TEMA 3
TIPOS ABSTRACTOS DE DATOS

1. Introducción a la programación con TADs
2. Especificación de TADs
3. Implementación de TADs
4. Módulos. Diseño modular
5. Estructuras de datos dinámicas

Bibliografía:

Fundamentals of Data Structures in C++
E. Horowitz, S. Sahni, D. Mehta
Computer Science Press, 1995

Data Abstraction and Problem Solving with C++, Second Edition
Carrano, Helman y Veroff

3.1 Introducción a la programación con tipos abstractos de datos

- La abstracción es un método de resolución de problemas

  Los problemas reales tienden a ser complejos porque involucran demasiados detalles, y resulta imposible considerar todas las posibles interacciones a la vez. Hay estudios en Psicología que afirman que en la memoria a corto plazo sólo es posible almacenar 7 elementos distintos.

- Una abstracción es un modelo simplificado de un problema, donde se contemplan los aspectos de un determinado nivel, ignorándose los restantes.

  Por ejemplo. Para indicarle a un robot cómo cambiar la rueda de un coche podemos utilizar una descripción con distintos niveles de abstracción:

  1. Bajar del coche
  2. Sacar la rueda de repuesto y el gato
  3. Quitar la rueda pinchada
  4. Poner la rueda de repuesto
  5. Guardar la rueda pinchada

  Para quitar la rueda pinchada

  3.1. Se coloca el gato debajo del coche

  3.2. Se levanta el coche haciendo girar la manivela del gato

  3.2. ...

  Este es un ejemplo de resolución de problemas por refinamientos sucesivos o diseño descendente.

- El razonar en términos de abstracciones conlleva una serie de ventajas:
  - La resolución de los problemas se simplifica
  - Las soluciones son más claras y resulta más sencillo razonar sobre su corrección
  - Pueden obtenerse soluciones (o fragmentos de ellas) de utilidad más general, que puedan adaptarse para resolver otros problemas
  - Permite la división de tareas

3.1.1 La abstracción como metodología de programación

- La programación es un proceso de resolución de problemas y como tal también se beneficia del uso de abstracciones.

- La evolución de la Programación muestra una tendencia a incluir mecanismos de abstracción cada vez de más alto nivel
  - Ensamblador
  - Lenguajes de alto nivel
    - Estructuras de control
    - Procedimientos y funciones
    - Tipos de datos
    - Módulos
    - Tipos abstractos de datos
    - Clases en programación orientada a objetos
  - Generadores de aplicaciones (hojas de cálculo, gestores de bases de datos, diseño visual de interfaces, ...)

- En todos estos mecanismos, aunque a diferente nivel, tenemos una situación como esta

Un mecanismo de abstracción es una barrera que, para una cierta *entidad*, deja a un lado la especificación, con la que los clientes de la abstracción pueden razonar, y a otro la implementación. De esta forma, los clientes sólo han de entender la especificación para utilizar la entidad, a través de su abstracción.

La abstracción funcional

- Hace tiempo que se *descubrió* y hoy en día es un lugar común en prácticamente todos los lenguajes de programación: las funciones y los procedimientos.
  - Se reúnen un conjunto de sentencias que realizan una determinada operación sobre unos datos (la implementación)
  - Se les asigna un nombre y se parametrizan por medio de argumentos, construyéndose así una abstracción del conjunto de sentencias: la cabecera del procedimiento o la función
  - Para que los clientes puedan usar esta abstracción (invocar al procedimiento o la función) se le adjunta una especificación que indica sus condiciones de uso –precondición– y la operación que realiza –postcondición–. La especificación también se puede realizar informalmente.

  De esta forma se puede utilizar ese conjunto de sentencias como si fuese una operación definida en el propio lenguaje, abstrayéndonos de los detalles de su implementación.

La abstracción de datos

- El nivel de abstracción de los datos ha ido aumentando a lo largo de los años
  - Tipos predefinidos

    Por ejemplo, en un lenguaje de programación de alto nivel, generalmente, no debemos preocuparnos por la representación binaria de los caracteres, los enteros o los reales.
  - Tipos de datos definidos por el programador

    Por ejemplo, un array de un cierto tipo se representa internamente como una sucesión de celdas de memoria adyacentes, donde el acceso directo se consigue mediante sencillas operaciones aritméticas.
  - Tipos abstractos de datos (TADs) definidos por el programador

    Una entidad donde se reúnen un tipo de datos y las operaciones que manipulan los valores de ese tipo.

- Para entender el concepto de tipo abstracto de datos analicemos la diferencia entre los tipos predefinidos y los tipos definidos por el programador.
  - Un tipo predefinido es un tipo abstracto de datos porque
    - Está fijado el conjunto de valores permitidos y los compiladores se encargan de garantizar que no se asignan valores erróneos. No es posible acceder directamente a la representación interna de los datos[1].
    - Está fijado el conjunto de operaciones permitidas sobre dichos valores y el compilador se encarga de garantizar que se usan correctamente.
    - Como programadores sólo necesitamos conocer el comportamiento de las operaciones (por ejemplo, los enteros se rigen por unas leyes algebraicas conocidas por todos) sin que tengamos acceso a su implementación concreta, ni nos interese lo más mínimo.
  - Un lenguaje no soporta la implementación de tipos abstractos de datos si
    - La estructura de datos donde se representa la información se define por una parte y las operaciones que manipulan esos datos por otra.
    - No es posible definir una abstracción que reúna la estructura de datos y las operaciones de forma que
      - la representación de los datos y la implementación de las operaciones sea oculta para los clientes y, por lo tanto,
      - el acceso a la información esté protegido y los clientes no puedan hacer ninguna suposición sobre la implementación de las operaciones que no aparezca explícitamente en la especificación.

  Por ejemplo, si queremos definir un tipo de datos para representar fechas, podemos utilizar una representación como esta:

```
typedef int TDia;
typedef enum { enero, febrero, ... , diciembre } TMes;
typedef int TAnyo;
typedef struct { TDia dia; TMes mes; TAnyo anyo; } TFecha;
```

  Si se tiene libre acceso a la representación del tipo de datos, es posible realizar operaciones absurdas con respecto a la semántica pretendida, como:

```
fecha.mes = febrero;
fecha.dia = 30;
```

[1] Esto no es cierto en general, pues normalmente se pueden hacer manipulaciones binarias sobre los valores de tipos predefinidos. Sin embargo, debemos considerar que este es un mecanismo "peligroso" con el cual saltarnos la barrera de la abstracción.

3.1.2 Tipos abstractos de datos

- Un tipo abstracto de datos se compone de
  - Una especificación
  - Una implementación

Especificación de un tipo abstracto de datos

- La especificación de un TAD debe incluir:
  - El dominio de valores del tipo
  - Las operaciones que permiten manipular los valores del tipo –y que pueden hacer referencia a otros tipos–
  - El comportamiento de las operaciones (especificado formal o informalmente)

- Como ejemplo, escribamos la especificación informal del TAD *fecha*
  - Dominio: las fechas desde el 1/1/1
  - Las operaciones, con su comportamiento especificado informalmente

```
TFecha NuevaFecha ( TDia d, TMes m, TAnyo a );
// P: d/m/a cumplen las restricciones de una fecha, según el
calendario
// Q: devuelve una fecha que representa a la fecha d/m/a

int distancia ( TFecha f1, TFecha f2 );
// P:
// Q: devuelve la distancia, medida en número de días, entre
f1 y f2

TFecha suma ( TFecha f, int d );
// P:
// Q: devuelve la fecha resultante de sumar d días a la
fecha f
```

¡Cuidado! *d* puede ser un número negativo de forma que *g* no sea una fecha posterior al 1/1/0. Algunas operaciones pueden fallar y es necesario determinar algún mecanismo de tratamiento de errores.

```
TDia dia( TFecha f );
// P:
// Q: devuelve el día de la fecha f

TMes mes( TFecha f );
// P:
// Q: devuelve el mes de la fecha f

TAnyo anyo( TFecha f );
// P:
// Q: devuelve el año de la fecha f

TDiaSemana diaSemana( TFecha f );
// P:
// Q: devuelve el día de la semana correspondiente a la
fecha f

TFecha primerDiaMes( TDiaSemana d, TMes m, TAnyo a );
// P:
// Q: devuelve la primera fecha del mes m del año a cuyo
//    día de la semana es d
```

Implementación de un tipo abstracto de datos

- La implementación de un TAD consiste en:
  - Elegir una estructura de datos para representar los valores del TAD con ayuda de otros tipos ya disponibles.
  - Implementar las operaciones del TAD como funciones o procedimientos, usando la representación elegida, y de modo que se satisfaga la especificación.

- En general cualquier TAD admite varias representaciones posibles.
  - Por ejemplo, una secuencia de números de longitud variable se puede implementar como
    - un registro con un vector y un campo longitud,
    - o como un vector con una marca al final.

- Cuando hay varias representaciones posibles, normalmente una representación concreta facilita unas determinadas operaciones y dificulta otras, con respecto a una representación alternativa.
  - Por ejemplo, para el TAD *fecha* que especificamos más arriba podemos optar entre
    - Los registros que vimos más arriba

      Con esta representa resulta trivial implementar *NuevaFecha*, *dia*, *mes* y *anyo*. Mientras que sería más difícil implementar el resto de las operaciones.
    - El número de días transcurridos desde el 1 de Enero de 1900

      Así es mucho más sencillo calcular el número de días transcurridos entre dos fechas dadas, pero resulta más difícil construir una fecha a partir del día/mes/año.

- Un lenguaje soporta la implementación de tipos abstractos de datos si incluye mecanismos que permitan separar la especificación de la implementación, en particular
  - Privacidad: la representación interna (estructura de datos e implementación de las operaciones) está oculta y es invisible para los usuarios.

    "Lo que no conoces no puede hacerte daño"
  - Protección: el tipo sólo puede usarse a través de sus operaciones; otros accesos incontrolados se hacen imposibles.

  Si un lenguaje no incluye estos mecanismos, entonces el *respeto* de la abstracción se convierte en una cuestión de disciplina del programador.

- ¿Por qué es importante que el lenguaje soporte la implementación de TADs?

  Aceptando que los TADS son un mecanismo de abstracción adecuado resulta mucho más cómodo y fiable que el lenguaje nos prohíba los malos usos –o al menos nos avise de ellos– a que se trate de una cuestión de autodisciplina. Y más aún en un entorno colaborativo con decenas de personas participando en el desarrollo de un sistema.

- En resumen, decimos que un lenguaje soporta el uso de tipos abstractos de datos si permite elevar los tipos definidos por el programador al rango de los tipos predefinidos, en el sentido indicado más arriba.

  El grado de soporte que proporcionan los lenguajes de programación para la definición de tipos abstractos de datos varía mucho. Aunque con el éxito de la programación orientada a objetos, cada vez es más habitual encontrar este tipo de mecanismos.

Los TADs como apoyo a la programación modular

- El desarrollo de programas grandes y complejos se realiza descomponiendo el programa en unidades menores llamadas *módulos*.

  El problema es, dado un cierto problema ¿cómo descompongo su solución en módulos? ¿cuál es el *diseño modular* de la aplicación?

  – La Ingeniería del software nos aconseja realizar diseños modulares en torno a las *clases de objetos* que maneja la aplicación, en lugar de organizarla en base a las funcionalidades que se implementan

  clase de objetos ≡ datos + operaciones ≡ implementación de TADs

- La abstracción en general y los TADs en particular facilitan el desarrollo modular y la división de tareas ya que la elección de una implementación concreta puede posponerse o variarse según convenga, sin afectar a los programas ya realizados, o en proceso de diseño, que utilicen el TAD en cuestión.

Un ejemplo

- Dado un vector *v* de números enteros con índices entre 1 y N y un número *k*, $0 \leq k \leq N$, se trata de determinar los *k* números mayores que aparecen en el vector –nótese que no tienen que ser *k* números diferentes–.

- Para resolver este problema necesitamos una estructura auxiliar donde vayamos almacenando los *k* mayores números encontrados hasta ahora. Tenemos dos opciones

  – Elegir una estructura de datos concreta –por ejemplo, un vector– e incluir su gestión dentro del propio algoritmo.

  Los inconvenientes de esta decisión:

  – La lógica del algoritmo queda oscurecida por los detalles de la representación.

  – La estructura de datos elegida a priori puede no ser la más eficiente, ya que para saber cuál es la elección más eficiente debemos saber cuáles son las operaciones necesarias sobre ella, y eso no lo sabremos hasta que hayamos diseñado el algoritmo que la utiliza.

- Pospongamos la elección de la estructura de datos, suponiendo que disponemos del TAD necesario para almacenar y acceder a la información necesaria, y concentrémonos en diseñar el algoritmo.

  Posteriormente, recolectaremos las operaciones que el algoritmo necesita y a partir de ellas especificaremos e implementaremos el TAD. En ese momento tendremos información adicional sobre qué operaciones debemos preocuparnos por optimizar.

  Ventajas de esta aproximación:

  - Descomponemos la tarea
  - El TAD resultante puede utilizarse para otros fines (reutilización)
  - La lógica del algoritmo debe quedar más clara

- Idea del algoritmo
  - Inicializamos la estructura auxiliar *m* con los *k* primeros elementos de *v*
  - Desde *k*+1 hasta *N*
    - Si v[*i*] es mayor que el mínimo de *m* entonces sustituimos éste por v[*i*]

Siguiendo esta idea, la implementación podría ser

```
TMCjtoInt mayores( int k, int v[], int num ) {
// P: ( 1 <= k <= num ) && ( num >= 1 )
//    v contiene al menos num elementos
  int n;
  TMCjtoInt m;

  vacio(m);
  n = 0;
  for ( n = 0; n < k; n++ )
    // m contiene los elementos de v[0..n]
    inserta( v[n], m );

  for ( n = k; n < num; n++ )
  // m contiene los k elementos mayores de v[0..n-1]
    if ( v[n] > min( m ) ) {
      borraMin( m );
      inserta( v[n], m );
    }
  return m;
// Q: devuelve un multiconjunto con los k elementos
//    mayores de v[0..num-1]
}
```

- El TAD ha de ser capaz de almacenar una colección de números enteros, algunos de los cuales pueden estar repetidos (*multiconjuntos*) y proporcionar operaciones para
  - Construir un multiconjunto vacío
  - Añadir un elemento
  - Quitar el mínimo de los elementos
  - Consultar por el mínimo de los elementos

  La especificación informal de las operaciones:

```
void vacio( TMCjtoEnt& m );
// P:
// Q: m representa ∅
void inserta( int x, TMCjtoEnt& m );
// P: m = M
// Q: m = M ∪ {x}
void borraMin( TMCjtoEnt& m );
// P: m = M ∧ m no es vacío
// Q: m es M menos una copia del elemento mínimo de M
int min( TMCjtoEnt m );
// P: m no es vacío
// Q: devuelve el elemento mínimo de m
```

  Un multiconjunto se puede implementar de distintas formas:
  - Un registro formado por un vector de enteros y un entero que indica cuántos elementos hay
  - Un registro con un entero que indica el número de elementos y un vector de registros de dos componentes, una que indica el número almacenado y otra el número de apariciones de ese valor
  - Para cada una de las dos anteriores
    - Sin orden
    - Ordenado creciente
    - Ordenado decreciente

  Si interesa optimizar las operaciones *inserta*, *min* y *borraMin*, ¿qué representación es más interesante?

Tipos abstractos de datos y estructuras de datos

- Una *estructura de datos* permite representar la información gestionada por un TAD, es decir, es la implementación que se utiliza para almacenar los valores del tipo.

- Sin embargo, en muchas ocasiones, se utiliza un TAD T_{aux} para implementar otro TAD T. Entonces ¿qué es T_{aux}?
  - ¿Una estructura de datos?
  - ¿Un TAD?
  - ¿Las dos cosas?

- Definimos, de manera no del todo precisa, una estructura de datos como un tipo definido a partir de los tipos primitivos (enteros, caracteres, reales, …) y las constructoras de tipos (registros, arrays, punteros, …) que incluyen la mayoría de los lenguajes de programación

- Sin embargo, una parte importante de esta segunda parte del curso la dedicaremos a estudiar las estructuras de datos más utilizadas junto con la implementación de las operaciones más habituales en su uso, explorando distintas alternativas.
  - Pero, ¡ esa es precisamente la definición de TAD ! : una estructura de datos junto con las operaciones que la manejan.
  - De hecho, resulta interesante abstraer las estructuras de datos más habituales y convertirlas en TADs.

- En resumen, nos vamos a dedicar a estudiar estructuras de datos diseñadas como tipos abstractos de datos, y a estudiar los tipos abstractos de datos más habituales junto con las estructuras de datos que permiten implementarlos eficientemente.

3.2 Especificación algebraica de TADs

- Una posible especificación para un tipo abstracto de datos consistiría en:
  - Identificar el conjunto de valores posibles
  - Escribir una especificación pre/post de cada una de las operaciones.

- La *especificación algebraica* es un enfoque diferente que consiste en imaginar las operaciones de un TAD como análogas a las operaciones de un álgebra[2] y describir su comportamiento por medio de *ecuaciones*: igualdades entre aplicaciones de las operaciones

  Las ecuaciones son más simples que las fórmulas lógicas utilizadas en la especificación pre/post formal y además
  - El razonamiento con ecuaciones es relativamente sencillo y natural
  - Las ecuaciones no sólo especifican las propiedades de las operaciones, sino que especifican también cómo construir los valores del TAD

Mecanismos básicos: especificación de los booleanos

- Una especificación algebraica consta fundamentalmente de tres componentes
  - Tipos: son nombres de dominios de valores. Entre ellos está siempre el *tipo principal* del TAD, aunque puede haber también otros que se relacionen con éste.
  - Operaciones: deben ser funciones con un perfil asociado, que indique el tipo de cada uno de los argumentos y el tipo del resultado. En una especificación algebraica no se permiten funciones que devuelvan varios valores, ni tampoco procedimientos no funcionales.
  - Ecuaciones entre términos formados con las operaciones y *variables de tipo* adecuadas.

[2] Un álgebra es un dominio de valores equipado con operaciones que cumplen axiomas algebraicos dados.

