Departamento
de Ingeniería de la
Información y las
Comunicaciones
UMU

Tecnología de la Programación

Título de Grado en Ingeniería Informática

Curso 2024/25

Grupos 2, 3 y 4

Tema 1. Abstracción de Datos

Dept. Ingeniería de la Información y las Comunicaciones

Universidad de Murcia

Tema 1. Abstracción de Datos

1.1 Abstracción en Programación

1.1.1 Concepto de Abstracción

Definición de *abstracción* (RAE): Acción y efecto de abstraer o abstraerse.

Definición de *abstraer* (RAE): Separar por medio de una operación intelectual las cualidades de un objeto para considerarlas aisladamente o para considerar el mismo objeto en su pura esencia o noción.

La **abstracción** es un proceso mental que consiste en **realzar los detalles relevantes** que interesan sobre el objeto de estudio mientras se **ignoran los detalles irrelevantes**, lo cual nos lleva a una **simplificación del problema**. Cada día utilizamos la abstracción en prácticamente todas las **actividades mentales**. para entender, explicar, conceptualizar, etc. Cuando se usa una palabra, por ejemplo, "mesa", no es necesario que se digan las características de una mesa, de manera que éstas se abstraen en un solo término, al cual llamamos "mesa". Otros ejemplos de abstracción en la **vida cotidiana** son los mapas, las señales de tráfico, los acrónimos, los emoticonos, etc.

1.1.2 Tipos de abstracción en programación

En ***programación de ordenadores***, la abstracción consiste básicamente en aislar un elemento de su contexto. La abstracción enfatiza el **¿qué hace?** sobre el **¿cómo se hace?**. Este proceso es fundamental para **manejar sistemas complejos** que contienen múltiples detalles y relaciones. Un programador no necesita mencionar todas las características de un proceso u objeto cada vez que éste se utiliza, sino que son declarados por separado en el programa y simplemente se utiliza su término abstracto para usarlos.

Dada la complejidad de los programas actuales, **la evolución de los lenguajes de programación** está marcada por un **uso creciente de la abstracción**.

En programación de ordenadores, podemos distinguir tres formas fundamentales de abstracción:

- **Abstracción de control**
- **Abstracción funcional**
- **Abstracción de datos**

1.1.2.1 Abstracción de control

Establece nuevos mecanismos de control sencillos ocultando los detalles internos más complejos de su implementación.

Ejemplo: Las sentencias `while`, `for` y `do while` usan internamente instrucciones `goto` y condicionales en sus implementaciones a más bajo nivel.

1.1.2.2 Abstracción funcional

Permite separar el propósito de una función de su implementación. Se considera el qué hace una función, obviando el cómo lo hace. En la abstracción funcional abstraemos un conjunto preciso de operaciones como una única operación en forma de función. De esta forma, los algoritmos que requieran muchas operaciones se engloban en una sóla, eliminándose así los detalles de cómo se implementa.

Ejemplo:

Operación	`int factorial(int n)`

Detalles (implementación)	`{`
	`  if (n==0) return 1;`
	`  return n*factorial(n-1);`
	`}`

1.1.2.3 Abstracción de datos

Permite crear nuevos tipos de datos (***tipos de datos abstractos***) para mejorar la representación de los datos del problema.

La abstracción de datos se refiere al proceso conjunto de **definir**, **implementar** y **utilizar tipos de datos abstractos** en el **desarrollo de programas**.

Ejemplo:

Podemos construir el nuevo tipo de dato Racional mediante el uso de estructuras de datos y abstracciones funcionales asociadas a cada una de las operaciones del tipo Racional, como por ejemplo, sumar, restar, simplificar, etc.

1.2 Concepto de Tipo de Dato Abstracto

1.2.1 Motivación del uso de TDAs

1.2.2 Proceso de definición, implementación y uso de TDAs

1.2.3 Clasificación de TDAs

1.2.1 Motivación del uso de TDAs (1/4)

Tipo de dato abstracto:

Colección de valores + Operaciones

Definición:

*Un **tipo de dato abstracto (TDA)** es una colección de valores y de operaciones que se definen mediante una especificación que es independiente de cualquier representación.*

1.2.1 Motivación del uso de TDAs (2/4)

El concepto de **TDA** fue introducido por *John Guttag* en 1974 y posteriormente *Barbara Liscov* en el 1975 lo propuso para el lenguaje *CLU*. Un TDA es un tipo de datos caracterizado por un **conjunto de operaciones** (que constituyen la **interfaz pública**) el cual representa el comportamiento del tipo, y se define mediante una **especificación**. La **implementación** de un TDA es la traducción a un código en un lenguaje de programación concreto del comportamiento descrito en la especificación. La implementación de un TDA es privada, permaneciendo oculta al programa cliente que lo usa.

Un TDA cumple las siguientes **dos propiedades**:

- **Privacidad de la representación:** Los usuarios del TDA no conocen la representación de sus valores en la memoria.
- **Protección:** Los datos del TDA sólo pueden ser manipulados a través de las operaciones previstas por la especificación.

1.2.1 Motivación del uso de TDAs (3/4)

Consideraciones:

