Capítulo 2

Implementación y uso 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 **tipos abstractos de datos** (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. Motivación

Te plantean el siguiente problema: Dado un número x, se cogen sus dígitos y se suman sus cuadrados, para dar x_1. Se realiza la misma operación, para dar x_2, y así mucho rato hasta que ocurra una de las dos cosas siguientes:

- se llega a 1, y se dice entonces que el número es "feliz"
- nunca se llega a 1 (porque se entra en un ciclo que no incluye el 1), y se dice entonces que el número es "infeliz"

Ejemplos:

- el 7 es feliz: $7 \to 49 \to 97 \to 130 \to 10 \to 1$
- el 38 es infeliz: $38 \to 73 \to 58 \to 89 \to 145 \to 42 \to 20 \to 4 \to 16 \to 37 \to 58$

Sin estructuras ni TADs, necesitarás bastante imaginación para encontrar una solución (existe, eso sí). Al final de este capítulo verás una solución mucho más legible, usando un TAD `Conjunto`.

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

2. Introducción

2.1. 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:

- 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;
}
```

los dominios de *float* e *int* no son equivalentes

⋆ 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;
}
```

optimización que rompe la encapsulación de los tipos *float* e *int*

⋆ 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.

2.2. TADs de usuario

⋆ Los `structs` de C, o los `records` 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) {}
};
```

estructura `Fecha`. Es fácil generar fechas inválidas.

⋆ 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`.

Ejemplo expandido: 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 (*this*): 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(`*this*`=`*f*`)`.

[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).

```
class Fecha {
public: // parte publica del TAD
  // Constructor
  //   dia, mes y año forman una fecha en el dominio esperado
  Fecha(int dia, int mes, int anyo);

  // Constructor alternativo
  //    dias es un número de días a sumar o restar a la fecha base
  Fecha(const Fecha &base, int dias);

  // distancia, en días, de la fecha actual con la dada (Obs.)
  int distancia(const Fecha &otra) const;

  // devuelve el dia de esta fecha (Obs.)
  int dia() const;

  // devuelve el día de la semana de esta fecha (Observadora)
  int diaSemana() const;

  // devuelve el mes de esta fecha (Observadora)
  int mes() const;

  // devuelve el año de esta fecha (Observadora)
  int anyo() const;

private: // parte privada, accesible solo para la implementacion
  int _dia;    // entre 1 y 28,29,30,31, en función de mes y año
  int _mes;    // entre 1 y 12, ambos inclusive
  int _anyo;
};
```

clase `Fecha`, con operaciones para acceder y modificar fechas de forma controlada

⋆ 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 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.

⋆ 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 públicas 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. También es frecuente referirse a la parte pública de un tipo o módulo como su **API** (*Application Programming Interface*).

⋆ 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 a 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 como privado. Una marca típica es comenzar identificadores privados con un '_', escribiendo, por ejemplo, '`_dia`'. Esto ya no es necesario en C++ (aunque sigue siendo útil como recordatorio), 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 la forma de llamar a las operaciones externas (la parte pública) no habría cambiado. En general, todo TAD contiene decisiones de implementación que deberían tener en cuenta los casos de uso a los que va a ser sometido, y sus frecuencias relativas.

2.3. 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

[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.

están implementadas internamente – o incluso, cómo se van a implementar, cuando llegue el momento de hacerlo).

- Ejemplo de 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.
- Ejemplo de 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*.

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 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
```

