmente el puntero `cima` no hace referencia a ningún nodo, puesto que la pila se crea vacía.

```
Pila::Pila ()
{
```

```
        cima = NULL;
}
```

Operación `PilaVacia`

Necesita verificar si el puntero a `cima` referencia algún nodo.

```
bool Pila::PilaVacia ()
{
        return (cima == NULL);
}
```

Operación `Apilar`

Ahora es preciso reservar memoria para cada nuevo elemento que se desee apilar, puesto que no hay espacio reservado para él a priori. Además de ésto, es preciso que el nuevo elemento se enlace adecuadamente con el resto de elementos de la pila. Para ello, el proceso a realizar consta de los siguientes pasos:

1) Reservar memoria para un nuevo nodo de la pila.
2) Establecer los valores adecuados para los campos del nodo:
   2.1) El campo de información será igual al valor que se desea apilar.
   2.2) El campo siguiente hará referencia a la cima actual de la pila, puesto que el nuevo elemento se debe situar delante de ella.
3) Como el nuevo nodo es la nueva cima de la pila, es preciso actualizar el puntero cima para apunte al nuevo nodo.

```
bool Pila::Apilar (Valor x)
{
   bool ok;
   Puntero p_aux;

   p_aux = new Nodo; //Paso 1
   if (p_aux == NULL)
     ok = false;
   else
   {
     p_aux->info = x; //Paso 2.1
     p_aux->sig = cima; //Paso 2.2
     cima = p_aux; //Paso 3
     ok = true;
   }

   return (ok);
}
```

Operación `Desapilar`

En una estructura dinámica es posible liberar la memoria reservada previamente si ya nos es necesaria. En este caso, cuando se desapila un elemento, es posible liberar la memoria asociado al nodo que ocupa en la pila. Luego, además de avanzar la cima para que el elemento quede fuera de la pila se debe liberar la memoria que ocupaba. La función sería la siguiente:

```
bool Pila::Desapilar (void)
{
   bool ok;
   Puntero p_aux;

   if (cima == NULL)
      ok = false;
   else
   {
      p_aux = cima;
      cima = cima->sig; //avanzar cima
      delete p_aux; //liberar la memoria (borrar nodo)
      ok = true;
   }

   return (ok);
}
```

Operación `CimaPila`

Esta operación debe devolver el valor almacenado en la pila de la cima. En este caso, la información almacenada en el campo `info` del nodo situado en la cima (no todo el nodo).

```
bool Pila::CimaPila (Valor & x)
{
   bool ok;

   if (cima == NULL)
      ok = false;
   else
   {
      x = cima->info;
      ok = true;
   }
   return (ok);
}
```

11.2.3 Clases y punteros en C++

Al programar con punteros hay que tener en cuenta aspectos irrelevantes cuando se utilizan estructuras estáticas. En particular, el problema de la copia o de la eliminación de las estructuras creadas. Hay que considerar, por ejemplo, que cuando se pasa un argumento por valor a una función, lo que se está haciendo es crear una copia del dato que se pasa en la llamada. Por ejemplo, sea el siguiente prototipo de función, `void f (Pila s)`, y sea `f (p)`, donde `p` es un objeto de tipo `Pila`, una llamada a la función `f`. Lo que se está haciendo en la llamada, al asociar el argumento `s` a `p`, es generar `s` como una copia exacta de `p`. Eso implica copiar el contenido de los datos declarados en la clase a la que pertenece el objeto, en este caso `Pila`. Como el único dato especificado en la clase es el puntero `cima`, lo que en realidad se está haciendo es asignar a `s.cima` el valor de `p.cima`. Por tanto, en realidad `s` y `p` comparten los mismos datos y no se está haciendo un duplicado real de la pila. Este es un problema que se reproduce siempre que se manejan clases con punteros.

En C++ es posible indicar cómo se debe hacer de manera correcta la copia de objetos. La solución consiste en definir en la clase un tipo especial de constructor, llamado constructor de copia, que debe especificar la forma correcta en la que se debe generar un duplicado del objeto. Cada vez que el lenguaje requiera crear un nuevo objeto como copia de otro (paso de argumentos por valor, valores de retorno de una función, variables temporales) buscará el constructor de copia y si está definido ejecutará el algoritmo de copia especificado en él. El constructor de copia no se suele invocar de manera explícita por parte del programador y su uso está un tanto oculto durante la ejecución del programa. Sin embargo, su ausencia puede provocar que los programas no funcionen correctamente.

