Abstracción en Programación

A. GARRIDO

Departamento de Ciencias de la Computación e I.A. ETS Ingeniería Informática
Universidad de Granada. 18071 Granada. Spain. Email: A.Garrido@decsai.ugr.es

Resumen

En este documento se pretende presentar brevemente los conceptos fundamentales de la abstracción, aplicados al desarrollo de programas. El objetivo es que el lector comprenda su importancia capital si se quieren obtener *buenos* programas. Se muestra como la evolución de las herramientas de programación ha sido acompañada de un creciente uso de este conjunto de conceptos, lo que confirma como una aproximación efectiva para enfrentarse a problemas de gran complejidad. Además, se propondrán distintas formas de aplicar estas ideas en programación, haciendo especial énfasis en los tipos de datos abstractos, como un medio eficaz para facilitar esta labor.

Se recomienda que el lector esté introducido en el lenguaje de programación C. A lo largo de la lección, justificado por el progresivo aumento de los mecanismos de abstracción que se presentan, se irán presentando características del lenguaje C++, que integra muchos de los conceptos que se discuten.

1 Introducción.

La abstracción es un proceso mental que consiste en realzar los detalles relevantes, es decir, los que nos interesan en un momento sobre el objeto de estudio, mientras se ignoran los detalles irrelevantes. Esto nos lleva a una simplificación del problema, ya que

- la cantidad de información que es necesario manejar en un momento dado disminuye y
- podemos tratar cosas diferentes como si fueran la misma

Este proceso de la mente humana es fundamental para comprender y manejar complejos sistemas que contienen multiples detalles y relaciones. Por ello, y dada la complejidad de los programas actuales, la evolución de los paradigmas y lenguajes de programación marca un uso creciente de la abstracción.

Cuando es un programa es pequeño (por ej. cientos de líneas de código), se puede obtener una solución basada en un único componente, en el que se pueden encontrar todos los detalles de ésta. Sin embargo, cuando el tamaño del problema aumenta (por ej. miles o cientos de miles de líneas de código), el ser humano es incapaz de manejar tal cantidad de detalles y es necesaria una descomposición en pequeñas partes independientes, que denominaremos **módulos** y que unidos constituyen la solución buscada. En esta lección, se presenta una metodología para la descomposición y construcción de módulos basada en la abstracción, consiguiendo con ello mejorar la calidad de los resultados y facilitar la construcción, modificación y mantenimiento del software.

1.1 Programación modular.

La programación modular constituye una metodología eficaz para abordar problemas de cualquier tamaño y la abstracción es una forma de llevarla a cabo. Sin embargo, es necesario considerar detenidamente las propiedades que deben cumplir para que la descomposición sea útil:

1. Las conexiones de cada módulo con el resto del programa deben ser mínimas, tanto en número como en complejidad, para obtener módulos más independientes.
2. Cada módulo lleva a cabo una tarea bien definida en el nivel de detalle correspondiente, para facilitar su solución y la independencia con el resto del programa.

3. La solución de cada subproblema debe ser general, para obtener módulos que, siendo igualmente útiles para el problema a resolver, puedan adaptarse a futuras modificaciones o incluso para otros problemas.

4. La solución de los subproblemas se puede combinar para resolver el problema original.

Una correcta aplicación de esta metodología, facilitará en gran medida la resolución del problema. Concretamente podemos destacar varios aspectos que ponen de relieve los beneficios que podemos obtener:

- Facilita el desarrollo de programa. La independencia entre las distintas partes del problema, permite que varios programadores puedan trabajar en equipo para resolverlo, minimizando las labores de coordinación necesarias para que se obtenga una solución correcta.
- Facilita el mantenimiento:
  - Facilita los cambios y mejoras. Este tipo de descomposición permite que la mayor parte de los cambios y mejoras se apliquen sobre un módulo o un número pequeño de ellos.
  - Facilita la prueba y depurado del software. El conjunto de tests para probar el software es más sencillo pues se puede analizar cada parte de manera independiente. Además, la detección y eliminación de errores se limitan a analizar un pequeño trozo del programa.
- Facilita la productividad evitando que un problema se resuelva múltiples veces:
  - Facilita la eliminación de redundacias. Es más fácil identificar los subproblemas que deben resolverse en distintas partes del programa.
  - Facilita la reutilización del software. El resultado no es una solución a un problema sino un conjunto de soluciones a múltiples subproblemas que se combinan en el programa que deseamos. Alguna de estas soluciones puede ser necesaria para resolver otros problemas.

1.2 Abstracción y ocultamiento de información.

La abstracción permite estudiar complejos sistemas usando un método jerárquico en sucesivos niveles de detalle. Así, un sistema se puede descomponer en distintos módulos. Dado que nuestra intención es manejar en un instante el menor número de características , cada módulo deberá exportar la menor cantidad de información siempre que con ello permita al resto de los módulos realizar su labor de forma sencilla y correcta.

Esto nos lleva a diferenciar dos partes muy importantes cuando se construye un módulo:

1. Parte **pública**. Todos los aspectos que son visibles para los demás módulos. El objetivo es que sean pocos. Es la única información necesaria y disponible para un correcto uso del módulo.

2. Parte **privada**. Todos los detalles que no son visibles para los demás módulos. Normalmente serán numerosos, pero no es necesario conocerlos para utilizar el módulo.

Por tanto, para que la descomposición sea útil, es necesario un proceso de **ocultamiento de información** de forma que el razonamiento, en la construcción del sistema a partir de sus módulos, utilice el mínimo de características relevantes (parte pública) ocultando las irrelevantes (parte privada). Nótese que cuanto mayor es el grado de ocultamiento de información, menor será la información que tendremos que manejar (se disminuye la complejidad a causa de haber minimizado la interdependencia entre módulos).

Es importante destacar que este ocultamiento no hace referencia a que el usuario del módulo no debe conocer ningún detalle interno por motivos relacionados con la privacidad del código, los derechos de copia o la explotación comercial[1]. Cuando usamos el término de ocultamiento en este documento nos referimos a la necesidad de que el usuario del módulo utilice el mínimo de características para el desarrollo de sus programas. Nuestras aportaciones para que le sea más difícil utilizar los detalles internos facilitarán un uso correcto del software.