Signatura de un TAD

- Definimos la *signatura* de un TAD como los tipos que utiliza, junto con los nombres y los perfiles de las operaciones –sin incluir las ecuaciones–.

- La signatura de los booleanos

```
tipo
  Bool
operaciones
  Cierto, Falso : → Bool
  not : Bool → Bool
  ( and ), ( or ) : (Bool, Bool) → Bool
```

  En la signatura de una operación utilizamos paréntesis para indicar notación infija, a diferencia de la notación prefija por defecto

- Para escribir las ecuaciones es necesario clasificar las operaciones según el papel que queremos que jueguen en relación con el tipo principal:
  - Generadoras (o constructoras). Pensadas para construir todos los valores de tipo τ. Tienen un perfil de la forma

    $c : \vec{\tau} \rightarrow \tau$
  - Modificadoras. Pensadas para hacer cálculos que produzcan resultados de tipo τ, pero que no son generadoras. Tienen un perfil de la forma

    $f : \vec{\tau} \rightarrow \tau$
  - Observadoras. Pensadas para obtener valores de otros tipos a partir de valores de tipo τ. Tienen un perfil de la forma

    $g : \vec{\tau} \rightarrow \tau'$ tal que $\tau' \equiv \tau$; y algún $\tau_i \equiv \tau$

- En este ejemplo
  - Generadoras: Cierto, Falso
  - Modificadoras: not, and, or

- Por convenio se suele escribir el nombre de las generadoras con la primera letra en mayúscula.

Términos

- Dada la signatura de un TAD, con tipo principal τ, y un conjunto de variables X es posible construir un conjunto (normalmente infinito) de *términos* de tipo τ mediante la aplicación de las operaciones del TAD.

  Cada término representa una aplicación sucesiva de operaciones del TAD.

- En el caso de los booleanos el conjunto de términos es de la forma

  T_{Bool} = { Cierto, Falso, not Cierto, Cierto, Falso or not Cierto, ... }

- Un tipo especialmente importante de términos son aquellos que sólo contienen operaciones generadoras: los *términos generados*.

  Es necesario que las generadoras permitan construir al menos un término distinto para cada posible valor del tipo que pretendemos especificar

  TG_{Bool} = { Cierto, Falso }

Ecuaciones

- Las ecuaciones deben reflejar el comportamiento de las operaciones para cualquier aplicación correcta de las mismas.
  - Las ecuaciones deben permitir convertir cualquier término en un término generado: el resultado de la secuencia de operaciones que representa el término.
  - Mediante las ecuaciones ha de ser posible deducir todas las equivalencias que son válidas entre los términos, es decir, identificar las secuencias de operaciones que producen el mismo resultado.
  - Se deben evitar las ecuaciones redundantes.

- ¿Qué ecuaciones necesitamos para especificar a los booleanos?
  - not

    ```
    (1) not Cierto  = Falso
    (2) not Falso                    = Cierto
    ```

  - and

    ```
    (3) Cierto and x= x
    (4) Falso and x  = Falso
    ```

  - or

    ```
    (5) Cierto or x  = Cierto
    (6) Falso or x   = x
    ```

- Usando estas ecuaciones podemos convertir cualquier término en un término generado. Por ejemplo,

```
Cierto and Cierto and Falso =3 Cierto and Falso =3 Falso
not Falso or Cierto =2 Cierto or Cierto =5 Cierto
```

- Y es posible detectar las equivalencias entre los términos

```
¿ Cierto and not Cierto = Falso and not Falso ?

      Cierto and not Cierto          Falso and not Falso
   3= not Cierto                  4= Falso
   1= Falso
```

Hemos obtenido el término generado al que era equivalente cada término y hemos comprobado que los dos son equivalentes al mismo.

- Con todo lo anterior, la especificación de los booleanos queda finalmente

```
tad BOOL
tipo
  Bool
operaciones
  Cierto, Falso : → Bool                        /* gen */
  not : Bool → Bool                             /* mod */
  ( and ), ( or ) : (Bool, Bool) → Bool         /* mod */
ecuaciones
    ∀ x : Bool
    not Cierto                                    = Falso
    not Falso    = Cierto
    Cierto and x = x
    Falso and x  = Falso
    Cierto or x  = Cierto
    Falso or x                                       = x
ftad
```

TADs genéricos

- Los TADs genéricos son TADs que dependen de otros –uno o más– de forma que es posible construir distintos ejemplares del TAD genérico según el tipo de los parámetros.

- Los TADs genéricos representan típicamente colecciones de elementos.

  Estos TADs tienen un cierto comportamiento independiente del tipo de los elementos, aunque también puede haber operaciones que dependan de los elementos.

  Para expresar las exigencias sobre los elementos en especificación algebraica se utilizan *clases de tipos*. De esta forma, sólo se pueden construir ejemplares del TAD genérico utilizando TADs que pertenezcan a la clase indicada en su especificación.

Clases de tipos

- Una clase de tipos especifica el conjunto de operaciones que deben incluir los TADs que pertenecen a dicha clase.

  Los TADs pueden incluir operaciones adicionales

- Las clases de tipos se organizan en una jerarquía con herencia, donde la clase ANY ocupa la raíz:

```
clase ANY
 tipo
   Elem
fclase
```

  Todos los TAD pertenecen a la clase de tipos ANY, siendo *Elem* un sinónimo del tipo principal del TAD.

- La clase de los tipos con igualdad

```
clase EQ
 hereda
   ANY
 usa
   BOOL
 operaciones
   ( == ), ( /= ) : (Elem, Elem) → Bool
 axiomas
   // ( == ) es una operación de igualdad.
 ecuaciones
   ∀ x, y : Elem :
   x /= y   =  not (x == y)
fclase
```

No se escriben las ecuaciones de la operación == porque no podemos suponer ningún conjunto de generadoras.

- La clase de los tipos con orden

```
clase ORD
 hereda
   EQ
 operaciones
   ( ≤ ), ( ≥ ), ( < ), ( > ) : (Elem, Elem) → Bool
 axiomas
   // ( ≤ ) es una operación de orden
 ecuaciones
   ∀ x, y : Elem :
   x ≥ y    =   y ≤ x
   x < y    =   (x ≤ y) and (x /= y)
   x > y    =   y < x
fclase
```

Operaciones parciales: especificación de las pilas

- En muchas ocasiones tendremos que especificar TADs que incluyan operaciones parciales. En ese caso existirán términos que no están definidos y que deben recibir un tratamiento *cuidadoso*.

- Como ejemplo construiremos la especificación del TAD genérico PILA. Una pila representa una colección de valores donde sólo es posible acceder al último elemento añadido.

```
tad PILA[E :: ANY]
 usa
   BOOL
 tipo
   Pila[Elem]
 operaciones
   PilaVacía: → Pila[Elem]                        /* gen */
   Apilar: (Elem, Pila[Elem]) → Pila[Elem]        /* gen */
   desapilar: Pila[Elem] – → Pila[Elem]           /* mod */
   cima: Pila[Elem] – → Elem                      /* obs */
   esVacía: Pila[Elem] → Bool                     /* obs */
```

  Las operaciones *desapilar* y *cima* son parciales porque no tiene sentido quitar un elemento ni consultar por la cima de una pila vacía. El carácter parcial de una operación se nota con una flecha discontinua, $-\!\!\rightarrow$, en su perfil.

- Para escribir las ecuaciones de operaciones parciales utilizamos un nuevo elemento en las especificaciones: los axiomas de definición.
  - Los axiomas de definición determinan la forma que tienen los términos definidos que involucran operaciones parciales. En el ejemplo de las pilas:

```
def desapilar(Apilar(x, xs))
def cima(Apilar(x, xs))
```

- Cuando hay operaciones parciales, además de los axiomas de definición, incluimos una sección *errores* es la especificación, donde se indica qué forma tienen los términos indefinidos. Aunque, en general, esto es información redundante, se incluye por claridad.
  - En el caso de las pilas

```
errores
  desapilar(Pilavacía)
  cima(PilaVacía)
```

- Con todo esto, la especificación de las pilas queda

```
tad PILA[E :: ANY]
usa
  BOOL
tipo
  Pila[Elem]
operaciones
  PilaVacía: → Pila[Elem]                      /* gen */
  Apilar: (Elem, Pila[Elem]) → Pila[Elem]      /* gen */
  desapilar: Pila[Elem] – → Pila[Elem]         /* mod */
  cima: Pila[Elem] – → Elem                    /* obs */
  esVacía: Pila[Elem] → Bool                   /* obs */

ecuaciones
  ∀ x : Elem : ∀ xs : Pila[Elem] :
  def desapilar(Apilar(x, xs))
  desapilar(Apilar(x, xs))                        = xs
  def cima(Apilar(x, xs))
  cima(Apilar(x, xs))  = x
  esVacía(PilaVacía)   = Cierto
  esVacía(Apilar(x, xs))                        = Falso

errores
  desapilar(Pilavacía)
  cima(PilaVacía)
ftad
```

3.3 Implementación de TADs

- Dada la especificación de un TAD T, su implementación consiste en:
  - Un *dominio concreto* D_τ para cada tipo τ incluido en T.

    Los dominios concretos se implementan mediante declaraciones de tipos, usando otros tipos ya implementados –por nosotros o incluidos en el lenguaje– que definen el *tipo representante*.
  - Una operación concreta

    $f_C : D_{\tau 1}, \ldots, D_{\tau n} \rightarrow D_\tau$

    para cada operación de la especificación

    $f : \tau_1, \ldots, \tau_n \rightarrow \tau$

    Las operaciones concretas pueden implementarse como procedimientos aunque en la especificación algebraica sólo se admitan funciones.

- La implementación ha de satisfacer dos requisitos:
  - Corrección: la implementación debe satisfacer las ecuaciones de la especificación
  - Privacidad y protección: la estructura interna de los datos debe estar oculta y el único acceso posible al tipo debe ser a través de las operaciones públicas de éste.

    Dependiendo del lenguaje de implementación y los mecanismos de modularidad que proporcione, este requisito se podrá reflejar directamente en la implementación o se convertirá en una cuestión de disciplina de uso.

- En las implementaciones utilizaremos C++.

3.3.1 Implementación correcta de un TAD: implementación de las pilas de enteros

Implementación del tipo

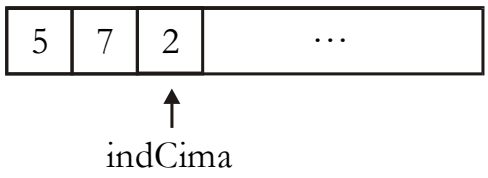
- Pretendemos implementar el TAD PILA[INT].
  - En primer lugar, consideramos que el tipo predefinido *int* implementa correctamente el TAD INT, donde el tipo representante de *Int* es *int* y las operaciones del TAD están implementadas como las operaciones predefinidas disponibles sobre *int*.
  - Una vez fijada la implementación del TAD de los elementos, debemos decidir cuál es el tipo representante de *Pila[Int]*.

    En este caso elegimos representar las pilas como un registro con dos campos, uno que es un vector donde se almacenan los elementos y otro que es un índice que apunta a la cima de la pila dentro del vector.

    Por ejemplo la representación de

    ```
    Apilar( 2, Apilar( 7, Apilar( 5, PilaVacía )))
    ```

    vendría dada por:



    Esta implementación tiene el inconveniente de que impone a priori un límite al tamaño máximo de la pila.

- El tipo representante escogido es:

```
int const limite = 100;

typedef struct {
  int espacio[limite];
  int indCima;
} TPilaInt;
```

- Para luego poder plantear una implementación correcta de las operaciones, conviene comenzar definiendo qué valores concretos queremos aceptar como *representantes válidos* de valores abstractos del TAD –la idea es que el tipo representante elegido puede tomar valores que no consideremos válidos–.

Invariante de la representación

- Empezamos formalizando qué condiciones les exigimos a los representantes válidos: el *invariante de la representación*.

$$
\begin{aligned}
& R_{Pila[Int]}\ (xs) \\
\Leftrightarrow_{def}\ & xs : TPilaInt \wedge \\
& -1 \leq xs.indCima < limite \wedge \\
& \forall\ i : 0 \leq i \leq xs.indCima : R_{Int}(xs.espacio[i])
\end{aligned}
$$

- Los representantes válidos son aquellos valores del tipo representante que cumplen el invariante de la representación.

- El invariante de la representación está relacionado con la corrección de la implementación de la siguiente forma:
  - Las operaciones sólo están obligadas a funcionar sobre representantes válidos.
  - Las operaciones que generen valores del tipo representante deben comprometerse a que éstos sean siempre representantes válidos.

- Lo importante no es escribir el invariante de la representación formalmente, sino determinar (aún con una descripción informal) de entre todos los valores que puede tomar el tipo representante cuáles representan valores válidos del TAD que estamos diseñando.

Implementación de las operaciones

- Para cada operación de la especificación

  $f : (\tau_1, \ldots, \tau_n) - \rightarrow \tau$

  debemos implementar una operación f_C que satisfaga una especificación pre/post de la forma

```
TRτ fC ( TRτ1 x1, … , TRτn xn ) {
// P : Rτ1(x1) ∧ … ∧ Rτn(xn) ∧ DOMf(x1, …, xn) ∧ LIMf(x1, …,
xn)
   TRτ y;
// Q : Rτ(y) ∧ y = f(x1, …, xn)
  return y;
}
```

  donde

  - $TR_{\tau i}$, TR_τ son los tipos representantes de los argumentos y el resultado
  - $R_{\tau i}$, R_τ son los invariantes de la representación para dichos tipos
  - DOM_f es la condición de dominio, donde se imponen las restricciones necesarias para que la operación esté definida

    $DOM_f(x_1, \ldots, x_n) \Leftrightarrow$ **def** $f(x_1, \ldots, x_n)$
  - LIM_f expresa limitaciones adicionales impuestas por la implementación

- Por ejemplo,
  - la especificación de *Apilar* de *Pila[Int]*

```
void Apilar ( int x, TPilaInt & xs ) {
 // P: RInt(x) ∧ RPila[Int](xs) ∧ xs.indCima < límite-1 ∧ xs =
 XS
 // Q : RPila[Int](xs) ∧ xs = Apilar(x, XS)
 }
```

    donde *xs.indCima* < *límite – 1* es una limitación de la implementación

  - la especificación de *cima* de *Pila[Int]*

```
int cima ( TPilaInt xs ) {
// P : RPila[Int](xs) ∧ xs.indCima > -1
  int x;
// Q : RInt(x) ∧ x = cima(xs)
   return x;
}
```

    donde *xs.indCima* > –1 es una condición de dominio, ya que, de acuerdo con la especificación, *cima* no está definido sobre una pila vacía.

- Existen técnicas formales que permiten demostrar la corrección de la implementación de las operaciones con respecto a la especificación algebraica. Estas técnicas se basan en el uso de *funciones de abstracción* que permiten traducir los valores representantes al modelo formal de la especificación, en términos del cual se puede realizar la demostración.

- Las anteriores especificaciones exponen detalles de la implementación elegida, violando así el principio de ocultación: *xs.indCima* < *limite–1* y *xs.indCima* > –1. Esto se puede resolver sustituyendo estas condiciones por funciones booleanas que las comprueben.
  - Normalmente las condiciones de dominio se podrán expresar directamente utilizando alguna operación de la especificación (*esVacía*).
  - Para expresar las limitaciones de la implementación será necesario añadir nuevas funciones a las que aparezcan en la especificación (*esLlena*).

```
void Apilar ( int x, TPilaInt & xs )
 // P: R_Int(x) ∧ R_Pila[Int](xs) ∧ ¬ esLlena(xs) ∧ xs = XS

 int cima ( TPilaInt xs )
 // P : R_Pila[Int](xs) ∧ ¬ esVacia(xs)
```

  Donde la función *esVacia* ya forma parte de la especificación y la función *esLlena* se puede especificar de la siguiente forma:

```
bool esLlena ( TPilaInt xs ) {
// P: R_Pila[Int](xs)
// Q: xs esLlena si tiene 'limite' elementos
}
```

  Nótese que aunque esta función viola el principio de ocultación (nos hace suponer que la pila se ha implementado en un array de *limite* elementos) es una información que se debe proporcionar a los usuarios del TAD.

- El uso de funciones booleanas en las precondiciones y su consiguiente implementación proporciona además la ventaja de permitir a los usuarios del TAD comprobar si una precondición se cumple antes de invocar a una determinada operación.

```
if ( ! esLlena(xs) )
  Apilar( x, xs );
```

- ¿Y qué hacer con el invariante de la representación?

  Podemos aplicar la misma idea e incluir en el TAD una función booleana *esValido* que, dado un valor del tipo representante, determine si es un representante válido.

  En el ejemplo de *Pila[Int]* que nos ocupa, dicha operación se especifica como:

```
bool esValido ( TPilaInt xs ) {
// P:
// Q: determina si xs es un representante válido de TPilaInt
}
```

  Una función como esta puede resultar de gran utilidad para detectar errores en la implementación de un TAD.

- Con todo esto, y considerando que
  - los valores de tipos primitivos siempre cumplen su invariante de la representación, y
  - las condiciones de la forma $y = f(x_1, \ldots, x_n)$ aportan poca información,

  la especificación pre/post de las operaciones queda:

```
void Apilar ( int x, TPilaInt & xs ) {
// P: esValido(xs) && ! esLlena(xs) && xs = XS
// Q: esValido(xs) && xs es el resultado de apilar x en XS
}

int cima ( TPilaInt xs ) {
// P: esValido(xs) && ! esVacia(xs)
// Q: devuelve la cima de xs
}
```

- Nótese que en la última versión de las especificaciones se han sustituido los operadores lógicos por sus equivalentes en C++. Esto nos hace pensar que las precondiciones son evaluables (salvo las condiciones sobre variables auxiliares de la especificación del tipo *xs = XS*) y que se pueden implementar las operaciones de un TAD comprobando que se cumple la precondición:

```
if ( P )
  // implementación de la operación
else
  error("No se cumple la precondición");
```

  Evidentemente, estas comprobaciones adicionales suponen una penalización en el tiempo de ejecución de las operaciones.