declaración del módulo fecha, en `fecha.h`

```
#include "fecha.h"

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

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

su implementación, en `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 están fuertemente relacionados 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

- Un TAD está formado por una colección de valores y un conjunto de operaciones sobre dichos valores.
- Una **estructura de datos** es una estrategia de almacenamiento en memoria de la información que se desea guardar. Muchos TADs se implementan utilizando estructuras de datos. Por ejemplo, los TADS pilas y colas pueden implementarse usando la estructura de datos de las listas enlazadas.
- Algunos TADs son de uso tan extendido y frecuente, que es de esperar que en cualquier lenguaje de programación de importancia exista una implementación. Por ejemplo, las librerías estándares de C++, Java ó Delphi soportan, todas ellos, árboles de búsqueda ó tablas hash.
- 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.
- La importancia de estudiar estructuras de datos radica no solamente en los detalles de su implementación (aunque sean buenos ejemplos de programación), sino en sus *características generales* incluyendo los costes de las operaciones que proporcionan,

que es imprescindible conocer si se quiere poder tomar decisiones acerca de cuándo preferir una estructura a otra.

3. Implementación de TADs

⋆ En este apartado se analizan con mayor detalle cómo se debe programar los TADs en C++. C++ es un lenguaje tremendamente flexible y poderoso; esto quiere decir que hay más de una forma de hacer las cosas, y las desventajas de ciertas implementaciones pueden no ser aparentes. El contenido de este apartado sigue las "mejores prácticas" reconocidas en la industria; úsalo como referencia cuando hagas tus propias implementaciones.

3.1. Tipos de operaciones y el modificador *const*

⋆ Las operaciones se pueden clasificar como *generadoras*, *modificadoras* u *observadoras*.

Generador 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.

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

Observador 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.

⋆ La **clasificación que usaremos** en la asignatura es equivalente, pero está mejor adaptada a lenguajes orientados a objetos:

Constructor Crea una nueva instancia del tipo. En C++, un constructor se llama siempre como el tipo que construye (con los argumentos que se desee utilizar). Se llaman automáticamente cuando se declara una nueva instancia del tipo. Si no hay nada que inicializar, se pueden omitir.

Mutador Modifica la instancia actual del tipo. En C++, *no pueden* llevar el modificador `const` al final de su definición.

Observador No modifican la instancia actual del tipo. En C++, *deben* llevar el modificador `const` (aunque el compilador no genera errores si se omite).

Destructor Destruye una instancia del tipo, liberando cualquier recurso que se haya reservado en el momento de crearla (por ejemplo, cerrando ficheros, liberando memoria, o cerrando conexiones de red). Los destructores se invocan automáticamente cuando las instancias a las que se refieren salen de ámbito. Si no hay que liberar nada, se pueden omitir.

⋆ *Se debe usar el modificador* ***const*** *siempre que sea posible* (es decir, para todas las operaciones observadoras), ya que hacer esta distinción tiene múltiples ventajas. Por ejemplo, una instancia de un tipo *inmutable* (sin mutadores) se puede compartir sin que haya riesgo alguno de que se modifique el original.

⋆ Es frecuente, durante el diseño de un TAD, poder elegir entre suministrar una misma operación como mutadora 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 (mutadora), o que devuelva una *nueva* fecha `delta` días en el futuro (observadora, y por tanto *const*).

3.2. Uso de sub-TADs y extensión de TADs

⋆ Es legal, y frecuente, que un TAD use a otro. 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.
 */
class Rectangulo {
private:
    ...
public:
    // constructor
    //    origen es la esquina inf. izquierda
    Rectangulo(Punto origen, float alto, float ancho);

    // constructor
    //    interpreta ambos puntos como esquinas opuestas
    Rectangulo(Punto uno, Punto otro);

    // devuelve el punto con coordenadas x e y mínimas (Obs.)
    const Punto &origen() const;
    float alto() const;
    float ancho() const;

    // igualdad (Observadora)
    //   todos los rectangulos vacios son iguales
    bool igual(const Rectangulo &r) const;

    // calcula area (Observadora)
    float area() const;

    // verifica si esta vacio, es decir, si tiene área 0 (Obs.)
    bool esVacio() const;

    // devuelve true si el punto está dentro (Obs.)
    //    el borde se considera dentro, excepto si el r. está vacio
    bool dentro(const Punto &p) const;

    // indica si este rectagulo contiene a r (Obs.)
    //    true si todos los puntos de r están dentro
    bool dentro(const Rectangulo &r) const;

    // calcula interseccion (Observadora)
    //    devuelve el mínimo r con todos los puntos dentro de ambos
    Rectangulo interseccion(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. En general, no se pedirá manejo de herencia en este curso – pero se debe saber que existe y está disponible en C++.

3.3. 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 `Conjunto`, no debería haber grandes diferencias entre un conjunto de enteros, `Puntos`, o palabras. 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.
- herencia: 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.
- 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`:

```
template <class A, class B>
class Pareja {
    // una pareja inmutable generica
    A _a; B _b;
public:
    // Constructor sencillo
    Pareja(A a, B b) { _a=a; _b=b; } // cuerpos en el .h (!)
    // Observadoras
    A primero() const { return _a; }
    B segundo() const { return _b; }
};
```

clase `Pareja` en `pareja.h`, con las implementaciones dentro de la clase

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

main.cpp que incluye a pareja.h

NOTA: en C++ es legal definir los cuerpos de funciones en el .h en lugar de en el .cpp (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:

```
template <class A, class B>
class Pareja {
    // una pareja inmutable generica
    A _a; B _b;
public:
    // Constructor sencillo
    Pareja(A a, B b);
    // Observadoras
    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; }
```

pareja.h alternativo, dejando más despejado el tipo

3.4. 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 dos pasos:

1. Implementar la parte privada, eligiendo para ello los tipos de implementación concretos que se va a usar para representar el TAD (los *tipos representantes*);

la interpretación que se va a hacer de sus valores (la *función de abstracción*, que incluye también decir cuándo dos valores del tipo representante representan lo mismo: la *función de equivalencia*); y las condiciones que se deben cumplir para que los valores se consideren válidos (los *invariantes de representación*).

2. Implementar las operaciones, de forma que nunca se rompan los invariantes de representación (posiblemente con restricciones adicionales debidas al tipo representante elegido).

⋆ Ejemplos de decisiones para implementar partes privadas, usando la nueva terminología:

- Un TAD `Rectangulo` en 2D alineado con los ejes se puede representar con
  - un `Punto` orígen y un par de `int` para ancho y alto (= *tipos representantes*). Todos los rectángulos vacíos se podrían considerar equivalentes, independientemente de orígen (ya que no contienen ningún punto); esta interpretación nos proporciona la *función de abstracción* y la de *equivalencia*; y el ancho y el alto tendrían que ser, para rectángulos no-vacíos, estrictamente mayores que 0 (*invariante de representación*).
  - dos `Puntos` en esquinas opuestas, con ambos puntos idénticos en el caso de un rectángulo vacío. Podríamos exigir que las coordenadas x e y del segundo punto fuesen siempre mayores que las del primero, lo cual nos proporcionaría un invariante adicional (y simplificaría muchas operaciones).
- El TAD `Complejo` se puede representar con
  - una pareja de `float`, uno para la parte entera y otro para la imaginaria. Las funciones de abstracción y de equivalencia son aquí triviales, y el invariante de representación se cumple siempre.
  - dos `double`, usando una representación polar ángulo-magnitud. La función de abstracción es en este caso la interpretación ángulo-magnitud (y la regla para pasar de esta representación a *(real, imaginario)*, y el invariante de representación podría usarse para restringir el rango de ángulos posibles al intervalo $[0, 2\pi]$.
- Un TAD `Fecha` se puede representar con tres `int` que almacenen los días, meses y años; o con un `int` que represente segundos desde el 1 de enero de 1970. Las reglas que limitan sus valores válidos para una fecha determinada constituirían los invariantes de representación de esta implementación.

⋆ Ejemplos de restricciones impuestas por el tipo representante al dominio de valores del TAD:

- Muchos sistemas desarrollados durante las décadas de los 70 y 80 representaban el año mediante dos caracteres "XX", interpretándolo como "19XX". Esto ocasionó numerosos problemas cuando se llegó al año 2000, que estos sistemas interpretaban como si fuese 1900.
- De forma similar, un valor entero representado mediante un `int` de 32 bits sólo puede tomar 2^{32} valores distintos - lo cual restringe el dominio de implementación de un TAD que use `int`s de 32 bits como tipos representantes. Si representamos una `Fecha` con un `int` para los segundos desde 1970, será imposible producir fechas más allá del 19 de enero de 2038[4].

[4]Esta representación es muy popular, y aunque muchas implementaciones ya se han actualizado a

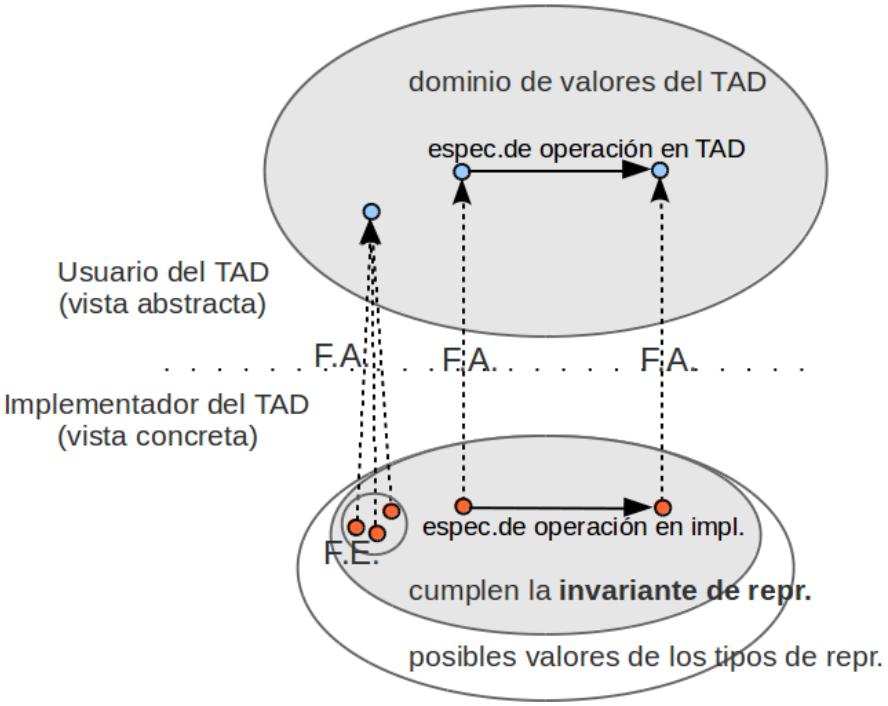

Figura 1: La función de abstracción (F.A.), la vista de usuario (abstracta), y la vista de implementador (concreta). El invariante de representación delimita un subconjunto de todos valores posibles del tipo representante como *válidos*, y la función de abstracción los pone en correspondencia con términos 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.

⋆ El **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:

- `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
- `Complejo` (usando `float _real`, `float _imaginario`):
  True (siempre se cumple)
- `Complejo` (usando `float _magnitud`, `float _angulo`):
  $0 \leq$ _angulo $\leq 2\pi$
- `Hora` (usando `int _horas`, `int _minutos`, `int _segundos`):
  $0 \leq$ _horas $\wedge\ 0 \leq$ _minutos $< 60 \wedge\ 0 \leq$ _segundos < 60
- `Hora` (usando sólo `int _segundos`):
  $0 \leq$ _segundos

enteros de 64 bits, el problema es comparable al del año 2000. Para más referencias, ver `http://en.wikipedia.org/wiki/Year_2038_problem`

⋆ La **relación de equivalencia** 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 3 se puede ver un ejemplo de conjuntos 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);
    };
    ...
};
```