[1] Aunque en la práctica, es la forma de proteger el programa fuente ofreciendo al usuario todas las posibilidades sin que conozca los detalles del desarrollo.

1.3 Documentación.

La naturaleza abstracta de una eficiente construcción de módulos implica que el programador debe crear dos documentos diferenciados:

1. **Especificación**. Corresponde al documento que presenta las características sintácticas y semánticas que describen la parte pública. Éste debiera ser suficiente (no necesitamos más información para poder usar todas las posibilidades del módulo) e independiente de los detalles internos de construcción de éste.

2. **Implementación**. Corresponde al documento que presenta las características internas del módulo. Es importante destacar que el software es algo dinámico, por tanto, es necesario mantener un sistema de documentación que permita el mantenimiento, tanto en lo que respecta a diseño como implementación.

2 Abstracción funcional.

El primer tipo de abstracción que históricamente aparece es la **funcional o procedimental** que surge a partir de separar el propósito de una función de su implementación. Se considera el *qué* de una función, obviando el *cómo* se implementa esa función. Así, los algoritmos complejos que requieren muchas operaciones se engloban en una sóla, eliminando todos los detalles de cómo se implementan. En la abstracción funcional o procedural, abstraemos un conjunto preciso de operaciones (detalles de *cómo* se realiza) como una única operación en forma de función o procedimiento.

Por ejemplo, si en el programa que se desarrolla es necesario calcular el índice del elemento máximo que se almacena en un vector, podemos olvidar todos los detalles relativos a cómo se debe calcular, las variables que se deben usar, etc (detalles irrelevantes) y considerar que disponemos de una operación *IndiceMaximo* que se ocupa de resolverlo (detalles relevantes). Así, el código correspondiente podría ser el presentado en la tabla 1.

Operación	**int** IndiceMaximo (**const int** vector[], **const int** nelem)

Detalles	{
	int i,max;
	
	max=0;
	for (i=1;i<nelem;i++)
	if (vector[max]<vector[i])
	max= i;
	
	return max;
	}

Tabla 1: Operacion IndiceMaximo

2.1 Especificación.

La especificación de una abstracción funcional corresponde a un documento que nos permite conocer los aspectos sintácticos y semánticos necesarios para poder usar la función o procedimiento, sin necesidad de conocer los detalles (parte privada) de su construcción.

La parte sintáctica se refiere a la forma correcta de escribir una llamada a la función. Para ellos será necesario establecer cual es el nombre así como el número, orden y tipo de entradas/salidas. La forma más simple, y por tanto más habitual, de realizarlo es utilizando directamente la sintaxis del lenguaje de programación. En nuestro caso, usando la cabecera de la función quedará especificada, sin ningún tipo de ambigüedad, la sintaxis a usar.

Por otro lado, la parte semántica se refiere al "significado", es decir a las condiciones que se dan antes y después de haber realizado una llamada a la función. En este caso, podemos afirmar que existen distintas formas de realizarla que dependen fundamentalmente del programador, aunque son equivalentes si

consideramos que todas son una forma de establecer el comportamiento de la función independientemente de los detalles internos de construcción.

La forma más simple de llevarla a cabo es definiendo dos cláusulas

1. **Precondiciones**. Presenta las condiciones que se deben de dar antes de la ejecución de la función.
2. **Postcondiciones**. Presenta las condiciones que serán ciertas después de la ejecución de la función.

Un ejemplo de este tipo de especificación se presenta a continuación

```
int IndiceMaximo (const int vector[], const int nelem)
/*
  Precondiciones:
    - 'nelem'>0
    - 'vector' es un vector con nelem elementos
  Postcondiciones:
    - Devuelve la posición del mínimo elemento del vector
        (devuelve i tal que v[i]<=v[j] para 0<=j<nelem)
*/
```

Aunque este método es muy conocido ya que aparece en gran medida en los primeros documentos acerca de abstracción, no es el único para expresar la especificación, ya que el programador es el responsable de escoger la forma de especificar que considere más conveniente. Así, podemos optar por estructurar mejor la información, ya que en muchos casos el usuario de la función desea consultar un aspecto concreto sin necesidad de tener que leer todos los detalles, dividiéndola en cinco cláusulas:

1. Parámetros. Explica el significado de cada parámetro de la función.
2. Valor de retorno. Si existe, se incluye esta cláusula que describe el valor que se devuelve en la función.
3. Precondiciones. Son los requisitos que se deben cumplir para que una llamada a la función se comporte como se describe en la especificación. Si no se cumplen estas condiciones, no se garantiza ningún tipo de resultado, es decir, no se asegura nada acerca del comportamiento del programa.
4. Efecto. Describe el resultado que se produce cuando se realiza una correcta llamada (según las precondiciones) a la función.
5. Excepciones. Si es necesario, se puede incluir una sección en la que se describe el comportamiento de la función cuando se da una circunstancia que no permite la finalización exitosa de su ejecución.

El siguiente es un ejemplo de este tipo de especificación

```
int IndiceMaximo (const int vec[], const int nelem)
/*
  Argumentos:
    vec: array 1-D que contiene los elementos donde encontrar
         el valor máximo.
    nelem: número de elementos en el vector 'vec'.
  Devuelve:
    indice en el vector 'vec'
  Precondiciones:
    nelem>0
    vec tiene al menos 'nelem' elementos
  Efecto:
    Busca el elemento más grande en el vector 'vec' y devuelve la
    posición de éste, es decir, el valor i tal que v[i]<=v[j]
    para 0<=j<nelem
*/
```

2.1.1 Herramientas para la especificación.

El uso de herramientas automáticas para la especificación es un aspecto importante para mejorar la calidad y productividad en el desarrollo de programas. Los lenguajes de programación son independientes de este tipo de herramientas por lo que la construcción y mantenimiento de especificaciones se debe realizar, por parte del programador, con el uso de herramientas adicionales que incluso podríamos encontrar en entornos de desarrollo integrados.

El uso de este tipo de software, nos facilita esta labor fundamentalmente porque

- Automatizan la generación de documentos y proporcionan herramientas para facilitar su uso.
- Permiten mantener la especificación junto con el código, haciendo más fácil para el programador relacionar ambas tareas.