- Es una técnica habitual añadir comprobaciones adicionales durante el desarrollo de un TAD para ayudar a depurarlo, para luego suprimir dichas comprobaciones en la versión definitiva. Un forma sencilla de implementar esta idea consiste en añadir una variable booleana global que controle si se han de evaluar las precondiciones.

```
int cima ( TPilaInt xs ) {
// P: esValido(xs) && ! esVacia(xs)
  int r;
  if ( ! depurando || ( esValido(xs) && ! esVacia(xs) ) )
    r = xs.espacio[xs.indCima];
  else
    error("No se cumple la precondición de cima");
  return r;
// Q: devuelve la cima de xs
}
```

- El problema de esta técnica es que supone que en algún momento estaremos seguros de que nuestra aplicación es correcta y podremos *desactivar* la evaluación de las condiciones de corrección.

  Sin embargo, en general, esta es una suposición demasiado arriesgada.

- Una solución alternativa, y más habitual, es mantener en la versión final una parte de las condiciones de corrección, las que se refieren a las condiciones de dominio y los límites de la implementación, pero no así la comprobación de la validez de los valores.

- Para facilitar el uso de los TADs incluiremos el tratamiento de los errores debidos a las condiciones de dominio y los límites de la implementación, para lo cual existen distintas alternativas:
  - mostrar un mensaje y detener la ejecución
  - mostrar el mensaje y no detener la ejecución
  - implementar un mecanismo que permita a los clientes del TAD saber si se ha producido un error y actuar en consecuencia

  La más *amigable* para el cliente del TAD es esta última, sin embargo, para llevarla a la práctica es aconsejable que el lenguaje incluya algún mecanismo que facilite su implementación.

Implementación de las operaciones de *TPilaInt*

- Con todo esto, la implementación de las operaciones de *TPilaInt* queda de la siguiente forma

```
// función auxiliar para identificar los representantes
válidos

bool esValido ( TPilaInt xs ) {
// P:
  return ( xs.indCima >= -1 ) && ( xs.indCima < limite );
// Q: determina si xs es un representante válido de TPilaInt
}

// función auxiliar para determinar si una pila está llena

bool esLlena ( TPilaInt xs ) {
// P: esValido(xs)
  return xs.indCima == limite - 1;
// Q: xs esLlena si tiene 'limite' elementos
}

// función auxiliar privada para mostrar un mensaje de error

void error( string mensaje ) {
  cout <<  "Error: " + mensaje;
}

// operaciones incluidas en la especificación

void PilaVacia( TPilaInt & xs ) {
// P:
  xs.indCima = -1;
// Q: esValido(xs) && xs representa a la pila vacía
}

void Apilar ( int x, TPilaInt & xs ) {
// P: esValido(xs) && ! esLlena(xs) && xs = XS
  if ( esLlena(xs) )
    error("No se puede apilar porque la pila está llena");
  else {
    xs.indCima = xs.indCima + 1;
    xs.espacio[xs.indCima] = x;
  }
// Q: esValido(xs) && xs es el resultado de apilar x en XS
}
bool esVacia ( TPilaInt xs ) {
// P: esValido(xs)
  return xs.indCima == -1;
// Q: determina si xs está vacía
}
```

```
int cima ( TPilaInt xs ) {
// P: esValido(xs) && ! esVacia(xs)
  int r;
  if ( esVacia( xs ) )
    error("La pila vacía no tiene cima");
  else
    r = xs.espacio[xs.indCima];
  return r;
// Q: devuelve la cima de xs
}

void desapilar( TPilaInt & xs ) {
// P: esValido(xs) && ! esVacia(xs) && xs = XS
  if ( esVacia(xs) )
      error("no se puede desapilar de la pila vacía");
  else
    xs.indCima = xs.indCima - 1;
// Q: esValido(xs) && xs es el resultado de desapilar de XS
}
```

3.3.2 Distintas implementaciones de un mismo TAD: implementación de los conjuntos de enteros

- Vamos a plantear tres posibles implementaciones para el TAD *CJTO[INT]*:
  - Vectores de elementos de tipo *int*
  - Vectores de *int* sin repetición
  - Vectores de *int* sin repetición y ordenados

- En los tres casos elegimos el mismo tipo representante

```
int const longMax = 100;
typedef struct {
  int espacio[longMax];
  int longitud;
} TCjtoInt;
```

- El invariante de la representación
  - Vectores

$$R_{Cjto[Int]}(xs) \Leftrightarrow_{def} xs : TCjtoInt \wedge 0 \leq xs.longitud \leq longMax \wedge \forall\, i : 0 \leq i < xs.longitud : R_{Int}(xs.espacio[i])$$

  - Vectores sin repetición

$$R_{Cjto[Int]}(xs) \Leftrightarrow_{def} xs : TCjtoInt \wedge 0 \leq xs.longitud \leq longMax \wedge \forall\, i : 0 \leq i < xs.longitud : R_{Int}(xs.espacio[I]) \wedge \forall\, i, j : 0 \leq i < j < xs.longitud : xs.espacio[i] \mathrel{!}= xs.espacio[j]$$

  - Vectores sin repetición y ordenados

$$R_{Cjto[Int]}(xs) \Leftrightarrow_{def} xs : TCjtoInt \wedge 0 \leq xs.longitud \leq longMax \wedge \forall\, i : 0 \leq i < xs.longitud : R_{Int}(xs.espacio[i]) \wedge \forall\, i, j : 0 \leq i < j < xs.longitud: xs.espacio[i] < xs.espacio[j]$$

Implementación de las operaciones

Operaciones auxiliares

- En los tres casos necesitamos una función de búsqueda, en los dos primeros búsqueda secuencial mientras que en el tercero podemos usar búsqueda binaria

- Búsqueda secuencial en un intervalo de un array de *int*. Necesitamos especificar a partir de dónde buscar porque tendremos que realizar búsquedas sucesivas (posibles repeticiones).

```
void busca ( int x , int v[], int ini, int num,
             bool & encontrado, int & pos       ) {
// P: v tiene al menos num elementos
  int i;

  if ( ini >= num ) {
    encontrado = false;
    pos = num;
  }
  else {
    encontrado = false;
    i = ini;
    while ( ! encontrado && (i < num) ) {
      encontrado = v[i] == x;
      if ( ! encontrado ) i++;
    }
    pos = i;
  }
// Q :  encontrado es true si existe un i en el intervalo
ini..num-1
//        tal que v[i] == x
//      si encontrado es true entonces pos = el minimo i en
//        el intervalo ini..num-1 tal que v[i] == x
//      si no encontrado entonces pos == num
}
```

- Búsqueda binaria en un array de enteros

```
int buscaBin( int v[], int x, int a, int b ) {
// Pre: v está ordenado entre 0 .. num-1
//      ( 0 <= a <= num ) && ( -1 <= b <= num ) && ( a <= b+1
)
//      todos los elementos a la izquierda de 'a' son <= x
//      todos los elementos a la derecha de 'b' son > x

  int p, m;

  if ( a == b+1 )
    p = a - 1;
  else if ( a <= b ) {
    m = (a+b) / 2;
    if ( v[m] <= x )
      p = buscaBin( v, x, m+1, b );
    else
      p = buscaBin( v, x, a, m-1 );
  }
  return p;
// Post: devuelve el mayor índice i (0 <= i <= num-1) que
cumple
//       v[i] <= x
//       si x es menor que todos los elementos de v, devuelve
-1
}

void buscaBin( int x, int v[], int num, bool & encontrado, int
& pos ) {
// Pre: los num primeros elementos de v están ordenados
//      num >= 0

  pos = buscaBin(v, x, 0, num-1);
  encontrado = ( pos >= 0 ) && ( pos < num ) && ( v[pos] == x
);

// Post : devuelve el mayor índice i (0 <= i <= num-1) que
cumple
//        v[i] <= x
//        si x es menor que todos los elementos de v, devuelve
-1
//        encontrado es true si x esta en v[0..num-1]
}
```

- En los tres casos necesitamos una función *esLleno* para comprobar la condición impuesta por las limitaciones de la implementación

```
  bool esLleno ( TCjtoInt xs ) {
  // P: esValido(xs)
    return xs.longitud == longMax;
  // Q: esLleno si xs tiene 'longMax' elementos
  }
```

Operaciones de la especificación

- La generadora *Vacio* y la observadora *esVacio* se implementan de la misma forma en los tres casos

```
void Vacio( TCjtoInt & xs ) {
// P:
  xs.longitud = 0;
// Q: esValido(xs) && xs representa al conjunto vacío
}


bool esVacio ( TCjtoInt xs ) {
// P: esValido(xs)
  return xs.longitud == 0;
// Q: determina si el conjunto está vacío
}
```

Vectores

- Función que comprueba la validez de los representantes

```
bool esValido ( TCjtoInt xs ) {
// P:
  return xs.longitud >= 0 && xs.longitud <= longMax;
// Q: determina si xs es un representante valido de TCjtoInt
}
```

- La generadora *Pon*

```
void Pon( int x, TCjtoInt & xs ) {
// P: xs = XS && esValido(xs) && ! esLleno(xs)
  if ( esLleno(xs) )
    error("No se puede insertar el elemento");
  else {
    xs.espacio[xs.longitud] = x;
    xs.longitud++;
  }
// Q: esValido(xs) && xs es el resultado de añadir x a XS
}
```

- La modificadora *quita*

```
void quita( int x, TCjtoInt & xs ) {
// P: xs = XS && esValido(xs)
  bool encontrado;
  int pos;

  busca( x, xs.espacio, 0, xs.longitud, encontrado, pos);
  while ( encontrado ) {
    xs.longitud--;
    xs.espacio[pos] = xs.espacio[xs.longitud];
    busca( x, xs.espacio, pos, xs.longitud, encontrado,
pos);
  }
// Q: esValido(xs) && xs es el resultado de quitar x de XS
}
```

- La observadora *pertenece*

```
bool pertenece( int x, TCjtoInt xs ) {
// P: esValido(xs)
  bool encontrado;
  int pos;

  busca( x, xs.espacio, 0, xs.longitud, encontrado, pos);
  return encontrado;
// Q: determina si x está en xs
}
```

Vectores sin repetición

- Función que comprueba la validez de los representantes

```
bool esValido ( TCjtoInt xs ) {
// P:
  bool valido;
  int i, j;

  valido = (xs.longitud >= 0) && (xs.longitud <= longMax);
  i = 0;
  while ( valido && (i < xs.longitud - 1) ) {
    j = i + 1;
    while ( valido && (j < xs.longitud) ) {
      valido = xs.espacio[i] != xs.espacio[j];
      j++;
    }
    i++;
  }
  return valido;
// Q: determina si xs es un representante valido de TCjtoInt
}
```

- La observadora *pertenece* tiene la misma implementación que en el caso anterior

- La generadora *Pon*

```
void Pon( int x, TCjtoInt & xs ) {
// P: xs = XS && esValido(xs) && ( ! esLleno(xs) ||
pertenece(x, xs) )
  if ( ! pertenece(x, xs) )
    if ( esLleno(xs) )
      error("No se puede insertar el elemento");
    else {
      xs.espacio[xs.longitud] = x;
      xs.longitud++;
    }
// Q: esValido(xs) && xs es el resultado de añadir x a XS
}
```

- La modificadora *quita*

```
void quita( int x, TCjtoInt & xs ) {
// P: xs = XS && esValido(xs)
  bool encontrado;
  int pos;

  busca( x, xs.espacio, 0, xs.longitud, encontrado, pos);
  if ( encontrado ) {
    xs.longitud--;
    xs.espacio[pos] = xs.espacio[xs.longitud];
  }
// Q: esValido(xs) && xs es el resultado de quitar x de XS
}
```

Vectores sin repetición y ordenados

- Función que comprueba la validez de los representantes

```
bool esValido ( TCjtoInt xs ) {
// P:
  bool valido;
  int i;

  valido = (xs.longitud >= 0) && (xs.longitud <= longMax);
  i = 0;
  while ( valido && (i < xs.longitud - 1) ) {
    valido = xs.espacio[i] < xs.espacio[i+1];
    i++;
  }
  return valido;
// Q: determina si xs es un representante valido de TCjtoInt
}
```

- La generadora *Pon*

```
void Pon( int x, TCjtoInt & xs ) {
// P: xs = XS && esValido(xs) && ( ! esLleno(xs) ||
pertenece(x, xs) )
  bool encontrado;
  int pos;

  buscaBin( x, xs.espacio, xs.longitud, encontrado, pos);
  if ( ! encontrado ) {
    if ( esLleno(xs) )
      error("No se puede insertar el elemento");
    else {
      for ( int k = xs.longitud; k > pos+1; k-- )
        xs.espacio[k] = xs.espacio[k-1];
      xs.espacio[pos+1] = x;
      xs.longitud++;
    }
  }
// Q: esValido(xs) && xs es el resultado de añadir x a XS
}
```

- La modificadora *quita*

```
void quita( int x, TCjtoInt & xs ) {
// P: xs = XS && esValido(xs)
  bool encontrado;
  int pos;

  buscaBin( x, xs.espacio, xs.longitud, encontrado, pos);
  if ( encontrado ) {
    for ( int k = pos; k < xs.longitud-1; k++ )
      xs.espacio[k] = xs.espacio[k+1];
    xs.longitud--;
  }
// Q: esValido(xs) && xs es el resultado de quitar x de XS
}
```

- La observadora *pertenece*

```
bool pertenece( int x, TCjtoInt xs ) {
// P: esValido(xs)
  bool encontrado;
  int pos;

  buscaBin( x, xs.espacio, xs.longitud, encontrado, pos);
  return encontrado;
// Q: determina si x está en xs
}
```

Comparación de las distintas implementaciones

- Para determinar cuál es la complejidad temporal en el caso peor de cada una de las implementaciones, debemos establecer qué consideramos como tamaño de los datos:
  - En los vectores con repetición

    n_1 = nº de elementos del array
  - En los vectores sin repetición, ordenados o no,

    n_2 = nº de elementos del conjunto

	Vector	Vector sin repetición	Vector ordenado
Vacio	$O(1)$	$O(1)$	$O(1)$
Pon	$O(1)$	$O(n_2)$	$O(n_2)$
quita	$O(n_1)$	$O(n_2)$	$O(n_2)$
esVacio	$O(1)$	$O(1)$	$O(1)$
pertenece	$O(n_1)$	$O(n_2)$	$O(\log n_2)$

¿Qué implementación es mejor?

Depende de la información adicional que podamos utilizar:

- ¿n_1 es mucho mayor que n_2 o son parecidas?
- ¿la operación *pertenece* es muy habitual o no?
- ¿queremos minimizar el consumo de espacio?

Si no dispusiésemos de información adicional, deberíamos ser pesimistas y elegir la implementación con vectores ordenados.

3.4 Implementación modular de TADs

- La modularidad es un mecanismo que permite descomponer el código de un sistema software. Esta descomposición puede resultar superflua en programas pequeños, pero es fundamental en aplicaciones de tamaño real.

  Al preocuparnos por la modularidad, vamos abandonando el ámbito de la Programación para adentrarnos en el de la Ingeniería del Software.

- El objetivo de la Ingeniería del Software es el estudio de las técnicas que permiten la construcción eficiente (en términos de recursos humanos y económicos) de sistemas software de calidad.

- Las fases del "ciclo de vida" del software, en su versión clásica, son
  - Análisis. De la interacción con los usuarios se obtiene una especificación de los requisitos.
  - Diseño. Se determina la estructura general del sistema. Dependiendo del tamaño puede convenir distinguir entre el "diseño de alto nivel" y el "diseño de bajo nivel". El resultado del diseño es una descomposición del sistema a desarrollar.
  - Implementación. Se escribe el código de cada uno de las partes identificadas en la fase de diseño, que, una vez implementadas y probadas, se integran para obtener el sistema final.
  - Prueba. Se intentan detectar y subsanar los errores que se hayan cometido en el desarrollo.
  - Mantenimiento. Una vez que el sistema ya está en uso, será necesario seguir incorporando cambios y correcciones.

- Los módulos son un mecanismo a nivel de implementación que permiten reflejar en el código de la aplicación la descomposición definida por el diseño:
  - Separación física en archivos que se pueden compilar por separado
  - Separación lógica en ámbitos de visibilidad separados

  En general, se entiende que el diseño de una aplicación es la descomposición en módulos de la misma, junto con las relaciones entre dichos módulos.