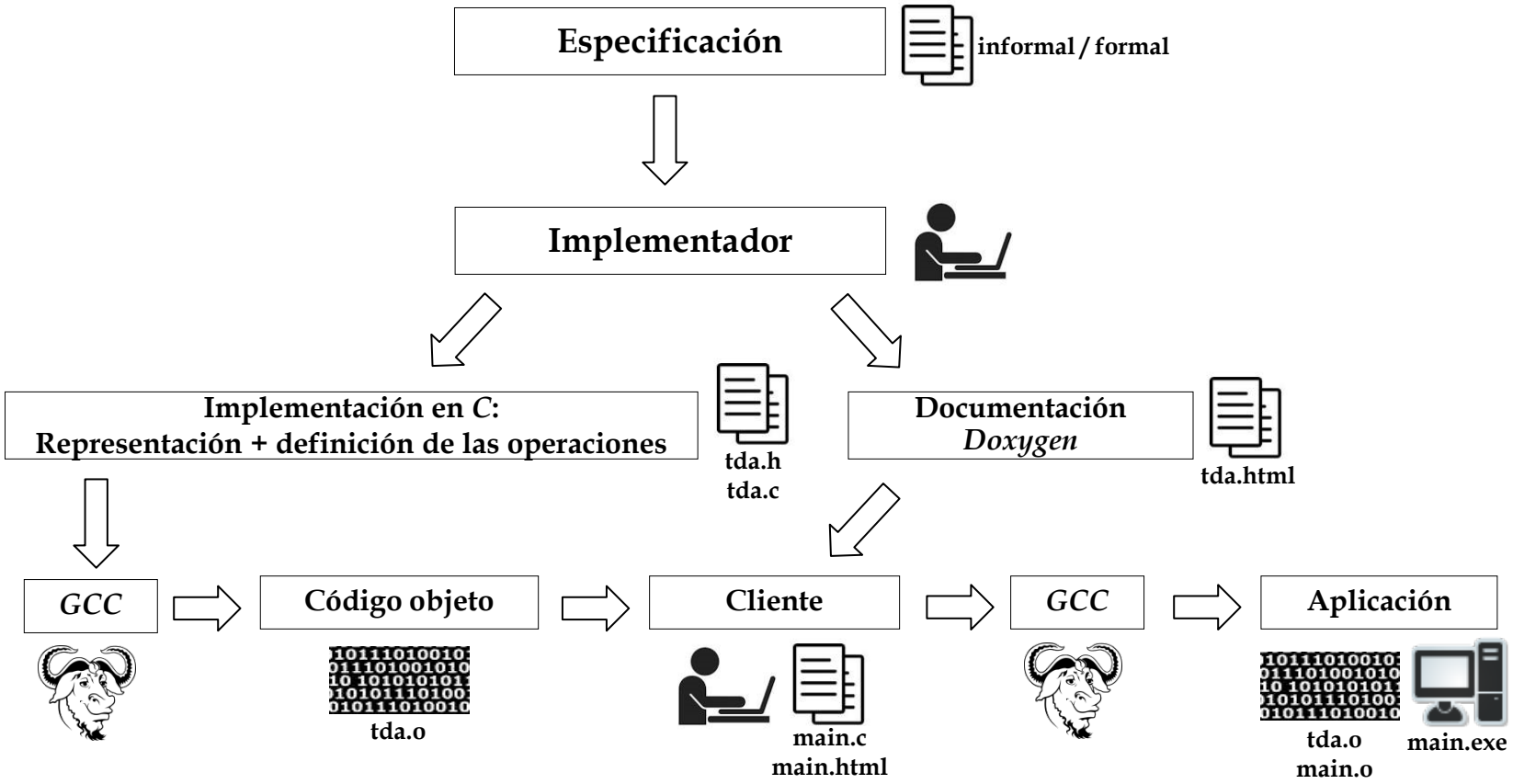
- Para manejar el TDA, el usuario no necesita conocer los detalles de la representación o la forma en que se implementan las operaciones. Sólo es necesario conocer un "manual de uso", al que más adelante haremos referencia como **documentación**.
- Para implementar el TDA, además de los aspectos de eficiencia, sólo es importante respetar que el comportamiento sea el especificado (según su **especificación**).
- Los tipos que incorpora un lenguaje de programación (en *C*, tipos `int`, `float`, `double`, `char`, `struct`, `enum`, `union`, array) son considerados como TDAs, ya que cumplen las propiedades de protección y privacidad. La idea fundamental en la construcción de nuevos TDAs por parte del programador radica en conseguir que esos nuevos tipos creados **sean tratados por el lenguaje como sus propios tipos**.

1.2.1 Motivación del uso de TDAs (4/4)

Ventajas del uso de TDAs:

- **Facilidad de uso:** No se requiere conocer los detalles internos del TDA, sólo la documentación, que es mucho más simple.
- **Desarrollo y mantenimiento:** Desarrollar y modificar el código es mucho más sencillo, ya que al establecer una interfaz pública, cualquier cambio interior al tipo de dato abstracto no afecta al resto del programa. De forma similar, cualquier cambio en el programa que siga respetando la interfaz no afectará al TDA. Además, localizar los errores es mucho más simple, ya que es más fácil su aislamiento.
- **Reusabilidad:** El TDA se puede usar en distintos programas.
- **Fiabilidad:** El código es más fiable, ya que es más fácil realizar pruebas sobre los módulos de forma independiente.

1.2.2 Proceso de definición, implementación y uso de TDAs



1.2.3 Clasificación de TDAs (1/2)

TDAs Simples

- Cambian su valor pero no su estructura:
  - Espacio de almacenamiento constante.
  - **Ejemplos:** Racional, complejo, punto, línea, cuadrado, cubo, etc.

TDAs Contenedores

- Cambian su valor y estructura:
  - Colecciones de elementos de número variable.
  - Espacio de almacenamiento variable.
  - **Ejemplos:** Lista, cola, pila, conjunto, árbol, grafo, polinomio, etc.

1.2.3 Clasificación de TDAs (2/2)

TDAs Inmutables

- Sus casos no pueden modificarse.
- No existen operaciones de modificación.

TDAs Mutables

- Sus casos pueden modificarse.
- Existen operaciones de modificación.

1.3 Especificación de TDAs

1.3.1 Clasificación de especificaciones

Especificaciones informales

- Lenguaje natural.
- Poco precisas, poco breves.
- Pueden producir ambigüedad.
- Sencillas de escribir, leer y entender.

Especificaciones formales

- Lenguaje algebraico.
- Precisas y breves.
- Permiten verificación formal.
- Complejas de escribir, leer y entender.

1.3.2 Especificaciones informales (1/5)

Contienen dos partes:

Definición:

En esta parte se define el nuevo TDA, así como todos los términos relacionados que sean necesarios para comprender el resto de la especificación.

Se define el dominio en el que tomará valores una instancia o caso del TDA. Para ello se podrán utilizar tipos de datos conocidos y otros TDAs, así como notaciones matemáticas, gráficos, y en general cualquier información que ayude a la descripción e interpretación del TDA.

Se podrá incluir una reseña acerca de la **mutabilidad** del TDA.

Operaciones:

En esta parte se especifican las operaciones, tanto sintáctica como semánticamente.

1.3.2 Especificaciones informales (2/5)

Operaciones

Para cada una de las operaciones del TDA, debe incluirse:

Sintaxis:

- Nombre de la operación.
- Nombre y tipos de los parámetros de entrada, salida y/o entrada/salida.
- Tipo de la salida.

En este documento se utilizará la sintaxis de C para los prototipos de las funciones.

Semántica:

- Restricciones de uso (precondiciones).
- Efecto de la operación.

1.3.2 Especificaciones informales (3/5)

Ejemplo: TDA Racional

Definición

Los números racionales son el conjunto de números fraccionarios y números enteros representados por medio de fracciones. Una **fracción** se expresa de la forma a / b con $a, b \in Z, b \neq 0$, donde a se llama **numerador** y b **denominador**. El denominador indica las partes iguales en que se divide a la unidad y el numerador las partes que tomamos. El **valor** de una fracción es el resultado de dividir numerador entre denominador. Se representa por:

$$Q = \left\{ \frac{a}{b} \mid a, b \in Z, b \neq 0 \right\}$$

1.3.2 Especificaciones informales (4/5)

Ejemplo: TDA Racional (Mutable)

Operaciones

`Racional crea(int n, int d)`
precondición: `d` ≠ 0.
efecto: devuelve un número racional con numerador `n` y denominador `d`.

`int numerador(Racional r)`
efecto: devuelve el numerador del número racional `r`.

`int denominador(Racional r)`
efecto: devuelve el denominador del número racional `r`.