Un ejemplo lo encontramos en el programa doc++[2], que permite incluir la especificación como comentarios junto con el código. Una vez incluida, podemos generar documentación de forma automáticas en distintos formatos. La forma de usar este programa, en el caso de la especificación de una función, consiste en incluir un comentario antes de la cabecera de la función que comience con *"/**"* en lugar de "/*", y que incluya una serie de cláusulas que describen de una manera estructurada el comportamiento de la función. Un ejemplo es el siguiente

```
/**
   @memo Calcula el indice del elemento máximo del vector
   @param vec: array 1-D que contiene los elementos donde encontrar el valor máximo.
   @param nelem: número de elementos en el vector 'vec'.
   @return indice en el vector 'vec'.
   @precondition vec: es un vector de al menos nelem elementos
   @precondition tamaño: nelem>0
   @doc
     Busca el elemento más grande en el vector vec y devuelve la posición de éste,
     es decir, el valor i tal que v[i]<=v[j] para 0<=j<nelem
*/
int IndiceMaximo (const int vec[],const int nelem)
```

Podemos ver que escribimos cada una de las partes de la especificación haciendo uso de palabras clave precedidas por el símbolo "@". El programa *doc++* procesa este código y genera la especificación en el formato solicitado.[3].

3 Tipos de datos abstractos.

En la historia de la programación, los algoritmos son cada vez más complejos, al igual que los datos manejados. Por ello, también se aplica la abstracción a éstos, que podemos denominar **abstracción de datos**. En primer lugar los lenguajes de alto nivel ofrecen una solución con los tipos simples (por ej. *float, int*, etc) y un conjunto de operaciones para manejarlos. Cuando la información es más compleja, se hace más difícil su manejo. Surge así la construcción de **tipos de datos abstractos**, que son nuevos tipos de datos con un grupo de operaciones que proporcionan la única manera de manejarlos. De esta forma, debemos conocer las operaciones que se pueden usar, pero no necesitamos saber

- la forma en que se almacenan los datos ni
- cómo se implementan las operaciones

Por ejemplo, si en el programa que se desarrolla es necesario gestionar fechas, es más complejo que el programador tenga que conocer los detalles de cómo se almacena, así como de las sentencias necesarias para realizar algún cálculo con ellas. En este caso, es conveniente crear un nuevo tipo de dato que podemos denominar *Fecha* junto con un conjunto de operaciones para manejarlo (ver tabla 2).

[2] *http://docpp.sourceforge.net*

[3] Existen otros programas similares. Por ejemplo, *kdoc* que puede usarse con el entorno integrado de desarrollo *kdevelop*

Tipo	*Fecha*
Operaciones	*Fecha CrearFecha(const int dia, cosnt int mes, const int anio)* *int DiaFecha(const Fecha f)* *int MesFecha(const Fecha f)* *int AgnoFecha(const Fecha f)* *Fecha IncrementarFecha(const Fecha f, const int dias)* *int DiferenciaFecha(const Fecha f1, const Fecha f2)* *int DiaSemanaFecha(const Fecha f)*

Tabla 2: Operaciones T.D.A. Fecha

Así, la implementación tendría distintas alternativas, por ejemplo una estructura

```
typedef struct {
            int dia;
            int mes;
            int anio;
          } Fecha;
```

o un simple entero codificando la distancia en dias desde cierta fecha conocida

```
typedef int Fecha;
```

De igual forma, podríamos tener una variedad de maneras para codificar cada una de las operaciones, incluso para una determinada representación.

Al igual que en el caso de abstracción funcional, esta multitud de detalles en la construcción de este tipo de dato abstracto son irrelevantes para el resto del programa, por lo que podemos ignorarlos, simplificando así el problema original que teníamos que resolver.

Por tanto, podemos definir un tipo de dato abstracto como una colección de valores junto con unas operaciones sobre ellos, definidos mediante una especificación que es independiente de cualquier implementación.

3.1 Selección de operaciones.

Una tarea fundamental en el desarrollo de un T.D.A. es la selección del conjunto de operaciones que se usarán para manejar el nuevo tipo de dato. Para ello, el diseñador deberá considerar los problemas que quiere resolver en base a este tipo, y ofrecer el conjunto de operaciones que considere más adecuado. Para una buena selección, podemos tener en cuenta que

- Un número muy pequeño de operaciones limitaría la utilidad del software desarrollado. Nótese que:
  - Debe existir un conjunto mínimo de operaciones para garantizar la abstracción. Es decir, no podemos encontrar problemas que, haciendo uso del nuevo tipo, no puedan resolverse por falta de operaciones.
  - Se deben incluir aquellas operaciones que vayan a ser usadas con bastante frecuencia ya que el no incluirlas obligaría al usuario a ampliar las posibilidades del tipo que le ofrecemos.
  - Podemos añadir algunas operaciones adicionales que, aunque resuelven problemas más puntuales, son mucho más simples de construir y/o eficientes en base a la estructura interna del tipo.
- Un número muy alto de operaciones no tiene por qué considerarse una opción más adecuada. Nótese que
  - El usuario usará un subconjunto del total y el añadir operaciones adicionales que probablemente no se usen, implicará una cantidad de información superior y por tanto un esfuerzo adicional para manejarlo.

– Debemos considerar la posibilidad de que el tipo resultante sufra alguna revisión en el futuro. Un número muy alto de operaciones puede provocar un esfuerzo superior para llevar a cabo dichos cambios. Además, es más sencillo añadir una nueva operación que no fue incluida inicialmente que eliminar una existente cuando el software está en explotación.

Una vez que disponemos del conjunto de operaciones, podemos clasificarlas en dos conjuntos

1. Fundamentales. Son aquellas que son necesarias para garantizar la abstracción. No es posible prescindir de ellas ya que habría problemas que no se podrían resolver sin acceder a la parte interna del tipo de dato.

2. No fundamentales. Corresponden a las operaciones prescindibles ya que el usuario podría construirlas en base al resto de operaciones.

Podemos considerar que el primero de ellos está compuesto por las operaciones **primitivas** que permiten resolver cualquier problema a partir de ellas. Sin embargo, las operaciones no fundamentales pueden estar o no construidas a partir de las operaciones básicas. Además, el usuario del tipo de dato abstracto las considera en el mismo nivel, es decir, como las operaciones más básicas de que dispone. Por tanto, usaremos el término "primitiva" y "operación" de un tipo de dato abstracto de forma indistinta.