3.4.1 Implementación modular de TADs en C++

Utilizando unidades

- En el archivo de cabecera (.h) se incluye:
  - El tipo representante
  - La cabecera de los procedimientos o funciones que implementan las operaciones del TAD.
  - La cabecera de otras operaciones auxiliares que aunque no apareciesen en la especificación pueden resultar útiles a los clientes del TAD

- En el archivo de implementación (.cpp) se incluye:
  - La implementación de las operaciones que aparecen en la interfaz
  - Otros tipos auxiliares
  - Otras operaciones auxiliares

- Por ejemplo, la implementación de los conjuntos de enteros

```
//---------------------------------------------------------
-------------
#ifndef cjto_intH
#define cjto_intH
//---------------------------------------------------------
-------------

/*
  Implementación de los conjuntos de enteros en vectores
ordenados sin
  repetición.
  Tamaño de los datos n = número de elementos almacenados en
el conjunto.
  La constructora Vacio proporciona un representante válido
del tipo,
  Todas las operaciones esperan representantes válidos y
devuelven
  representantes válidos.
  Como máximo se pueden almacenar longMax elementos
*/

int const longMax = 100;

typedef struct {
  int espacio[longMax];
  int longitud;
} TCjtoInt;
```

```
void Vacio( TCjtoInt & xs );
// P: true
// Q: esValido(xs) && xs representa al conjunto vacío
// O(1)

bool esVacio ( TCjtoInt xs );
// P: esValido(xs)
// Q: determina si el conjunto está vacío
// O(1)

void Pon( int x, TCjtoInt & xs );
// P: xs = XS && esValido(xs) && ( ! esLleno(xs) ||
pertenece(x, xs) )
// Q: esValido(xs) && xs es el resultado de añadir x a XS
// O(n)

void quita( int x, TCjtoInt & xs );
// P: xs = XS && esValido(xs)
// Q: esValido(xs) && xs es el resultado de quitar x de XS
// O(n)

bool pertenece( int x, TCjtoInt xs );
// P: esValido(xs)
// Q: determina si x está en xs
// O(log n)

bool esLleno ( TCjtoInt xs );
// P: esValido(xs)
// Q: esLleno si xs tiene 'longMax' elementos
// O(1)

bool esValido ( TCjtoInt xs );
// P: true
// Q: determina si xs es un representante valido de TCjtoInt
// O(n)

void escribe( TCjtoInt xs );
// P: esValido(xs)
// Q: muestra por pantalla el contenido de xs

#endif
```

- Y el archivo de implementación `cjto_int.cpp`

```
//-------------------------------------------------------------
-------------

#include "cjto_int.h"
#include <iostream>
#include <string>
using namespace std;

////////////////////////////////////////////////////////////////
/////////////
//
//
//                OPERACIONES AUXILIARES PRIVADAS
//

//
// función auxiliar privada para mostrar un mensaje de error
//
void error( string mensaje ) {
  cout <<  "Error: " + mensaje;
}

//
// función auxiliar de búsqueda en el array
//

...

////////////////////////////////////////////////////////////////
/////////////
//
//
//                     OPERACIONES PUBLICAS
//

void Vacio( TCjtoInt & xs ) {
// P: true
  xs.longitud = 0;
// Q: esValido(xs) && xs representa al conjunto vacío
}

...
```

```
////////////////////////////////////////////////////////////
/////////////
//
//
//                      OPERACIONES DE ESCRITURA
//

void escribe( TCjtoInt xs ) {
// P: esValido(xs)
  cout << "(";
  for ( int i = 0; i < xs.longitud; i++ ) {
    cout << xs.espacio[i];
    if ( i < xs.longitud-1 )
      cout << ", ";
  }
  cout << ")" << endl;
// Q: muestra por pantalla el contenido de xs
}


////////////////////////////////////////////////////////////
/////////////
//
//
//             OPERACIONES AUXILARES DE LA ESPECIFICACION
//

//
// Limitación impuesta por la implementación (esLleno)
//
bool esLleno ( TCjtoInt xs ) {
// P: esValido(xs)
  return xs.longitud == longMax;
// Q: esLleno si xs tiene 'longMax' elementos
}

//
// Función que determina si un valor del tipo es un
representante válido
//

...

//-----------------------------------------------------------
------------
```

Utilizando unidades y clases

- Implementando los TADs como clases en C++, podemos proteger el acceso a la representación de la información.

- En el archivo de cabecera se escribe la definición de la clase
  - Implementamos al menos una de las generadoras como una constructora de objetos de la clase.
  - La estructura de datos se declara como variables privadas de la clase.
  - El dato del tipo principal que las operaciones observan o modifican se convierte en el parámetro implícito de los métodos de la clase: el objeto que recibe el mensaje.
  - En las especificaciones pre/post sustituimos el parámetro del tipo principal por la pseudo-variable *this* o simplemente lo obviamos.
  - Dado que la implementación con clases sí que garantiza la privacidad y protección del tipo representante, asumimos que los clientes del TAD no pueden construir valores del tipo que no sean representantes válidos y eliminamos la condición *esValido* de las especificaciones.

```
/*
  Implementación de los conjuntos de enteros en vectores
ordenados sin
  repetición.
  Tamaño de los datos n = número de elementos almacenados en
el conjunto.
  Como máximo se pueden almacenar longMax elementos
*/


class TCjtoInt {
  public:

  // Tamaño máximo
     static const int longMax = 100;

  // Constructora
     TCjtoInt( );
     // Pre: true
     // Post: construye un valor que representa al
     //conjunto vacío O(1)
```

```
  // Operaciones de los conjuntos
     bool esVacio ( );
     // Pre: true
     // Post: determina si el conjunto está vacío
     // O(1)

     void Pon( int x );
     // Pre: ! esLleno( ) || pertenece(x)
     // Post: modifica el conjunto, añadiéndole x
     // O(n)

     void quita( int x );
     // Pre: true
     // Post: modifica el conjunto, quitando x
     // O(n)

     bool pertenece( int x );
     // Pre: true
     // Post: determina si x está en el conjunto
     // O(log n)

     bool esLleno ( );
     // Pre: true
     // Post: determina si el conjunto tiene 'longMax'
elementos
     // O(1)

  // Escritura
     void escribe( );
     // Pre: true
     // Post: muestra por pantalla el contenido del conjunto

  private:
  // Variables privadas
     int _espacio[longMax];
     int _longitud;
};
```

Por convenio escribimos los nombres de las variables miembro precedidas de un guión bajo _.

- Al construirse un valor de un tipo definido por una clase, se ejecuta necesariamente una constructora de esa clase, asegurando así que los objetos están inicializados.

```
TCjtoInt cjto;  // se ejecuta la constructora por defecto
```

- En la implementación de las operaciones eliminamos las referencias explícitas al parámetro del tipo principal.

Por ejemplo

```
bool pertenece( int x, TCjtoInt xs ) {
// P: esValido(xs)
  bool encontrado;
  int pos;

  buscaBin( x, xs.espacio, xs.longitud, encontrado, pos);
  return encontrado;
// Q: determina si x está en xs
}
```

Se convierte en

```
bool TCjtoInt::pertenece( int x ) {
// Pre: true
  bool encontrado;
  int pos;

  buscaBin( x, _espacio, _longitud, encontrado, pos);
  return encontrado;
// Post: determina si x está en el conjunto
}
```

Nótese que la versión orientada a objetos es equivalente a

```
bool TCjtoInt::pertenece( int x ) {
// Pre: true
  bool encontrado;
  int pos;

  buscaBin( x, this->_espacio, this->_longitud, encontrado,
pos);
  return encontrado;
// Post: determina si x está en el conjunto
}
```

- Y el archivo de implementación `cjto_int.cpp`

```
//-------------------------------------------------------------
-------------

#include "cjto_int.h"
#include <iostream>
#include <string>
using namespace std;

////////////////////////////////////////////////////////////
/////////////
//
//
//                OPERACIONES AUXILIARES PRIVADAS
//

//
// función auxiliar privada para mostrar un mensaje de error
//
void error( string mensaje ) {
  cout <<  "Error: " + mensaje;
}

//
// función auxiliar de búsqueda en el array
//

...


////////////////////////////////////////////////////////////
/////////////
//
//
//                     OPERACIONES PUBLICAS
//


TCjtoInt::TCjtoInt( ) {
// Pre: true
  _longitud = 0;
// Post: construye un valor que representa al conjunto vacío
}

bool TCjtoInt::esVacio ( ) {
// P: true
  return _longitud == 0;
// Q: determina si el conjunto está vacío
}
```

```
void TCjtoInt::Pon( int x ) {
// Pre: ! esLleno( ) || pertenece(x)
  bool encontrado;
  int pos;

  buscaBin( x, _espacio, _longitud, encontrado, pos);
  if ( ! encontrado ) {
    if ( esLleno( ) )
      error("No se puede insertar el elemento");
    else {
      for ( int k = _longitud; k > pos+1; k-- )
        _espacio[k] = _espacio[k-1];
      _espacio[pos+1] = x;
      _longitud++;
    }
  }
// Post: modifica el conjunto, añadiéndole x
}

...

////////////////////////////////////////////////////////////
/////////////
//
//
//                      OPERACIONES DE ESCRITURA
//

...

////////////////////////////////////////////////////////////
/////////////
//
//
//           OPERACIONES AUXILARES DE LA ESPECIFICACION
//

//
// Limitación impuesta por la implementación (esLleno)
//
bool TCjtoInt::esLleno ( ) {
// Pre: true
  return _longitud == longMax;
// Post: determina si el conjunto tiene 'longMax' elementos
}

//-----------------------------------------------------------
-------------
```

Implementación de TADs genéricos

- En C++ las clase genéricas se implementan mediante plantillas (*templates*).
  Una plantilla es una clase con una o más *variables de tipo* en términos de las cuales se realiza la implementación. Luego se declaran ejemplares concretos de la plantilla instanciando las variables con tipos concretos.

- Podemos definir tanto clases como funciones plantilla.
  - Una plantilla se declara precediendo la declaración con
    template<class *VarTipo1*, ... , **class** *VarTipoN*>
    donde *VarTipo1, ... , VarTipoN* identifican *variables de tipo* cuyo ámbito coincide con el de la declaración de la clase o la función plantilla.
  - En un uso particular de un plantilla se ha de proporcionar un tipo concreto para cada parámetro de tipo. Se pueden utilizar tipos primitivos o declarados por el programador.
  - El compilador comprueba que el uso de la plantilla es correcto y genera la instancia solicitada, i.e., es como si generase una copia del código de la plantilla donde se han sustituido las apariciones de las variables de tipo por los tipos concretos.
  - No supone ineficiencias en tiempo de ejecución.
  - En las plantillas tanto la definición de la clase como la implementación de las operaciones deben estar en el mismo archivo de cabecera.

- Primer ejemplo: la clase de las parejas de valores.
  Declaración de la clase:

```
template <class T1, class T2>
class TPareja {
  public:
    TPareja( T1 v1, T2 v2 );
    T1 v1( );
    T2 v2( );
    void ponV1( T1 v1 );
    void ponV2( T2 v2 );
    bool operator<=( TPareja<T1,T2>& par );
  private:
    T1 _v1;
    T2 _v2;
};
```

Implementación de las operaciones:

```
template <class T1, class T2>
TPareja<T1,T2>::TPareja( T1 v1, T2 v2 ) :
    _v1(v1), _v2(v2) {};

template <class T1, class T2>
T1 TPareja<T1,T2>::v1( ) { return _v1; };

template <class T1, class T2>
T2 TPareja<T1,T2>::v2( ) { return _v2; };

template <class T1, class T2>
void TPareja<T1,T2>::ponV1( T1 v1 ) { _v1 = v1; };

template <class T1, class T2>
void TPareja<T1,T2>::ponV2( T2 v2 ) { _v2 = v2; };

template <class T1, class T2>
bool TPareja<T1,T2>::operator<=( TPareja<T1,T2>& par ) {
  return (_v1 <= par._v1) && (_v2 <= par._v2); };

template <class T1, class T2>
ostream& operator<<( ostream& salida, TPareja<T1,T2> par ) {
  salida << "(" << par.v1() << ", " << par.v2() << ")";
  return salida;
};
```

- Ejemplo de uso

```
TPareja<int,int> p1(1, 1), p2(1, 2);
TPareja<int,TPareja<int,int> > q1(1, p1), q2(1, p2);
// el espacio entre los dos > > es imprescindible

cout << p1 << p2 << q1 << q2 << endl;
cout << ((q1 <= q2) ? "menor" : "mayor");
```

- La implementación de una clase plantilla puede imponer restricciones sobre las variables de tipo relativas a las operaciones que ese tipo debe implementar.

  Por ejemplo, las parejas sólo se pueden instanciar con tipos que implementen el operador <=. Tipos que pertenecen a la clase de tipos ordenados.

  El compilador se encarga de comprobar que la instanciación es correcta, pero no hay ningún mecanismo para indicar explícitamente estas restricciones, por lo que es aconsejable incluirlo como comentario.

Implementación genérica de los conjuntos

```
/*
  Implementación de los conjuntos en vectores ordenados sin
repetición.
  Tamaño de los datos n = número de elementos almacenados en
el conjunto.
  Como máximo se pueden almacenar longMax elementos
*/

#include <iostream>
#include <string>
using namespace std;

template<class TElem>
class TCjto {
  public:

  // Tamaño máximo
     static const int longMax = 100;

  // Constructora
     TCjto( );
     // Pre: true
     // Post: construye un valor que representa al conjunto
vacío
     // O(1)

  // Operaciones de los conjuntos
     bool esVacio ( );
     // Pre: true
     // Post: determina si el conjunto está vacío
     // O(1)

     void Pon( TElem x );
     // Pre: ! esLleno( ) || pertenece(x)
     // Post: modifica el conjunto, añadiéndole x
     // O(n)

     void quita( TElem x );
     // Pre: true
     // Post: modifica el conjunto, quitando x
     // O(n)

     bool pertenece( TElem x );
     // Pre: true
     // Post: determina si x está en el conjunto
     // O(log n)
```

```
bool esLleno ( );
     // Pre: true
     // Post: determina si el conjunto tiene 'longMax'
     //elementos O(1)

  // Escritura
     void escribe( ostream& salida );
     // Pre: true
     // Post: escribe en 'salida' el contenido del conjunto

  private:
  // Variables privadas
     TElem _espacio[longMax];
     int _longitud;
};

template <class TElem>
ostream& operator<< ( ostream& salida, TCjto<TElem> cjto );
```

```
////////////////////////////////////////////////////////////
/////////////
//
//
//                OPERACIONES AUXILIARES PRIVADAS
//

//
// Búsqueda en el array
//
template<class TElem>
int buscaBin( TElem v[], TElem x, int a, int b ) {
// Pre: v está ordenado entre 0 .. num-1
//      ( 0 <= a <= num ) && ( -1 <= b <= num ) && ( a <= b+1
)
//      todos los elementos a la izquierda de 'a' son <= x
//      todos los elementos a la derecha de 'b' son > x

  int p, m;

  if ( a == b+1 )
    p = a - 1;
  else if ( a <= b ) {
    m = (a+b) / 2;
    if ( v[m] <= x )
      p = buscaBin( v, x, m+1, b );
    else
      p = buscaBin( v, x, a, m-1 );
  }
  return p;
// Post: devuelve el mayor índice i (0 <= i <= num-1) que
cumple
//       v[i] <= x
//       si x es menor que todos los elementos de v, devuelve
-1
}

template<class TElem>
void buscaBin( TElem x, TElem v[], int num, bool & encontrado,
int & pos ) {
// Pre: los num primeros elementos de v están ordenados
//      num >= 0

  pos = buscaBin(v, x, 0, num-1);
  encontrado = ( pos >= 0 ) && ( pos < num ) && ( v[pos] == x
);

// Post : devuelve el mayor índice i (0 <= i <= num-1) que
cumple
//        v[i] <= x
//        si x es menor que todos los elementos de v, devuelve
-1
//        encontrado es true si x esta en v[0..num-1]
```

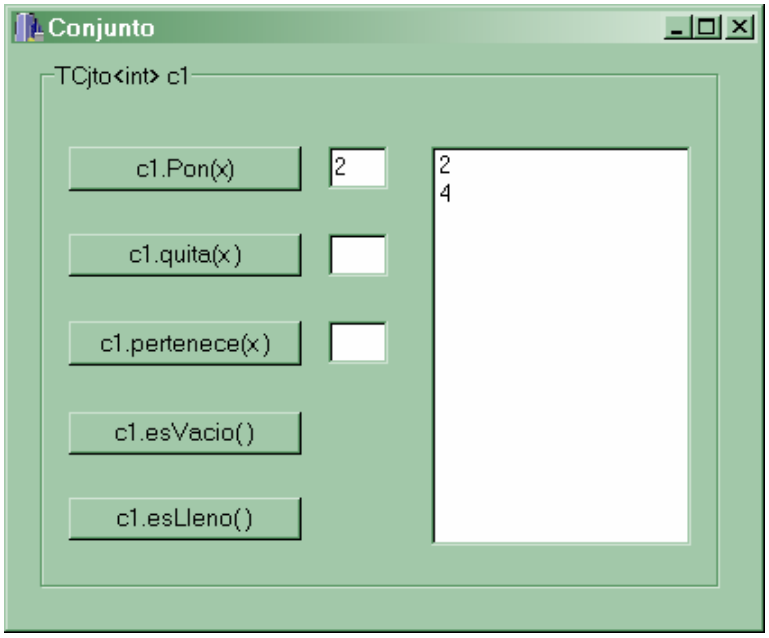
```
}
//
// función auxiliar privada para mostrar un mensaje de error
//
void error( string mensaje ) {
  cout <<  "Error: " + mensaje;
}

////////////////////////////////////////////////////////////
////////////
//
//
//                    OPERACIONES PUBLICAS
//

template<class TElem>
TCjto<TElem>::TCjto( ) {
// Pre: true
  _longitud = 0;
// Post: construye un valor que representa al conjunto vacío
}

template<class TElem>
bool TCjto<TElem>::esVacio ( ) {
// P: true
  return _longitud == 0;
// Q: determina si el conjunto está vacío
}

template<class TElem>
void TCjto<TElem>::Pon( TElem x ) {
// Pre: ! esLleno( ) || pertenece(x)
  bool encontrado;
  int pos;

  buscaBin( x, _espacio, _longitud, encontrado, pos);
  if ( ! encontrado ) {
    if ( esLleno( ) )
      error("No se puede insertar el elemento");
    else {
      for ( int k = _longitud; k > pos+1; k-- )
        _espacio[k] = _espacio[k-1];
      _espacio[pos+1] = x;
      _longitud++;
    }
  }
// Post: modifica el conjunto, añadiéndole x
}
```

```
template<class TElem>
void TCjto<TElem>::quita( TElem x ) {
// Pre: true
  bool encontrado;
  int pos;

  buscaBin( x, _espacio, _longitud, encontrado, pos);
  if ( encontrado ) {
    for ( int k = pos; k < _longitud-1; k++ )
      _espacio[k] = _espacio[k+1];
    _longitud--;
  }
// Post: modifica el conjunto, quitando x
}

template<class TElem>
bool TCjto<TElem>::pertenece( TElem x ) {
// Pre: true
  bool encontrado;
  int pos;

  buscaBin( x, _espacio, _longitud, encontrado, pos);
  return encontrado;
// Post: determina si x está en el conjunto
}
```

```
////////////////////////////////////////////////////////////
/////////////
//
//
//                    OPERACIONES DE ESCRITURA
//

template<class TElem>
void TCjto<TElem>::escribe( ostream& salida ) {
// Pre: true
  salida << "(";
  for ( int i = 0; i < _longitud; i++ ) {
    salida << _espacio[i];
    if ( i < _longitud-1 )
      salida << ", ";
  }
  cout << ")" << endl;
// Post: muestra por pantalla el contenido del conjunto
}

template <class TElem>
ostream& operator<< ( ostream& salida, TCjto<TElem> cjto ) {
  cjto.escribe(salida);
  return salida;
}


////////////////////////////////////////////////////////////
/////////////
//
//
//          OPERACIONES AUXILARES DE LA ESPECIFICACION
//

//
// Limitación impuesta por la implementación (esLleno)
//
template<class TElem>
bool TCjto<TElem>::esLleno ( ) {
// Pre: true
  return _longitud == longMax;
// Post: determina si el conjunto tiene 'longMax' elementos
}
```

Prueba de los módulos

- Una de las ventajas de la implementación modular es que resulta más sencillo probar cada módulo por separado.