Constructor de copia para la clase Pila

El constructor de copia tiene el mismo nombre que la clase y tiene como argumento argumento un objeto de la misma clase pasado por referencia (si fuese por valor se tendría un bucle recursivo infinito) pero "protegido" con el calificativo `const`, para que no se pueda modificar. En el caso que nos ocupa, suponiendo que existe una función auxiliar que realiza la copia efectiva de las pilas, lo que permite utilizarla desde cualquier otro método, el constructor de copia tendría la siguiente forma:

```
Pila::Pila (const Pila& p)
{
   if (! Copiar(p))
      cerr << "Error de memoria al copiar pila." << endl;
}
```

La función Copiar realiza el duplicado del original sobre el objeto destino de la copia (receptor de la petición). Los pasos que realiza son:

1) Vaciar la pila sobre la que se va a copiar. Esto permite una aplicación más general de esta función, incluso para realizar asignaciones de objetos.

2) Recorrer uno a uno todos los elementos de la pila original y generar una copia de cada elemento en la pila destino.

```
bool Pila::Copiar (const Pila& p)
{
   Puntero p_aux, dup, fin;
   bool ok=true;
```

```
    Vaciar(); //función que se especifica más adelante

    p_aux = p.cima;
    while ( (p_aux != NULL) && ok )
    {
       dup = new Nodo;
       if (dup == NULL)
          ok = false;
       else
       {
          dup->info = p_aux->info;
          dup->sig = NULL;
          if (cima == NULL)
            cima = dup;
          else
            fin->sig = dup;
          fin = dup;
          p_aux = p_aux->sig;
       }
    }
    return (ok);
}

void Pila::Vaciar()
{
    while (Desapilar());
}
```

Operador asignación

El operador asignación tiene el mismo problema que la copia de objetos cuando se manejan punteros. Se trata también de copiar un objeto en otro. Para que se pueda utilizar de manera correcta el operador asignación sobre elementos de la clase pila implementada con punteros es preciso realizar lo que se llama una *sobrecarga* del operador asignación (=). Esto significa especificar cómo se debe realizar la asignación de objetos de esta clase. La posibilidad de ampliar el funcionamiento de los operadores estándar del lenguaje para nuevos tipos de datos es una característica específica de C++ que no posee otros lenguajes de programación. La implementación de la sobrecarga del operador = (`operator=`) sería idéntica a la del operador de copia (de ahí la conveniencia de haber definido la función `Copiar`).

```
const Pila& Pila::operator= (const Pila& p)
{
    if (! Copiar(p))
      cerr << "Error de memoria al copiar pila." << endl;
    return (*this);
```

```
}
```

Esto permite utilizar el operador = de manera natural sobre pilas. Por ejemplo,

```
Pila p, q;
p = q;
```

A la hora de implementar la sobrecarga de cualquier operador se debe tener en cuenta que:

1) El nombre del operador es siempre la palabra `operator` seguida del símbolo(s) que identifica al operador.

2) El número de argumentos del operador sobrecargado debe coincidir con los del operador original.

En el caso del operador asignación, se tiene un argumento del tipo pila (representa a la pila a la derecha de la asignación, ya que el objeto receptor de la operación es el que se sitúa a la izquierda del operador y está implícito en la función) y el valor de retorno es también una pila (por referencia constante), de igual manera que se define habitualmente el operador.

Destructor

El problema inverso a la copia de pilas se tiene cuando se debe liberar la memoria de las pilas (destruir). En C++ todos los objetos y variables definidas tiene un ámbito de declaración. Tiene un periodo de "vida". Fuera de ese ámbito los objetos dejan de existir, se destruyen. Eso implica la liberación de la memoria que ocupan. Sin embargo, cuando se manejan pilas con punteros el único espacio que se libera es (de nuevo) el ocupado por la variable cima. Mientras que los nodos creados para la pila permanecen en la memoria, ocupando un espacio que no se puede liberar. En C++, la solución es definir un destructor, cuya misión es inversa a la del constructor. Se trata de especificar que debe se debe de hacer antes de liberar la memoria del objeto. En nuestro, liberar antes la memoria ocupada por los nodos.

El destructor de una clase es único, a diferencia del constructor, y no tiene argumentos. Su nombre es el mismo de la clase, pero precedido del símbolo ~. Suponiendo que existe la función auxiliar `Vaciar` que se ha utilizado ya en la función `Copiar`, el destructor de la clase `Pila` tendría la siguiente forma:

```
Pila::~Pila ()
{
   Vaciar();
}
```

Si se reescribe el interfaz de la clase `Pila` para reflejar todas las operaciones introducidas se tiene que:

```
class Pila
{
   public:
      Pila ();
      Pila (const Pila&);
      ~Pila ();
      const Pila& operator= (const Pila&);
      bool Apilar (Valor);
```

```
        bool Desapilar ();
        bool CimaPila (Valor &);
        bool PilaVacia ();
    private:
        struct Nodo;
        typedef Nodo* Puntero;
        struct Nodo
        {
            Valor info;
            Puntero sig;
        };
        Puntero cima;
        void Vaciar();
        bool Copiar (const Pila&);
};
```

Las operaciones Copiar y Vaciar se han declarado como operaciones privadas, puesto que se trata de operaciones auxiliares que ayudan a construir otras operaciones pero que no se plantea que puedan ser utilizadas desde fuera de la clase.