3.2 Especificación.

En el caso de la abstracción de datos también es necesaria una especificación. Los objetivos son los mismos que para el caso de la abstracción funcional, es decir, poder establecer una especificación que determine la forma de usar un tipo de dato abstracto, sin necesidad de conocer los detalles internos de su construcción. En este caso, no sólo vamos a especificar una función sino un nuevo tipo de dato junto con un conjunto de operaciones, por lo tanto aparecerán dos partes:

1. **Definición**. En esta parte deberemos definir el nuevo tipo de dato abstracto, así como todos los términos relacionados que sean necesarios para comprender el resto de la especificación.

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

Adicionalmente, podemos añadir a la especificación una sección con información sobre la implementación usada. Ésta no es, propiamente, una parte de la especificación ya que depende de la implementación interna, pero resulta útil para el usuario cuando desarrolla nuevos algoritmos. Así, podemos incluir información sobre la eficiencia del tipo en espacio en notación O-mayúscula, la eficiencia en tiempo de las operaciones, la localización de los recursos del tipo (memoria estática o dinámica), o incluso una breve reseña sobre el tipo de estructura de datos que se ha usado.

Esta parte dependiente puede cambiar si modificamos los implementación del tipo de dato abstracto. Sin embargo, las implementaciones del usuario seguirán siendo válidas ya que el desarrollo de sus programas se basa en la especificación del tipo. Una modificación interna del T.D.A. sólo afectará a la eficiencia y estabilidad de sus programas que, por otro lado, probablemente se verán afectadas positivamente.

3.2.1 Definición.

El resultado de un tipo de dato abstracto en una nueva clase de objetos que tendremos que traducir en uno o varios tipos nuevos, definidos por el usuario en el lenguaje escogido. Por tanto, tenemos que definir el dominio en el que tomará valores una instancia de la nueva clase de objetos. Para ello, tenemos distintas formas de hacerlo, por ejemplo

- Si el dominio es finito y pequeño, puede ser "enumerado". Por ejemplo, el dominio de un nuevo tipo *booleano* es $\{false, true\}$.

- Podemos utilizar otro dominio conocido. Por ejemplo, los enteros positivos desde el 0 al 255 ($[0..255]$).

- Se puede hacer estableciendo las reglas para poder construirlo. Por ejemplo, el dominio de las cadenas de caracteres se puede definir indicando que la cadena vacía en un elemento del conjunto y que cualquier cadena seguida de un caracter también es una cadena.

Además, en la parte de definición tendremos que describir los elementos necesarios para poder entender correctamente el resto de la especificación. Así por ejemplo, podemos definir algún valor concreto de un elemento destacado del conjunto, algún término o concepto relacionado con el tipo y que se usa en la descripción de las operaciones, etc.

3.2.2 Operaciones

En esta parte, se realiza una especificación de las operaciones que se usarán sobre el tipo de dato abstracto que se está construyendo. Dado que cada una de ellas se puede construir como una función o procedimiento, podemos establecer un método similar a los comentados para la abstracción funcional. Así, una forma de realizarla es utilizando cláusulas de precondiciones y postcondiciones, o un esquema más estructurado como se indicó anteriormente.

3.3 Implementación.

El resultado de la fase de implementación es doble: por un lado obtendremos el código que implementa el comportamiento descrito en la fase de especificación y por otro, la documentación del software escrito.

Para implementar un T.D.A., es necesario, en primer lugar, escoger una representación interna adecuada, es decir, una forma de estructurar la información de manera que podamos representar todos los objetos de nuestro tipo de dato abstracto de una manera eficaz. Por tanto, debemos seleccionar una **estructura de datos** adecuada para la implementación, es decir, un tipo de dato que corresponde a esta representación interna y sobre el que implementamos las operaciones. A éste tipo, se le denomina **tipo rep**. Ejemplos:

- T.D.A. *Racional*.
  - Un vector de dos posiciones para almacenar el numerador y denominado

```
typedef int Racional[2];
```

  - Una estructura con dos campos, para almacenar el numerador y denominadorr

```
typedef struct {
            float numerador;
            float denominador;
          } Racional;
```

  –

- T.D.A. *Fecha*.
  - Un vector para almacenar tres valores indicando el día, mes y año correspondiente

```
typedef int Fecha[3];
```

  - Una estructura con tres campos.

```
typedef struct {
            int dia;
            int mes;
            int anio;
          } Fecha;
```

  - Un entero, que almacena el número de días que han transcurrido desde cierta fecha base o inicial.

```
typedef int Fecha;
```

  - etc.

- T.D.A. *Polinomio*[4].
  - Una estructura con una matriz que almacena los coeficientes y un entero que indica el grado.

```
typedef struct {
          float coeficientes[];
          int grado;
        } Polinomio;
```

  - Dos matrices (de enteros y reales) para almacenar el grado y el coeficiente de cada monomio más un entero que indica cuantos monomios componen el polinomio.

```
typedef struct {
          int grado[];
          float coeficientes[];
          int nmonomios;
        } Polinomio;
```

  - Una matriz de estructuras de pares *(grado,coeficiente)* más un entero con un sentido similar al punto anterior.

```
typedef struct {
          int grado;
          float coeficiente;
        } Monomio;
typedef struct {
          Monimio *mon;
          int nmonomios;
        } Polinomio;
```

  - etc.

- T.D.A. *Conjunto*.
  - Una matriz del tipo base del conjunto con los elementos que pertenecen más un entero indicando el número de elementos.

```
typedef int TipoBase;
typedef struct {
          TipoBase elementos[];
          int nelem;
        } Conjunto;
```

  - Un puntero a un conjunto de celdas enlazadas que almacenan los elementos, junto con un entero que indica el número de elementos[5]

```
typedef int TipoBase;
typedef struct celda {
          TipoBase elemento;
          struct celda *nelem;
        }Celda;
typedef struct {
          Celda *elementos;
          int nelem;
        } Conjunto;
```

  - etc.

[4] véase más adelante el T.D.A. polinomio
[5] véase la implementación de listas con celdas enlazadas.