- Para llevar a cabo esta prueba, es necesario que el TAD exporte una operación que genere una cadena con los elementos separados por saltos de línea, a partir de la cual se genera el contenido del cuadro de lista.

```
template<class TElem>
class TCjto {
  public:
  ...
     void escribeLn( ostream& salida );
     // Pre: true
     // Post: escribe en 'salida' el contenido del conjunto
     //       incluyendo saltos de línea entre cada dos
datos
  ...
};
...
template<class TElem>
void TCjto<TElem>::escribeLn( ostream& salida ) {
// Pre: true
  for ( int i = 0; i < _longitud; i++ )
    salida << _espacio[i] << endl;
// Post: muestra por pantalla el contenido del conjunto
//       incluyendo saltos de línea entre cada dos datos
}
```

- La clase que implementa el formulario

  La definición de la clase que genera automáticamente el entorno.

```
class TForm1 : public TForm
{
__published: // IDE-managed Components
        TGroupBox *GroupBox1;
        TListBox *List;
        TButton *Pon;
        TEdit *Ponx;
        TButton *Quita;
        TButton *EsLleno;
        TButton *Pertenece;
        TEdit *Quitax;
        TEdit *Pertenecex;
        TButton *esVacio;
        void __fastcall PonClick(TObject *Sender);
        void __fastcall QuitaClick(TObject *Sender);
        void __fastcall PerteneceClick(TObject *Sender);
        void __fastcall esVacioClick(TObject *Sender);
        void __fastcall EsLlenoClick(TObject *Sender);
private: // User declarations
public:    // User declarations
        __fastcall TForm1(TComponent* Owner);
};
```

La implementación de la clase

```
#include <iostream>
#include <sstream> //STR 640
#include <string>
#include <list>
#include "cjto.h"

using namespace std;

typedef TCjto<int> TCjtoInt;

TCjtoInt c1;

TForm1 *Form1;
```

Función que actualiza el contenido del cuadro de lista

```
//---------------------------------------------------------
----------
void actualizaContenido( TCjtoInt cjto, TStrings* lista )
{
  ostringstream aux;
  int n, inicio, fin;
  string contenido;
  string separadores = "\n";

  cjto.escribeLn(aux);
  contenido = aux.str();
  n = contenido.length();
  inicio = contenido.find_first_not_of(separadores);
  lista->Clear();

  while ((inicio >= 0) && (inicio < n)) {
    fin = contenido.find_first_of(separadores, inicio);
    if ((fin < 0) || (fin > n)) fin = n;
    lista->Add((contenido.substr(inicio, fin -
inicio)).data());
    inicio = contenido.find_first_not_of(separadores,
fin+1);
  }
}
```

Creación del formulario

```
//---------------------------------------------------------
----------
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
}
```

Funciones asociadas con la pulsación de los botones

```
//---------------------------------------------------------
----------
void __fastcall TForm1::PonClick(TObject *Sender)
{
  c1.Pon(StrToInt(Ponx->Text));
  actualizaContenido(c1, List->Items);
}
//---------------------------------------------------------
----------
void __fastcall TForm1::QuitaClick(TObject *Sender)
{
  c1.quita(StrToInt(Quitax->Text));
  actualizaContenido(c1, List->Items);
}
```

```
//---------------------------------------------------------
----------
void __fastcall TForm1::PerteneceClick(TObject *Sender)
{
  if( c1.pertenece(StrToInt(Pertenecex->Text)) )
    ShowMessage("Sí está");
  else
    ShowMessage("No está");
}
//---------------------------------------------------------
----------
void __fastcall TForm1::esVacioClick(TObject *Sender)
{
    if( c1.esVacio( ) )
      ShowMessage("Está vacío");
    else
      ShowMessage("No está vacío");
}
//---------------------------------------------------------
----------
void __fastcall TForm1::EsLlenoClick(TObject *Sender)
{
    if( c1.esLleno( ) )
      ShowMessage("Está lleno");
    else
      ShowMessage("No está lleno");
}
//---------------------------------------------------------
----------
```

Implementación de TADs con operaciones parciales: manejo de excepciones

- La mayoría de los lenguajes de programación modernos incluyen algún mecanismo para facilitar el tratamiento de los errores y las situaciones no previstas en el código: *excepciones*

- En C++ una excepción se representa como un objeto de una cierta clase.
  - Para definir un cierto tipo de excepción, se declara un nuevo tipo de clase.
  - Cuando el código cliente es susceptible de generar una excepción, se utiliza la sentencia *try ... catch* para capturarla y tratarla.
  - Para lanzar una excepción se utiliza la sentencia *throw* que recibe como parámetro un objeto del tipo de excepción que se quiere lanzar.

    ```
    throw objeto;
    ```

- La sintaxis de la sentencia *try ... catch*

```
try {
  lista_sentencias
}
catch( identificador_clase1 ) {
   lista_sentencias_excepcion1
}
...
catch( identificador_claseN ) {
  lista_sentencias_excepcionN
}
catch( ... ) {   // captura cualquier excepción
  lista_sentencias_otras_excepciones
}
```

Se ejecuta la *lista de sentencias* que siguen a *try*

- Si las sentencias se ejecutan sin que se lancen excepciones, se ignoran los bloques de tratamiento de excepciones.
- Cuando se genera una excepción, se transfiere el control al *manejador de excepciones* más interno (puede haber sentencias *try ... catch* anidadas) que puede manejar las excepciones de la clase dada.

  La propagación de la excepción continua hasta que se encuentra el manejador adecuado o hasta que no hay más sentencias *try ... catch* activas, en cuyo caso se produce un error en tiempo de ejecución y se finaliza la aplicación.

- Para determinar si el *bloque de excepciones* puede manejar una excepción, se examinan los *manejadores de excepción* en orden de aparición.
  - Un manejador puede tratar una excepción si la clase de ésta coincide con el correspondiente *identificador de clase* o es una especialización suya.
  - Un manejador de la forma *catch (...)* captura cualquier excepción.

- En una cláusula *catch* se puede declarar una variable local cuyo ámbito es el bloque de código asociado.

  ```
  catch( identificador_clase identificador_variable )
  ```

  Esta variable se inicializa con el objeto excepción.

- En el bloque asociado con una cláusula *catch* se puede volver a *lanzar* el mismo objeto sin más que realizar una nueva invocación de *throw*; (sin parámetros).

- En la cabecera de una función se puede indicar explícitamente qué excepciones puede generar esa función, de forma que si genera cualquier otra acabe la ejecución del programa.

Uso de excepciones en la implementación de los TADs

- Definimos dos tipos de excepciones comunes a las implementaciones de los TADs:
  - Inserciones en estructuras que están llenas: *EDesbordamiento*
  - Accesos ilegales a los valores del TAD: *EAccesoIndebido*

```
class EDesbordamiento {
  public:
    EDesbordamiento( string mensaje = "" ) :
_mensaje(mensaje) { };
    string mensaje ( ) {
      return _mensaje;
    };
  private:
    string _mensaje;
};

class EAccesoIndebido {
  public:
    EAccesoIndebido( string mensaje = "" ) :
_mensaje(mensaje) { };
    string mensaje ( ) {
      return _mensaje;
    };
  private:
    string _mensaje;
};
```

Aunque podemos definir estas clases en los archivos donde implementemos las clases que las pueden lanzar, es mejor idea incluirlas en un archivo aparte (*excepciones.h*) de forma que no se produzcan redefiniciones cuando usemos varios TADs en una misma aplicación.

- En las implementaciones de los TAD con operaciones parciales, ya sea por situaciones indicadas en la especificación o por limitaciones impuestas por la propia implementación, elevamos excepciones cuando se viola la precondición. En algunos casos puede ser conveniente definir un nuevo tipo de excepción, propio del TAD en cuestión.

- Por ejemplo, en la implementación genérica de las pilas, utilizando arrays.
  Anotamos las excepciones y las añadimos a los comentarios, en la definición de la clase.

```
template<class TElem>
class TPila {
  public:
      ...
      void apila( TElem x ) throw (EDesbordamiento);
      // Pre: ! esLleno( )
      // Post: modifica la pila, añadiéndole x en la cima
      // Lanza la excepción EDesbordamiento si la pila está
llena
      // O(1)

      void desapila( ) throw (EAccesoIndebido);
      // Pre: ! esVacio()
      // Post: modifica la pila, quitando el elemento de la
cima
      // Lanza la excepción EAccesoIndebido si la pila está
vacía
      // O(1)

      TElem cima( ) throw (EAccesoIndebido);
      // Pre: ! esVacio()
      // Post: devuelve el elemento que está en la cima de la
pila
      // Lanza la excepción EAccesoIndebido si la pila está
vacía
      // O(1)
      ...
};
```

- Y en las implementaciones de las operaciones parciales lanzamos una excepción cuando no se cumple la precondición.

```
template<class TElem>
void TPila<TElem>::apila( TElem x ) throw (EDesbordamiento){
// Pre: ! esLleno( )
  if ( esLleno() )
    throw EDesbordamiento("Pila llena");
  else {
    _indCima++;
    _espacio[_indCima] = x;
  }
// Post: modifica la pila, añadiéndole x en la cima
}
```

```
template<class TElem>
void TPila<TElem>::desapila( ) throw (EAccesoIndebido){
// Pre: ! esVacio()
  if ( esVacio() )
    throw EAccesoIndebido("No se puede desapilar de la pila
vacía");
  else
    _indCima--;
// Post: modifica la pila, quitando el elemento que está en
la cima
}

template<class TElem>
TElem TPila<TElem>::cima( ) throw (EAccesoIndebido){
// Pre: ! esVacio()
  if ( esVacio() )
    throw EAccesoIndebido("No existe la cima de la pila
vacía");
  else
    return _espacio[_indCima];
// Post: devuelve el elemento que está en la cima de la pila
}
```

- En el uso de estas operaciones parciales, se deben capturar las excepciones que se puedan producir.

```
TPila<char> pila;

try {
  pila.apila('a');
  pila.desapila();
  pila.desapila();
  pila.apila('b');
}
catch( EDesbordamiento e ) {
  cout << e.mensaje();
}
catch ( EAccesoIndebido e ) {
  cout << e.mensaje();
}
catch ( ... ) {
  cout << "otro tipo de excepción";
}
```

3.5 Estructuras de datos dinámicas

- Una estructura de datos se llama estática cuando el espacio que va a ocupar está determinado en tiempo de compilación.

  Puede que ese espacio se ubique al principio de la ejecución –variables globales– o en la invocación a procedimientos o funciones –variables locales–, pero, en cualquier caso, no es responsabilidad del programador ocuparse de su gestión.

- Un estructura de datos se llama dinámica cuando el espacio que ocupa en memoria puede variar durante la ejecución del programa, y no está determinado en tiempo de compilación.

- Una de las razones para utilizar estructuras de datos dinámicas es la representación de colecciones de valores.
  - La estructura de datos estática apta para representar colecciones de datos es el array. Sin embargo, se presentan dos problemas por el hecho de tener que reservar el espacio a priori:
    - Se desaprovecha memoria si el espacio reservado resulta ser excesivo.
    - Se pueden producir errores de desbordamiento en tiempo de ejecución si el espacio reservado resulta ser insuficiente.
  - Las estructuras de datos dinámicas, por su parte, permiten representar colecciones de datos, de forma que
    - no se malgasta espacio a priori, y
    - el único límite en tiempo de ejecución es la cantidad total de memoria disponible.

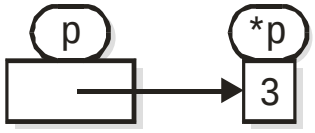
- Los *punteros* son el mecanismo que permite construir y manejar estructuras de datos dinámicas.

3.5.1 Punteros

- Una variable de un cierto tipo lleva asociado: un nombre, un espacio de memoria y un valor del tipo declarado para la variable.

  Una variable de tipo puntero tiene igualmente asignado un identificador, un espacio de memoria y su valor es la dirección de otra variable, tal que
  - El nombre de la variable apuntada por un puntero se puede obtener a partir del identificador del puntero mediante el *operador de indirección*. En C++ el operador de indirección (o *dereferencia*) se escribe *.
  - El espacio de memoria de la variable a la que apunta un puntero se puede ubicar dinámicamente durante la ejecución.
    - Llamamos *variable dinámica* a una variable creada de esta forma.
    - Una variable dinámica no existe hasta que es creada explícitamente en tiempo de ejecución.
  - Es posible "anular" una variable apuntada por un puntero, liberando así el espacio que ocupa.

  Gráficamente:



- Como estructura para representar colecciones de datos, la propiedad fundamental de los punteros es que permiten obtener y devolver memoria dinámicamente durante la ejecución, de forma que sólo solicitaremos el espacio imprescindible para los datos que en cada momento necesitemos representar.

Uso de punteros

- Para cualquier tipo τ admitimos que puede formarse un nuevo tipo de punteros a variables de tipo τ, declarado como

  τ*
  - Independientemente del tipo de variable al que apunte, siempre se reserva la misma cantidad de espacio para un puntero: la necesaria para representar una dirección de memoria.
  - El espacio reservado para la variable dinámica es el necesario para almacenar un valor de tipo τ.

- Las dos operaciones básicas con los punteros son ubicar la variable a la que apuntan y anular dicha variable, liberando el espacio que ocupa.

  En C++ la ubicación y la liberación de punteros se realiza mediante los operadores *new* y *delete*.

  – La ubicación de un puntero se hace con el operador *new*

  ```
  variable_puntero = new nombre_tipo
  ```

  El efecto de *new* es:

  – Crear una variable del tipo *nombre_tipo*.
  – Devolver la dirección del espacio de memoria asignado a dicha variable.

  Ejemplo:

  ```
  int* p;

  p = new int;
  ```

  – La liberación de una variable apuntada por un puntero se hace con el operador *delete*

  ```
  delete variable_puntero
  ```

  El efecto de *delete* es:

  – destruir la variable apuntada por *variable_puntero*, liberando el espacio de memoria que ocupaba

  Nótese que

  – no se debe liberar una variable que no esté ubicada, y
  – no se debe utilizar la variable **p* después de ejecutar *delete p*

  Ejemplo:

  ```
  int *p, *q;   // si se declara más de un puntero en la
misma
                // declaración, hay que escribir * delante
de cada uno

  p = new int;
  *p = 2;
  delete p;
  *p = 3;
  q = new int;
  *q = 4;
  cout << *p << endl;
  ```

  – ¿qué se muestra por pantalla?
  – ¿qué sentencias son erróneas?

- Otras operaciones que podemos realizar con los punteros
  - Asignaciones entre punteros del mismo tipo

    ```
    p = q
    ```

    Con el siguiente efecto
    - *p* pasa a apuntar al mismo sitio al que esté apuntado *q*.
    - Si antes de la asignación *p* apuntaba a una variable, después de la asignación **p* ya no es un identificador válido para dicha variable.

      No debemos perder el puntero a una variable dinámica sin liberar previamente el espacio que ésta ocupa, a menos que la variable sea accesible desde otro puntero.
    - Si **q* no está ubicada entonces **p* tampoco lo está.
    - **p* y **q* son dos identificadores de la misma variable.

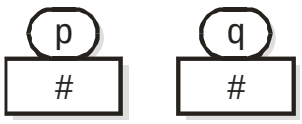
  - Comparaciones entre punteros del mismo tipo

    ```
    p == q   p != q
    ```

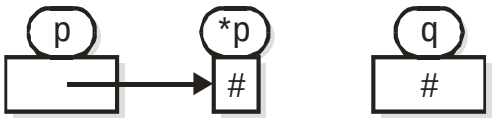
    Nótese que comparamos direcciones y no los valores de las variables a las que apuntan los punteros, **p* y **q*.

- Como conclusión de este apartado, vamos a seguir gráficamente con detalle un ejemplo del uso de las operaciones sobre punteros.

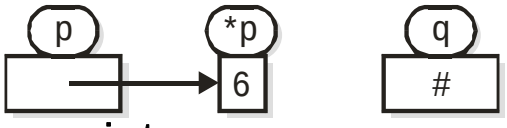
```
int *p, *q;
```




```
p = new int;
```



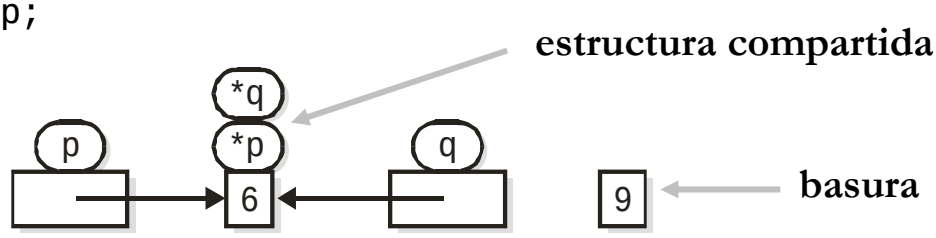

```
*p = 3;
*p = 2 * *p;
```



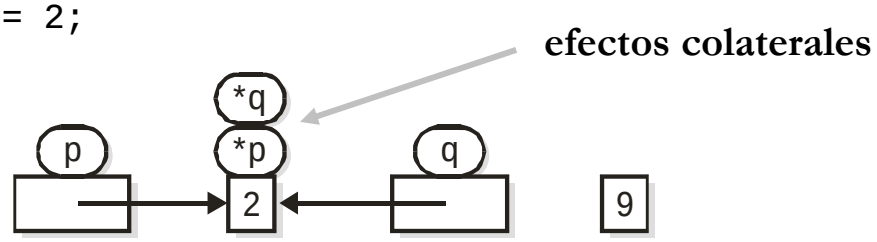

```
q = new int;
```


```
*q = 3;
*q = *p + *q;
```

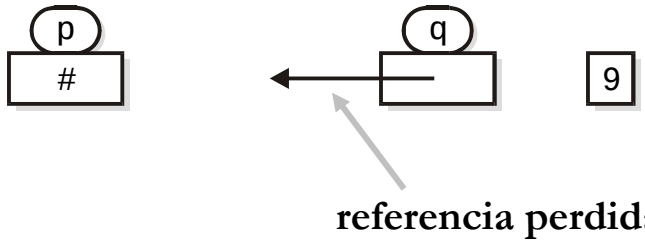

```
q = p;
```




```
*q = 2;
```




```
delete p;
```



- Dos punteros *comparten estructura* si permiten acceder a las mismas variables dinámicas.
- Una variable ubicada dinámicamente es *basura* si no es posible acceder a ella mediante ningún puntero (estático).
- Un puntero almacena una *referencia perdida* si apunta a una zona de la memoria dinámica que ha sido marcada como libre.