`void suma(Racional r1, Racional r2)`
efecto: suma el número racional `r2` al número racional `r1`.

`void resta(Racional r1, Racional r2)`
efecto: resta el número racional `r2` al número racional `r1`.

`void producto(Racional r1, Racional r2)`
efecto: multiplica el número racional `r2` al número racional `r1`.

`void division(Racional r1, Racional r2)`
precondición: el numerador de `r2` es distinto de `0`.
efecto: divide el número racional `r2` al número racional `r1`.

`void simplifica(Racional r)`
efecto: simplifica el número racional `r`.

1.3.2 Especificaciones informales (5/5)

Ejemplo: TDA Racional (Inmutable)

Operaciones

`Racional crea(int n, int d)`
precondición: `d` ≠ 0.
efecto: devuelve un número racional con numerador `n` y denominador `d`.

`int numerador(Racional r)`
efecto: devuelve el numerador del número racional `r`.

`int denominador(Racional r)`
efecto: devuelve el denominador del número racional `r`.

`Racional suma(Racional r1, Racional r2)`
efecto: devuelve la suma del número racional `r1` y el número racional `r2`.

`Racional resta(Racional r1, Racional r2)`
efecto: devuelve la resta del número racional `r1` y el número racional `r2`.

`Racional producto(Racional r1, Racional r2)`
efecto: devuelve el producto del número racional `r1` y el número racional `r2`.

`Racional division(Racional r1, Racional r2)`
precondición: el numerador de `r2` es distinto de `0`.
efecto: devuelve la división del número racional `r1` por el número racional `r2`.

`Racional simplifica(Racional r)`
efecto: devuelve la simplificación del número racional `r`.

1.3.3 Especificaciones formales (1/2)

Las especificaciones formales permiten establecer un sistema de comunicación claro, simple y conciso. Por otro lado permiten deducir formalmente propiedades que satisface el tipo y posibilitan la verificación formal de programas.

Contienen tres partes:

Tipo: Nombre del TDA

Sintaxis: Forma de las operaciones

nombre de la función (tipo de los argumentos) → tipo del resultado

Semántica: Significado de las operaciones

nombre de la función (valores particulares) ⇒ expresión del resultado

1.3.3 Especificaciones formales (2/2)

Ejemplo: TDA Polinomio

Tipo: `Polinomio`

Sintaxis:

```
crea() → Polinomio
suma(Polinomio,Real,Natural) → Polinomio
evalua(Polinomio,Real) → Real
```

Semántica: $\forall$ `P` $\in$ `Polinomio`, $\forall$ `x,c` $\in$ `Real`, $\forall$ `e` $\in$ `Natural`:

$$\texttt{evalua(crea(),x)} \Rightarrow 0$$

$$\texttt{evalua(suma(P,c,e),x)} \Rightarrow \texttt{evalua(P,x)} + c \cdot x^{e}$$

1.3.4 Selección de operaciones (1/6)

El conjunto de operaciones de un TDA **no es único**, ya que se puede optar por un conjunto distinto de operaciones teniendo en cuenta las **siguientes consideraciones:**

- Un TDA definido sobre un módulo debe representar **un único modelo de datos**.
- El conjunto de operaciones debe ser **suficiente, pero no obligatoriamente mínimo**.
- Podría ser conveniente añadir a las operaciones básicas nuevas operaciones no básicas si existe alguno de estos **motivos**:
  - La función va a ser **muy utilizada**.
  - La función va a ser utilizada con cierta asiduidad y su **eficiencia empeora si se implementa haciendo uso de las operaciones básicas**.

1.3.4 Selección de operaciones (2/6)

Según el lenguaje de programación sobre el que recaiga la implementación del TDA, se pueden requerir nuevas operaciones para, por ejemplo, **liberación de memoria** o **gestión de errores** (sección 1.4.4). Dado que este tipo de operaciones dependen del lenguaje de programación utilizado, no se incluyen en la especificación del TDA, aunque sí se deberán incluir en la documentación una vez implementado el TDA.

También pueden incluirse operaciones de **gestión de datos del módulo** para acceder a datos que son comunes a todos los casos del TDA. Estas operaciones definidas sobre el módulo en lugar de sobre los casos del TDA se llaman **estáticas**.

La **sintaxis de las operaciones** se puede modificar para hacer eficiente su implementación. Por ejemplo, un paso de parámetro por valor puede ser ineficiente por su tamaño, siendo lo recomendado un paso por referencia.

Un TDA está sujeto a **mantenimiento**. Desde este punto de vista es menos costoso la adición de nuevas operaciones que la eliminación de operaciones ya existentes por lo que resulta conveniente retrasar la adición de nuevas operaciones cuya necesidad sea dudosa.

1.3.4 Selección de operaciones (3/6)

Consideraciones sobre los nombres de las operaciones

Dado que C no puede encapsular datos y operaciones en una misma entidad (clase) surgen problemas cuando una aplicación trabaja con distintos TDAs que contienen nombres iguales para sus operaciones. Por ejemplo, las operaciones `crea` y `libera` aparecerán en todos los TDAs.

Para salvar este problema, una opción posible consiste en hacer distinción de los nombres de las operaciones, para lo cual podemos seguir los siguiente criterios:

- **TDAs simples:** Se antepone el nombre del TDA al nombre de la operación. Ejemplo: `RacionalCrea`
- **TDAs contenedores:** Se antepone el nombre del TDA y el nombre del tipo de sus elementos al nombre de la operación. Ejemplo: `ConjuntoEnteroCrea`

Nota: Por simplicidad y legibilidad de código, en este documento no se tendrán en cuenta estas consideraciones sobre nombres.

1.3.4 Selección de operaciones (4/6)

Consideraciones sobre los nombres del TDA

El nombre debe identificar el tipo de forma adecuada sin indicar nada sobre la representación interna del mismo.

Consideraciones sobre las operaciones del TDA

Las operaciones deben devolver un caso del TDA, recibirlo como parámetro, o ambos (excepto en operaciones estáticas). También es posible incluir operaciones que reciban más de un caso del TDA y devuelvan otro caso construido a partir de ellos.

1.3.4 Selección de operaciones (5/6)

TDAs simples:

Es recomendable incluir operaciones tipo `get` y `set` (**acceso y modificación**) para acceder a los campos de la estructura privada.

Si el TDA representa un modelo matemático (grupo, cuerpo, anillo, álgebra, etc.) es usual incluir en el TDA las **operaciones definidas sobre el modelo matemático** (ejemplo, en el TDA Racional las operaciones de suma, resta, multiplicación, división, simplificación, etc.).

Si el TDA representa un modelo geométrico (punto, recta, círculo, triángulo, etc.) usualmente se incluyen las **operaciones propias asociadas al modelo** (distancia entre dos puntos, punto de intersección de dos rectas, área de un círculo o triángulo, etc.).

En aplicaciones gráficas en movimiento (por ejemplo, TDA Nave) es usual incluir **operaciones de movimiento** a partir de una posición y una velocidad (módulo), estableciéndose una nueva posición del objeto gráfico.