sobrecarga del operador == en C++

⋆ Un TAD puede incluir aspectos *parciales*, es decir, no totalmente 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 se debe considerar *parcial*, y debe estar debidamente comentada explicando las precondiciones que garantizar un funcionamiento predecible.

⋆ Ejemplo: un conjunto. En un TAD `Conjunto`, es posible añadir elementos, consultar si existe un elemento dado, y eliminar un elemento. En este caso, sería un error intentar eliminar un elemento que no exista.

```
template <class E>
class Conjunto {
    ... // aquí iría el tipo representante
public:
    // Constructor
    Conjunto();
    // inserta un elemento (mutadora)
    void inserta(const E &e);
    // elimina un elemento (mutadora)
    //    parcial: si no contiene(e), error
    void elimina(const E &e);
    // si contiene el elemento, devuelve 'true' (observadora)
    bool contiene(const E &e) const;
};
```

un TAD `Conjunto`. La implementación de `elimina()` es parcial.

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

```
static const int MAX = 100;
int _tam = 0;
E _elementos[MAX];
```

tipo representante para `Conjunto`

Debido a la limitación de tamaño, `inserta()` también es parcial:

```
// inserta un elemento (mutadora)
//    parcial: si se intenta insertar más de MAX elementos, error
void inserta(const E &e);
```

debido al uso de un vector estático, la inserción es parcial

2. Invariante de representación: $0 \leq \_\mathsf{tam} \leq \mathsf{MAX}$ (siempre se deben usar constantes[5] para los valores límite).

   La elección del invariante tiene impacto en la forma de implementar las operaciones y por tanto también en el coste de las mismas. Podríamos considerar tres invariantes de representación diferentes.:

   a) No hay ninguna restricción adicional a lo que se ha dicho hasta el momento.

   b) Exigimos que los elementos del vector entre 0 y $\_tam-1$ no estén repetidos.

   c) Exigimos que los elementos del vector entre 0 y $\_tam-1$ no estén repetidos y además estén ordenados (supuesto que exista una relación de orden entre ellos).

   Supongamos $\_tam = 3$. Así, mientras que el vector

2	1	2	...

   es válido en la representación (1), no lo es ni en la (2) ni en la (3) porque el 2 está repetido.

   Análogamente, el vector

2	1	3	...

   es válido en las representaciones (1) y (2) pero no en la (3), porque los elementos no están ordenados.

3. Función de abstracción y relación de equivalencia: Para pasar de un valor de `_elementos` y `_tam` a un conjunto abstracto, entenderemos que los elementos con índice 0 a `_tam` (exclusive) forman parte del conjunto salvo posibles repeticiones.