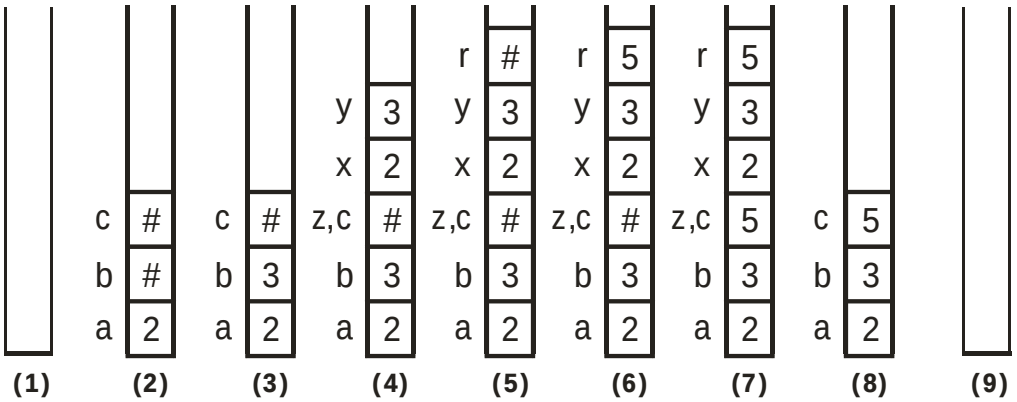
3.5.2 Construcción, destrucción, copia y asignación de estructuras dinámicas

- Cuando se utilizan punteros en los programas hay que ser especialmente cuidadoso por dos razones:
  - el programador es el responsable de ubicar y anular las variables dinámicas,
  - la asignación entre punteros produce compartición de estructura que puede provocar efectos colaterales y, unida a la anulación, generar basura o referencias perdidas.

- ¿Por qué no ocurre esto con las variables estáticas?

  Porque:
  - es el compilador quien genera el código encargado de ubicar y anular automáticamente las variables estáticas (también llamadas *automáticas* por razones obvias), y
  - la asignación no da lugar a compartición de estructura.

```
void suma ( int x, int y, int & z ) {          // 4
  int r ;                                      // 5

  r = x + y;                                   // 6
  z = r;                                 // 7
}
void main() {                                  // 1
  int a = 2, b, c;                       // 2

  b = 3;                                 // 3
  suma( a, b, c );                       // 8
}                                              // 9
```



- Cuando manejamos punteros la cosa se complica, sobre todo cuando manejamos punteros a estructuras que contienen a su vez punteros a otras estructuras. Afortunadamente, las clases de C++ incluyen los mecanismos necesarios para controlar los procesos de creación, destrucción copia y asignación.
  - Siempre que se crea un objeto, se ejecuta una de sus constructoras. Un objeto puede, al igual que cualquier otro valor, ser:
    - estático, si se declara explícitamente, o
    - dinámico si se crea mediante una invocación a *new*.

    Si en una clase no definimos ninguna constructora, entonces el compilador genera una que realiza inicializaciones por defecto.

    Si se define alguna constructora, entonces el compilador no genera ninguna, lo que quiere decir que si la constructora definida requiere parámetros, entonces no será posible construir ningún objeto sin suministrar esos parámetros.

    A una constructora sin parámetros se le denomina *constructora por defecto* y, en general, es conveniente que todas las clases tengan una. Si una clase no tiene constructora por defecto, entonces no es posible construir un array de elementos de ese tipo.
  - Siempre que se destruye un objeto, se invoca a su destructora. Un objeto, igual que cualquier otro valor, se destruye cuando sale de ámbito (si es estático) o cuando se invoca el operador *delete* sobre un puntero que lo apunta.

    Si en una clase no definimos destructora, entonces el compilador genera una que se limita a devolver el espacio ocupado por los datos estáticos del objeto.
  - Si un objeto se construye a partir de otro, entonces es la *constructora de copia* quien se encarga de la construcción.

    La constructora de copia se invoca cuando:
    - Se declara un objeto inicializado con el valor de otro.
    - Se construye un parámetro por valor. Los parámetros por valor son copias de los parámetros reales.
    - Se devuelve el valor de una función, siempre que ese valor no sea de tipo referencia.
  - Es posible redefinir el operador de asignación para objetos de un tipo determinado.

- Vamos a desarrollar un ejemplo que maneja estructuras dinámicas complejas. En primer lugar definimos la clase *TComplejo*

```
class Complejo {
  private:
    double _real;
    double _img;
  public:
    Complejo( ) : _real(0), _img(0) { };
    Complejo( double r, double i = 0 ) : _real(r), _img(i)
{ };
    double real() { return _real; }
    double img()  { return _img; }
};

void main( )
{
  Complejo z1, z2(1), z3(1,1);
  Complejo *pz1 = new Complejo, *pz2 = new Complejo(1);
  delete pz1;
  delete pz2;
}
```

Esta clase no necesita destructora ni constructora de copia porque todos sus atributos son estáticos.

Definimos ahora una clase que almacena punteros a objetos de tipo *Complejo*

```
class PuntoComplejo {
  private:
    Complejo *_x, *_y;
  public:
    PuntoComplejo( double r = 0, double i = 0 ) {
      _x = new Complejo(r, i);
      _y = new Complejo(r, i); };
    Complejo x() { return *_x; }
    Complejo y() { return *_y; }
    ~PuntoComplejo( ) {
      delete _x;
      delete _y; };
};
```

En general, la destructora ha de liberar, al menos, lo que la constructora ubicó.

- ¿Qué ocurre al ejecutar el siguiente programa?

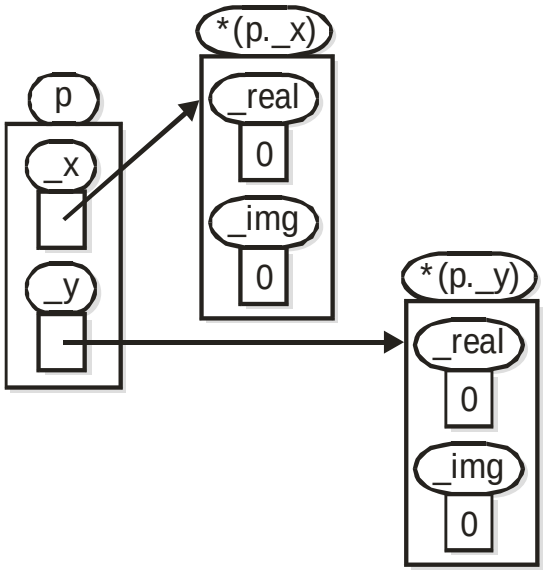
```
void trampa( PuntoComplejo& a )
{
  PuntoComplejo b(1,1);

     a = b;
}

void main( )
{
  PuntoComplejo p;

  trampa( p );

  cout << p.x().real() << "+" << p.x().img() << "i\n";
  cout << p.y().real() << "+" << p.y().img() << "i\n";
}
```

La memoria evoluciona de la siguiente manera:

– Construcción del objeto referenciado por *p*

```
PuntoComplejo p;
```

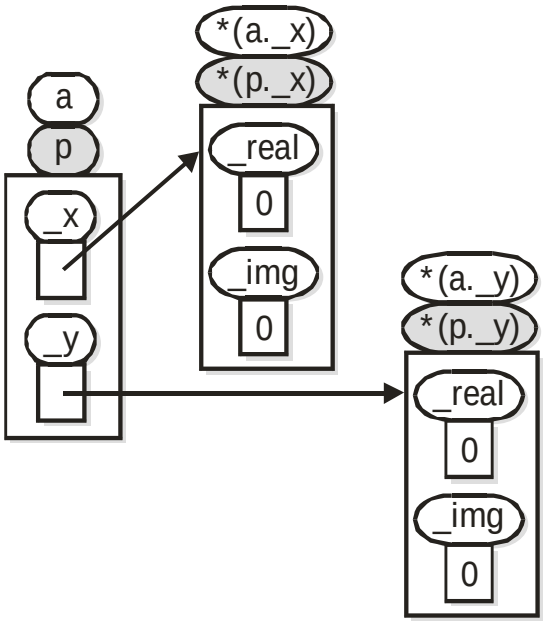
  – obtiene automáticamente espacio para *p* y ejecuta su constructora
  – la constructora de *PuntoComplejo* solicita explícitamente espacio para *_x* e *_y*, para cada uno de los cuales se ejecuta la constructora de *Complejo*



– Invocación a la función trampa

```
void trampa( PuntoComplejo& a )
```

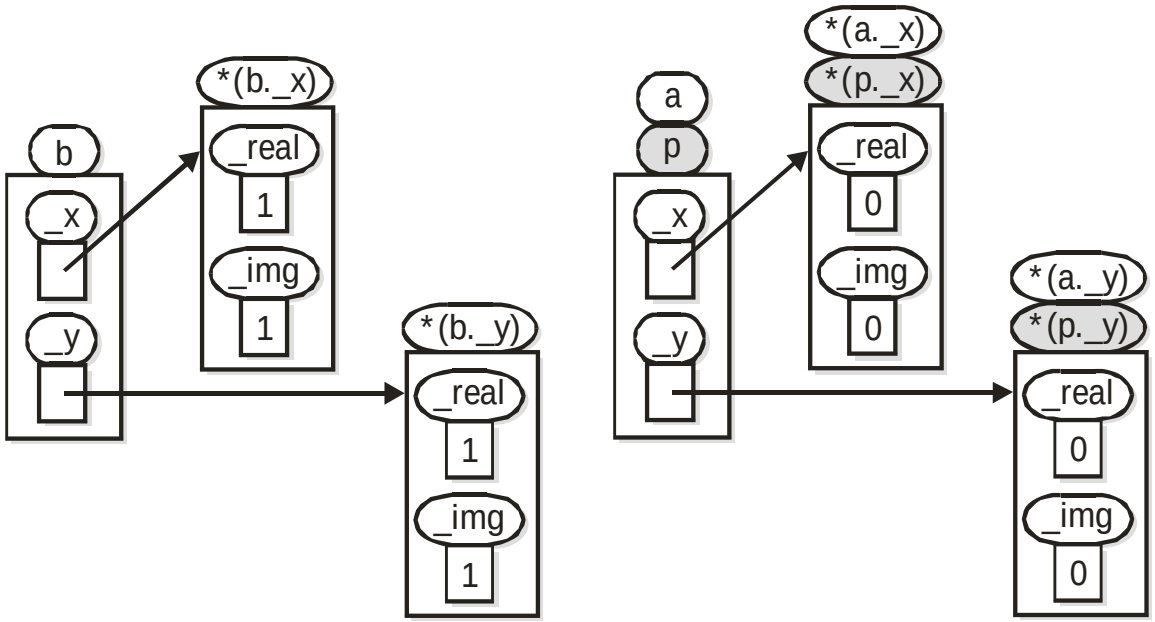
en este ámbito, *a* es un alias de *p* y *p* no es visible



– Construcción del objeto referenciado por *b*

```
PuntoComplejo b(1,1);
```

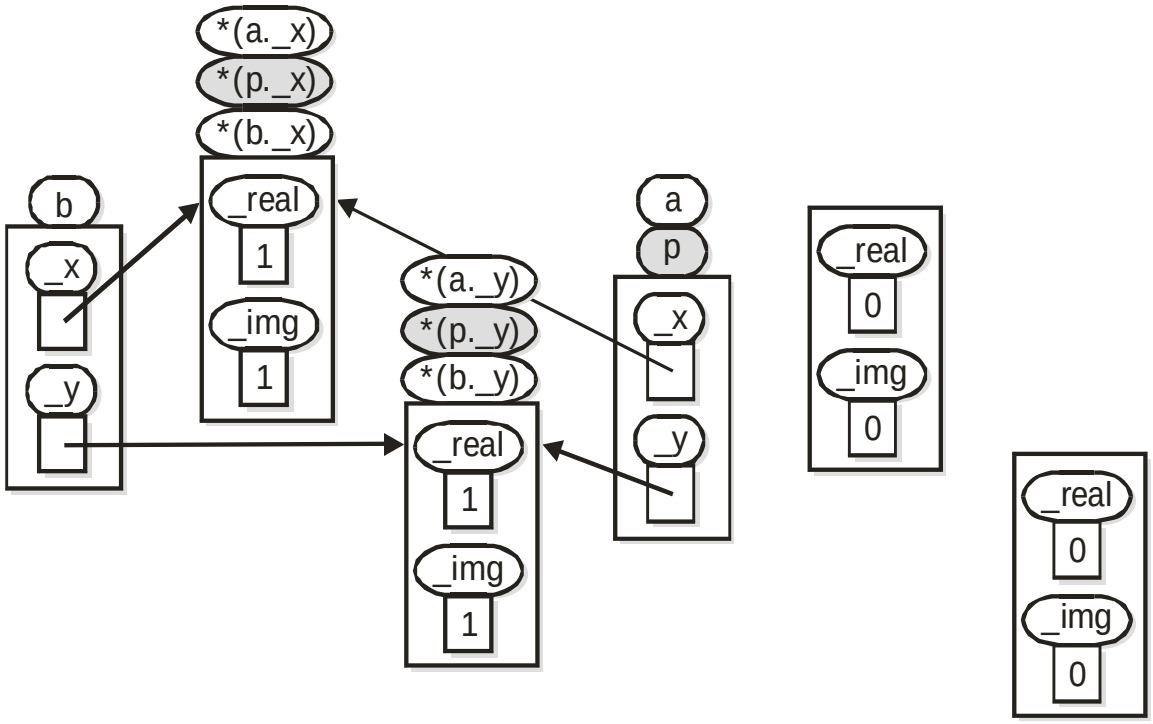
  – obtiene automáticamente espacio para *b* y ejecuta su constructora
  – la constructora de *PuntoComplejo* solicita explícitamente espacio para *_x* e *_y*, para cada uno de los cuales se ejecuta la constructora de *Complejo*



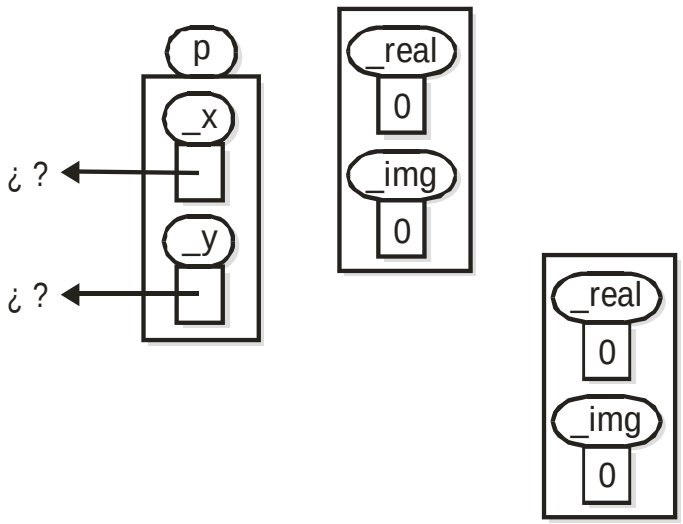
– Asignación *a = b;*

  – se realizan las asignaciones

```
a._x = b._x; a._y = b._y;
```



– Termina la ejecución de trampa
  – se destruye automáticamente el espacio ocupado por *b*, lo que da lugar a que se ejecute la destructora *~PuntoComplejo*, quien libera explícitamente el espacio ocupado por *b._x* y *b._y*
  – *b* y *a* salen de ámbito y dejan de existir
  – Se ha generado basura y referencias perdidas



- El problema que se presenta en el ejemplo anterior es que estamos realizando una asignación entre objetos que contienen variables de tipo puntero.
  – El comportamiento por defecto de la asignación entre objetos consiste en asignar sucesivamente cada uno de sus datos miembro. De esta forma, es-

tamos realizando una asignación entre punteros, con la consiguiente compartición de estructura.

– Para evitar este problema, en C++ es posible redefinir el operador de asignación para asignaciones entre objetos de una clase determinada. De esta forma, será este método el que se ejecute, en lugar del comportamiento por defecto de la asignación.

- La implementación habitual para las redefiniciones del operador de asignación –salvo optimizaciones–
  – comprueba que el origen y el destino de la asignación son diferentes,
  – destruye el valor original del objeto,
  – copia el nuevo valor,
  – y devuelve un puntero al propio objeto

- En el ejemplo, añadimos el nuevo método a la declaración de la clase

```
class PuntoComplejo {
    ...
    PuntoComplejo& operator=(const PuntoComplejo& p);
    ...
};
```

Y lo implementamos

```
PuntoComplejo& PuntoComplejo::operator=(const
PuntoComplejo& p)
{
  if (this != &p) {
    delete _x;
    delete _y;
    _x = new Complejo(p.x().real(), p.x().img());
    _y = new Complejo(p.y().real(), p.y().img());
  }
  return *this;
}
```

donde,

– la pseudovariable *this* contiene un puntero al objeto receptor el mensaje, y
– el operador & devuelve la dirección en la que está ubicada una variable.