1.3.4 Selección de operaciones (6/6)

TDAs contenedores:

En TDAs específicos (por ejemplo TDA Agenda) es habitual incluir operaciones de **búsqueda, sustitución o eliminación** (insertar un nuevo contacto, eliminar un contacto, buscar un contacto, modificar los datos de un contacto, etc.).

En TDAs contenedores genéricos (Listas, Árboles, Grafos, etc.) es habitual incluir **operaciones de posicionamiento** para poder realizar recorridos o acceso a datos individuales (en listas: primero, siguiente, anterior, fin; en árboles: raiz, hijoizquierdo, hijoderecho, hermanoderecho, padre).

Al igual que en los TDAs simples, si el TDA representa un modelo matemático (por ejemplo, TDA Polinomio) es usual incluir en el TDA las **operaciones definidas sobre el modelo matemático** (suma, resta, multiplicación, evaluación, simplificación, etc.).

En aplicaciones gráficas en movimiento (por ejemplo, TDA Ejercito) es usual incluir **operaciones de movimiento** que mueven todos los elementos del conjunto.

1.4 Implementación de TDAs en C

1.4.1 La fase de implementación (1/2)

El resultado de la ***fase de implementación*** es doble:

- Por un lado se escribe el **código**, en un lenguaje de programación concreto, que implementa el comportamiento descrito en la especificación.

- Por otro lado se genera la **documentación del software** realizado (usualmente con **generadores automáticos de documentación**).

1.4.1 La fase de implementación (2/2)

Para obtener el código se debe identificar:

1. **Representación:**

Una forma de estructurar la información de manera que se puedan representar todos los valores del tipo de dato abstracto de una manera eficiente.

2. **Implementación de las operaciones:**

Un código eficiente que realice el comportamiento establecido en la especificación.

1.4.2 Representación

1.4.2.1 Tipo *rep*

1.4.2.2 Función de abstracción

1.4.2.3 Invariante de la representación

1.4.2.4 Ejemplos

1.4.2.1 TIPO *REP*

Para implementar un TDA es necesario, en primer lugar, escoger una representación de sus valores. Se debe seleccionar un **tipo de datos adecuado** que permita representar **todos los posibles valores del TDA de forma eficiente**. A este tipo se le denominará ***tipo rep*** en este documento.

Ejemplo: En el TDA Racional, el tipo *rep* puede ser:

- Un `struct` con dos miembros enteros para almacenar numerador y denominador.

- Un array de dos elementos enteros que representan numerador y denominador respectivamente.

1.4.2.2 Función de abstracción (1/2)

Se debe establecer una relación entre el tipo *rep* y el tipo de datos que se está definiendo. Para ello se define una función entre los valores que se pueden representar en el tipo *rep* y los valores en el tipo abstracto que le corresponden. A esta función se le denomina ***función de abstracción.***

$$f_{abs} : rep \rightarrow \mathcal{A}$$

La función de abstracción es una ***aplicación sobreyectiva no inyectiva***, es decir, cada elemento origen se aplica a un único elemento abstracto (aplicación), cada valor abstracto tiene al menos una representación (sobreyectiva) y, dos representaciones pueden corresponder al mismo elemento abstracto (no inyectiva).

1.4.2.2 Función de abstracción (2/2)

Caracterización de la función de abstracción:

- Es *parcial*, ya que no todos los valores que podemos representar corresponden a valores del tipo de dato abstracto. **Ejemplo:** La representación de la estructura con campo denominador igual a cero no corresponde a ningún número racional.
- Todos los elementos del tipo de dato abstracto deben tener una representación, es decir, **un elemento origen** en esta función.
- **Varios valores** de la representación podrían representar un **mismo valor** del tipo de dato abstracto. **Ejemplo:** el valor con numerador 2 y denominador 4 representan el mismo racional que el valor con numerador 1 y denominador 2.

1.4.2.3 Invariante de la representación

Para indicar el significado de la representación, es necesario establecer el conjunto de valores de representación que son válidos, es decir, que representan a un TDA. Es por tanto necesario establecer una **condición** sobre el conjunto de valores del tipo *rep* que nos indique si corresponden a un valor válido. Esta condición se denomina ***invariante de la representación***:

$$f_{inv} : rep \rightarrow boolean$$

El invariante de la representación **es siempre cierto** para la representación de **cualquier valor del TDA**.

Se debe asegurar que los valores construidos con las operaciones del TDA cumplan siempre el invariante de la representación.

1.4.2.4 EJEMPLOS (1/2)

1. **TDA Racional**

Representación:

```
struct RacionalRep
{
    int num, den;
};
typedef struct RacionalRep Racional;
```

Invariante de la representación:

$r.den \neq 0$

Función de abstracción:

$$f_{abs} : rep \rightarrow Racional$$
$$r \rightarrow \{ r.num / r.den \}$$

1.4.2.4 EJEMPLOS (2/2)

2. TDA Conjunto

Representación

```
struct ConjuntoRep
{
    int n;        // Número de elementos
    int * elem;   // Vector de elementos
};
typedef struct ConjuntoRep Conjunto;
```

Invariante de la representación:

$$r.elem[i] \neq r.elem[j] \ \forall\ i,j \ t.q.\ 0 \leq i < j < r.n$$

Función de abstracción:

$$f_{abs} : rep \rightarrow Conjunto$$
$$r \rightarrow \{ r.elem[i] \ t.q.\ 0 \leq i < r.n \}$$

1.4.3 Mecanismos de ocultación en *C* (1/7)

El lenguaje *C* **no permite** ***encapsulamiento***. Con el encapsulamiento se juntan los datos y las operaciones que los manipulan (concepto de *clase* en lenguajes orientados a objetos, como C++ o Java) para mantenerlos aislados de posibles interferencias o usos indebidos. De esta forma, el acceso a los datos se realiza de una forma controlada a través de una interface pública definida.

Con el encapsulamiento se consigue ***ocultar información***. Se ocultan todos los detalles que no contribuyen a sus características esenciales, como son el **tipo usado para la representación** y el **código de las operaciones**, siendo la **interface** lo único visible al exterior.

No obstante, *C* permite ciertos mecanismos para ocultar información mediante el uso de ***programación modular***, ***tipos incompletos*** o ***apuntadores a void.*** Este último mecanismo es normalmente usado en *C* para ***genericidad***, que no será tratada en este curso. La genericidad es un mecanismo usado en programación para crear herramientas de propósito general y posteriormente especializarlas para situaciones concretas.

1.4.3 Mecanismos de ocultación en *C* (2/7)

La implementación de cada TDA recaerá en un módulo que puede **compilarse de forma independiente**. El **fichero de cabecera .h** contendrá la **parte pública**, es decir, el **nombre del tipo** y los **prototipos de las funciones**, mientras que el **fichero .c** contendrá la **parte privada**, que será la **representación interna del tipo abstracto** y la **implementación de las operaciones**. El proceso de compilación del módulo genera un **fichero objeto .o**. Los comandos asociados a la **documentación del TDA** se incluirán en la parte pública (fichero .h).

