Capítulo 5

Especificación e implementación de TADs[1]

La perfección no se obtiene
cuando se ha añadido todo lo añadible,
sino cuando se ha quitado todo lo superfluo.

Antoine de Saint-Exupéry (Escritor francés, 1900-1944)

Resumen: En este tema se introducen los (TADs), que permiten definir y abstraer tipos de forma similar a como las funciones definen y abstraen código. Presentaremos la forma de definir un TAD, incluyendo tanto su especificación externa como su representación interna, y veremos, como caso de estudio, el TAD "iterador".

1. Introducción

Abstracción

⋆ El exceso de detalles hace más difícil tratar con los problemas. Para concentrarnos en lo realmente importante, abstraemos el problema, eliminando todos los detalles innecesarios.

⋆ Por ejemplo, para leer un fichero de configuración desde un programa, el programador no tiene que preocuparse del formato del sistema de archivos, ni de su soporte físico concreto (si se trata de una unidad en red, una memoria USB o un disco que gira miles de veces por segundo) - en lugar de eso, las librerías del sistema y el sistema operativo le permiten usar la abstracción "fichero" – una entidad de la que se pueden leer y a la que se pueden escribir bytes, y que se identifica mediante una ruta y un nombre. Un fichero es un ejemplo de tipo abstracto de datos (TAD).

⋆ Se puede hablar de dos grandes tipos de abstracción:

[1] Manuel Freire es el autor principal de este tema.

- funcional: abstrae una *operación*. Es la que hemos visto hasta ahora. La operación se divide en "especificación" (qué es lo que hago) e "implementación" (cómo lo llevo a cabo). De esta forma, un programador puede llamar a `ordena(v, n)` con la plena confianza de que obtendrá un vector ordenado en $O(n \log n)$, y sin tener que preocuparse por los detalles de implementación del algoritmo concreto.
- de datos: abstrae un *tipo de dato*, o mejor dicho, el uso que se puede hacer de este tipo (por ejemplo, un fichero) de la forma en que está implementado internamente (por ejemplo, bloques de un disco magnetizado estructurados como un sistema de archivos NTFS). Si la abstracción funcional es una forma de *añadir operaciones* a un lenguaje, la abstracción de datos se puede ver como una forma de *añadir tipos* al lenguaje.

TADs predefinidos

⋆ Algunos tipos de TADs vienen predefinidos con los lenguajes. Por ejemplo, enteros, caracteres, números en coma flotante, booleanos o arrays tienen todos representaciones internas "opacas" que no es necesario conocer para poderlos manipular. Por ejemplo:

- los enteros (`char`, `int`, `long` y derivados) usan, internamente, representación binaria en complemento a 2. Las operaciones +, -, *, /, % (entre otras) vienen predefinidas, y en general no es necesario preocuparse por ellas
- los números en coma flotante (`float` y `double`)usan una representación binaria compuesta por mantisa y exponente, y disponen de las operaciones que cabría esperar; en general, los programadores evitan recurrir a la representación interna.
- los vectores (en el sentido de *arrays*) tienen su propia representación interna y semántica externa. Por ejemplo, `v[i]` se refiere a la i-ésima posición, y es posible tanto escribir como leer de esta posición, sin necesidad de pensar en cómo están estructurados los datos físicamente en la memoria.

⋆ Aunque, en general, los TADs se comportan de formas completamente esperadas (siguiendo el *principio de la mínima sorpresa*), a veces es importante tener en cuenta los detalles. Por ejemplo, este código demuestra una de las razones por las cuales los `float` de C++ no se pueden tratar igual que los `int`: sus dominios se solapan, pero a partir de cierto valor, son cada vez más distintos.

```
int ejemplo() {
  int i=0;    // 31 bits en complemento a 2 + bit de signo
  float f=0;  // 24 bits en complemento a 2 + 7 de exponente + signo
  while (i == f) { i++; f++; } // int(2^24+1) ≠ float(2^24+1) (!)
  return i;
}
```

⋆ Este otro fragmento calcula, de forma aproximada, la inversa de la raíz cuadrada de un número n - abusando para ello de la codificación de los enteros y de los flotantes en C/C++. Es famosa por su uso en el videojuego *Quake*, y aproxima muy bien (y de forma más rápida que `pow`) el resultado de `pow(n, .5)`. Los casos en los que está justificado recurrir a este tipo de optimizaciones son *muy* escasos; pero siempre que se realiza una abstracción que no permite acceso a la implementación subyacente,

se sacrifican algunas optimizaciones (como ésta, y a menudo más sencillas) en el proceso.

```
float Q_rsqrt(float n) {
  const float threehalfs = 1.5f;
  float x2 = n * 0.5f;
  float y  = n;
  int i  = * ( long * ) &y;      // convierte de float a long (!)
  i  = 0x5f3759df - ( i >> 1 );
  y  = * ( float * ) &i;         // convierte de long a float (!)
  y  = y * ( threehalfs - ( x2 * y * y ) );
  return y;
}
```

⋆ Un ejemplo clásico de TAD predefinido de C++ es la cadena de caracteres de su librería estándar: `std::string`. Dada una cadena `s`, es posible saber su longitud (`s.length()` ó `s.size()`), concatenarle otras cadenas (`s += t`), sacar copias (`t = s`), o mostrarla por pantalla (`std::cout << s`), entre otras muchas operaciones – y todo ello sin necesidad de saber cómo reserva la memoria necesaria ni cómo codifica sus caracteres.

TADs de usuario

⋆ Los `struct`s de C, o los `record`s de Pascal, sólo permiten definir nuevos tipos de datos, pero no permiten ocultar los detalles al usuario (el programador que hace uso de ellos). Esta ocultación sí es posible en C++ u otros lenguajes que soportan orientación a objetos.

⋆ Un tipo de datos Fecha muy básico, sin ocultación o abstracción alguna, sería el siguiente:

```
struct Fecha {
  int dia;
  int mes;
  int anyo;  // identificadores en C++: sólo ASCII estándar
  Fecha(int d, int m, int a): dia(d), mes(m), anyo(a) {}
};
```

⋆ Ahora, es posible escribir `Fecha f = Fecha(14, 7, 1789)` ó `Fecha f(14, 7, 1789)`, y acceder a los campos de esta fecha mediante `f.dia` ó `f.mes`: hemos definido un nuevo tipo, que no existía antes en el lenguaje. Por el lado malo, es posible crear fechas inconsistentes; por ejemplo, nada impide escribir `f.mes = 13` ó, más sutil, `f.mes = 2; f.dia = 30` (febrero nunca puede tener 30 días). Es decir, tal y como está escrita, es facil salirse fuera del *dominio* de las fechas válidas.

⋆ Además, esta `Fecha` es poco útil. No incluye operaciones para sumar o restar días a una fecha, ni para calcular en qué día de la semana cae una fecha dada, por citar dos operaciones frecuentes. Tal y como está, cada programador que la use tendrá que (re)implementar el mismo conjunto de operaciones básicas para poder usarlas, con el consiguiente coste en tiempo y esfuerzo; y es altamente probable que dos implementaciones distintas sean incompatibles entre sí (en el sentido de distintas precondiciones/postcondiciones). Una buena especificación de TAD debería incluir las *operaciones* básicas del tipo, facilitando así su uso estándar.

⋆ Todo TAD especifica una serie de **operaciones** del tipo; estas operaciones son las que el diseñador del tipo prevee que se van a querer usar con él. Pueden hacer uso de otros tipos; por ejemplo, un tipo `Tablero` podría usar un tipo `Ficha` y un tipo `Posicion` en su operación `mueve`.

Fechas

Este ejemplo presenta un tipo Fecha[2] algo más completo, donde se especifica (de forma informal) el dominio de una implementación concreta y una serie de operaciones básicas.

Nota: en este ejemplo se usa notación de C++ "orientado a objetos". El lector que no sea familiar con esta notación debe entender que, en todas las operaciones de una clase (excepto los constructores), se está pasando un parámetro adicional implícito: el objeto del tipo que se está definiendo (en este caso una `Fecha`) sobre el que operar. De esta forma, dada una `Fecha f`, la instrucción `f.dia()` se convierte internamente a `dia(f)`.

```
// dominio: 1/1/1 al (1/1/2^15 - 1) AD, ambos inclusive
class Fecha {
public: // parte publica del TAD
  // Constructor / Generador
  // {P0 : dia, mes y año forman una fecha en el dominio esperado}
  // {Q0 : devuelve la Fecha con el dia, mes y año correspondientes}
  Fecha(int dia, int mes, int anyo);

  // Constructor / Generador
  // {P0 : delta es un número de días a sumar (o restar),
  //   tal que base+delta esta dentro del dominio de Fecha}
  // {Q0 : una nueva Fecha, delta días en el futuro (o pasado) de base}
  Fecha(const Fecha &base, int delta);

  // {P0 : }
  // {Q0 : devuelve distancia en días de fecha actual con la dada}
  int distancia(const Fecha & otra) const;

  // Accesores

  // {P0 : }
  // {Q0 : devuelve dia del mes de la fecha actual}
  int dia() const;

  // {P0 : } - {Q0 : devuelve el día de la semana}
  int diaSemana() const;

  // {P0 : } - {Q0 : devuelve el mes de esta fecha}
  int mes() const;

  // {P0 : } - {Q0 : devuelve el año de esta fecha}
  int anyo() const;
```

[2] Para ver un ejemplo real (y muy, muy flexible) de TAD fecha para su uso en C++, `http://www.boost.org/doc/libs/1_48_0/doc/html/date_time.html`. También hay una discusion de Fecha más instructiva en la sección 10.2 de Stroustrup (1998).

```
  // {P0 : } - {Q0 : devuelve el dia del año}
  int diaAnyo() const;

private: // parte privada, accesible solo para la implementacion
  int _dia;
  int _mes;
  int _anyo;
};
```

⋆ La *parte privada* de la clase determina la estructura interna de los objetos del tipo, y es la que limita el dominio. Con la estructura interna de `Fecha`, podríamos ampliar el dominio a cualquier año representable con un entero de 32 bits. Puede contener operaciones auxiliares, que por su posibilidad de romper y/o complicar innecesariamente el tipo, no se quieren exponer en la parte pública.