- Equipados con nuestro flamante operador de asignación, ¿qué ocurrirá en este ejemplo?

```
void trampa2( PuntoComplejo a ) {
  cout << a.x().real() << "+" << a.x().img() << "i\n";
  cout << a.y().real() << "+" << a.y().img() << "i\n";
}
```

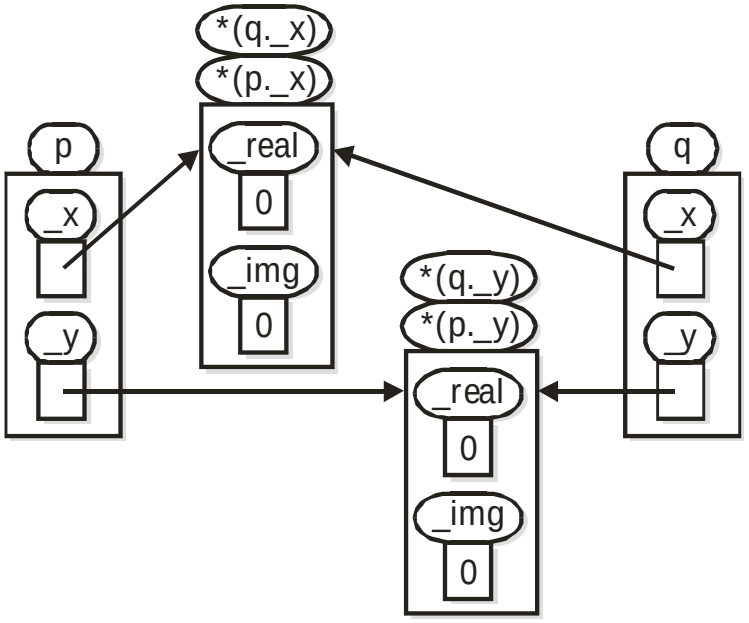
```
void main( ) {
  PuntoComplejo p, q = p;

  trampa2( p );

  cout << p.x().real() << "+" << p.x().img() << "i\n";
  cout << p.y().real() << "+" << p.y().img() << "i\n";
  cout << q.x().real() << "+" << q.x().img() << "i\n";
  cout << q.y().real() << "+" << q.y().img() << "i\n";
}
```

La memoria evoluciona de la siguiente manera:

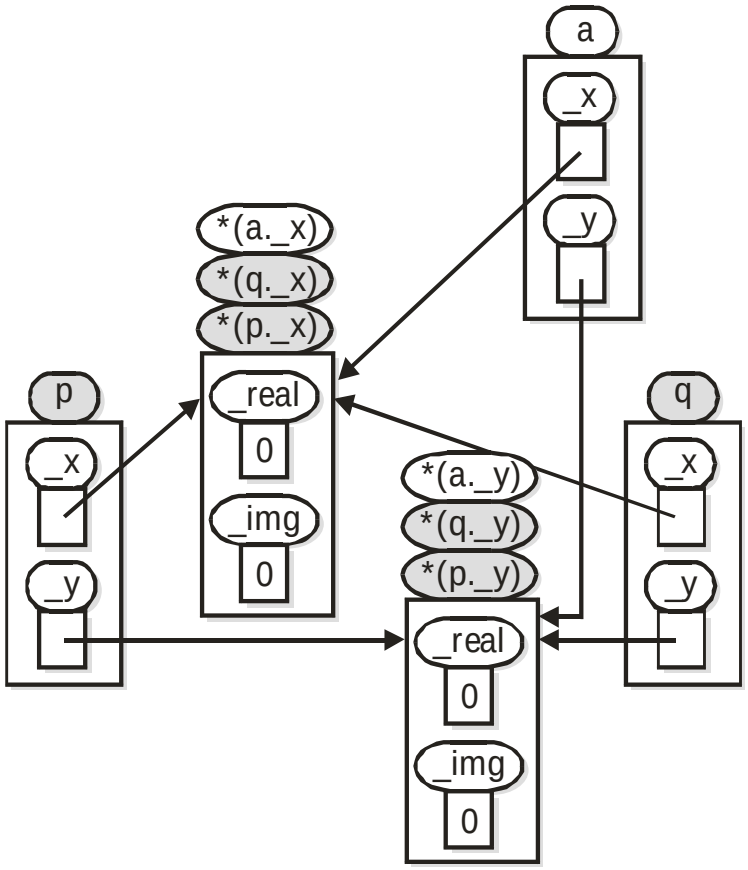
– Construcción de los objetos referenciados por p y q

  `PuntoComplejo p, q = p;`

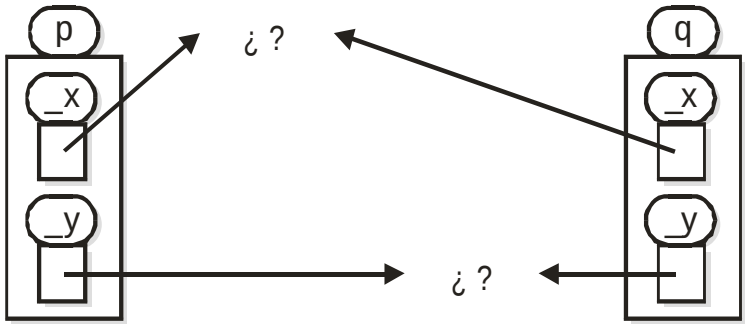
  Mientras que *p* se construye con la constructora por defecto, en la construcción de *q* se obtiene automáticamente espacio para un objeto *PuntoComplejo* cuyos datos se inicializan con el valor de los datos de *p* –comportamiento de la *constructora de copia* por defecto–



– Invocación de la función *trampa2*
  – Se crea automáticamente el objeto al que referencia el parámetro formal *a*, que es inicializado como una copia del parámetro real *p* mediante asignaciones consecutivas de sus datos miembro –comportamiento por de la *constructora de copia* por defecto–.
  – Los identificadores *p* y *q* están fuera de ámbito



– Termina la ejecución de *trampa2*
  – se destruye automáticamente el espacio ocupado por *a*, lo que da lugar a que se ejecute la destructora *~PuntoComplejo*, quien libera explícitamente el espacio ocupado por *a._x* y *a._y*
  – *a* sale de ámbito y deja de existir
  – Se han generado referencias perdidas



➢ La solución radica en añadir a la clase una *constructora de copia* que será el método que se ejecute cuando se cree un objeto a partir de otro creado previamente

```
class PuntoComplejo {
    ...
    PuntoComplejo(const PuntoComplejo& p);
    ...
};
```

e implementarla de forma que se evite la compartición de estructura

```
PuntoComplejo::PuntoComplejo(const PuntoComplejo& p)
{
  _x = new Complejo(p.x().real(), p.x().img());
  _y = new Complejo(p.y().real(), p.y().img());
}
```

Nótese que el parámetro se pasa como una referencia constante para evitar que se ejecute la constructora de copia al invocar a la constructora de copia.

- Cabe preguntarse ¿por qué es tan complicado C++?

  La respuesta es que lo complicado no es –sólo– C++ sino la gestión de la memoria dinámica. Es esta complejidad la razón de que algunos lenguajes incorporen *recolección automática de basura*, un mecanismo que se encarga de liberar las zonas de la memoria dinámica que dejan de ser accesibles. De esta forma, el programador sólo se debe ocupar de ubicar las variables dinámicas, pero no así de liberarlas, con lo cual se evita el problema de las referencias perdidas (aunque no el problema de los efectos colaterales debidos a la compartición de estructura que, de todas formas, puede resultar útil). El precio a pagar es el coste temporal de la recolección automática de basura.

  C++ ofrece mecanismos para tratar de forma elegante todos los problemas relativos a la gestión de la memoria dinámica:

  – Todos los objetos se inicializan pues siempre se ejecuta una constructora en el momento de su creación.
  – Cuando un objeto deja de existir, se invoca automáticamente a su destructora, si la hay.
  – Es posible redefinir la asignación.
  – Es posible redefinir el mecanismo de copia por defecto que se utiliza el paso de parámetros por valor y la inicialización de variables.

  En cualquier caso, es necesario haberse enfrentado a este tipo de problemas para apreciar las bondades de un recolector de basura.

3.5.3 Construcción de estructuras de datos dinámicas

- La utilidad de los punteros para construir colecciones de datos se obtiene cuando un puntero *p* apunta a una variable de tipo registro que contiene en uno de sus campos un puntero del mismo tipo que *p*.

  Pueden usarse entonces los punteros para encadenar entre sí registros, formando estructuras dinámicas, ya que *new* y *delete* podrán usarse en tiempo de ejecución para añadir o eliminar registros de la estructura.

  Por ejemplo:

```
struct TNodo {
  int info;
  TNodo* sig;
};
```

- Dado que en C++ el operador de acceso a los campos de un registro es más prioritario que el de indirección, debemos utilizar paréntesis cuando accedemos a un registro a través de un puntero:

```
TNodo *p;
...
(*p).info = 2;
```

  Afortunadamente, existe una notación alternativa:

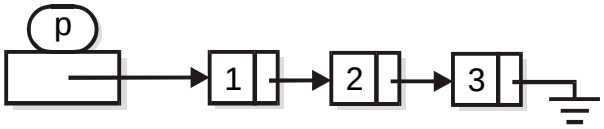
```
p->info = 2;
```

- Para poder construir estructuras dinámicas es necesario poder indicar de alguna forma el final de las estructuras. Esto se hace con un valor especial de tipo puntero: el *puntero vacío* que en C++ se representa con el valor 0.

  0 es una constante polimórfica, compatible con cualquier variable de tipo puntero, que representa un puntero ficticio que no apunta a ninguna variable, de forma que

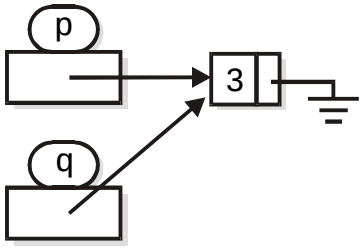
  – **0* está indefinido, y
  – *new 0* y *delete 0* no tienen sentido

- Utilizando registros con punteros como el anterior y la constante 0, podemos construir estructuras dinámicas como esta

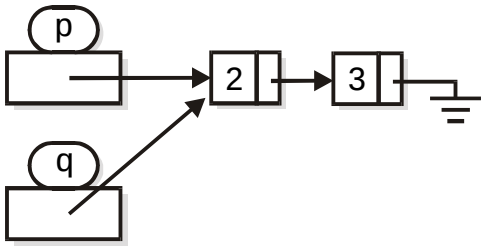


- Por ejemplo

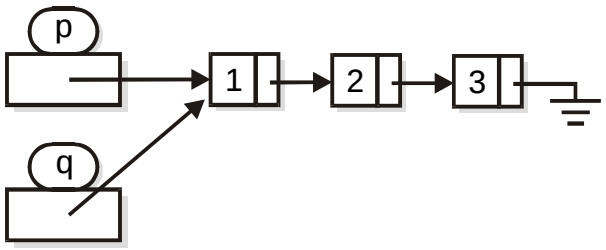
```
TNodo *p, *q;

q = new TNodo;
q->info = 3;
q->sig = 0;
p = q;
```




```
q = new TNodo;
q->info = 2;
q->sig = p;
p = q;
```



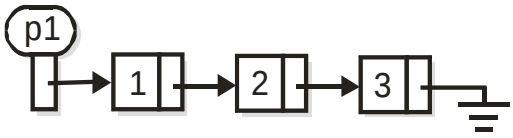

```
q = new TNodo;
q->info = 1;
q->sig = p;
p = q;
```



3.5.4 Implementación de TADs mediante estructuras de datos dinámicas: implementación de las pilas

Tipo representante

- Representamos las pilas como una lista enlazada de nodos, donde la cima es el primer elemento de la lista.



En muchas ocasiones necesitaremos definir estructuras auxiliares para representar a los *nodos* de una estructura dinámica. Para aprovecharnos de los mecanismos que C++ proporciona para las clases, en lugar de registros

```
template <class TElem>
struct TNodoPila {
  TElem elem;
  TNodoPila* sig;
};
```

utilizaremos clases para los nodos de las estructuras dinámicas

```
template <class TElem>
class TNodoPila {
  TElem _elem;
  TNodoPila<TElem>* _sig;
  ...
}
```

de forma que el tipo representante de las pilas utilice esta clase auxiliar

```
template <class TElem>
class TPilaDinamica {
  ...
  private:
     TNodoPila<TElem>* _cima;
}
```

- Invariante de la representación

  Dada la cima de una pila *p : TNodoPila**, definimos

```
      R_Pila[Elem](p)
⇔def
      p = 0 ∨
    ( p ≠ 0 ∧ ubicado(p) ∧
      R_TElem(p->_elem) ∧ R_Pila[TElem](p->_sig) ∧
      p ∉ cadena(p->_sig)                        )
```

  donde la función auxiliar cadena permite obtener el conjunto de punteros accesibles a partir de uno dado:

```
cadena( 0 ) =def ∅
cadena( p ) =def {p} ∪ cadena(p^.sig)   si p ≠ 0
```

  En este invariante de la representación se expresa que

  – la lista enlazada está vacía (*p=0*) o no lo está
  – si no está vacía, entonces
    – todos los nodos están ubicados,
    – el campo *_elem* de cada nodo es un representante válido del tipo *TElem*,
    – no hay ciclos en la estructura.

  El problema para implementar una función que determine si un valor de tipo *TNodoPila** es un representante válido radica en que no existe forma de saber si un puntero está o no ubicado.

- Dado que las clases que implementan a los nodos sólo han de ser accesibles para la clase en cuya implementación colaboran, declararemos sus datos y su constructora como privadas, dando acceso privilegiado, en forma de *clase amiga*, a la clase principal, que ha de ser declarada por anticipado debido a la existencia de referencias cruzadas.

```
template <class TElem>
class TPilaDinamica;
template <class TElem>
class TNodoPila {
  private:
    TElem _elem;
    TNodoPila<TElem>* _sig;
    TNodoPila( const TElem&, TNodoPila<TElem>* );
  public:
    const TElem& elem() const;
    TNodoPila<TElem> * sig() const;
    friend TPilaDinamica<TElem>;
};
```

La implementación de la clase de los nodos queda entonces:

```
template <class TElem>
TNodoPila<TElem>::TNodoPila( const TElem& elem,
TNodoPila<TElem>* sig ) :
  _elem(elem), _sig(sig) {
};

template <class TElem>
const TElem& TNodoPila<TElem>::elem() const {
  return _elem;
}

template <class TElem>
TNodoPila<TElem>* TNodoPila<TElem>::sig() const {
  return _sig;
}
```

Y la constructora de las pilas:

```
template <class TElem>
TPilaDinamica<TElem>::TPilaDinamica( ) :
  _cima(0) {
};
```

Copias, destrucciones y asignaciones

- Dado que el uso de estructuras dinámicas puede dar lugar a los problemas mencionados en un apartado anterior, adoptaremos una estrategia conservadora que evite la compartición de estructura y equiparemos a las implementaciones de los TADs con:
  - constructora de copia,
  - destructora, y
  - operador de asignación.

  Dado que en C++ no puede existir un objeto que no haya sido inicializado mediante la invocación a una constructora, en una asignación entre objetos de la forma

  ```
  a = b
  ```

  a siempre contendrá un objeto válido del tipo en cuestión. Es por ello, que la asignación consistirá en la anulación del valor existente seguida de una copia del nuevo valor.

  Definimos entonces dos operaciones privadas auxiliares *libera* y *copia* de forma que, salvo por el tipo de los parámetros, la implementación de la constructora de copia, la destructora y el operador de asignación será siempre la misma. Por ejemplo, en el caso de las pilas:

```
 template <class TElem>
 TPilaDinamica<TElem>::TPilaDinamica( const
TPilaDinamica<TElem>& pila ) {
    copia(pila);
 };

 template <class TElem>
 TPilaDinamica<TElem>::~TPilaDinamica( ) {
   libera();
 };

 template <class TElem>
 TPilaDinamica<TElem>&
 TPilaDinamica<TElem>::operator=( const TPilaDinamica<TElem>&
pila ) {
   if( this != &pila ) {
     libera();
     copia(pila);
   }
   return *this;
 };
```

- En el caso de las pilas *copia* y *libera* se implementan de la siguiente forma:

```
template <class TElem>
void TPilaDinamica<TElem>::libera() {
  while (_cima != 0) {
    TNodoPila<TElem>* tmp = _cima;
    _cima = _cima->_sig;
    delete tmp;
  }
};

template <class TElem>
void TPilaDinamica<TElem>::copia(const
TPilaDinamica<TElem>& pila) {
  if ( pila.esVacio() )
    _cima = 0;
  else {
    TNodoPila<TElem> *antCopia, *actCopia, *act;
    act = pila._cima;
    _cima = new TNodoPila<TElem>( act->_elem, 0 );
    actCopia = _cima;
    while ( act->_sig != 0 ) {
      act = act->_sig;
      antCopia = actCopia;
      actCopia = new TNodoPila<TElem>( act->_elem, 0 );
      antCopia->_sig = actCopia;
    }
  }
};
```

- Sin embargo esta política conservadora que evita la compartición de estructura, penaliza el coste temporal de las operaciones.