El usuario del TDA usará el fichero .h y el fichero .o, por lo que **la implementación del tipo queda oculta**. Las funciones implementadas en el fichero .c cuyo prototipo no aparezca en el fichero .h se considerarán como **funciones privadas**, mientras que el resto de funciones serán **funciones públicas**.

Público (fichero cabecera .h):

- Nombre del TDA.
- Interface, prototipos de las funciones públicas.
- Documentación.

Privado (fichero .c):

- Representación interna del TDA.
- Implementación de las funciones públicas y privadas.

1.4.3 Mecanismos de ocultación en *C* (3/7)

Una forma de **ocultar la representación de un TDA** es mediante el uso de ***tipos incompletos***.

De esta forma, **el fichero .h sólo define el nombre del tipo *rep* (tipo incompleto), y en el fichero .c se completa el tipo *rep* con el resto de su definición**.

Por otra parte, y por razones de eficiencia, resulta conveniente declarar (o renombrar) el nombre del TDA como **un apuntador a su tipo *rep***. Esto permite una gestión eficiente de los recursos tanto de CPU como de memoria ya que en las operaciones se pasarán, como parámetros, las direcciones de memoria de las variables del tipo, en lugar de copias de las variables.

Puesto que el TDA se construye como un apuntador a una variable dinámica que representa un caso del TDA, las operaciones que construyen nuevos valores del TDA devolverán un **valor `NULL`** cuando no se pueda construir un valor válido. Además será necesario también incluir una **operación pública `libera`** para liberar la memoria asignada en su construcción.

1.4.3 MECANISMOS DE OCULTACIÓN EN *C* (4/7)

En este ejemplo se muestran los ficheros **Racional.h** y **Racional.c** que definen el TDA Racional. En Racional.h se define el tipo *rep* como el tipo incompleto `struct RacionalRep`, y el nombre del TDA (`Racional`) está definido como un apuntador al tipo *rep*. El TDA es inmutable. No hemos incluido algunas operaciones.

Racional.h

```
#ifndef __RACIONAL_H
#define __RACIONAL_H
typedef struct RacionalRep * Racional;
Racional crea(int n, int d);
void libera(Racional r);
int numerador(Racional r);
int denominador(Racional r);
Racional simplifica(Racional r);
Racional suma(Racional r1, Racional r2);
#endif
```

1.4.3 MECANISMOS DE OCULTACIÓN EN *C* (5/7)

En el fichero Racional.c se completa el tipo `struct RacionalRep` con la definición de sus miembros, y se implementan las operaciones, incluyendo operaciones privadas como `mcd` (*máximo común divisor*, el cual se requiere para la operación `simplifica`).

Racional.c

```
#include "Racional.h"
#include <stdlib.h>
struct RacionalRep
{
    int num, den;
};
int mcd(int n, int m) // operación privada, algoritmo de Euclides
{
    if (n<0) n=-n;
    if (m<0) m=-m;
    if (n<m) { int t=m;  m=n; n=t; }
    while(m>0) { int t=n; n=m; m=t%m; }
    return n;
}
```

1.4.3 Mecanismos de ocultación en *C* (6/7)

Racional.c

```
Racional crea(int n, int d)
{
     if (d==0) return NULL;
     Racional r = malloc(sizeof(struct RacionalRep));
     r->num = n;
     r->den = d;
     return r;
}
void libera(Racional r)
{
     free(r);
}
```

1.4.3 Mecanismos de ocultación en *C* (7/7)

Racional.c

```
int numerador(Racional r)
{
    return r->num;
}
int denominador(Racional r)
{
    return r->den;
}
Racional simplifica(Racional r)
{
    int m = mcd(r->num,r->den);
    return crea(r->num/m,r->den/m);
}
Racional suma(Racional r1, Racional r2)
{
    int num = r1->num * r2->den + r2->num * r1->den;
    int den = r1->den * r2->den;
    return crea(num,den);
}
```

1.4.4 GESTIÓN DE ERRORES (1/7)

Cuando una operación de un TDA tiene precondiciones, es el usuario del TDA el responsable de comprobar que éstas se cumplen antes de realizar la llamada a la operación. No obstante, una implementación puede auto protegerse frente a posibles errores, comprobando que se cumplen las precondiciones de las operaciones e informando de una situación de error cuando éstas se violan.

Para la comunicación de las situaciones de error entre un TDA y su usuario, otros lenguajes de programación, como *C++* o *Java*, disponen de mecanismos automáticos denominados ***manejadores de excepciones***. No es el caso de *C*, por lo que **el programador deberá implementar sus propios mecanismos de manejo de errores**. Dos de estos mecanismos utilizados en algunas bibliotecas de *C* son los siguientes:

a) En el fichero .h se declara una variable de tipo `extern int` (llamada por ejemplo `error`) que almacene un código de error asociado al último error ocurrido, y una función (por ejemplo con prototipo `char * mensajeError(int codigoError)`) que devuelva el mensaje de error asociado a un código de error pasado como parámetro a la función. En el fichero .c se declara la variable `error` de tipo `int` y se implementa la función `mensajeError`. En este caso, tanto la variable `error` como la función `mensajeError` son públicas.

b) En el fichero .h se declara una función (por ejemplo con prototipo `char * mensajeError()`) que devuelve el mensaje de error asociado al último error ocurrido. En el fichero .c se declara una variable `static int error` y se implementa la función `mensajeError`, la cual accede a la variable `error` para devolver su correspondiente mensaje. En este caso la función `mensajeError` es pública mientras que la variable `error` es privada al fichero en el que se ha declarado.

1.4.4 Gestión de errores (2/7)

Ejemplo: TDA Triángulo. Especificación

Definición

El TDA Triángulo define triángulos.

Operaciones

```
Triangulo crea(double x1, double x2, double y1, double y2, double
    a, double b);
```

precondición: `(x1,y1)` ≠ `(x2,y2)`, `a>0`, `b>0`, `a+b<180`.
efecto: Devuelve un triángulo definido por los puntos (`x1,y1`) y (`x2,y2`) y los ángulos `a` y `b` en grados.

```
double area(Triangulo t);
```

efecto: Devuelve el área del triángulo `t`.

```
double perimetro(Triangulo t);
```

efecto: Devuelve el perímetro del triángulo `t`.