- `int diasDesde1AD()` facilitaría mucho la implementación interna de `distancia()`, pero se podría ver como una complicación innecesaria del tipo (ya que no tendría mucho sentido fuera de esta operación). Por tanto, se podría declarar como privada.
- internamente, se podrían usar dos operaciones, `suma(int delta)` y `reconcilia()`, para encontrar una fecha `delta` días en el futuro (o pasado) de la actual. Si `reconcilia()` corrige los errores antes de que un usuario pueda detectarlos, `suma()` puede, temporalmente, producir fechas inválidas. No obstante, `suma()` nunca debería entrar a formar parte de la parte pública del tipo, ya que "rompe" el tipo (sale fuera de su dominio).

⋆ La *parte pública* corresponde a las operaciones que se pueden usar externamente, y es completamente independiente (desde el punto de vista del usuario) de la parte privada. Las operaciones del tipo constituyen un *contrato* entre el autor del TAD y estos usuarios. En tanto en cuanto el autor del TAD se limite a modificar la parte privada (y las implementaciones de la parte pública), y los usuarios se limiten a usar las operaciones de la parte pública, el contrato seguirá vigente, y todo el código escrito contra el contrato seguirá funcionando.

⋆ Cada lenguaje de programación ofrece unas ciertas facilidades para la realización de este tipo de contratos entre implementador y usuario. Para que el usuario no acceda la parte privada, el lenguaje puede ofrecer

- **privacidad**: los detalles de implementación quedan completamente ocultos e inaccesibles para el usuario. Este es el caso, por ejemplo, de las `Interface` (interfaces) de Java. Es posible simularlo[3] con C++, pero no forma, estrictamente hablando, parte del lenguaje.
- **protección**: intentar acceder a partes privadas de TADs produce errores de compilación. Tanto C++ como Java tienen este tipo de controles, que se regulan mediante las notaciones `public:` ó `private:` en los propios TADs.
- **convención**: si no hay ningún otro mecanismo, sólo queda alcanzar un "acuerdo entre caballeros" entre usuario e implementador para no tocar aquello marcado

[3] mediante la técnica de "puntero a implementación", abreviada generalmente a "pimpl": un campo privado `void *_pimpl;` contiene y oculta toda la implementación. Sólo el desarrollador de la librería tiene que preocuparse de trabajar con el contenido de ese puntero para cumplir la especificación.

como privado. Una marca típica es comenzar identificadores privados con un '_', escribiendo, por ejemplo, '`_dia`'. Esto ya no es necesario en C++, pero sí lo era en C.

⋆ Con la versión actual de la parte privada, será muy fácil implementar `dia()` ó `mes()`, pero harán falta más cálculos en `distancia()`, `diaSemana()` y `diaAnyo()`. Podríamos haber elegido una implementación que almacenase sólo `_diaAnyo` y `_anyo`. Esto facilitaría calcular `distancia()` entre fechas del mismo año y haría muy sencillo implementar `diaAnyo()`, pero complicaría `dia()` ó `mes()`. En cualquier caso, el usuario del TAD no sería capaz de ver la diferencia, ya que las operaciones externas (la parte pública) no habría cambiado. En general, todo TAD contiene decisiones de implementación que deberían tener en cuenta de los casos de uso a los que va a ser sometido, y sus frecuencias relativas.

Diseño con TADs

⋆ El uso de TADs simplifica el desarrollo, ya que establece una diferenciación clara entre las partes importantes de un tipo (sus operaciones públicas) y aquellas que no lo son (la implementación interna). Por tanto, *disminuye la carga cognitiva* del desarrollador.

⋆ Al igual que la abstracción funcional, la abstracción de tipos es vital para la *reutilización*: un único TAD se puede usar múltiples veces (para eso se escribe). Además, es más fácil reutilizar un código que usa TADs: será más compacto y legible que un código equivalente que no los use, y en general los TADs están (y deben estar) mucho mejor documentados que otros tipos de código.

⋆ Además, facilita el *diseño descendente*: empezando por el máximo nivel de abstracción, ir refinando conceptos y tipos hasta alcanzar el nivel más bajo, el de implementación. Y también el diseño *ascendente*: empezar por las abstracciones más sencillas, e ir construyendo abstracciones más ambiciosas a partir de éstas, hasta llegar a la abstracción que resume todo el programa. En ambos casos, el programador puede concentrarse sobre lo importante (cómo se usan las cosas) y olvidar lo accesorio (cómo están implementadas internamente – o incluso, cómo se van a implementar, cuando llegue el momento de hacerlo).

  - Diseño descendente: Quiero programar un juego de ajedrez. Empiezo definiendo un `Tablero` con sus operaciones básicas: generar jugadas, realizar una jugada, ver si ha acabado la partida, guardar/cargar el tablero, etcétera. De camino, empiezo a definir lo que es una `Jugada` (algo que describe cómo se mueve una pieza sobre un tablero), una `Posicion` (un lugar en un tablero) y una `Pieza` (hay varios tipos; cada una tiene Jugadas distintas). Esta metodología permite decidir qué operaciones son importantes, pero no permite hacer pruebas hasta que todo está bastante avanzado.
  - Diseño ascendente: Quiero programar un juego de ajedrez. Empiezo definiendo una `Posicion` y una `Jugada` (que tiene posiciones inicial y final). Añado varios tipos de `Pieza`, cada una de las cuales genera jugadas distintas. Finalmente, junto todo en un `Tablero`. Esta metodología permite ir haciendo pruebas a medida que se van implementando los tipos; pero dificulta la correcta elección de las operaciones a implementar, ya que se tarda un tiempo en adquirir una visión general.

⋆ En la práctica, se suele usar una combinación de ambos métodos de diseño; el método *top-down* (descendente) para las especificaciones iniciales de cada tipo, y el *bottom-up* (ascendente) para la implementación real y pruebas, convergiendo en la estrategia *meet-in-the-middle* (fuera-a-dentro).

TADs y módulos

⋆ Todos los lenguajes incluyen el concepto de *módulo* - un conjunto de datos y operaciones fuertemente relacionadas que, externamente, se pueden ver como una unidad, y a las cuales se accede mediante una interfaz bien definida. El concepto de módulo incluye de forma natural a los TADs, y por tanto, en general, los TADs se deben implementar como módulos.

⋆ En C++, esto implica la declaración de una clase en un fichero .h (por ejemplo, `fecha.h`), y la definición de su implementación en un fichero .cpp asociado (por ejemplo, `fecha.cpp`):

```
// includes imprescindibles para que compile el .h
#include <libreria_sistema>
#include "modulo_propio.h"

// protección contra inclusión múltiple
#ifndef _FECHA_H_
#define _FECHA_H_

// ... declaración de la clase Fecha aqui ...

// fin de la protección contra inclusión múltiple
#endif
```

fecha.h

```
#include "fecha.h"

// includes adicionales para que compile el .cpp
#include <libreria_sistema>
#include "modulo_propio.h"

// ... definicion de Fecha aqui ...
```

fecha.cpp

⋆ En general, la documentación de un módulo C++ se escribe en su .h. No es recomendable duplicar la documentación del .h en el .cpp, aunque sigue siendo importante mantener comentarios con versiones reducidas de las cabeceras de las funciones, como separación visual y para añadir comentarios "de implementación" dirigidos a quienes quieran mantener o consultar esta implementación.

⋆ En un módulo C++ más general, el .h podría contener más de una declaración de clase, o mezclar declaraciones de tipos con prototipos de funciones.

⋆ El diseño modular y la definición de TADs está fuertemente relacionada on el diseño orientado a objetos (OO). En la metodología OO, el programa entero se estructura

en función de sus tipos de datos (llamados *objetos*), que pasan así al primer plano. Por el contrario, en un diseño imperativo tradicional, el énfasis está en las *acciones*.

⋆ C++ soporta ambos tipos de paradigmas, tanto OO como imperativo, y es perfectamente posible (y frecuente) mezclar ambos. Un módulo OO estará caracterizado por clases (objetos) que contienen acciones, mientras que un módulo imperativo contendrá sólo funciones que manipulan tipos.

TADs y estructuras de datos

⋆ Una **estructura de datos** es una *implementación* de un TAD relacionado con el acceso y mantenimiento de *colecciones de datos*. Algunos de estos TADs son de uso tan extendido y frecuente, que es de esperar que en cualquier lenguaje de programación de importancia exista una implementación equivalente. Por ejemplo, las librerías estándares de C++, Java ó Delphi soportan, todas ellos, árboles de búsqueda ó tablas hash. También hay estructuras que sólo se usan en dominios reducidos. Por ejemplo, ninguna de estas librerías estándares soporta el TAD *quadtree*, muy popular en sistemas de información geográfica, y que permite la búsqueda del punto más cercano a otro dado en tiempo logarítmico.

⋆ Como su nombre indica, las estructuras de datos se limitan sólamente a estructuras para acceder y mantener datos – de forma que, aunque todos los lenguajes proporcionan soporte para fechas, ficheros, o entrada/salida con formato, ninguno de los TADs correspondientes recibe el nombre de "estructura de datos".

⋆ Las estructuras de datos describen más bien una *estrategia* de implementación para un TAD que una implementación en particular. Hay muchas formas de implementar un árbol binario de búsqueda en C++ – pero más allá de los detalles, todas ellas (incluyendo la que viene en la librería estándar) serán ejemplos particulares de la estructura "árbol de búsqueda". La importancia de estudiar estructuras de datos no está en los detalles de su implementación (aunque sean buenos ejemplos de programación), sino en sus *características generales*, que es imprescindible conocer si se quiere poder tomar decisiones acerca de cuándo preferir una estructura a otra.

2. Especificación e implementación de TADs

⋆ Para la especificación de TADs usaremos notación algebraica. Es decir, en lugar de definir los conjuntos de estados que se cumplen en la precondición y postcondición, usaremos ecuaciones algebraicas para dar igualdades que se deberán cumplir a cuando se usan las operaciones del mismo.