En la documentación, debe aparecer la estructura de datos que se utiliza como **tipo rep** y cómo se almacena un elemento del nuevo tipo de dato que se define en dicha estructura. En los ejemplos anteriores, lo hemos descrito de una manera poco precisa, aunque se deberá establecer una manera más formal que ayude en la eliminación de ambigüedades. Para ello, debemos tener en cuenta que establecer la relación entre el tipo *rep* y el tipo abstracto que se está construyendo, consiste en definir una función entre los objetos que se pueden representar en el tipo rep y los objetos del tipo abstracto a los que corresponden. Ésta se denomina **función de abstracción**. Propiedades de esta función son:

- Parcial, ya que no todos los valores que podemos representar corresponden a un elemento del tipo abstracto. Por ejemplo:
  - Fecha: La representación con la terna de enteros {40,5,2001} no corresponde a ninguna fecha pues no existe el día 40.
  - Conjunto: La matriz con elementos enteros {1,1,3} y un entero indicando que existen 3 elementos (lo denotamos {3: 1,1,3}) no corresponde a una representación válida para un conjunto si consideramos que no pueden repetirse elementos.
- Todos los elementos del tipo abstracto tienen que tener una representación, es decir, un elemento origen en esta función
- Varios valores de la representación podrían representar a un mismo valor abstracto. Por ejemplo, {2: 1,2} y {2: 2,1} podrían en cierta implementación ser equivalentes para representar un conjunto de dos elementos que contenga los enteros 1 y 2.

Si restringimos la función de abstracción a los elementos del tipo *rep* que representan un objeto válido de tipo abstracto, descubrimos que constituye una aplicación (todos los elementos origen se aplican a un único objeto abstracto) sobreyectiva (todos los objetos abstractos tienen al menos una representación) y no inyectiva (dos representaciones pueden corresponder al mismo objeto abstracto).

Por tanto en la documentación podemos incluir esta aplicación para indicar el significado de la representación. Esta aplicación tiene dos partes:

1. Indicar exactamente cual es el conjunto de valores de representación que son válidos, es decir, que representan a un tipo abstracto. Por tanto, será necesario establecer una condición sobre el conjunto de valores del tipo *rep* que nos indique si corresponden a un objeto válido. Esta condición se denomina **invariante de la representación**.

$$f_{inv} : rep \longrightarrow booleanos$$

2. Indicar para cada representación válida cómo se obtiene el tipo abstracto correspondiente, es decir, la **función de abstracción**.

$$f_{Abs} : rep \longrightarrow \mathcal{A}$$

Un *invariante de la representación* es "invariante" porque siempre es cierto para la representación de cualquier objeto abstracto. Por tanto, cuando se llama a una función del tipo de dato se garantiza que la representación cumple dicha condición y cuando se devuelve el control de la llamada, debemos asegurarnos que se sigue cumpliendo. Sin embargo, a lo largo de la ejecución de dicha operación, no es necesario que se cumpla, y la representación podrá contener valores temporales que no sean válidos.

Aunque estos conceptos sean triviales, debemos incluirlos como documentación, ya que cuando se realiza la implementación de un tipo de dato abstracto, los conceptos de función de abstracción e invariante de la representación contribuyen a precisar el diseño y mejorar la documentación, por lo que constituyen una pieza importante que debe aparecer siempre en la implementación. Así, por ejemplo, si encontramos una representación para la que cualquier valor es correcto para representar un tipo abstracto, deberíamos incluir un comentario del tipo *"el invariante de la representación es: verdadero"*.

Ejemplos de funciones e invariantes de la representación son:

- T.D.A. Racional con una estructura de dos campos (ver pág. 8)

– Invariante de la representación:

$$r.denominador \neq 0$$

– Función de abstracción:

$$\begin{array}{rcl} f_{Abs} : rep & \longrightarrow & Racional \\ r & \longrightarrow & \dfrac{r.numerador}{r.denominador} \end{array}$$

- T.D.A. Fecha con una estructura de tres campos (ver pág. 8).
  – Invariante de la representación. Un objeto r que representa una Fecha debe cumplir que:
    * $1 \leq r.dia \leq 31$ y $1 \leq r.mes \leq 12$
    * $r.mes \in \{4, 6, 9, 11\} \rightarrow r.dia \leq 30$
    * $r.mes = 2$ *y* $bisiesto(r.anio) \rightarrow r.dia \leq 29$
    * $r.mes = 2$ *y no* $bisiesto(r.anio) \rightarrow r.dia \leq 28$

    donde $bisiesto(i)$ es la función booleana que devuelve verdadero para los años bisiestos

$$bisiesto(i) \equiv ((i\%4 = 0)\, y\, (i\%100 \neq 0))\, o\, (i\%400 = 0)$$

  – Función de abstracción:

$$\begin{array}{rcl} f_{Abs} : rep & \longrightarrow & Fecha \\ r & \longrightarrow & r.dia/r.mes/r.anio \end{array}$$

- T.D.A. Conjunto con una estructura que contiene un entero, el número de elementos, y una matriz que almacena cada uno de ellos (ver pág. 9).
  – Invariante de la representación:

$$(r.elementos[i] \neq r.elementos[j])\ \forall i, j\ 0 \leq i < j < r.nlem$$

  – Función de abstracción:

$$\begin{array}{rcl} f_{Abs} : rep & \longrightarrow & Conjunto \\ r & \longrightarrow & \{r.elementos[i]/0 \leq i < r.nlem\} \end{array}$$

- T.D.A. Conjunto con una estructura que contiene un entero, el número de elementos, y una matriz que almacena cada uno de ellos en orden ascendente(ver pág. 9).
  – Invariante de la representación:

$$(r.elementos[i] < r.elementos[j])\ \forall i, j\ 0 \leq i < j < r.nlem$$

  – Función de abstracción:

$$\begin{array}{rcl} f_{Abs} : rep & \longrightarrow & Conjunto \\ r & \longrightarrow & \{r.elementos[i]/0 \leq i < r.nlem\} \end{array}$$

Para algún ejemplo adicional, véase más adelante los tipos de datos abstractos que se desarrollan.