… resto de operaciones …

1.4.4 Gestión de errores (3/7)

TDA Triángulo. Fichero Triangulo.h

```
#ifndef __TRIANGULO_H
#define __TRIANGULO_H
typedef struct TrianguloRep * Triangulo;
extern int error;
/* Precondición: (x1,y1)!=(x2,y2) y a>0 y b>0 y a+b<180.
Efecto: Si se cumple la precondición, devuelve un nuevo triángulo con los puntos
     (x1,y1) y (x2,y2) y los ángulos a y b en grados. Si (x1,y1) es igual a
     (x2,y2), devuelve NULL y asigna error=1. Si a<=0, devuelve NULL y asigna
     error=2. Si b<=0, devuelve NULL y asigna error=3. Si a+b>=180 devuelve NULL
     y asigna error=4. */
Triangulo crea(double x1, double y1, double x2, double y2, double a, double b);
/* Efecto: Libera la memoria asociada al triángulo t. */
void libera(Triangulo t);
/* Efecto: Devuelve el error asociado al codigo de error. */
char * mensajeError(int codigoError);
/* Efecto: Devuelve el área del triángulo t. */
double area(Triangulo t);
/* Efecto: Devuelve el perímetro del triángulo t. */
double perimetro(Triangulo t);
#endif
```

1.4.4 Gestión de errores (4/7)

TDA Triángulo. Fichero Triangulo.c (1/2)

```
#include "Triangulo.h"
#include <stdlib.h>
#include <float.h>
#include <math.h>
int error;
struct TrianguloRep { double x1, y1, x2, y2, a, b; };
Triangulo crea(double x1, double y1, double x2, double y2, double a, double b)
{
      if ((fabs(x1-y1)<DBL_MIN)&&(fabs(x2-y2)<DBL_MIN))     { error = 1; return NULL;}
      if (a<=0)                                             { error = 2; return NULL;}
      if (b<=0)                                             { error = 3; return NULL;}
      if (a+b>=180)                                         { error = 4; return NULL;}
      Triangulo t = malloc(sizeof(struct TrianguloRep));
      t->x1 = x1;
      t->x2 = x2;
      t->y1 = y1;
      t->y2 = y2;
      t->a = a;
      t->b = b;
      return t;
}
```

1.4.4 Gestión de errores (5/7)

TDA Triángulo. Fichero Triangulo.c (2/2)

```
void libera(Triangulo t)
{
      free(t);
}

char * mensajeError(int codigoError)
{
      switch(codigoError)
      {
            case 1: return "Los dos puntos deben ser distintos";
            case 2: return "El primer angulo debe ser mayor que cero";
            case 3: return "El segundo angulo debe ser mayor que cero";
            case 4: return "La suma de los dos angulos debe ser menor que 180 grados";
      }
      return NULL;
}
double area(Triangulo t) { ... }

double perimetro(Triangulo t) { ... }

// ... resto de operaciones ...
```

1.4.4 Gestión de errores (6/7)

TDA Triángulo. Fichero Main.c

```
#include "Triangulo.h"
#include <stdio.h>

int main()
{
    Triangulo t;
    do
    {
        double x1,y1,x2,y2,a,b;
        printf("Introduzca valores del triangulo: ");
        scanf("%lf %lf %lf %lf %lf %lf",&x1,&y1,&x2,&y2,&a,&b);
        t = crea(x1,y1,x2,y2,a,b);
        if (t==NULL) fprintf(stderr,mensajeError(error));
    } while (t==NULL);
    // Operaciones sobre el triangulo t
    // ...
    libera(t);
    return 0;
}
```

1.4.4 Gestión de errores (7/7)

Nota:

Por legibilidad y simplicidad de código, en este documento no se presentarán implementaciones que incluyan gestión de errores. Se asume que el usuario de la abstracción comprueba que se cumplen las precondiciones de las operaciones antes de usarlas.

1.5 Ejercicios resueltos (1/19)

Objetivos:

- Interpretar especificaciones de TDAs.
- Implementar TDAs simples partir de su especificación.
- Implementar TDAs con representación contigua partir de su especificación.

1.5 EJERCICIOS RESUELTOS (2/19)

1) Implementar en *C* el TDA Monomio con el siguiente fichero de cabecera:

```
#ifndef MONOMIO_H
#define MONOMIO_H
typedef struct MonomioRep * Monomio;
// Crea un nuevo monomio con grado g y coeficiente c.
Monomio MonomioCrea(int g, double c);
// Libera el monomio m.
void MonomioLibera(Monomio m);
// Evalúa en x el monomio m.
double MonomioEvalua(Monomio m, double x);
// Devuelve el grado del monomio m.
int MonomioGetGrado(Monomio m);
// Devuelve el coeficiente del monomio m.
double MonomioGetCoeficiente(Monomio m);
#endif // MONOMIO_H
```

Nota: La función `double pow(double x, double y)` de `math.h` devuelve x^y.

1.5 Ejercicios resueltos (3/19)

```
#include <stdlib.h>
#include <math.h>
#include "Monomio.h"
struct MonomioRep{
    int g;
    double c;
};
Monomio MonomioCrea(int g, double c) {
    Monomio m = malloc(sizeof(struct MonomioRep));
    m->g = g;
    m->c = c;
    return m;
}
void MonomioLibera(Monomio m)
{
    free(m);
}
double MonomioEvalua(Monomio m, double x)
{
    return m->c*pow(x,m->g);
}
int MonomioGetGrado(Monomio m)
{
    return m->g;
}
double MonomioGetCoeficiente(Monomio m)
{
    return m->c;
}
```

1.5 EJERCICIOS RESUELTOS (4/19)

2) Implementar en *C* el TDA Recta (en un fichero `Recta.c`) sujeto a la especificación y fichero cabecera (`Recta.h`) adjuntos. El TDA Recta utiliza un TDA Punto, cuya especificación y fichero cabecera (`Punto.h`) también se adjuntan.

```
#ifndef __RECTA_H
#define __RECTA_H
#include "Punto.h"
// Los casos del TDA Recta definen una recta en el plano de la forma
// y=ax+b donde a es la pendiente de la recta y b es el término
// independiente.
typedef struct RectaRep * Recta;
// Crea una nueva recta de la forma y=ax+b
Recta RectaCrea(double a, double b);
// Libera la recta r.
void RectaLibera(Recta r);
```

1.5 Ejercicios resueltos (5/19)

```
// Calcula la distancia de un punto p a una recta r.
double RectaDistancia(Recta r, Punto p);
// Devuelve el punto intersección de dos rectas r1 y r2 o NULL si son
// paralelas.
Punto RectaIntereseccion(Recta r1, Recta r2);
#endif
```

```
#ifndef __PUNTO_H
#define __PUNTO_H
//Los casos del TDA Punto definen puntos en el plano de la forma (x,y)
typedef struct PuntoRep * Punto;
// Crea un nuevo punto de coordenadas (x,y)
Punto PuntoCrea(double x, double y);
// Libera el punto p.
void PuntoLibera(Punto p);
// Devuelve la coordenada x del punto p.
double PuntoX(Punto p);
// Devuelve la coordenada y del punto p.
double PuntoY(Punto p);
#endif
```