⋆ Por ejemplo, vamos a especificar e implementar el TAD "booleano":

```
#include <ostream>
#ifndef _BOOLEANO_H_
#define _BOOLEANO_H_
class Bool {
public:
    // Generadores
    static Bool Falso, Cierto;
    // Falso.bNot() = Cierto; Cierto.bNot() = Falso
```

```
    Bool bNot() const;
    // Cierto.bAnd(x) = x; Falso.and(x) = Falso
    Bool bAnd(Bool &otro) const;
    // Cierto.bOr(x) = Cierto; Falso.or(x) = x
    Bool bOr(Bool &otro) const;
private:
    // tipo interno
    char _b;
    // constructor por defecto: privado
    Bool(char b): _b(b) {}

// amigos: permiten a una funcion externa acceder a privados
    friend std::ostream &operator<<(
        std::ostream &out, const Bool b);
};
#endif
```

booleano.h

```
#include "booleano.h"
#include <iostream>

using namespace std;
Bool Bool::Falso(0);
Bool Bool::Cierto(1);
Bool Bool::bNot() const { return _b ? Falso : Cierto; }
Bool Bool::bAnd(Bool &o) const { return _b ? o : Falso; }
Bool Bool::bOr(Bool &o) const { return _b ? Cierto : o; }

// op. externo sobrecargado (nuevos args para mismo nombre)
//   declarado amigo en la clase (tiene acceso a _b)
//   'cout << b;' se traduce a 'operator<<(cout, b)'
std::ostream &operator<<(std::ostream &out, const Bool b) {
    out << (b._b ? "1" : "0");
    return out;
}
```

booleano.cpp

⋆ NOTA: En las ecuaciones usadas para definir las operaciones, consideraremos equivalente e intercambiable la notación "orientada a objetos":

objeto.operación(operandos) $\rightsquigarrow \mathsf{Falso.or}(\mathsf{x}) = \mathsf{x}$

y la notación clásica:

operación(objeto-y-operandos) $\rightsquigarrow \mathsf{or}(\mathsf{Falso}, \mathsf{x}) = \mathsf{x}$

prefiriendo la clásica para razonamientos matemáticos, y la orientada a objetos para código.

⋆ ¿Hasta dónde especificar? Solamente hasta donde sea imprescindible, y no más. Si es posible decir lo mismo con menos palabras, y ambas formulaciones son igualmente claras, se debe preferir la más corta.

⋆ Las operaciones se pueden diferenciar en *generadoras*, *modificadoras* y *observadoras*. Nótese que la frontera entre generadoras y modificadoras es algo arbitraria, como veremos más adelante.

Generadora Crea una nueva instancia del tipo (puede o no partir de otras instancias anteriores). Usando operaciones generadoras, es posible construir todos los valores posibles del tipo.

Modificadora Igual que las generadoras, pero no pertenece al "conjunto mínimo" que permite construir los valores del tipo.

Observadora Permite acceder a aspectos del tipo principal, codificados en otros tipos - pero *no modifica* el tipo al que observa (y, por tanto, su declaración lleva el calificativo `const`). Por ejemplo, conversión a cadena, obtención del día-del-mes de una `Fecha` como entero, etcétera.

Además, frecuentemente se puede elegir entre suministrar una misma operación como modificadora o como observadora. Por ejemplo, en un TAD `Fecha`, podríamos elegir entre suministrar una operación `suma(int dias)` que modifique la fecha actual sumándola `delta` días (modificadora), o que devuelva una *nueva* fecha `delta` días en el futuro (observadora).

⋆ Se denomina *términos generados* a los términos del tipo que se pueden construir exclusivamente usando operaciones generadoras. Si el conjunto de términos generados no incluye todos los posibles términos del tipo, la especificación es incompleta. Por el contrario, si todo término puede ser reducido a un término generado, entonces se cumplirá una de las condiciones imprescindibles para que el tipo esté bien construido.

⋆ Ejemplo: términos generados del tipo `Bool`:

$$\mathsf{TG_{Bool}} = \{\mathsf{Falso}, \mathsf{Cierto}\} \equiv \mathsf{DOM_{Bool}}$$

⋆ Es posible, para un mismo tipo, elegir distintos subconjuntos de operaciones como "generadoras" ó "modificadoras". Por ejemplo, dado un tipo `Nat` (de los números naturales con el 0), estos dos conjuntos de formulaciones serían equivalentes:

- `Cero` y `sucesor(Nat n)` (donde, para $\mathsf{n} \in \mathsf{Nat}$, $\mathsf{sucesor}(\mathsf{n}) = \mathsf{n}+1$) – con ellas puedo formar todos los números naturales; no necesito incluir `suma(...)`:

  $$\mathsf{TG_{Nat}} = \{\mathsf{Cero}\} \cup \{\mathsf{sucesor}^{\mathsf{n}}(\mathsf{Cero}) \mid \mathsf{n} > 0\} \equiv \mathsf{Dominio_{Nat}}$$

  Esto se puede demostrar por inducción:

  $$\begin{aligned} \text{Base}: &\quad 0+1 = \mathsf{sucesor}(\mathsf{Cero}) = 1 \\ \text{Inducción}: &\quad \mathsf{n}+1 = \mathsf{sucesor}(\mathsf{n}),\ \text{luego } \mathsf{TG_T} \supset \mathbb{N} \end{aligned}$$

- `Cero`, `Uno` y `suma(Nat a, Nat b)` – con ellas puedo formar todos los naturales; ya no necesito incluir `sucesor(...)`.

  $$\mathsf{TG_{Nat}} = \{\mathsf{Cero}, \mathsf{Uno}\} \cup \{\mathsf{suma}(\mathsf{a}, \mathsf{b}) \mid \mathsf{a}, \mathsf{b} \in \mathsf{TG_{Nat}}\} \equiv \mathsf{Dominio_{Nat}}$$

⋆ Es legal, y frecuente, que un TAD use a otro. En estos casos, el TAD externo debe evitar añadir nuevas ecuaciones exclusivamente sobre el interno, o construir términos del tipo interno fuera del dominio correspondiente. Un ejemplo de buen uso de TAD interno sería, dentro de un `Rectangulo`, el uso de `Punto`:

```
#include "Punto.h" // usa Punto
/*
 * Un Rectangulo en 2D alineado con los ejes.
 * Dominio: si vacio, entonces r.ancho() = r.alto()
 */
class Rectangulo {
private:
    ...
public:
    // Constructor; Generador
    Rectangulo(Punto origen, float alto, float ancho);

    // Constructor;
    // si p = Punto(min(uno.x, otro.x), min(uno.y, otro.y))
    // Rectangulo(uno, otro)
    //    = Rectangulo(p, |uno.x-otro.x|, |uno.y-otro.y|)
    Rectangulo(Punto uno, Punto otro);

    // Observadoras
    const Punto &origen() const;
    float alto() const;
    float ancho() const;
    // igualdad
    //   r1.igual(r2) = (r1.vacio() and r2.vacio())
    //                or (r1.origen().igual(r2.origen())
    //                   and r1.ancho() = r2.ancho()
    //                   and r1.alto() = r2.alto())
    bool igual(const Rectangulo &r) const;
    // verifica si esta vacio, cumpliendo
    //   r.vacio() = (r.ancho() > 0) = (r.alto() > 0)
    bool esVacio() const;
    // calcula si un punto cae dentro o fuera de otro
    //    r.dentro(p) =
    //           (origen.x <= p.x < origen.x + ancho)
    //      and  (origen.y <= p.y < origen.y + alto)
    //    r.dentro(p) = not r.vacio()
    bool dentro(const Punto &p) const;
    // calcula area, cumpliendo
    //    r.area() = r.alto() * r.ancho()
    float area() const;
    // calcula interseccion; si p es un Punto,
    //    (r1.dentro(p) and r2.dentro(p)) = r1.interseccion(r2).dentro(p)
    Rectangulo interseccion(const Rectangulo &r) const;
    // indica si r1 contiene r2; si p es un Punto,
    //    r1.dentro(r2) and r2.dentro(p) = r1.interseccion(r2).dentro(p)
    bool dentro(const Rectangulo &r) const;
};
```

⋆ Es frecuente también *enriquecer* tipos: partiendo de un tipo, añadir operaciones y/o extender su dominio. En lenguajes que soportan orientación a objetos, el uso de de herencia permite hacer esto de una forma compacta y explícita:

```
class Hora {
    // Dominio: horas() >=0, minutos() >=0 y <60, segundos() >=0 y <60
public:
```

```
    // Generadores
    Hora(unsigned int h, unsigned int m, unsigned int s);
    // Mutadores
    //   a.suma(s).segundos() = a.segundos() + s%60
    //   a.suma(s).minutos() = a.minutos() + (s/60)%60
    //   a.suma(s).horas() = a.horas() + (s/60)/60
    void suma(unsigned int s);
    //   a.suma(h, m, s) = a.suma(60*60*h + 60*m + s)
    void suma(unsigned int h, unsigned int m, unsigned int s);
    // Observadores
    //   segundos
    int segundos() const;
    //   minutos
    int minutos() const;
    //   horas
    int horas() const;
protected:      // permite acceder a esta zona a sus subclases
    // representacion protegida, usando total-de-segundos
    long _s;
    // permite a una 'funcion amiga' acceder a la parte privada
    friend std::ostream &operator<<(std::ostream &out, const Hora &h);
};
```

hora.h

```
#include "hora.h"
class IHora : public Hora { // extiende hora
    // Dominio: el de hora, pero ahora con signo
public:
    // Generadores (ahora con cualquier signo)
    IHora(int h, int m, int s);
    // Mutadores
    //   a.suma(b) = a.suma(b.horas(), b.minutos(), b.segundos)
    void suma(const IHora &otra);
    //   (mismas ecuaciones; pero ahora aceptan numeros negativos)
    void suma(int s);
    void suma(int h, int m, int s);
    //   a.resta(b) = a.suma(-b.horas(), -b.minutos(), -b.segundos)
    void resta(const IHora &otra);
    // Observadores
    //   mantiene definiciones y ecuaciones de los de la clase padre
    //   si negativa: 'true'; si no: 'false'
    bool negativa() const;
private:
    bool _negativa;

// el uso de friend permite a una f. externa accedera a partes privadas
    friend std::ostream &operator<<(
        std::ostream &out, const IHora &h);
};
```

ihora.h

En este ejemplo también se puede ver el uso de ecuaciones condicionales (con cláusulas "si"), el uso de `protected:` en lugar de `private:` para marcar código como accesible por subclases, una ampliación de dominio, y de operaciones soportadas. En

general, no se pedirá manejo de herencia en este curso – pero se debe saber que existe y está disponible en C++.