3.4 Especificación formal de T.D.A.

En las secciones anteriores hemos expuesto una forma de realizar la especificación de un T.D.A., que podemos denominar mediante **modelos abstractos**, ya que se basa en que el usuario está familiarizado con algún otro dominio. De esta forma, podemos definir el tipo y establecer las precondiciones y postcondiciones necesarias haciendo referencia a ese tipo conocido. Por tanto, se podría decir que es un método informal al asumir casos de ambigüedades cuando el usuario tenga un conocimiento intuitivo, sobre ese dominio, distinto al de la persona que especifica el tipo.

Para establecer un sistema de comunicación claro, simple y conciso, podemos realizar una especificación independiente del lenguaje de programación. Para llevarla a la práctica, deberemos de establecer una especificación del interface (dependiente del lenguaje) en términos de ella, con la ventaja de que los términos que se usan tendrán un significado preciso que no darán lugar a ambigúedades.

Por otro lado, el disponer de este tipo de especificación, nos permite deducir formalmente propiedades que satisface el tipo o cualquier implementación válida de éste, así como posibilitar la verificación formal de programas.

Un ejemplo de este tipo de especificación es la **especificación algebraica** para la cual, se define una sintaxis con la que indicar dominios, operaciones, etc. y un conjunto de axiomas u operaciones que definen el significado (semántica) del nuevo tipo. En la figura 1 se presenta un esbozo del aspecto de una especificación algebraica, a fin de que el lector disponga de algún ejemplo que clarifique la idea de que la semántica puede aparecer como un conjunto de axiomas. Si el lector está interesado, puede consultar, por ejemplo, las referencias Liskov[5] y Peña[6] para una discusión más profunda sobre especificaciones formales.

Especificación de Sucesión

- **Operaciones**
  - $<>: \longrightarrow Sucesion$ /* sucesión sin elementos*/
  - $< \cdot >: elemento \longrightarrow Sucesion$ /* sucesión con un sólo elemento */
  - $\cdot + \cdot : Sucesion \times Sucesion \longrightarrow Sucesion$ /* Concatenación de sucesiones */
  - $\cdot \hookrightarrow \cdot : elemento \times Sucesion \longrightarrow Sucesion$ /* Añadir elemento en la izquierda */
  - $\cdot \hookleftarrow \cdot : Sucesion \times elemento \longrightarrow Sucesion$ /* Añadir elemento en la derecha */
  - $\| \cdot \| : Sucesion \longrightarrow natural$ /* longitud(*natural* previamente especificado) */
- **Semántica**
  - Variables:
    - x: de tipo *elemento*
    - s, s_1, s_2, s_3: de tipo *Sucesion*
  - Ecuaciones:
    - $s+ <>= s$
    - $<> +s = s$
    - $(s_1 + s_2) + s_3 = s_2 + (s_2 + s_3)$
    - $x \hookrightarrow s =< x > +s$
    - $s \hookleftarrow x = s+ < x >$
    - $\| <> \| = 0$
    - $\| < x > \| = 1$
    - $\|s_1 + s_2\| = \|s_1\| + \|s_2\|$

Figura 1: Esbozo de especificación algebraica.

Sin embargo, este tipo de especificación no se usa de manera generalizada ya que su complejidad, la necesidad de herramientas adicionales para facilitar su uso y el hecho de que muchos profesionales consideren que su utilidad está aún por demostrar, hacen que se opte por otras alternativas de especificación.

Es este documento, asumiremos la necesidad de una especificación cuasi-formal, considerando que con este nombre no queremos decir una notación matemática o verificable automáticamente, sino un conjunto

de reglas de notación y documentación que permitan la división del problema y la comunicación entre las personas involucradas en el desarrollo de un programa.

3.5 Diseño modular y tipos de datos abstractos.

Si revisamos las características básicas del diseño modular que se han expuesto anteriormente y el concepto de tipo de dato abstracto, podemos observar claramente que un tipo de dato abstracto constituye un buen candidato para ser considerado un módulo en nuestra solución:

- Las conexiones del módulo con el resto del programa son pocas y bien definidas mediante la especificación.
- El módulo lleva a cabo una tarea bien definida por el tipo de dato y las operaciones que se asocian a éste.
- El problema de diseár un T.D.A. se puede enfocar de forma general para obtener un módulo fácilmente reutilizable en otros problemas.
- La independencia de los detalles internos con el resto del programa por medio de la especificación facilita las tareas de desarrollo y mantenimiento.

Por tanto, el desarrollo de tipos de datos abstractos es uno de los fundamentos para la división de problemas en módulos.

Es interesante destacar que la metodología de la programación modular no acaba cuando se determina un tipo de dato abstracto a desarrollar. Nótese que para construir un módulo que corresponde a este tipo podemos volver a utilizar la misma metodología. Por ejemplo,

- podemos dividir el conjunto de primitivas de un tipo en varios subconjuntos considerando que algunas de ellas acceden directamente a la representación interna del tipo y otras se pueden implementar a partir de las primeras,
- en estos módulos podemos construir funciones adicionales que implementen alguna operación interna del tipo y que no estén disponibles para el usuario final,
- podemos volver a dividir el módulo en base a nuevos tipos de datos abstractos que faciliten su implementación,
- etc.

3.6 Un ejemplo: El TDA Fecha.

En primer lugar, es necesario revisar algunos conocimientos sobre el calendario que nos faciliten el desarrollo. En este ejemplo, el tipo de dato va a permitir el manejo de fechas según el calendario cristiano. En éste, el año tiene una longitud de 365 o 366 días dependiendo de si se considera no bisiesto o bisiesto respectivamente[6]. Podemos distinguir dos calendarios, que se distinguen según el criterio para considerar si un año es bisiesto:

1. Juliano[7]. Uno de cada cuatro años se considera bisiesto (los años divisibles por 4). El problema de este calendario es que se desvía 1 días cada 128 años aproximadamente (considera años de 365.25 días).
2. Gregoriano[8]. Los años divisibles por 4 son bisiestos excepto aquellos divisibles por 100 que no lo sean por 400. En este caso se aproxima mejor la longitud del año ya que son necesarios 3225 años aproximadamente para desviarse 1 día (considera años de 365.2425 días).