1.5 EJERCICIOS RESUELTOS (6/19)

NOTAS: La distancia de un punto *p* definido por las coordenadas *(x,y)* a una recta *r* definida por la pendiente *a* y término independiente *b* se calcula de la siguiente forma:

$$dist(p,r) = \frac{|ax - y + b|}{\sqrt{a^2 + 1}}$$

Se pueden utilizar las funciones `fabs` y `sqrt` de la librería `math.h` con la siguiente sintaxis:

```
double fabs(double x); // Devuelve el valor absoluto de x
double sqrt(double x); // Devuelve la raíz cuadrada de x
```

La intersección de dos rectas r_1 y r_2 definidas por las pendientes a_1 y a_2 y términos independientes b_1 y b_2 respectivamente, suponiendo que r_1 y r_2 no son paralelas, es decir, $a_1 \neq a_2$, es un punto *p* cuyas coordenadas *(x,y)* pueden obtenerse resolviendo el sistema de ecuaciones resultante de igualar las dos ecuaciones de las rectas, de forma que la coordenada *x* puede calcularse como:

$$x = \frac{b_2 - b_1}{a_1 - a_2}$$

La coordenada *y* se puede calcular a partir de la coordenada *x* como $y = a_1 x + b_1$

1.5 EJERCICIOS RESUELTOS (7/19)

```
// Fichero Recta.c
#include <stdlib.h>
#include <math.h>
#include "Recta.h"
struct RectaRep
{
     double a, b;
};
Recta RectaCrea(double a, double b)
{
     Recta r = malloc(sizeof(struct RectaRep));
     r->a = a;
     r->b = b;
     return r;
}
void RectaLibera(Recta r)
{
     free(r);
}
```

1.5 Ejercicios resueltos (8/19)

```
double RectaDistancia(Recta r, Punto p)
{
     double a = r->a;
     double b = r->b;
     double x = PuntoX(p);
     double y = PuntoY(p);
     return fabs(a*x-y+b)/sqrt(a*a+1);
}
Punto RectaIntereseccion(Recta r1, Recta r2)
{
     double a1 = r1->a;
     double a2 = r2->a;
     if (a1==a2) return NULL;
     double b1 = r1->b;
     double b2 = r2->b;
     double x = (b2-b1)/(a1-a2);
     double y = a1*x+b1;
     return PuntoCrea(x,y);
}
```

1.5 EJERCICIOS RESUELTOS (9/19)

2) Una *tienda virtual* o *Webstore* es un sitio web que vende productos o servicios y por lo general tiene catálogo de productos y un carro de compra en línea. Implementar en *C* el TDA Catálogo de productos (en un fichero `Catalogo.c`) sujeto a la especificación y fichero cabecera (`Catalogo.h`) adjuntos, teniendo en cuenta las siguientes consideraciones:

- Debe utilizarse una representación contigua, con un array de productos.
- El tamaño del array se determina en la operación `CatalogoCrea` y no cambia posteriormente.
- Cada producto deberá contener el nombre del producto y su precio.
- Los nombres de los productos se almacenarán en cadenas de 20 caracteres.
- Para la implementación de la función `CatalogoNuevoProducto` se asume que se cumplen los requerimientos.

1.5 Ejercicios resueltos (10/19)

```
#ifndef __CATALOGO_H
#define __CATALOGO_H
typedef struct CatalogoRep * Catalogo;
// Crea un catálogo de productos vacío que puede contener un máximo de
// productos nmax.
Catalogo CatalogoCrea(int nmax);
// Libera un catálogo de productos.
void CatalogoLibera(Catalogo catalogo);
// Devuelve el precio de un producto en un catálogo, ó el valor 0.0 si el
// producto no existe en el catálogo.
float CatalogoPrecio(Catalogo catalogo, char * nombre);
// Cataloga un nuevo producto, con su nombre y precio, en un catálogo.
// Pre.: No existe previamente un producto con el mismo nombre en el
// catálogo.
// Pre.: El número de productos en el catálogo es menor que el número
// máximo.
void CatalogoNuevoProducto(Catalogo catalogo, char * nombre, float
precio);
#endif // __CATALOGO_H
```

1.5 Ejercicios resueltos (11/19)

```
#include <stdlib.h>
#include <string.h>
#include "Catalogo.h"
struct Producto
{
     char nombre[20];
     float precio;
};
struct CatalogoRep
{
     struct Producto * productos;
     int n;
};
Catalogo CatalogoCrea(int nmax)
{
     Catalogo c = malloc(sizeof(struct CatalogoRep));
     c->productos = malloc(sizeof(struct Producto)*nmax);
     c->n = 0;
     return c;
}
```

1.5 Ejercicios resueltos (12/19)

```
void CatalogoLibera(Catalogo catalogo)
{
      free(catalogo->productos);
      free(catalogo);
}
void CatalogoNuevoProducto(Catalogo catalogo, char * nombre, float precio)
{
      strcpy(catalogo->productos[catalogo->n].nombre,nombre);
      catalogo->productos[catalogo->n].precio = precio;
      catalogo->n++;
}
float CatalogoPrecio(Catalogo catalogo, char * nombre)
{
      int i=0;
      while((i<catalogo->n)&&(strcmp(catalogo->productos[i].nombre,nombre)))
            i++;
      if (i<catalogo->n)
            return catalogo->productos[i].precio;
      return 0.0;
}
```

1.5 Ejercicios resueltos (13/19)

3) Considérense los TDA Carta y TDA Baraja con las siguientes especificaciones y ficheros de cabecera (`Carta.h` y `Baraja.h`).

```
#ifndef __CARTA_H
#define __CARTA_H
#define NPALOS 4
#define NCARTAS 12
typedef struct CartaRep * Carta;
// Crea y devuelve una carta de la baraja española con el palo p y el número n
// Palos: 1 - Oros, 2 - Bastos, 3 - Espadas, 4 - Copas
// Número: 1 - 9, 10 - Sota, 11 - Caballo, 12 - Rey
// Devuelve NULL si el palo p<1 ó p>4 ó el número n<1 ó n>12
Carta CartaCrea(int p, int n);
// Libera la carta c
void CartaLibera(Carta c);
// Devuelve el palo de la carta c
int CartaPalo(Carta c);
// Devuelve el número de la carta c
int CartaNumero(Carta c);
#endif
```

1.5 Ejercicios resueltos (14/19)

```
#ifndef __BARAJA_H
#define __BARAJA_H
#include "Carta.h"
typedef struct BarajaRep * Baraja;
// Crea y devuelve una baraja española consistente en 48 cartas
// clasificadas en 4 palos y cada palo numerado del 1 al 12
Baraja BarajaCrea();
// Libera la baraja b
void BarajaLibera(Baraja b);
// Extrae y devuelve una carta de la baraja b de forma aleatoria
// Req: La baraja b no debe estar vacía
Carta BarajaExtrae(Baraja b);
// Devuelve 1 (cierto) si la baraja b está vacía y 0 (falso) en
// caso contrario
int BarajaVacia(Baraja b);
#endif
```

1.5 EJERCICIOS RESUELTOS (15/19)

3.1) Implementar en *C* el TDA Baraja, el cual usa el TDA Carta.