   Nótese que puede haber múltiples valores del tipo representante (en este caso, múltiples vectores de `_elementos`) que se refieran al mismo conjunto abstracto:

   - Por una parte, a partir de `_tam` (inclusive), el contenido de `_elementos` es completamente indiferente desde el punto de vista del TAD, como se muestra en la Figura 3.
   - Por otra parte, las repeticiones y el orden son irrelevantes. Los ejemplos anteriores representan los conjuntos $\{1, 2\}$ y $\{1, 2, 3\}$ en aquellas representaciones en las que eran válidos. Por ejemplo, vectores equivalentes en la

[5] El uso de `static const` declara `MAX` como constantes (`const`) para esta implementación del TAD, definidas una única vez para cualquier número de conjuntos (`static`), en lugar de una vez para cada conjunto individual (ausencia de `static`). Es preferible usar constantes estáticas de clase que `#defines`.

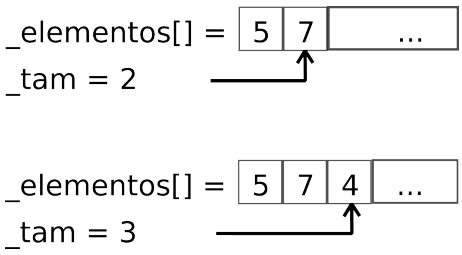

Figura 2: Conjunto antes y después de insertar el elemento 4

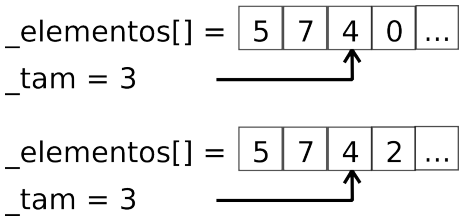

Figura 3: Conjuntos equivalentes (según la función de equivalencia del TAD)

representación (1) son: $\{1, 1, 2, 1, 2, \ldots\}$ con $\_tam = 5$ y $\{1, 2, \ldots\}$ con $\_tam = 2$. Vectores equivalentes en la representación (2) son:$\{1, 2, \ldots\}$ con $\_tam = 2$ y $\{2, 1, \ldots\}$ con $\_tam = 2$. En la representación (3) los vectores equivalentes solamente difieren en las posiciones $\_tam$ en adelante.

4. Implementaciones de cada operación, respetando los invariantes.Vamos a ver aquí la representación con invariante (1):

```
Conjunto() { _tam = 0; }
```

constructor

```
void inserta(const E &e) {
    if (_tam == MAX) throw "Conjunto Lleno";
    _elementos[_tam] = e;
    _tam ++;
}
```

inserta()

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. Por ejemplo, se podrían definir los tipos `ConjuntoLleno` y `ElementoInvalido`, que se usarían luego en las sentencias throw en substitución de las actuales cadenas de caracteres.

Para implementar la operación de eliminación, hemos de tener en cuenta que puesto que los elementos pueden estar repetidos, hay que eliminar todas las apariciones de dicho elemento:

```
void elimina(const E &e) {
    bool esta = false;
    int i=0;
    while (i<_tam)
      { if (_elementos[i]==e){
             _elementos[i] = _elementos[_tam-1];
             esta = true;
             _tam --;}
             // paso el ultimo elemento a su lugar
         else i++;
      };
    if (!esta) throw "Elemento Invalido";

   }
```

elimina()

```
bool contiene(const E &e) const {
    bool esta = false;
    int i=0;
    while (i<_tam && !esta)
      {esta = (_elementos[i]==e);
       i++;
      };
    return esta;
}
```

contiene()

Con esta forma de representar los conjuntos, las operaciones tienen los siguientes costes:

- El constructor y la operación `inserta` están en $\Theta(1)$.
- Las operaciones `elimina` y `contiene` están en $\Theta(\_tam)$ en el caso peor.

Obsérvese que puesto que puede haber elementos repetidos, $\_tam$ puede ser mucho más grande que el cardinal del conjunto al que representa. En las otras representaciones, que exigen que no haya repetición de elementos, $\_tam$ representa exactamente el cardinal del conjunto.

⋆ 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; Java, C++ ó Delphi sí). En general, casi 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 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. Esto es común en librerías con un fuerte componente asíncrono; por ejemplo, cuando se hacen llamadas "AJAX" desde JavaScript.