2.1. TADs genéricos

⋆ Un TAD genérico es aquel en el que uno o más de los tipos que se usan se dejan sin identificar, permitiendo usar las mismas operaciones y estructuras con distintos tipos concretos. Por ejemplo, en un TAD `Pila`, no debería haber grandes diferencias entre apilar enteros, punteros arbitrarios, o `IHoras` de las anteriormente descritas. Hay varias formas de conseguir esta genericidad, entre ellas:

- plantillas: usado en C++; permite declarar tipos como "de plantilla" (*templates*), que se resuelven en tiempo de compilación para producir todas las variantes concretas que se usan realmente. Mantienen un tipado fuerte y transparente al programador.
  Ejemplo:

```
// requiere <stack>, un TAD generico 'pila' via plantillas
    stack<bool> s;
    s.push(true);      // s.push("ey") falla
    bool b = s.top(); // b == true
```

- herencia: muy usado en Java (que a partir de la versión 5 empezó a implementar algo similar a las platillas de C++), y disponible en cualquier lenguaje con soporte OO. Requiere que todos los tipos concretos usados desciendan de un tipo base que implemente las operaciones básicas que se le van a pedir. Tienen mayor coste que los templates, y debilita el sistema de tipos (una vez reducido a una superclase, puede ser difícil volver a la subclase concreta).

```
// requiere java.util.Stack un TAD generico 'pila' via herencia
    Stack s = new Stack(); // conteniene Object
    s.push(Boolean.FALSE); // s.top() devuelve Object
    s.push("ey");          // funciona sin problemas
// Java SE 5 en adelante: versión semi-generica
    Stack<Boolean> t = new Stack<Boolean>();
    s.push(Boolean.FALSE); // s.top() devuelve Boolean
    s.push("ey");          // error de compilacion
```

- lenguajes dinámicos: JavaScript o Python son lenguajes que permiten a los tipos adquirir o cambiar sus operaciones en tiempo de ejecución. En estos casos, basta con que los objetos de los tipos introducidos soporten las operaciones requeridas en tiempo de ejecución (pero se pierde la comprobación de tipos en compilación).

⋆ En C++, se pueden definir TADs genéricos usando la sintaxis

`template <class` T_1**`, ... class`** T_n**`>`** *contexto*

y refiriéndose a los T_i igual que se haría con cualquier otro tipo a partir de este momento. Generalmente se escogen mayúsculas que hacen referencia a su uso; por ejemplo, para un tipo cualquiera se usaría `T`, para un elemento `E`; etcétera.

Ejemplo de TAD genérico pareja:

```
// en el .h
template <class A, class B>
```

```
class Pareja {
    // una pareja inmutable generica
    A _a; B _b;
public:
    // Generador (un constructor sencillo)
    Pareja(A a, B b) { _a=a; _b=b; } // cuerpos en el .h (!)
    // Observadores
    //   Pareja(a, b).primero() = a
    A primero() const { return _a; }
    //   Pareja(a, b).segundo() = b
    B segundo() const { return _b; }
};

// en el main.cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    Pareja<int,string> p(4, "hola");
    cout << p.primero() << " " << p.segundo() << "\n";
    return 0;
}
```

NOTA: en C++ es legal definir los cuerpos de funciones en el .h en lugar de en el prototipo (tal y como se hace en el ejemplo). En el caso de TADs genéricos, esto es **obligatorio** (en caso contrario se producirán errores de enlazado), aunque para implementaciones más grandes es preferible usar esta versión alternativa, que deja el tipo más despejado, a costa de repetir la declaración de los tipos de la plantilla para cada contexto en el que se usa:

```
// en el .h
    ...
    // Generador (un constructor sencillo)
    Pareja(A a, B b);
    // Observadores
    A primero() const;
    B segundo() const;
};
template <class A, class B>
Pareja<A,B>::Pareja(A a, B b) { _a = a; _b = b; }
template <class A, class B>
A Pareja<A,B>::primero() const { return _a; }
template <class A, class B>
B Pareja<A,B>::segundo() const { return _b; }
```

2.2. Implementación, parcialidad y errores

⋆ Es vital hacer bien la distinción entre un **TAD** y una **implementación** concreta del mismo. Un TAD es, por definición, una abstracción. Al elegir una implementación concreta, siempre se introducen limitaciones - que podrían ser distintos en otra implementación distinta. Por ejemplo, las implementaciones típicas de los enteros tienen sólo 32 bits (pero hay otras con 64, y otras que tienen longitud limitada sólo por la memoria disponible, pero que son mucho más lentas). Es interesante ver que *todo lo*

que sea cierto para el TAD lo será para todas sus implementaciones. No obstante, los límites de una implementación no tienen porqué afectar a otra del mismo TAD.

⋆ Es posible descomponer la implementación de un TAD en tres pasos:

1. Elegir un *tipo representante*, es decir, los tipos de implementación concretos que se va a usar para representar el TAD; y las condiciones que deben cumplir estos tipos para considerarse como válidos (llamadas ***invariantes de representación***).
   - El TAD `complejo` se puede representar con una pareja de `float`, uno para la parte entera y otro para la imaginaria - pero esta es sólo una de las posibilidades; el TAD subyacente no cambiaría si usásemos dos `double` y una representación polar (ángulo-magnitud), o un `quad` para la parte entera y un `float` para la imaginaria, o cualquier otra representación.
   - Un TAD `Fecha` se puede representar con tres `int` que almacenen los días, meses y años. Las reglas que limitan sus valores válidos para una fecha determinada constituirían los *invariantes de representación* de esta implementación.
2. Elegir una *función de abstracción*, que permita pasar de un representante válido a uno de los términos generados del TAD. Es decir, elegir cómo se van a interpretar los tipos representantes del paso anterior.
   - En una representación *(real, imaginario)* de un número complejo, podríamos usar estos valores directamente. En una representación polar, esta función especificaría cómo convertir de *(ángulo, magnitud)* a *(real, imaginario)*.
3. Implementar las *operaciones*, especificando para cada una de ellas condiciones *pre* y *post* que, en todos los casos, deberán mantener todos las invariantes de representación (posiblemente con restricciones adicionales debidas al tipo representante elegido)
   - Por ejemplo, un valor entero representado mediante un `int` de 32 bits sólo puede tomar 2^{32} valores distintos - lo cual impone limitaciones al dominio de implementación de un TAD que use `int`s de 32 bits como tipos representantes.

Figura 1: La función de abstracción (F.A.), la vista de usuario (abstracta), y la vista de implementador (concreta). La invariante de representación delimita un subconjunto de todos valores posibles del tipo representante como *válidos*, y la función de abstracción los pone en correspondencia con términos del TAD. Es frecuente que varios valores del tipo representante puedan describir al mismo término del TAD – la función de equivalencia (F.E.) permite encontrar estos casos.

⋆ La **invariante de representación** (ver figura 1) consiste en el *conjunto de condiciones que se tienen que cumplir para que una representación se considere como válida.* Ejemplos:

- `Hora` (usando `int _horas`, `int _minutos`, `int _segundos`):
  $0 \leq \texttt{\_horas} \wedge 0 \leq \texttt{\_minutos} < 60 \wedge 0 \leq \texttt{\_segundos} < 60$

- `Hora` (usando sólo `int _segundos`):
  $0 \leq$ _segundos
- `Rectangulo` (usando `Punto _origen`, `float _ancho`, `float _alto`)
  $0 \leq$ _ancho $\wedge\ 0 \leq$ _alto
- `Rectangulo` (usando `Punto _origen`, `Punto _extremo`)
  _origen.x $\leq$ _extremo.x $\wedge$ _origen.y $\leq$ _extremo.y

⋆ Otra relación importante es la **relación de equivalencia**, que indica cuándo dos valores del tipo implementador representan el *mismo valor* abstracto. Usando el tipo `Rectangulo`, todos los rectángulos vacíos son idénticos, tengan los orígenes que tengan. En la Figura 4 se puede ver un ejemplo de pilas equivalentes. La relación de equivalencia está ilustrada en la Figura 1. Esta relación (o función) es abstracta (es decir, existe independientemente de que se implemente o no). No obstante, para hacer pruebas, es muy útil implementarla. En C++, la forma de hacerlo es sobrecargando el operador '`==`':

```
class Rectangulo {
    ...
    // permite ver equivalencia de rectangulos mediante r1 == r2
    bool operator==(const Rectangulo &r) const {
        return (esVacio() && r.esVacio())
            || (_alto == r._alto && _ancho == r._ancho
                && _origen == r._origen);
    };
    ...
};
```

⋆ Un TAD puede incluir aspectos *parciales*, es decir, no totalmente definidos. Marcando estas operaciones como `parcial`, se anuncia que tienen precondiciones adicionales a las normales - y que, si no se cumplen estas precondiciones, el resultado se debe tratar como excepcional. A la hora de razonar sobre el TAD, sólo se pueden hacer razonamientos sobre aspectos definidos.

Es frecuente que las limitaciones impuestas por recursos finitos (memoria, ficheros) o debidas a los tipos de representante seleccionados (límites de `int` o `float`, vectores de tamaño fijo, etcétera) se traduzcan en precondiciones adicionales.

Ciertas operaciones son por definición erróneas (intentar acceder más allá del final de un vector, o intentar dividir por cero). Cualquier operación que conlleve estas posibilidades debe ir marcada como `parcial`.

⋆ Ejemplo: una pila. En el TAD `Pila`, es posible añadir elementos (operación *push* o *apilar*), y extraerlos en el orden inverso al que fueron insertados (operación *pop* o *desapilar*). Es posible consultar el último elemento apilado (operación *cima* o *top*), y saber si la pila está vacía (operación *isEmpty* o *esVacia*). Las pilas empiezan estando vacías, y es un error intentar extraer elementos cuando están vacías.

```
template <class E>
class Pila {
    ... // aquí iría el tipo representante
public:
    // Generadores
    //   devuelve una pila vacía
    //   Pila().esVacia() = true
```

```
    Pila();
    //   inserta un elemento; si p ∈ Pila ∧ e ∈ E,
    //   p.apila(e).esVacia() = false
    //   p.apila(e).desapila() = p
    //   p.apila(e).cima() = e
    Pila &apila(const E &e);
    //   extrae un elemento
    //   parcial: si p.esVacia(), error
    Pila &desapila();
    // Observadores
    //   devuelve el elemento de la cima
    //   parcial: si p.esVacia(), error
    const E &cima() const;
    //   si vacia, devuelve 'true'
    bool esVacia() const;
};
```

1. Tipo representante: Usaremos un vector de tamaño fijo, con el índice del elemento "cima" indicado mediante un entero:

   ```
   static const int MAX = 100;
   static const int VACIA = -1;
   int _posCima;
   E _elementos[MAX];
   ```

   Consideraremos como válidos $\texttt{VACIA} \leq \texttt{\_posCima} < \texttt{MAX}$ (siempre se deben usar constantes[4] para los valores límite); y por tanto, esta será nuestra invariante de representación. El valor inicial de `_posCima`, -1, se usará para señalizar "pila vacía". Debido a la implementación con un límite duro de tamaño, `apila()` pasa a ser parcial:

   ```
   // parcial: si p._posCima = MAX-1, error
   Pila &apila(const E &e);
   ```

Figura 2: Pila tras insertar el elemento 7

Figura 3: Pila tras insertar el elemento 4

2. Función de abstracción: Para pasar de un valor de `_elementos` y `_posCima` a una pila abstracta, entenderemos que los elementos con índice 0 a `_posCima` (inclusive) forman parte de la pila, y que están en el mismo orden en el que fueron apilados.