3.2) Programar una aplicación en *C* que, utilizando los TDA Carta y TDA Baraja, cree una baraja y realice un descarte aleatorio, extrayendo una a una todas las cartas de la baraja e imprimiendo por pantalla los valores enteros correspondientes al palo y número de la carta extraída.

Notas:

- Se debe elegir la estructura adecuada para la baraja, teniendo en cuenta que se trata de una baraja española que contiene siempre 4 palos y 12 números cada palo.
- Téngase en cuenta que es necesario liberar las cartas según se van descartando, así como la baraja al terminar la aplicación.
- Se dispone de una función `aleatorio`, para la generación de números enteros aleatorios, con la siguiente especificación y fichero de cabecera (`Aleatorio.h`)

1.5 Ejercicios resueltos (16/19)

```
#ifndef __ALEATORIO_H
#define __ALEATORIO_H
// Devuelve un entero aleatorio comprendido entre i y j.
// Req.: i<=j
int aleatorio(int i, int j);
#endif
```

```
#include <stdlib.h>
#include "Baraja.h"
#include "Aleatorio.h"
#include "Carta.h"
#define NCARTASTOTAL NPALOS*NCARTAS
struct BarajaRep {
     Carta cartas[NCARTASTOTAL];
     int n;
};
```

1.5 Ejercicios resueltos (17/19)

```
Baraja BarajaCrea()
{
     Baraja b = malloc(sizeof(struct BarajaRep));
     b->n = 0;
     for(int i=1; i<=NPALOS; i++)
          for(int j=1; j<=NCARTAS; j++)
          {
               b->cartas[b->n] = CartaCrea(i,j);
               b->n++;
          }
     return b;
}
void BarajaLibera(Baraja b)
{
     for(int i=0; i<b->n; i++) CartaLibera(b->cartas[i]);
     free(b);
}
```

1.5 Ejercicios resueltos (18/19)

```
Carta BarajaExtrae(Baraja b)
{
     int i = aleatorio(0,b->n-1);
     Carta carta = b->cartas[i];
     b->cartas[i] = b->cartas[b->n-1];
     b->n--;
     return carta;
}
int BarajaVacia(Baraja )
{
     return (b->n==0);
}
```

1.5 EJERCICIOS RESUELTOS (19/19)

```
#include <stdio.h>
#include <stdlib.h>
#include "Baraja.h"
int main()
{
     Baraja b = BarajaCrea();
     while(!BarajaVacia(b)) {
          Carta c = BarajaExtrae(b);
          printf("Extrae: palo = %d, numero = %d\n",CartaPalo(c),
          CartaNumero(c));
          CartaLibera(c);
          system("PAUSE");
     }
     BarajaLibera(b);
     system("PAUSE");
     return 0;
}
```

1.5 Ejercicios propuestos (1/6)

Objetivos:

- Interpretar especificaciones de TDAs.
- Implementar TDAs con representación enlazada partir de su especificación.
- Usar TDAs dada su especificación.

1.5 Ejercicios propuestos (2/6)

1) Implementar en *C* el TDA Grupo, que representa un grupo de Whatsapp, mediante una estructura enlazada lineal con el siguiente fichero de cabecera. El TDA Grupo debe utilizar el TDA Contacto cuyo fichero de cabecera se adjunta.

```
#ifndef GRUPO_H
#define GRUPO_H
#include "Contacto.h"
typedef struct GrupoRep * Grupo;
// Crea un nuevo grupo vacío.
Grupo GrupoCrea();
// Libera el grupo g.
void GrupoLibera(Grupo g);
// Inserta el contacto c en el grupo g.
// Pre: El contacto c no existe previamente en el grupo g.
void GrupoInserta(Grupo g, Contacto c);
// Elimina el contacto c del grupo g.
// Pre: El contacto c existe previamente en el grupo g.
void GrupoElimina(Grupo g, Contacto c);
// Envía el mensaje m a todos los contactos del grupo g.
void GrupoEnviaMensaje(Grupo g, char * m);
#endif // GRUPO_H
```

1.5 Ejercicios propuestos (3/6)

```
#ifndef CONTACTO_H
#define CONTACTO_H
typedef struct ContactoRep * Contacto;
// Crea un contacto con un nombre y un número de teléfono.
Contacto ContactoCrea(char * nombre, char * numero);
// Libera el contacto c.
void ContactoLibera(Contacto c);
// Envía el mensaje m al contacto c.
void ContactoEnviaMensaje(Contacto c, char * m );
// Devuelve 1 (cierto) si los contactos c1 y c2 son iguales
// y 0 (falso) en caso contrario.
int ContactoIgual(Contacto c1, Contacto c2);
#endif // CONTACTO_H
```

Nota: No hay que implementar el TDA Contacto, solamente utilizarlo para la implementación del TDA Grupo.

1.5 Ejercicios propuestos (4/6)

2) Implementar en *C* el TDA Carro de Compra (en un fichero `CarroCompra.c`) sujeto a la especificación y fichero cabecera (`CarroCompra.h`) adjuntos, teniendo en cuenta las siguientes consideraciones:
   - Debe usarse el TDA Catálogo de los ejercicios resueltos, 2).
   - Debe utilizarse una representación enlazada con simple enlace. Se sugiere el uso de una celda de encabezamiento al inicio de la estructura enlazada.
   - Cada celda debe contener el nombre del producto y el número de pedidos para ese producto.
   - Los nombres de los productos se almacenarán en cadenas de 20 caracteres.
   - No deben existir celdas con el mismo nombre de producto.
   - El número de pedidos en cada celda debe ser mayor que cero.

1.5 Ejercicios propuestos (5/6)

```
#ifndef __CARROCOMPRA_H
#define __CARROCOMPRA_H
#include "Catalogo.h"

typedef struct CarroCompraRep * CarroCompra;

// Crea un carro de compra vacío.
CarroCompra CarroCompraCrea();

// Libera el carro de compra.
void CarroCompraLibera(CarroCompra carroCompra);

// Añade un nuevo pedido de un producto en un carro de compra.
void CarroCompraRealizaPedido(CarroCompra carroCompra, char * nombre);

// Cancela un pedido de un producto en un carro de compra.
void CarroCompraCancelaPedido(CarroCompra carroCompra, char * nombre);
```

1.5 Ejercicios propuestos (6/6)

```
// Imprime los pedidos de un carro de compra de acuerdo a los nombres
// y precios de un catálogo indicando, para cada pedido, nombre del
// producto, precio del producto, número de pedidos y precio total para
// ese producto. Finalmente, se indica el precio total de la compra.
// Un ejemplo sería el siguiente:
// Portatil 635.50 x 1 = 635.50
// Impresora 129.80 x 3 = 389.40
// Tablet 238.30 x 2 = 476.60
// Total: 1501.50
void CarroCompraImprime(CarroCompra carroCompra, Catalogo catalogo);

#endif // __CARROCOMPRA_H
```