[6] El incorporar años con 1 día adicional se debe al reajuste necesario para que la posición del sol en el cielo sea la misma en las mismas fechas. El tiempo necesario para que el sol se situe en el mismo lugar es de 365.24219 días (en media).

[7] Introducido por Julio Cesar en el 45 antes de Cristo

[8] Introducido por Gregorio XIII en el 1582

El momento en que se cambió del primero de ellos al actual, es decir, al gregoriano varía según los países. Así, el papa Gregorio XIII determinó que después del 4 de octubre de 1582 (usándose el Juliano), se debía pasar al 15 de octubre de 1582 (empezándose a usar el Gregoriano). Algunos países realizaron este cambio en esa fecha pero otros no lo llevaron a cabo incluso hasta el siglo XX.

Para eliminar la confusión y los cambios que ocurren en estos calendarios, se ha introducido el uso del **Día Juliano**. Para ello, se considera que el período juliano comienza el 1 de enero del 4713 antes de cristo (a las 12:00 UTC). A partir de ahí se asigna un número entero positivo que numera los días transcurridos[9].

3.7 Especificación del TDA Fecha.

3.7.1 Definición.

Una instancia f del tipo de dato abstracto Fecha almacena el valor de una fecha en el período comprendido entre el 1 de enero de 4713 A.C. al 1 de enero de 3268 D.C. (fin del período juliano).

Una fecha f se puede representar como:

- $f \equiv n(DJ)$ donde n es un entero que representa el día juliano.
- $f \equiv d/m/a(CJ)$ donde d, m, a son enteros indicando día, mes y año respectivamente, en el calendario juliano.
- $f \equiv d/m/a(CG)$ donde d, m, a son enteros indicando día, mes y año respectivamente, en el calendario gregoriano.
- $f \equiv d/m/a$ donde d, m, a son enteros indicando día, mes y año respectivamente, en el calendario juliano si $d/m/a$ es anterior a $d_g/m_g/a_g$ (CG) o en el calendario gregoriano si es esa o posterior.

Por defecto, el valor de $d_g/m_g/a_g$ (CG) es 15/10/1582 (CG), aunque puede cambiar debido a que distintos países cambiaron al calendario gregoriano en distintas fechas.

Por tanto, consideraremos que cualquier fecha anterior a $d_g/m_g/a_g$ (CG) se entiende que se expresa según el calendario juliano, mientras que el resto se expresan según el gregoriano.

A cada fecha, le corresponde un día de la semana, considerándose la primera aquella que contenga el día 4 de enero.

No existe el año 0, y los años antes de cristo de expresarán con su correspondiente valor negativo. Por ejemplo, el 40 A.C., se expresará como el año -40.

Para poder usar el nuevo tipo de dato se debe incluir el fichero

fecha.h

Una variable *f* del T.D.A. Fecha se declara con la sentencia

Fecha f;

El espacio requerido para el almacenamiento es O(1).

3.7.2 Operaciones del TDA Fecha.

1. *FijarCambioGregoriano(const int dia, const int mes, const int anio)*. Determina la fecha en que se empieza a usar el gregoriano.
   - *Parámetro dia*: día de la fecha a asignar
   - *Parámetro mes*: mes de la fecha a asignar
   - *Parámetro anio*: año de la fecha a asignar. $-4713 \leq anio \leq 3267$ y $anio \neq 0$.
   - *Precondición*: *dia/mes/anio* es una fecha válida del calendario gregoriano
   - *Efecto*: Fija la primera fecha en que se empieza a usar el calendario gregoriano. Esta fecha es *dia/mes/anio* expresada en el calendario gregoriano

[9] También existe un día juliano modificado que corresponde a restar 2,400,000.5 días al día juliano, y por tanto comienza el 17 de noviembre de 1858 a las 00 UTC

2. *int AsigFecha (Fecha* f, const int dia, const int mes, const int anio)*. Asigna un valor a una variable de tipo fecha.
   - *Parámetro f*: puntero a la fecha que tomará el nuevo valor.
   - *Parámetro dia*: día de la fecha a asignar
   - *Parámetro mes*: mes de la fecha a asignar
   - *Parámetro anio*: año de la fecha a asignar. $-4713 \leq anio \leq 3267$ y $anio \neq 0$.
   - *Devuelve*: 1 si ha tenido éxito, 0 en caso contrario.
   - *Efecto*: Almacena el valor dia/mes/anio en **f* (de tipo *Fecha*). Si el trío de valores no corresponde a una fecha válida devolverá un valor de 0. La operación se realiza en tiempo O(1)
3. *void AsigPascua (Fecha *f, const int anio)*. Asigna la fecha de pascua.
   - *Parámetro f*: puntero a la fecha que tomará el nuevo valor.
   - *Parámetro anio*: año al que calcular la fecha de pascua. $1 \leq anio \leq 3267$
   - *Efecto*: Calcula la fecha de pascua del año *anio* y la asigna a **f*. La operación se realiza en tiempo O(1).
4. *void AsigDJFecha (Fecha* f, const int DJ)*. Asigna un valor a una variable de tipo fecha.
   - *Parámetro f*: puntero a la fecha que tomará el nuevo valor.
   - *Parámetro DJ*: día juliano de la fecha a asignar. $0 \leq DJ < 2914695$
   - *Efecto*: Almacena el valor de la fecha que corresponde al día juliano *DJ* en **f* (de tipo *Fecha*). La operación se realiza en tiempo O(1).
5. *int DJFecha (const Fecha f)*. Calcula el día juliano de una fecha.
   - *Parámetro f*: fecha desde la que calcular el día juliano.
   - *Devuelve*: Día juliano que corresponde a la fecha *f*.
   - *Efecto*: La operación se realiza en tiempo O(1).
6. *int DiaFecha (const Fecha f)*. Día de una fecha.
   - *Parámetro f*: fecha desde la que extraer el día. *f* está inicializada.
   - *Devuelve*: día que corresponde a la fecha *f*.
   - *Efecto*: La operación se realiza en tiempo O(1).
7. *int MesFecha (const Fecha f)*. Mes de una fecha.
   - *Parámetro f*: fecha desde la que extraer el Mes. *f* está inicializada.
   - *Devuelve*: Mes que corresponde a la fecha *f*.
   - *Efecto*: La operación se realiza en tiempo O(1).
8. *int AnioFecha (const Fecha f)*. Año de una fecha.
   - *Parámetro f*: fecha desde la que extraer el Año. *f* está inicializada.
   - *Devuelve*: Año que corresponde a la fecha *f*.
   - *Efecto*: La operación se realiza en tiempo O(1).
9. *void IncrementarFecha (Fecha *res, const Fecha f, const int d)*. Desplazamiento de fecha
   - *Parámetro res*: un puntero a la fecha que tomará el valor resultante.
   - *Parámetro f*: fecha base a la que sumar un número de días. *f* está inicializada.
   - *Parámetro d*: número de días a incrementar.