   Nótese que puede haber múltiples valores del tipo representante (en este caso, múltiples vectores de `_elementos`) que se refieran a la misma pila abstracta: a partir de la cima, el contenido del vector `_elementos` es completamente indiferente desde el punto de vista del TAD.

[4] El uso de `static const` declara `MAX` y `VACIA` como constantes (`const`) para esta implementación del TAD, definidas una única vez para cualquier número de pilas (`static`), en lugar de una vez para cada pila individual (ausencia de `static`). Es preferible usar constantes estáticas de clase que `#define`s.

Figura 4: Pilas abstractas equivalentes

3. Especificación de pre- y post-condiciones para cada una de las operaciones, e implementaciones definitivas:

   - Pila()

```
// func Pila() dev Pila p
// {P: }
Pila() { _posCima = VACIA; }
// {Q: p Pila y p.esVacia()}
```

   - apila(const E &e)

```
// func apila(in E e) dev Pila p;
// {P: p Pila y p._posCima < MAX-1 y q = p}
Pila &apila(const E &e) {
    if (_posCima == MAX-1) throw "demasiados elementos";
    _elementos[_posCima + 1] = e;
    _posCima ++;
    return *this;
}
// {Q: p Pila y
//   p.esVacia() = false y p.cima() = e y p.desapila() = q}
```

     Donde, en caso de error, se usa la sentencia "throw" para *lanzar* una excepción y salir de la función (tiene efectos similares, pero más fuertes, que un `return`); en el punto siguiente se muestran más estrategias posibles, y se recomienda que, en lugar de lanzar excepciones de tipo `const char *`, se usen otros tipos de dato que permitan al programa interpretar fácilmente de qué excepción se trata.
     El hecho de devolver `*this` (una referencia a la misma pila[5] , después de modificarla) permite encadenar operaciones, lo cual resulta en una forma muy compacta y legible de escribir código: `p.apila(1).apila(2).apila(3);` es legal y apila, sucesivamente, los enteros 1, 2 y 3. (las implementaciones de TADs que llevan esto a su extremo se denominan "fluidas").

   - desapila()

```
// func desapila() dev Pila p;
// {P: p Pila y no p.esVacia() y p = q}
Pila &desapila() {
    if (_posCima == VACIA) throw "sin elementos";
    _posCima --;
    return *this;
}
// {Q: p Pila y p.apila(q.cima()) = q}
```

   - cima()

```
// func cima() dev E e;
// {P: p Pila y no p.esVacia()}
```

[5] en el interior de una función no-estática de clase, `this` es un puntero al argumento implícito de la función: el objeto de la clase sobre el que se está trabajando. Por ejemplo, llamando a `p.apila(5)`, `this` contiene un puntero a `p`. Escribir `*this` desreferencia este puntero, y devuelve `p`.

```
const E &cima() const {
    if (_posCima == VACIA) throw "sin elementos";
    return _elementos[_posCima];
}
// {Q: p, q Pila y p = q.apila(e)}
```

- esVacia()

```
// func esVacia() dev bool b;
// {P: p Pila}
bool esVacia() const {
    return _posCima == VACIA;
}
// {Q: p, q Pila y ((b y p = q.apila(e)) o
//                  (no-b y no existe p con p = q.apila(e)))}
```

⋆ Hasta ahora, hemos ignorado el manejo de los errores que se pueden producir en un programa (ya sea una función independiente o una función que forma parte de un TAD). Hay varias estrategias posibles para tratar errores:

- Devolver (vía `return`) un "valor de error".
  - Requiere que haya un valor de error disponible (típicamente NULL ó -1) que no puede ser confundido con una respuesta de no-error; cuando no lo hay, lo común es devolver un booleano indicando el estado de error y pasando el antiguo valor devuelto a un argumento por referencia.
  - Requiere que el código que llama a la función verifique si ha habido o no error mediante algún tipo de condicional, y lleva a código difícil de leer.

  No obstante, este patrón se usa en muchos sitios; por ejemplo, la API C++ de Windows, la de Linux, o la librería estándar de C. En todas estas librerías se permite recabar más información sobre el error con llamadas adicionales, lo cual implica guardar información sobre el último error que se produjo por si alguien la pide luego[6].
- Lanzar una excepción.
  - Requiere soporte del lenguaje de programación (C ó Pascal no las soportan, C++ ó Delphi sí). En general, todos los lenguajes con orientación a objetos soportan excepciones.
  - Requiere que el código que llama a la función decida si quiere manejar la excepción (lanzada con `throw`), usando una sentencia `catch` apropiada.

  El código resultante es más limpio (requiere menos condicionales), y la excepción puede contener toda la información disponible sobre cómo se produjo - no hace falta que las librerías que lanzan excepciones mantengan siempre el último error. Esta es la estrategia que se sigue en las librerías orientadas a objetos de, por ejemplo, la API .NET de Windows ó la librería estándar de Java.
- Mostrar un mensaje por pantalla y devolver cualquier valor. Esta estrategia sólo es admisible cuando se está depurando, la librería es muy pequeña, y somos

[6] Se suele usar para ello una variable global (en el caso de Unix/Linux, `errno`). El uso de esta variable permite adoptar una convención adicional: en algunos casos, en lugar de devolver un "valor de error", se espera que se consulte el valor de `errno` para saber si hubo o no error. Esto requiere mucha disciplina por parte del programador para consultar el valor de `errno` tras cada llamada, antes de que la siguiente lo sobreescriba con un nuevo valor – lo cual puede ser imposible en presencia de programación concurrente.

su único usuario; e incluso entonces, es poco elegante. Su única ventaja es que resulta fácil de implementar.

- Exigir una función adicional, pasada como parámetro (en muchos lenguajes, C++ inclusive, es posible pasar funciones como parámetros), a la que llamar en caso de error. La mayoría de las librerías evitan este sistema salvo en casos muy específicos.

⋆ Ejemplo: Lanzando y capturando excepciones en C++

```
#include <iostream>
int divide(int a, int b) {
    if (b==0) {
        throw "imposible dividir por cero"; // tipo 'const char *'
    }
    return a/b;
}

using namespace std;
int main() {
    try {
        cout << divide(12,3) << "\n";
        cout << divide(1, 0) << "\n";
    } catch (const char * &s) {             // ref. al tipo lanzado
        cout << "ERROR: " << s << "\n";
    }
    return 0;
}
```

⋆ Las excepciones lanzadas pueden ser de cualquier tipo. No obstante, **se deben usar clases específicas**, tanto por legibilidad como por la posibilidad de usarlas para suministrar más información dentro de las propias excepciones. En el siguiente ejemplo, usamos una clase base `Excepcion`, con subclases `DivCero` y `Desbordamiento`:

```
#include <iostream>
#include <string>
#include <climits>

class Excepcion {
    const std::string _causa;
public:
    Excepcion(std::string causa) : _causa(causa) {};
    const std::string &causa() const { return _causa; };
};
class DivCero : public Excepcion {
    public: DivCero(std::string causa) : Excepcion(causa) {};
};
class Desbordamiento : public Excepcion {
    public: Desbordamiento(std::string causa) : Excepcion(causa) {};
};

int suma(int a, int b) {
    if ((a>0 && b>0 && a>INT_MAX-b) || (a<0 && b<0 && a<INT_MIN-b)) {
        throw Desbordamiento("excede limites en suma");
    }
    return a+b;
```

```
}

int divide(int a, int b) {
    if (b==0) {
        throw DivCero("en division");
    }
    return a/b;
}

using namespace std;
int main() {
    try {
        cout << divide(12,3) << "\n";
        cout << suma(INT_MAX, -10) << "\n";
        cout << divide(1, 0) << "\n";        // lanza excepcion
        cout << suma(INT_MAX,  10) << "\n";  // esto nunca se evalua
    } catch (DivCero &dc) {          // trata DivCero (si se produce)
        cout << "Division por cero: " << dc.causa() << "\n";
    } catch (Desbordamiento &db) {   // trata Desbordamiento (si hay)
        cout << "Desbordamiento: " << db.causa() << "\n";
    } catch (...) {                  // trata cualquier otra excepcion
        cout << "Excepcion distinta de las anteriores!\n";
    }
    return 0;
}
```

⋆ Las operaciones de un TAD pueden devolver valores o referencias, tanto como valor de la función como a través de argumentos. A continuación proponemos una serie de **convenciones** a seguir en cuanto a los valores devueltos:

- Si la función no tiene un "valor devuelto" claro, como es el caso de las operaciones de mutación, se puede devolver una referencia al objeto (usando el mismo `*this` presente en los mutadores de `Pila`); los errores se indicarán lanzando excepciones, por lo que no es necesario reservar el valor de retorno para notificarlos. Esto permite escribir "código encadenado" y resulta en programas más compactos y legibles:

  ```
  // en Pila: apila un nuevo elemento
  Pila &apila(const E &e);
  // en un Poligono: inserta un nuevo punto
  Poligono &inserta(float x, float y);
  ```

- Es posible también devolver `void` o algún otro valor útil (por ejemplo, al insertar una palabra en un `Diccionario`, un `bool` que indique si hemos substituido una definición anterior o no). En general, dentro de un proyecto dado, se deben seguir las convenciones ya existentes. Por ejemplo, si el código circundante no suele devolver valores que permitan encadenamiento, podríamos haber escrito el fragmento anterior como

  ```
  // en Pila: apila un nuevo elemento
  void apila(const E &e);
  // en un Poligono: inserta un nuevo punto
  void inserta(float x, float y);
  ```

- En el caso de observadores, si *sólo se devuelve un valor*, el tipo de retorno de la función debe ser el del valor devuelto.