  Si implementamos la constructora de los nodos y la operación apila de la siguiente forma:

```
template <class TElem>
void TPilaDinamica<TElem>::apila( TElem elem ) {
  _cima = new TNodoPila<TElem> ( elem, _cima );
};

template <class TElem>
TNodoPila<TElem>::TNodoPila( TElem elem, TNodoPila<TElem>*
sig ) :
  _elem(elem), _sig(sig) {
};
```

  ¿cuál es el coste temporal de la siguiente invocación a *apila*?

```
TPilaDinamica< TPilaDinamica<int> > pilaDePilas;
TPilaDinamica<int> pila;
...

pilaDePilas.apila( pila );
```

  considerando que

n_1 = número de elementos de pilaDePilas
n_2 = número de elementos de pila

  se tiene

$$
\begin{aligned}
t(n_1, n_2) > {} & n_2 + \text{// copia de pila a elem en la invocación de apila} \\
& n_2 + \text{// copia de elem a elem en la invocación de la} \\
& \quad \text{// constructora de TNodoPila} \\
& n_2 + \text{// copia de elem al dato \_elem en la construcción} \\
& \quad \text{// del nuevo nodo} \\
& n_2 + \text{// destrucción del parámetro elem en la} \\
& \quad \text{// constructora de TNodoPila} \\
& n_2 + \text{// destrucción del parámetro elem en apila} \\
= {} & 5\, n_2
\end{aligned}
$$

Una copia es inevitable para evitar la compartición de estructura, pero no así las dos copias adicionales con sus consiguientes anulaciones. La solución podría ser pasar los parámetros siempre por referencia, evitándose así las copias, sin embargo esto iría contra la lógica de los algoritmos y pondría en peligro la integridad de los datos. C++ ofrece un tercer tipo de parámetros: las referencias constantes. Un parámetro de tipo referencia constante se pasa por variable pero sin permitir que el código de la función modifique el valor.

```
template <class TElem>
void TPilaDinamica<TElem>::apila(const TElem& elem) {
  _cima = new TNodoPila<TElem> ( elem, _cima );
};

template <class TElem>
TNodoPila<TElem>::TNodoPila( const TElem& elem,
TNodoPila<TElem>* sig ) :
  _elem(elem), _sig(sig) {
};
```

que sólo realiza una copia y ninguna destrucción.

- Así pues, para ser estrictos, en la complejidad de las implementaciones de los TAD deberemos incluir la complejidad de las copias y destrucciones de los elementos, con la siguiente notación
  - T(TElem::TElem( ))
    coste de la constructora de los elementos
  - T(TElem::TElem(TElem&))
    coste de la constructora de copia de los elementos
  - T(TElem::~TElem( ))
    coste de la destructora de los elementos
  - T(TElem::operator=(TElem&))
    coste de la asignación entre elementos

  Así, por ejemplo, el coste de la última versión de *apila* queda

  $T(\text{TElem::TElem(TElem\&)}) + O(1)$

  siendo $O(1)$ sobre un tipo predefinido.

- Y lo mismo ocurre con los resultados de las funciones.

  Dada esta implementación de la operación *cima*

```
template <class TElem>
TElem TPilaDinamica<TElem>::cima( ) throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido ();
  else
    return _cima->_elem;
};
```

  ¿cuál es el coste temporal de la siguiente invocación a *apila*?

```
TPilaDinamica< TPilaDinamica<int> > pilaDePilas;
TPilaDinamica<int> pila;
...

pila = pilaDePilas.cima();
```

  considerando que

  n_1 = número de elementos de la pila que está en la cima de pilaDePilas
  n_2 = número de elementos de pila

  se tiene

$$t(n_1, n_2) > n_1 + \text{// copia de \_cima->\_elem al devolver el resultado}$$
$$n_2 + \text{// destrucción del valor antiguo de pila}$$
$$n_1 + \text{// copia de la copia de \_cima->\_elem sobre pila}$$
$$n_1 \quad \text{// destrucción de la copia de \_cima->\_elem}$$
$$= 3\,n_1 + n_2$$

  La solución de nuevo es utilizar referencias constantes

```
template <class TElem>
const TElem& TPilaDinamica<TElem>::cima( ) throw
(EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido ();
  else
    return _cima->_elem;
};
```

  de forma que en el ejemplo anterior sólo realiza una copia de *_cima->_elem* además de la destrucción del valor antiguo de *pila*.

- Es importante que estas referencias también sean constantes porque si no lo fueran podríamos modificar los valores almacenados en el TAD encadenando mensajes:

```
pilaDePilas.cima().desapila();
```

aunque esta decisión también nos prohíbe encadenamientos válidos de mensajes que no producen efectos colaterales, como por ejemplo

```
cout << pilaDePilas.cima().cima();
```

porque a un referencia constante no se le pueden pasar mensajes que la puedan modificar …

Pero ¡ si *cima* no modifica la pila !, pero el compilador no lo sabe, para indicárselo es necesario marcar el método *cima* como constante

```
template <class TElem>
const TElem& TPilaDinamica<TElem>::cima( ) const throw
(EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido ();
  else
    return _cima->_elem;
};
```

- Con todo esto, las operaciones propias de las pilas quedan:

```
template <class TElem>
void TPilaDinamica<TElem>::apila(const TElem& elem) {
  _cima = new TNodoPila<TElem> ( elem, _cima );
};
// Complejidad: T(TElem::TElem(TElem&)) + O(1)

template <class TElem>
const TElem& TPilaDinamica<TElem>::cima( ) const throw
(EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido();
  else
    return _cima->_elem;
};
// Complejidad: O(1)

template <class TElem>
void TPilaDinamica<TElem>::desapila( ) throw (EAccesoIndebido)
{
  if( esVacio() )
    throw EAccesoIndebido();
  else {
    TNodoPila<TElem>* tmp = _cima;
    _cima = _cima->_sig;
    delete tmp;
  }
};
// Complejidad: T(TElem::~TElem()) + O(1)

template <class TElem>
bool TPilaDinamica<TElem>::esVacio( ) const {
  return _cima == 0;
};
// Complejidad: O(1)
```

- ¿Con esta política de copias estamos condenados a manejar estructuras de datos seguras pero ineficientes?

  No, en primer lugar, la compartición o no de las estructuras depende de la implementación del constructor de copia de los elementos, y, por lo tanto, es posible instanciar las plantillas sobre elementos que no realicen copias.

  En segundo lugar, si queremos evitar por completo las copias no tenemos más que instanciar las plantillas con punteros a elementos. Aunque los punteros se copien, las estructuras a las que apunten serán compartidas.

  Sin embargo, de nuevo, aunque mejoremos la eficiencia de los programas habremos de ser más cuidadosos y ser conscientes de los posibles efectos colaterales, las referencias perdidas y la generación de basura.

  Por ejemplo, si utilizamos una variable auxiliar local a una función que contenga elementos de tipo puntero, nos deberemos ocupar de anular los elementos explícitamente antes de salir de la función; a no ser que esos elementos formen parte también de alguno de los resultados, en cuyo caso, si los anulásemos estaríamos generando referencias perdidas.

  Por ejemplo, en esta función es necesario anular todos los elementos de *pilas* excepto la cima, que se devuelve como resultado.

```
typedef TPilaDinamica< TPilaDinamica<int>* > TPuntPilas;

const TPilaDinamica<int>& pilaN ( int num ) {
  TPuntPilas pilas;
  TPilaDinamica<int>* pila;
  pila = new TPilaDinamica<int>();
  pila->apila(0);
  pilas.apila(pila);
  for ( int i = 1; i < num; i++ ) {
    pila = new TPilaDinamica<int>( * pilas.cima() );
    pila->apila( i );
    pilas.apila(pila);
  }
  pila = pilas.cima();
  pilas.desapila();
  while ( ! pilas.esVacio() ) {
    delete pilas.cima();
    pilas.desapila();
  }
  return *pila;
}
```

Aunque, en realidad, la anterior implementación es intencionadamente ineficiente, ya que las pilas pueden compartir estructura entre sí, no siendo necesaria la construcción de copias.

```
const TPilaDinamica<int>& pilaN2 ( int num ) {
  TPuntPilas pilas;
  TPilaDinamica<int>* pila;

  pila = new TPilaDinamica<int>();
  pila->apila(0);
  pilas.apila(pila);
  for ( int i = 1; i < num; i++ ) {
    pila = pilas.cima();
    pila->apila( i );
    pilas.apila(pila);
  }

  return * pilas.cima();
}
```

¿Podrías dibujar cómo evoluciona la memoria cuando se realizan las invocaciones *pilaN(4)* y *pilaN2(4)*?

Almacenamiento de los datos en disco

- Los datos representados por estructuras dinámicas no pueden escribirse y leerse en archivos directamente, porque los valores de los punteros –direcciones de memoria–serán diferentes en cada ejecución particular. Es por ello que
  - la escritura de estructuras dinámicas se limitará a escribir la información –los elementos– sin guardar los valores de los punteros, y
  - la lectura deberá reconstruir las estructuras dinámicas, a partir de los datos leídos del archivo, solicitando nueva memoria dinámica donde almacenar los datos leídos.

- Normalmente incluiremos en los TAD una operación de escritura y redefiniremos el operador de inserción para el tipo en cuestión.

```
template <class TElem>
void TPilaDinamica<TElem>::escribe( ostream& salida ) const {
  TNodoPila<TElem>* tmp = _cima;
  while ( tmp != 0 ) {
    salida << tmp->elem() << endl;
    tmp = tmp->sig();
  }
};

template <class TElem>
ostream& operator<<( ostream& salida, const
TPilaDinamica<TElem>& pila ){
  pila.escribe(salida);
  return salida;
};
```

La razón de no declarar directamente el operador de inserción como amigo de la clase *TPilaDinamica*, evitando así la necesidad de definir la operación *escribe*, radica en que el mecanismo de plantillas interfiere con la resolución de identificadores y obliga a que la implementación del operador de inserción aparezca directamente en la definición de clase.

Nótese que la escritura de las pilas se basa en la escritura de sus elementos, y que, por lo tanto, ahora sólo podremos instanciar la plantilla *TPilaDinamica<TElem>* con tipos sobre los que esté definido *operador<<*.

Igualdad

- El tipo representante de un tipo abstracto puede no admitir el uso de la comparación de igualdad. Y aunque la admitiese, la identidad entre valores del tipo representante en general no corresponderá a la igualdad entre los valores abstractos representados. Este problema se presenta por igual en las implementaciones estáticas como en las dinámicas; aunque en las estáticas puede o no ocurrir mientras que en las dinámicas sucede siempre[3].

```
int *p1, *p2;
p1 = new int;
p2 = new int;
*p1 = 1;
*p2 = 1;
cout << ( p1 == p2 ? “iguales” : “diferentes” ) << endl;
```

Aunque dos estructuras dinámicas almacenen exactamente los mismos valores, lo harán en posiciones de memoria diferentes, es decir, apuntados por punteros diferentes. La solución radica en redefinir los operadores correspondientes a las comparaciones que se deseen permitir sobre los valores del TAD, haciendo que dichas implementaciones recorran las estructuras comparando los valores e ignorando el valor de los punteros.

```
template <class TElem>
bool TPilaDinamica<TElem>::operator==( const TPilaDinamica&
pila ) const {
  bool iguales;
  TNodoPila<TElem> *p, *q;
  iguales = true;
  p = _cima;
  q = pila._cima;
  while ( ( p != 0 ) && (q != 0) && iguales ) {
    iguales = p->elem() == q->elem();
    p = p->sig();
    q = q->sig();
  }
  return iguales && ( p == 0 ) && ( q == 0 );
}
```

Nótese que la comparación entre pilas se basa en la comparación entre sus elementos, y que, por lo tanto, ahora sólo podremos instanciar la plantilla *TPilaDinamica<TElem>* con tipos sobre los que esté definido *operador==*.

[3] Por otra parte, C++ no permite realizar comparaciones entre objetos a no ser que se haya redefinido el operador correspondiente.

La clase *TPilaDinamica*

```
template <class TElem>
class TPilaDinamica {
  public:
  // Constructoras, destructora y operador de asignación
     TPilaDinamica( );
     TPilaDinamica( const TPilaDinamica<TElem>& );
     ~TPilaDinamica( );
     TPilaDinamica<TElem>& operator=( const
TPilaDinamica<TElem>& );

  // Operaciones de las pilas
     void apila(const TElem&);
     // Pre: true
     // Post: Se añade 'elem' a la cima de la pila
     const TElem& cima( ) const throw (EAccesoIndebido);
     // Pre: ! esVacio( )
     // Post: Devuelve el elemento que está en la cima
     // Lanza la excepción TPilaArray<TElem>::Vacio si la pila
está vacía
     void desapila( ) throw (EAccesoIndebido);
     // Pre: ! esVacio( )
     // Post: Elimina el elemento que está en la cima
     // Lanza la excepción TPilaArray::Vacio si la pila está
vacía
     bool esVacio( ) const;
     // Pre: true
     // Post: Devuelve true | false según si la pila está o no
vacía

  // Comparación
     bool operator==( const TPilaDinamica& ) const;
     // Pre: true
     // Post: Determina si dos pilas coinciden

  // Escritura
     void escribe( ostream& ) const;

  private:
  // Variables privadas
     TNodoPila<TElem>* _cima;

  // Operaciones privadas
     void libera();
     void copia(const TPilaDinamica<TElem>& pila);
};
```

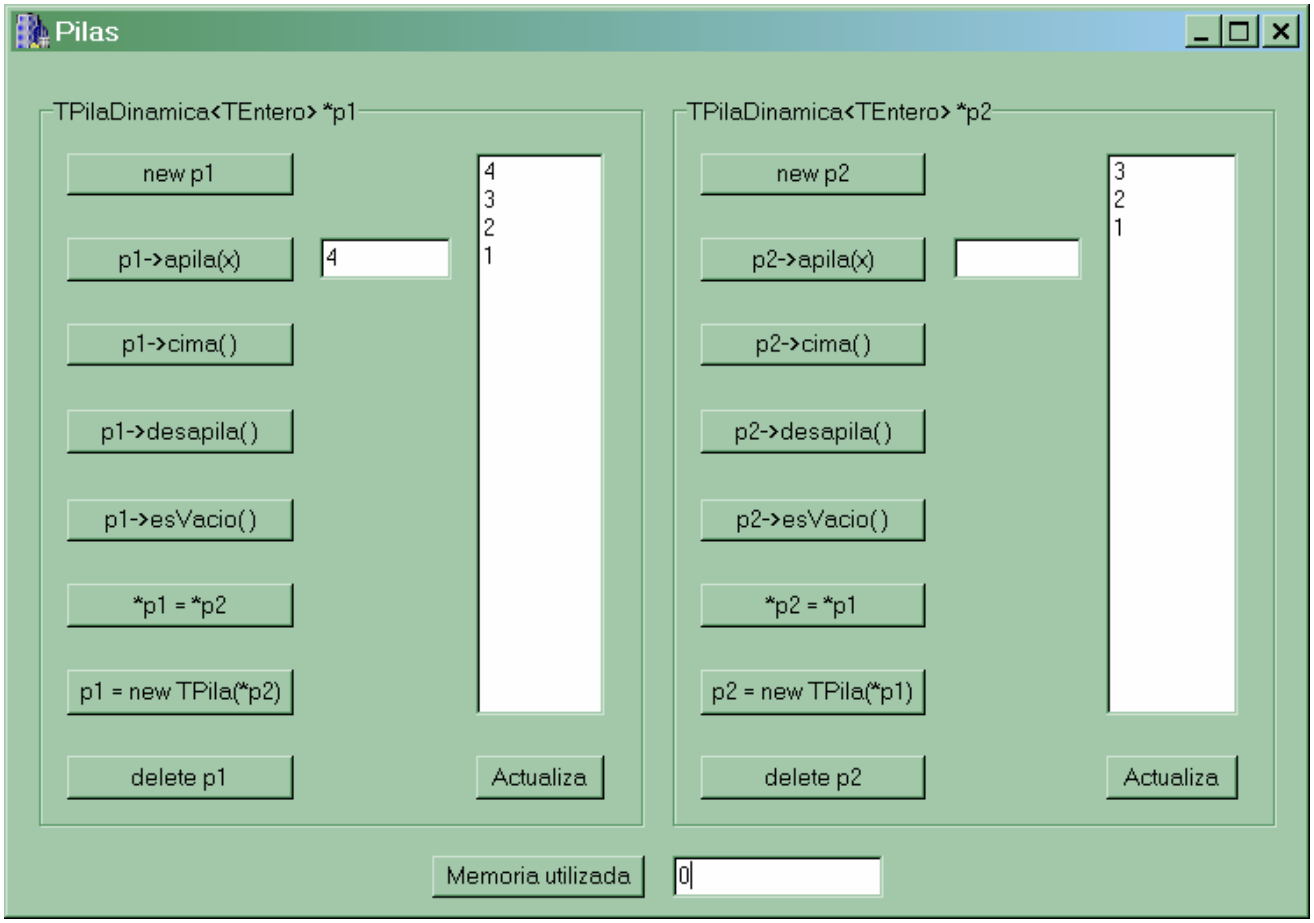
3.5.5 Prueba de los TADs

- Ampliamos la aplicación de prueba para que también se ocupe de las nuevas operaciones. Además, utilizaremos como tipo de los elementos la clase *TEntero* que encapsula al tipo *int* y que nos permite comprobar el funcionamiento de las constructoras, la asignación y la destructora.

```
class TEntero {
  public:
    TEntero( int valor = 0 ): _valor(new int(valor)) {
//      ShowMessage("TEntero: constructora por defecto");
    };
    TEntero( const TEntero& entero ) : _valor(new
int(*entero._valor)) {
//      ShowMessage("TEntero: constructora de copia");
    };
    ~TEntero( ) {
//      ShowMessage("TEntero: destructora");
      delete _valor;
    };
    TEntero& operator=( const TEntero& entero ) {
//      ShowMessage("TEntero: asignación");
      if( this != &entero )
        *_valor = *entero._valor;
      return *this;
    };
    bool operator==( const TEntero& entero ) {
      return *_valor == *entero._valor;       };
    bool operator<( const TEntero& entero ) {
      return *_valor < *entero._valor;        };
    bool operator>( const TEntero& entero ) {
      return *_valor > *entero._valor;        };
    int valor( ) const { return *_valor; };
    TEntero& operator++ ( ) {
      (*_valor)++;
      return *this;
    };
    friend std::ostream& operator<<( std::ostream& salida,
                                     const TEntero& entero ) {
      salida << *entero._valor;
      return salida;
    };
  private:
    int* _valor;
};
```

- El formulario maneja dos punteros a valores del tipo en cuestión

```
typedef TPilaDinamica<TEntero> TPilaEnt;
TPilaEnt *p1, *p2;
```



y si la función *GetHeapStatus* funcionase, también se mostraría la memoria dinámica utilizada en cada instante …