⋆ 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 su-

[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 – algo que puede ser imposible en presencia de programación concurrente.

ministrar 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;
}
```

⋆ NOTA: en C++, es posible usar el modificador **const** en 3 contextos completamente distintos:

para indicar una constante – usado en la declaración de la constante, y preferible al uso de `#define`.

para indicar un método observador – en el interior del cual no se modifica el valor de la instancia actual de una clase ó estructura.

como modificador de tipo – en los argumentos de entrada ó salida de un método, para indicar que el valor pasado no se puede cambiar dentro de la función. Tiene más sentido para argumentos pasados por referencia (`&`) o puntero (`*`), porque los argumentos pasados por valor siempre son copias, y por tanto modificarlos nunca tiene efectos externos a la función.

Ejemplo:

```
// devuelve el vértice del polígono más cercano al punto pasado
const Punto &Poligono::contiene(const Punto &p) const;
```

El primer y segundo `const` indican que el punto devuelto y el punto pasado son referencias no-modificables (uso como modificador de tipo). El último const indica que esta operación es observadora, y no modifica el polígono sobre el que se llama.

⋆ 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, asumiendo que se están usando excepciones para el manejo de las condiciones de error:

- Si la función no tiene un "valor devuelto" claro, como es el caso de las operaciones de mutación, no se debe devolver nada (declarando la función como `void`).
- 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), el tipo del resultado es el del valor devuelto, lo que significa que la instrucción `return` hará una copia del valor devuelto.
  - Si el valor es más grande, pero se trata de un nuevo objeto construido en el método, el tipo del resultado es el del valor devuelto y análogamente se hará una copia del valor devuelto por la instrucción `return`.
  - Si el valor es grande, pero comparte con un objeto ya existente, es más eficiente devolver una referencia, ya que en tal caso no se hará copia. Para evitar que modificaciones sobre ese valor afecten al objeto con el que comparte (lo que habitualmente recibe el nombre de efecto lateral), dicha referencia debe ser constante.

    ```
    // copia: bool no es grande,
    bool contiene(const E &e) const;
    // ref. cte: ocupa más que un int y es parte de un rectangulo
    const Punto &origen() const;
    // copia: devuelve un nuevo rectangulo
    Rectangulo Rectangulo::interseccion(const Rectangulo &r1) const;
    ```

- Si hay que devolver *más de un valor a la vez*, se puede devolver a través de argumentos "por referencia". El tipo de retorno de la función queda libre, y de nuevo se debería devolver `void`. Alternativamente, es posible agrupar todos los valores a devolver en una estructura ó clase, y devolverla como único valor.

  ```
  // devuelve coordenadas del i-esimo punto en x e y
  void Poligono::punto(int i, float &x, float &y) const;
  ```

```
// version mejorada: devuelve referencia a Punto
const Punto &Poligono::punto(int i) const;
```

3.5. Implementaciones dinámicas y estáticas

⋆ Algunos TADs pueden llegar a requerir mucho espacio. Por ejemplo, el tamaño de un `Conjunto` puede variar entre uno o dos elementos y cantidades astronómicas de los mismos, en función de las necesidades de la aplicación que lo use. 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**:

```
Conjunto<int> *c = new Conjunto<int>();
// conjunto de enteros en memoria dinámica
c->inserta(4);              // inserto un 4
(*c).inserta(2);            // "(*algo)." equivale a "algo->"
c->elimina(4);              // quito el primer 4
delete c;                   // libero el conjunto
```

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

```
int cuantos = 1<<30;                // = 2^30 (y<<x equivale a y·2^x)
int *dinamicos = new int[cuantos]; // reserva 1GB de memoria dinamica
...                                 // usa la memoria
delete[] dinamicos;                 // libera memoria anterior
```

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**. Para el conjunto estático con 100 elementos declarado antes, `sizeof(Conjunto<int>)` será `sizeof(_tam) + sizeof(_elementos)` = 404 bytes (es decir, el tamaño ocupado por sus tipos representantes: tanto `_tam` como cada uno de los 100

[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" ó "desbordamiento de pila", indica que tu pila ha desbordado su tamaño máximo prefijado.

`_elementos` ocupan 4 bytes, por ser de tipo `int`). De igual forma, `sizeof(Conjunto<char>)` habría sido 104 (ya que los `_elementos` de tipo `char` ocuparían sólo 1 byte cada uno). En una versión dinámica de este conjunto, `_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(_tam) + 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: el operador `sizeof` ignora completamente la existencia de memoria dinámica.

⋆ 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", o "escape de memoria" (*memory leak*), 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 difícil 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 produzcan errores inmediatos (y por tanto más fáciles de diagnosticar).

[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.

- Nada más liberar un puntero, asígnale el valor NULL (= 0); **delete()** nunca libera punteros a NULL; de esta forma puedes evitar algunas liberaciones múltiples:

```
// asigno un valor en la misma declaracion
Conjunto *c = new Conjunto();
delete c;              // comprueba NULL antes de liberar
c = 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;
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
    void celda(int i, int j, unsigned short v) {
        _celdas[i*_ancho + j] = v;
    }
    // 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:

```
Conjunto<int> c1().inserta(42);
Conjunto<int> c2;
c2 = c1;            // operador asignacion
Conjunto<int> c3 = c1; // constructor de copia, equivale a c3(c1)
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, `c1` y `c2` 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).

4. 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, teniendo en cuenta el 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 (como 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.

Unas pruebas de caja negra deberían incluir, para cada operación, comprobaciones de que produce la salida esperada para cada uno de los ejemplos de entrada con los que se va a enfrentar. En general, es imposible probar todos los casos, por lo que se elige un subconjunto representativo; cuantos más casos se puedan cubrir, más fiables serán las pruebas. Un ejemplo de pruebas de caja negra para `Conjunto` podrían ser las siguientes:

```
#include <iostream>
#include <cassert>
using namespace std;
int main() {
    Conjunto<int> vacio;
    Conjunto<int> c;
    // tras insertar un elemento, el elemento existe
    c.inserta(21);
    assert(c.contiene(21));
    // y se puede borrar
    c.elimina(21);
    // y despues ya no existe
    assert( ! c.contiene(21));
    // y si intento borrarlo se produce un error
    bool error = false;
    try {
      c.elimina(12);
    } catch (ElementoInvalido e) {
```

algunas pruebas de caja negra para `Conjunto`

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 conjuntos (que deberá reflejar correctamente la *relación de equivalencia* entre conjuntos - ver Figura **??**). Para estar razonablemente seguros de que la implementación funciona como se espera, habría que ejecutar el código anterior con muchos conjuntos `p` y elementos 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, herramientas de cobertura, o (en algunos casos), compiladores suficientemente avanzados.

4.1. 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 casos particularmente complejos, tiene sentido incluir pequeños ejemplos de uso.

⋆ 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 (resumidas) en los `.cpp`.

⋆ Los TADs auxiliares de un TAD principal, si son visibles para un usuario (es decir, accesibles mediante operaciones públicas) deben tener el mismo nivel de 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    conjunto.h
 * @author  manuel.freire@fdi.ucm.es
 * @date    2014-02-12
 * @brief   Conjunto sencillo
 *
 * Un conjunto estatico muy sencillo, que usa un vector
 * fijo para los elementos
 */
#ifndef CONJUNTO_H_
#define CONJUNTO_H_

/// excepcion de conjunto lleno
class ConjuntoLleno {};

/// excepcion de elemento inexistente
class ElementoInvalido {};

template <class E>
class Conjunto {
    static const int MAX = 100;  /**< max. de elementos */
    int _tam;            /**< numero de elementos actuales */
    E _elementos[MAX];   /**< vector estatico de elementos */

public:

    /**
     * Inicializa un conjunto vacio (constructor).
     */
    Conjunto() { _tam = 0; }

    /**
     * Inserta un elemento en la el conjunto
     * (mutador parcial; si se llena, lanza ConjuntoLleno).
     * @param e elemento a insertar
     */
    void inserta(const E &e) {
        if (_tam == MAX) throw ConjuntoLleno();
        _elementos[_tam] = e;
        _tam ++;
    }

    /**
     * Borra un elemento del conjunto
     * (mutador parcial; si no existe, lanza ElementoInvalido)
     */

   void elimina(const E &e) {
```

```
    bool esta = false;
    int i=0;
    while (i<_tam)
      { if (_elementos[i]==e){
            _elementos[i] = _elementos[_tam-1];
            esta = true;
            _tam --;}
            // paso el ultimo elemento a su lugar
        else i++;
      };
    if (!esta) throw ElementoInvalido();

   }

    /**
     * Devuelve true si el elemento esta contenido en el conjunto
     * (observador)
     */
    bool contiene(const E &e) const {
      bool esta = false;
      int i=0;
       while (i<_tam && !esta)
         {esta = (_elementos[i]==e);
        i++;
       };
    return esta;

    }

};
#endif

#include <iostream>
#include <cassert>
using namespace std;
int main() {
    Conjunto<int> vacio;
    Conjunto<int> c;
    // tras insertar un elemento, el elemento existe
    c.inserta(21);
    assert(c.contiene(21));
    // y se puede borrar
    c.elimina(21);
    // y despues ya no existe
    assert( ! c.contiene(21));
    // y si intento borrarlo se produce un error
    bool error = false;
    try {
      c.elimina(12);
    } catch (ElementoInvalido e) {
      error = true;
    }
```

listado completo de la implementación de una clase `Conjunto`

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.