- Si el valor es un bool, un int, un puntero, o similarmente pequeño (y por tanto una referencia no ahorraría nada); o si el valor es más grande, pero no se va a guardar una copia en ninguna parte (y por tanto sería un error usar referencias), se deberá devolver el resultado de la forma tradicional, sin referencias.
- En caso contrario (valores grande de los que se guarda copia), es más eficiente devolver una referencia constante.

```
// en Pila: devuelve el elemento de la cima,
//    E puede ser grande (no se sabe), y hay una copia
const E &cima() const;
// en Pila: un bool no es grande, y no hay copia
bool esVacia() const;
// en Rectangulo: grande, y hay copia
const Punto &origen() const;
// en Rectangulo: grande, pero no hay copia
Rectangulo interseccion(const Rectangulo &r1) const;
```

- Si hay que devolver *más de un valor a la vez*, se debe devolver a través de argumentos "por referencia". El tipo de retorno de la función queda libre para otras opciones, como devolver `*this` para permitir encadenamiento de llamadas, o devolver códigos de error o un simple `void`.

```
// en un Poligono: devuelve el i-esimo punto en x e y
//    (este TAD quedaria mejor devolviendo Puntos)
Poligono &punto(int i, float &x, float &y) const;
// mejora de Poligono: usando una clase auxiliar Punto
const Punto &punto(int i) const;
```

2.3. implementaciones dinámicas y estáticas

⋆ Algunos TADs pueden llegar a requerir mucho espacio. Por ejemplo, el tamaño de una pila puede variar entre uno o dos elementos y cantidades astronómicas de los mismos, en función del uso concreto que se haga de ella. Cuando un TAD tiene estos requisitos, no tiene sentido asignarle un espacio fijo "en tiempo de compilación", y es mejor solicitar memoria al sistema operativo (SO) a medida que va haciendo falta. **La memoria obtenida en tiempo de ejecución mediante solicitudes al SO se denomina *dinámica*. La que no se ha obtenido así se denomina *estática*[7].**

⋆ La memoria dinámica se gestiona mediante punteros. Un puntero no es más que una dirección de memoria; pero en conexión con la memoria dinámica, se usa también como identificador de una reserva de memoria (que hay que devolver). En C++, la memoria se reserva y se libera mediante los operadores **`new`** y **`delete`**:

```
Pila<int> *p = new Pila<int>(); // una pila de enteros en memoria dinámica
p->apila(4).apila(2);           // apilo un par de elementos
p->desapila().cima();           // quito uno; despues, cima() = 4
delete p;                       // libero la pila
```

[7] El sistema operativo asigna, a cada proceso, un espacio fijo de pila de programa (o *stack*) de unos 8MB, que es de donde viene la memoria estática; la memoria dinámica (también denominada *heap* o "del montón") se obtiene del resto de la memoria libre del sistema, y suele ser de varios GB. Puedes probar a quedarte sin memoria estática escribiendo una recursión infinita: el mensaje resultante, "stack overflow", indica que tu pila ha desbordado su tamaño máximo prefijado.

Y sus variantes **new[]** y **delete[]**, que se usan para reservar y liberar vectores, respectivamente:

```
int cuantos = 1<<30;                // = 2^30 (<<X es el operador ·2^X)
int *dinamicos = new int[cuantos]; // reserva 1GB de memoria dinamica
...                                 // usa la memoria
delete[] dinamicos;                 // libera  1GB de memoria dinamica
```

Recordatorio sobre punteros:

```
int x = 10, y[] = {3, 8, 7};
Pareja<int, int> *par = new Pareja<int, int>(4, 2);
int *px = &x;     // &x: dirección de memoria de la variable x
int z = *px;      // *px: acceso a los datos apuntados por px == 10
int x = par->primero();  // = (*par).primero() = 4
int py = y;       // un vector es tambien un puntero a su comienzo
int w = *(py+1); // = 8 = py[1] = y[1]
```

⋆ Puedes saber cuánta memoria "directa" (es decir, sin contar la memoria apuntada mediante punteros) ocupa una estructura o clase C++ usando el operador **sizeof**. Así, dada la pila estática con 100 elementos declarada anteriormente, `sizeof(Pila<int>)` será `sizeof(_posCima) + sizeof(_elementos)` = 404 bytes (es decir, el tamaño ocupado por sus tipo representante: tanto `_posCima` como cada uno de los 100 `_elementos` ocupan 4 bytes, por ser de tipo `int`). De igual forma, `sizeof(Pila<char>)` habría sido 104 (ya que los `_elementos` de tipo `char` ocuparían sólo 1 byte cada uno). En una versión dinámica de esta pila, `_elementos` se habría reservado mediante un new (y por tanto sería de tipo puntero); en este caso, el tamaño de la pila sería siempre `sizeof(_posCima) + sizeof(E *)` = 12 (en sistemas operativos de 64 bits; u 8 si tu sistema operativo es de 32), independientemente del tamaño del vector y del número de elementos que contenga.

⋆ Errores comunes con punteros y memoria dinámica:

- **Usar un puntero sin haberle asignado memoria** (mediante un `new` / `new[]` previo). El puntero apuntará a una dirección al azar; el resultado queda indefinido, pero frecuentemente resultará en que la dirección estará fuera del espacio de direcciones del programa, ocasionando una salida inmediata.
- **No liberar un puntero tras haber acabado de usarlo** (mediante un `delete` / `delete[]`). La memoria seguirá reservada, aunque no se pueda acceder a ella. Esto se denomina "fuga de memoria", y al cabo de un tiempo (en función de la frecuencia y cantidad de memoria perdida), el sistema operativo podría quedarse sin memoria. En este momento empezarán a fallar las reservas de los programas; si falla una reserva por falta de memoria, se lanzará una excepción (de tipo `bad_alloc`) que hará que acabe tu programa.
- **Liberar varias veces el mismo puntero.** La librería estándar de C++ asume que sólo liberas lo que no está liberado, e intentar liberar cosas ya liberadas o que nunca reservaste en primer lugar conduce a comportamiento no definido (típicamente a errores de acceso).
- **Acceder a memoria ya liberada.** No hay ninguna garantía de que esa memoria siga estando disponible para el programa (si no lo está, se producirá error de acceso), o de que mantenga su contenido original.

- **Intentar escribir y leer el valor de un puntero** (por ejemplo, a un archivo) **fuera de una ejecución concreta.** Un puntero sólo sirve para apuntar a una dirección de memoria. Cuando esa dirección deja de contener lo que contenía, el valor del puntero ya no sirve para nada. Por ejemplo, si en un momento un programa tiene, en la dirección `0xA151E0FF`, el comienzo de un array dinámico, no hay ninguna garantía de que esa misma dirección vaya a contener nada interesante (o incluso accesible) en la siguiente ejecución[8].

⋆ Para evitar estos errores, se recomiendan las siguientes prácticas:

- Usa memoria dinámica sólo cuando sea necesaria. No es "más elegante" usar memoria dinámica cuando no es necesaria; pero sí es más farragoso y resulta en más errores.
- Inicializa tus punteros nada más declararlos. Si no puedes inicializarlos inmediatamente, asígnales el valor NULL (= 0) para que esté garantizado que todos los accesos resultarán en un error inmediato.
- Nada más liberar un puntero, asígnale el valor NULL (= 0); **delete()** nunca libera punteros a NULL; de esta forma puedes evitar algunas liberaciones múltiples:

```
Pila *p = new Pila(); // asigno un valor en la misma declaracion
delete p;             // comprueba NULL antes de liberar
p = NULL;             // para evitar accesos/liberaciones futuros
```

Constructores y destructores

⋆ En cualquier implementación de TAD dinámico (los TADs en sí no son ni dinámicos ni estáticos; sólo sus implementaciones pueden serlo), habrá dos fragmentos de código especialmente importantes: la inicialización de una nueva instancia del TAD (donde, como parte de las inicializaciones, se hacen los `new`s) , y la liberación de recursos una vez se acaba de trabajar con una instancia dada (donde se hacen los `delete`s que falten). En C++, estos fragmentos reciben el nombre de **constructor** y **destructor**, respectivamente, y tienen una sintaxis algo distinta de las demás funciones:

- No devuelven nada (sólo pueden comunicar errores mediante excepciones).
- Sus nombres se forman a partir del nombre del TAD. Dado un TAD `X`, su constructor se llamará `X`, y su destructor `~X`.
- Si no se especifican, el compilador genera versiones "por defecto" de ambos, públicas, con 0 argumentos y totalmente vacías. Es equivalente escribir `class X {};` o escribir `class X { public: X() {} ~X() {} };`.

Veamos un ejemplo con una implementación con memoria dinámica de `Mapa`:

```
class Mapa {
    unsigned short *_celdas; // 2 bytes por celda
    int _ancho;
    int _alto;
```

[8] Es más, hay sistemas operativos que intentan, a propósito, hacer que sea muy difícil adivinar dónde acabarán las cosas (vía *address space randomization*); esto tiene ventajas frente a programas malintencionados.

```
public:
    // Generador - Constructor
    Mapa(int ancho, int alto) : _ancho(ancho), _alto(alto) {
        if (ancho<1||alto<1) throw "dimensiones mal";
        _celdas = new unsigned short[_alto*_ancho];
        for (int i=0; i<alto*ancho; i++) _celdas[i] = 0;
        std::cout << "construido: mapa de "
                  << _ancho << "x" << _alto << "\n";
    }
    // Generador - Mutador
    Mapa &celda(int i, int j, unsigned short v) {
        _celdas[i*_ancho + j] = v;
        return *this;
    }
    // Observador
    unsigned short celda(int i, int j) {
        return _celdas[i*_ancho + j];
    }
    // Destructor
    ~Mapa() {
        delete[] _celdas;
        std::cout << "destruyendo: mapa de "
                  << _ancho << "x" << _alto << "\n";
    }
};
int main() {
    Mapa *m0 = new Mapa(128, 128); // llama a constructor (m0)
    Mapa m1 = Mapa(2560, 1400);    // llama a constructor (m1)
    {   // comienza un nuevo bloque
        Mapa m2(320, 200);         // llama a constructor (m2)
    }   // al cerrar bloque llama a los destructores (m2)
    delete m0;                     // destruye m0 explicitamente
    return 0;
}       // al cerrar bloque llama a destructores (m1; m0 es puntero)
```

Es importante la distinción entre la forma en que se liberan las variables estáticas (m1 y m2: automáticamente al cerrarse los bloques en que están declarados) y la forma en que se liberan las variables dinámicas (m0: explícitamente, ya que fue reservado con un `new`). Si no se hubiese liberado explícitamente m0, se habría producido una fuga de memoria. El ejemplo produce la siguiente salida:

```
construido: mapa de 128x128
construido: mapa de 2560x1400
construido: mapa de 320x200
destruyendo: mapa de 320x200
destruyendo: mapa de 128x128
destruyendo: mapa de 2560x1400
```

⋆ Finalmente, además de constructores y destructores, hay una otras dos operaciones que se crean por defecto en cualquier clase nueva: el **operador de copia**, con la cabecera siguiente (por defecto, pública):

```
UnTAD& operator=(const UnTAD& otro);
```

Y el **constructor de copia**, que es similar en función:

```
UnTAD(const UnTAD& otro);
```

Ambos se usan como sigue:

```
Pila<int> p1().apila(42);
Pila<int> p2;
p2 = p1;           // operador asignacion
Pila<int> p3 = p1; // constructor de copia, equivale a p3(p1)
Mapa m1(100, 100);
Mapa m2;
m2 = m1;       // operador asignacion
Mapa m3 = m1; // constructor de copia, equivale a m3(m1)
```

En sus versiones por defecto (generadas por el compilador), ambos copian campo a campo del objeto *otro* al objeto actual (la razón de que exista un constructor de copia es que, si no existiera, primero habría que crear un objeto que no se usaría, y luego machacarlo con los valores del otro; es decir: ahorra una copia). Esto sólo funciona como se espera si la implementación no usa punteros. En el caso de una implementación dinámica, copiar un campo de tipo puntero resulta en que ambos punteros (el original y la copia) apuntan a lo mismo; y modificar la copia resultará en que también se modifica el original, y viceversa. Si esto no es lo que se desea en una implementación dada, se debe escribir una versión explícita de este operador. En el ejemplo anterior, modificar `m1` y `m2` es equivalente (ambos comparten los mismos datos). En cambio, `p1` y `p2` son independientes (se trataba de implementaciones estáticas).