5. Para terminar...

Terminamos el tema con la implementación de la función descrita en la primera sección de motivación. Como se ve, al hacer uso de un TAD `Conjunto` el código queda realmente legible: nos podemos centrar en la idea central (detectar el primer duplicado) sin preocuparnos para nada de los detalles. Además, si mejoramos la implementación del `Conjunto` (y la de este tema es muy ineficiente), no habrá que modificar en nada nuestra solución – gracias al uso de un TAD, nos hemos aislado de los detalles de implementación.

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

using namespace std;

int siguiente(int n) {
    int suma = 0;
    while (n>0) {
        int digito = n%10;
        suma += digito*digito;
        n /= 10; // avanzo de digito
    }
    return suma;
}

void psicoanaliza(int n) {
    Conjunto<int> c;
    while ( ! c.contiene(n)) {
        cout << n << " ";  // para ver los numeros por los que pasa
        c.inserta(n);
        n = siguiente(n);
    }
    if (n==1) {
        cout << "feliz (llega a 1) " << endl;
    } else {
        cout << n << " infeliz (repite del " << n << " en adelante)" << endl;
    }
}

int main() {
    psicoanaliza(7);
    psicoanaliza(38);
}
```

una solución que hace uso de un `Conjunto` de enteros

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

Notas bibliográficas

Parte de este capítulo se basa en el capítulo correspondiente de (Rodríguez 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] }
```

Implementación de TADs

6. Implementa las siguientes operaciones de los conjuntos utilizando el invariante de representación (1) presentado en el texto:

   - `esVacio`, que determina si un conjunto es vacío
   - `unitario`, que crea un conjunto unitario conteniendo un elemento dado
   - `cardinal`, que devuelve el cardinal del conjunto
   - `union`, `interseccion` y `diferencia` para llevar a cabo respectivamente la unión, intersección y diferencia entre dos conjuntos.
   - el operador de igualdad de dos conjuntos, mediante la sobrecarga del operador ==

[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.

Proporciona el coste de todas las operaciones.

7. Implementa todas las operaciones vistas hasta el momento de los conjuntos usando los invariantes de representación (2) y (3) vistos en el texto.

8. Implementa el TAD de los polinomios con coeficientes naturales, representando un polinomio de grado g $\sum_{i=0}^{g} c_i * x^i$ mediante un vector $P[0..N]$ con $N \geq g$ de forma que $P[i] = c_i$ para cada i, $0 \leq i \leq g$.

9. Implementa las operaciones del TAD `Fecha`, usando la versión que toma un `int` para indicar los segundos transcurridos desde el 1 de enero de 1970.

10. 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.

11. 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`.

12. Implementa el .h y el .cpp correspondientes a un TAD `Rectangulo` (alineado con los ejes), 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, entendidos como opuestos?

13. 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.

14. 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

15. Escribe un programa que determine cuánta memoria estática tiene disponible un PC. 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 marcos de las funciones que se han ido llamando, corresponde al número que buscas. Ejecuta el mismo programa en otro sistema operativo y/o PC.

16. 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 y compara los resultados.

17. 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)` para acceder al elemento i-esimo, otra llamada `dimension()` para consultar la dimensión actual del vector y un mutador `set(i, valor)` para cambiar el valor del elemento i-esimo. Lanza excepciones para evitar accesos fuera del vector. En cualquier momento, usando la operación `dimensiona(d)`, debe ser 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.

18. 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].

19. Modifica el código de ejemplo del `Conjunto` parametrizable estático para escribir un conjunto dinámico basado 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

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

21. Escribe una batería de pruebas de caja blanca para el TAD Conjunto (estático) - haz énfasis en los aspectos que no puedes comprobar con las pruebas de caja negra: todos los casos límites de intentar más elementos de los que caben, o intentar eliminar elementos cuando no hay ninguno.

22. Escribe una batería de pruebas de caja blanca para el TAD Conjunto (dinámico, según el ejercicio 19). ¿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"?.

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

[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.

Documentando TADs

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

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

Brassard, G. y Bratley, P. *Fundamentos de Algoritmia*. Prentice Hall, 1997.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. y Stein, C. *Introduction to Algorithms*. MIT Press, 2nd edición, 2001.

Martí Oliet, N., Ortega Mallén, Y. y Verdejo López, J. A. *Estructuras y datos y métodos algorítmicos: 213 Ejercicios resueltos*. Ibergarceta Publicaciones, 2013.

Rodríguez 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.