⋆ Ventajas de las implementaciones estáticas:

- Más sencillas que las dinámicas, ya que no requieren código de reserva y liberación de memoria.
- El operador de copia por defecto funciona tal y como se espera (asumiendo que no se usen punteros de ningún tipo), y las copias son siempre independientes.
- Más rápidos que los dinámicos; la gestión de memoria puede requerir un tiempo nada trivial, en función de cuántas reservas y liberaciones hagan los algoritmos empleados. Esto se puede solucionar parcialmente haciendo pocas reservas grandes en lugar de muchas pequeñas.

⋆ Ventajas de las implementaciones dinámicas:

- Permiten tamaños mucho más grandes que las estáticas: hasta el máximo de memoria disponible en el sistema, típicamente varios GB; en comparación con los ∼8MB que pueden ocupar (en total) las reservas estáticas.
- No necesitan malgastar memoria - pueden reservar sólo la que realmente requieren.
- Pueden compartir memoria, mediante el uso de punteros (pero esto requiere mucho cuidado para evitar efectos indeseados).

3. Probando y documentando TADs

⋆ Las verificaciones formales son una buena forma de convencernos de la corrección de los programas. No obstante, es posible cometer errores en las demostraciones, o demostrar algo distinto de lo que realmente se ha implementado.

⋆ Las pruebas se pueden dividir en varios tipos:

- Pruebas **formales**: verifican la corrección teórica de los algoritmos, sin necesidad de acceso a una implementación concreta. Son las que hemos visto en los temas 3 (iterativos) y 4 (recursivos).
- Pruebas de **caja negra**: verifican una implementación sólo desde el punto de vista de su interfaz, sin necesidad de acceder a su código. Se usan para ver si "desde fuera de la caja" todo funciona como debe.
- Pruebas de **caja blanca**: verifican aspectos concretos de una implementación, accediendo al código de la misma.

⋆ La verificación formal, si se realiza correctamente, *es la única que puede demostrar la corrección de un algoritmo o la consistencia de una especificación.* Las pruebas de implementaciones sólo pueden encontrar errores, pero no encontrar un error no es garantía de que no exista. Desgraciadamente, requiere un trabajo sustancial, y (a no ser que se use asistencia de herramientas informáticas) es susceptible a errores en las demostraciones. En un escenario real, sólo se deben usar en casos muy puntuales, o en campos de aplicación donde la seguridad e integridad de los datos es realmente crítica (eg.: aviónica o control de centrales nucleares).

⋆ Las pruebas de caja negra permiten verificar la especificación de una implementación "desde fuera", contrastando una serie de comportamientos esperados (según la especificación) con los comportamientos obtenidos. No requieren acceso a los fuentes.

Si se dispone de los axiomas algebráicos de un TAD, unas pruebas de caja negra deberían incluir, para cada ecuación, una comprobación de que se cumple. Así, unas pruebas de 'caja negra'mínimas para "Pila" podrían ser las siguientes:

```
//   Pila().esVacia() = true
assert( true == Pila<int>().esVacia() );
//   p.apila(e).esVacia() = false
assert( false == p.apila(e1).esVacia() );
//   p.apila(e).cima() = e
assert( e1 == p.cima() );
//   p.apila(e).desapila() = p
Pila<int> q = p;
assert(p.apila(e2).desapila() == q);
```

Donde el uso de `assert` (importado mediante `#include <cassert>`) permite introducir comparaciones "todo o nada": si en algún caso el contenido es `false`, se sale del programa inmediatamente con un mensaje indicando la línea del error. El código también asume que se ha implementado un operador "`==`" para verificar igualdad entre pilas (que deberá reflejar correctamente la *relación de equivalencia* entre pilas - ver Figura 4). Para estar razonablemente seguros de que la implementación funciona como se espera, habría que ejecutar el código anterior con muchas pilas `p` y elementos `e1` y `e2` distintos.

⋆ Las pruebas de caja blanca van un paso más allá que las de caja negra, y partiendo del código de una implementación, ejercitan ciertas trazas de ejecución para verificar que fucionan como se espera.

- Una *traza de ejecución* (*execution path*) es una secuencia de concreta de instrucciones que se pueden ejecutar con una entrada concreta; por ejemplo, en este fragmento hay 3 trazas posibles:

```
if (a() && b()) c(); else d();
```

1. `a()`, `d()` (asumiendo que `a()` devuelva `false`)
2. `a()`, `b()`, `d()` (asumiendo que `a()` devuelva `true` y `b()` devuelva `false`)
3. `a()`, `b()`, `c()` (asumiendo que `a()` y luego `b()` devuelvan `true`)

- Las trazas relacionadas se agrupan en "tests unitarios" (*unit tests*), cada uno de los cuales ejercita una función o conjunto de funciones pequeño.
- Al porcentaje de código que está cubierto por las trazas de un conjunto de pruebas de caja blanca se le denomina "cobertura". Idealmente, el 100 % del código debería estar cubierto - pero en un comienzo, lo más importante es concentrarse en los fragmentos más críticos: aquellos que se usan más o que pueden producir los problemas más serios. Existen herramientas que ayudan a automatizar el seguimiento de la cobertura de un programa.
- Es posible que exista código que no puede ser cubierto por ninguna traza (por ejemplo, `if (false) cout << "imposible!\n";`). A este código se le denomina "código muerto" - y sólo puede ser detectado por inspección visual o herramientas de cobertura.

3.1. Documentando TADs

⋆ La documentación de un TAD es crítica, ya que es el único sitio en el que se describe la abstracción que representa el tipo, cómo se espera que se use, y las especificaciones a las que se adhiere. Uno de los principales objetivos de un TAD es ser reutilizable. Típicamente, el equipo o persona encargados de reutilizarlo no será el mismo que lo implementó (e incluso si lo es, es muy fácil olvidarse de cómo funciona algo que se escribió hace meses, semanas o años). Leer código fuente es difícil, y a veces no estará disponible - una buena forma de entender un programa complejo es examinar la documentación de sus TADs para hacerse una idea de las abstracciones que usa.

⋆ Todo TAD debe incluir, en su documentación:

- Una **descripción de alto nivel** de la abstracción usada. Por ejemplo, en un TAD `Sudoku`, la descripción podría ser *Un tablero de Sudoku 9x9, que contiene las soluciones reales, las casillas iniciales, y los números marcados por el usuario en casillas inicialmente vacías.*. En algunos casos, el TAD es tan bien conocido que no hace falta extenderse en esta descripción; por ejemplo, un `Punto2D` o una `Pareja` son casi auto-explicativos.
- Una **descripción de cada uno de sus campos públicos**, ya sean operaciones, constantes, o cualquier otro tipo. La descripción debe especificar cómo se espera que se use, y en general puede incluir notación tanto formal como informal. En esta asignatura, exigiremos que se use notación la notación formal explicada en este tema.

⋆ En C++, la documentación de un TAD se escribe exclusivamente en su `.h`. Los comentarios en el `.cpp` se refieren a la implementación concreta de las operaciones, aunque es útil y recomendable incluir cabeceras de función mínimas en los `.cpp`.

⋆ Los TADs auxiliares de un TAD principal, si son visibles para un usuario (es decir, accesibles mediante operaciones públicas) deben tener la misma documentación que

el TAD principal. Así, si un TAD `Poligono` usa un TAD `Punto` devolviendo o aceptando puntos, el TAD Punto también debe estar bien documentado. No es necesario (pero sí recomendable) tener el mismo cuidado con los TADs auxiliares privados.

⋆ El estilo de la documentación de un proyecto, junto con el estilo del código del proyecto, se especifican al comienzo del mismo (idealmente mediante una "guía de estilo"). A partir de este momento, todos los desarrollos deberán atenerse a esta guía. Para nuevos proyectos, puedes usar un estilo similar al siguiente:

```
/**
 * @file    pila.h
 * @author manuel.freire@fdi.ucm.es
 * @date   2012-01-09
 * @brief  Pila sencilla
 *
 * Una pila estatica muy sencilla, que usa un vector
 * fijo para los elementos
 */
#ifndef PILA_H_
#define PILA_H_

// excepcion de pila llena
class PilaLlena {};

// excepcion de pila vacia
class PilaVacia {};

template <class E>
class Pila {
    static const int MAX = 100;  /**< max. de elementos */
    static const int VACIA = -1; /**< indica pila vacia */
    int _posCima;        /**< posicion del elemento cima */
    E _elementos[MAX];  /**< vector estatico de elementos */
public:

    /**
     * Inicializa una pila vacía (generador).
     * Axiomas:
     *   Pila().esVacia() = true
     * P: {cierto}
     * Q: {p.esVacia()}
     */
    Pila() { _posCima = VACIA; }

    /**
     * Inserta un elemento en la pila
     * (generador; parcial; si p llena, error).
     * Axiomas: (si p es Pila, y e es de tipo E),
     *  p.apila().esVacia() = false
     *  p.apila().desapila() = p
     *  p.apila(e).cima() = e
     * P: {p Pila y p._posCima < MAX-1 y q = p}
     * Q: {p Pila y
     *  p.esVacia() = false y p.cima() = e y p.desapila() = q}
     * @param e elemento a insertar
     */
```

```
    Pila &apila(const E &e) {
        if (_posCima == MAX-1) throw PilaLlena();
        _elementos[_posCima + 1] = e;
        _posCima ++;
        return *this;
    }

    /**
     * Extrae un elemento de la pila
     * (mutador; parcial: si p.esVacia, error)
     * P: {p Pila y no p.esVacia() y p = q}
     * Q: {p Pila y p.apila(q.cima()) = q}
     */
    Pila &desapila() {
        if (_posCima == VACIA) throw PilaVacia();
        _posCima --;
        return *this;
    }

    /**
     * Devuelve el elemento de la cima de la pila
     * (observador; parcial: si p.esVacia(), error)
     * P: {p Pila y no p.esVacia()}
     * Q: {p, q Pila y p = q.apila(e)}
     * @return e: una referencia constante al elemento de la cima
     */
    const E &cima() const {
        if (_posCima == VACIA) throw PilaVacia();
        return _elementos[_posCima];
    }

    /**
     * Devuelve si la pila esta vacia
     * (observador; parcial: si p.esVacia(), error)
     * P: {p Pila y no p.esVacia()}
     * Q: {p, q Pila y ((b y p = q.apila(e)) o
     *       (no-b y no existe p con p = q.apila(e)))}
     * @return b: true si esta vacia, false en caso contrario
     */
    bool esVacia() const {
        return _posCima == VACIA;
    }
};
#endif
```

pila.h

En este ejemplo, se usan anotaciones tipo *doxygen*[9]. El sistema doxygen permite generar documentación pdf y html muy detallada a partir de los fuentes y las anotaciones que contiene su documentación.

[9] Puedes leer más sobre *doxygen* en `http://www.doxygen.org`

Notas bibliográficas

Gran parte de este capítulo se basa en el capítulo correspondiente de (Rodriguez Artalejo et al., 2011). Algunos ejemplos están inspirados en (Stroustrup, 1998).

Ejercicios

Introducción

1. Proporciona 3 ejemplos de TADs que podrían resultar útiles para implementar un juego de cartas (elige un único juego concreto para tus ejemplos). Para cada uno de esos TADs, enumera sus operaciones básicas.

2. Un diccionario de palabras permite, dada una palabra, buscar su significado. ¿Qué operaciones de tipo necesitaría un TAD `Diccionario` que permita tanto crear como consultar el diccionario?

3. Modifica el TAD diccionario del ejercicio anterior para que pueda almacenar más de un significado por palabra; define para ello un TAD básico `Lista` (que almacene sólo definiciones de tipo `string`), que usarás dentro del diccionario.

4. Implementa el .h y el .cpp para un TAD `Complejo` que permita representar números complejos en C++. Usa `float` para ambas partes, real e imaginaria, y proporciona operaciones para suma, resta, multiplicación (no es necesario usar sobrecarga de operadores[10]), y observadores que devuelvan las partes real e imaginaria.

5. Supón que existe un TAD `MultiConjunto` para enteros, de forma que, dado un multiconjunto con capacidad para n elementos `mc = MultiConjunto(n)`, `mc.pon(e)` añade un nuevo entero (descartando el más grande, si es que ya hay n), y `mc.min(e)` devuelve el entero más bajo. Implementa

```
func menores (k: Natural; v[n] enteros) devuelve mc : MultiConjunto
{ P0 : 0 <= k < n y n >= 1 }
{ Q0 : mc contiene los k elementos menores de v[0..n-1] }
```

Especificación e implementación de TADs

6. Amplía el tipo `Bool` para añadir las operaciones *xor*, *nor* y *nand*, con sus acepciones habituales, especificando estas nuevas operaciones mediante las ecuaciones correspondientes.

7. Especifica e implementa un tipo `Bool3` que, además de `Cierto` y `Falso`, incluya un valor `Incierto`. Todas las operaciones con `Incierto` deben producir como resultado `Incierto`.

8. Demuestra que, para el tipo `Bool`, las operaciones *and* y *or* son conmutativas, es decir, que dados $x, y : \mathsf{Bool}$:

   - $\mathsf{and}(x, y) =_{\mathsf{Bool}} \mathsf{and}(y, x)$

[10] En C++, es posible sobrecargar casi todos los operadores (`+`, `-`, `*`, `/`, ...) para tipos del usuario; por ejemplo, TAD `std::complex` de la librería estándar redefine todos estos operadores.

- $or(x, y) =_{Bool} or(y, x)$

9. Implementa una versión genérica (usando *templates* C++) del `Complejo` descrito en el el ejercicio 4, de forma que `Complejo<float>` sea un complejo que usa precisión sencilla, y `Complejo<double>` uno con precisión doble.

10. Implementa una versión genérica que empiece por

```
template <class E, int i>
class MultiConjunto {
    E _elems[i]; // vector generico de i elementos de tipo E
    ...
};
```

del MultiConjunto descrito en el ejercicio 5, donde E será el tipo de elemento del que está compuesto el multiconjunto.

11. Implementa el .h y el .cpp correspondientes a un TAD `Rectangulo`, que use por debajo un TAD `Punto` muy básico (con un `struct` vale), y con las mismas operaciones del ejemplo visto en la teoría. Usa `float` para coordenadas y dimensiones. ¿Cuánto ocupa (en bytes) un `Rectangulo` si lo implementas con un punto origen, un ancho, y un alto? ¿y si lo implementas con puntos origen y extremo?

12. Implementa el .h y el .cpp correspondientes a un TAD `TresEnRaya`, que deberá contener la posición de las piezas en un tablero del juego de 3 en raya, y el turno actual. Especifica e implementa las operaciones imprescindibles para poder jugar con este tablero, y describe qué otras operaciones podrían facilitar una implementación completa del juego.

13. Implementa el .h y el .cpp correspondientes a un TAD `BarajaPoker`, que deberá contener las cartas correspondientes a una baraja para jugar al poker. ¿Qué partes podrían ser comunes con una baraja española?

TADs dinámicos

14. Escribe un programa que determine cuánta memoria estática tiene disponible. Para ello, usa una función recursiva infinita, que reserve memoria de 100 Kb en 100 Kb (y muestre por pantalla cuánta ha reservado) antes de volver a llamarse. El último valor mostrado, sumando un poquito más dedicado a los punteros de las funciones que se han ido llamando, corresponde al número que buscas. Ejecuta el mismo programa en otro sistema operativo.

15. Escribe un programa que determine cuánta memoria dinámica tienes disponible, por el método de hacer infinitos `new`s de arrays de 100 Kb hasta que uno de ellos falle (en este momento, finalizará el programa). Ve mostrando el progreso a medida que avancen las reservas – y ejecuta este programa en una máquina que no te moleste reiniciar, porque durante las últimas reservas se ralentizará significativamente la velocidad del sistema. Ejecuta el mismo programa en otro sistema operativo.

16. Implementa un tipo "vector dinámico" de enteros llamado `Vector`. En el constructor, habrá que especificar un tamaño (también llamado dimensión) inicial. Habrá una operación de acceso `get(i)` que devuelva el elemento i-esimo, otra llamada `dimension()` que devuelva la dimensión actual del vector y un mutador

`set(i, valor)` que permita cambiar el valor del elemento i-esimo. Lanza excepciones si se intenta acceder fuera del vector. En cualquier momento, usando la operación `dimensiona(d)`, será posible cambiar el tamaño del vector a uno *d*, que podrá ser tanto mayor como menor que el actual. En ambos casos, se deberán mantener todos los elementos que quepan. Ten cuidado con el destructor y el operador de copia por defecto.

17. Modifica el `Vector` del ejercicio anterior para que, además de enteros, acepte cualquier otro tipo de elemento; es decir, convierte a tu `Vector` en parametrizable[11].

18. Modifica el código de ejemplo de la `Pila` parametrizable estática para escribir una pila dinámica basada en el `Vector` parametrizable del ejercicio anterior. Inicialmente, usarás un `Vector` de `INICIAL` elementos (una constante, que puede ser por ejemplo 100; ya no necesitarás la constante `MAX`). Cuando se inserten muchos elementos, en lugar de lanzar excepciones de tamaño agotado, llama a `dimensiona(_elementos.dimension() + INICIAL)`.

Probando TADs

19. Escribe una batería de pruebas de caja negra para el TAD Pareja.

20. Escribe una batería de pruebas de caja blanca para el TAD Pila (estática) - haz énfasis en los aspectos que no puedes comprobar con las pruebas de caja negra: todos los casos límites de intentar extraer o mirar la cima de una pila vacía, o de tratar de insertar en una pila llena.

21. Escribe una batería de pruebas de caja blanca para el TAD Pila (dinámica, según el ejercicio 18). ¿Cómo tendrás que adaptar las pruebas del ejercicio anterior para este cambio en la implementación del TAD? ¿Cómo se te ocurre que puedes probar el caso de "pila llena"?.

22. Escribe una batería de pruebas de caja negra usando `assert` para los TAD diccionario de los ejercicios 2 y 3.

Documentando TADs

23. Aplica el estilo de documentación propuesto en la sección de Documentando TADs a los .h de todos los ejercicios anteriores.

[11] La librería estándar de C++ ya proporciona un vector dinámico parametrizable, llamado `vector`. Siempre que no sepas de antemano el tamaño de un array, y que ese tamaño va a ser constante, deberías usar `vector` (definido mediante `#include <vector>`). En general (excepto cuando el ejercicio pide lo contrario), siempre es mejor usar implementaciones de la librería estándar en lugar de las propias.

Bibliografía

Y así, del mucho leer y del poco dormir, se le secó el celebro de manera que vino a perder el juicio.

Miguel de Cervantes Saavedra

Collins-Sussman, B., Fitzpatrick, B. W. y Pilato, C. M. *Version Control with Subversion*. O'Reilly, 2004. ISBN 0-596-00448-6. Disponible en `http://svnbook.red-bean.com/` (último acceso, Octubre, 2009).

Martí Oliet, N., Segura Díaz, C. M. y Verdejo López, J. A. *Especificación, Derivación y Análisis de Algoritmos: ejercicios resueltos*. Colección Prentice Práctica, Pearson Prentice-Hall, 2006.

Peña, R. *Diseño de Programas: Formalismo y Abstracción*. Tercera edición. Pearson Prentice-Hall, 2005.

Rodriguez Artalejo, M., González Calero, P. A. y Gómez Martín, M. A. *Estructuras de datos: un enfoque moderno*. Editorial Complutense, 2011.

Stroustrup, B. *The C++ Programming Language, 3rd Edition*. Addison-Wesley, 1998. ISBN 0201889544.