- *Precondición*: La suma de *d* a *f* no resulta una fecha fuera del período juliano.
- *Efecto*: Obtiene la fecha que resulta de añadir *d* días a *f*. Nótese que d puede ser negativo y por tanto puede igualmente realizarse un desplamiento hacia el pasado. La fecha **res* puede ser la misma que el parámetro que se pasa en *f*. La operación se realiza en tiempo O(1).

10. *int DiferenciaFecha (const Fecha f1, const Fecha f2)*. Resta de fechas.
    - *Parámetro f1*: Primer operando de la resta.
    - *Parámetro f2*: Segundo operando de la resta.
    - *Devuelve*: número de días desde la fecha *f2* a la fecha *f1*.
    - *Efecto*: Realiza la operación *f1-f2*, dando como resultado el número de días que tienen que pasar desde *f2* a *f1*. Si *f1* es anterior a *f2*, el resultado será negativo. La operación se realiza en tiempo O(1).
11. *int DiaSemanaFecha (const Fecha f)*. Día de la semana
    - *Parámetro f*: Fecha desde la que queremos obtener el día de la semana.
    - *Devuelve*: El día de la semana (0 equivale a lunes, 6 a domingo) que corresponde a la fecha *f*.
    - *Efecto*: La operación se realiza en tiempo O(1).
12. *int NumeroSemanaFecha (const Fecha f)*. Calcula el número de semana.
    - *Parámetro f*: Fecha desde la que queremos obtener el número de semana.
    - *Devuelve*: número de semana de *f*.
    - *Efecto*: Devuelve un entero que corresponde al número de semana donde se sitúa *f*. La primera semana es aquella que contiene el 4 de enero. Lo que significa que el 1 de enero de un año puede estar situado en la última semana del año anterior (ISO-8601). La operación se realiza en tiempo O(1).

3.8 Implementación del TDA Fecha.

El tipo de dato *Fecha* se ha implementado en base a un entero.

Fecha ≡ int

3.8.1 Función de abstracción.

Una fecha *f* del conjunto de valores en la representación (véase invariante de la representación), se aplica a la fecha que corresponde al día juliano *f*. Para más detalles acerca de la correspondencia entre este entero y los habituales tríos día-mes-año, consúltense las funciones privadas del módulo.

3.8.2 Invariante de la representación.

La condición para que un objeto representado sea válido es

$$f \geq 0$$

3.8.3 Funciones y variables privadas.

Para realizar la conversión del entero al trío *día-mes-año* y viceversa se han desarrollado funciones privadas que resuelven el problema para el calendario juliano y para el gregoriano, y que se llaman desde las funciones que implementan las operaciones del tipo. Son las siguientes:

1. *int Gregoriano2JD (int dia, int mes, int ano)*. Conversión de Gregoriano a DJ.
    - *Parámetro dia*: Día de la fecha a convertir.

- *Parámetro mes*: Mes de la fecha a convertir.
- *Parámetro ano*: Año de la fecha a convertir. $-4713 \leq anio \leq 3267$ y $anio \neq 0$
- *Devuelve*: día juliano de la fecha *dia/mes/ano (CG)*
- *Efecto*: Obtiene el día juliano que corresponde a la fecha *dia/mes/ano* del calendario gregoriano. La operación se realiza en tiempo O(1).

2. *int Juliano2JD (int dia, int mes, int ano)*. Conversión de Juliano a DJ.
   - *Parámetro dia*: Día de la fecha a convertir.
   - *Parámetro mes*: Mes de la fecha a convertir.
   - *Parámetro ano*: Año de la fecha a convertir. $-4713 \leq anio \leq 3267$ y $anio \neq 0$
   - *Devuelve*: día juliano de la fecha *dia/mes/ano (CJ)*
   - *Efecto*: Obtiene el día juliano que corresponde a la fecha *dia/mes/ano* del calendario gregoriano. La operación se realiza en tiempo O(1).
3. *void JD2Gregoriano(int *dia, int *mes, int *ano, int JD)*. Conversión de DJ a Gregoriano.
   - *Parámetro dia*: puntero al entero donde almacenar el día.
   - *Parámetro mes*: puntero al entero donde almacenar el mes.
   - *Parámetro ano*: puntero al entero donde almacenar el ano.
   - *Parámetro JD*: Día juliano que convertir.
   - *Efecto*: Obtiene la fecha **dia/*mes/*ano* del calendario gregoriano que corresponde al día juliano *JD*. La operación se realiza en tiempo O(1).
4. *void JD2Juliano(int *dia, int *mes, int *ano, int JD)*. Conversión de DJ a Juliano.
   - *Parámetro dia*: puntero al entero donde almacenar el día.
   - *Parámetro mes*: puntero al entero donde almacenar el mes.
   - *Parámetro ano*: puntero al entero donde almacenar el ano.
   - *Parámetro JD*: Día juliano que convertir.
   - *Efecto*: Obtiene la fecha **dia/*mes/*ano* del calendario juliano que corresponde al día juliano *JD*. La operación se realiza en tiempo O(1).

Se definen 4 variables privadas que almacenan el primer día en que entra en vigor el calendario gregoriano. Son las siguientes:

1. *static int diaGregoriano*. Almacena el valor del día en que se asume el comienzo del calendario gregoriano. Por defecto tiene el valor 15.
2. *static int mesGregoriano*. Almacena el valor del mes en que se asume el comienzo del calendario gregoriano. Por defecto tiene el valor 10.
3. *static int anioGregoriano*. Almacena el valor del año en que se asume el comienzo del calendario gregoriano. Por defecto tiene el valor 1582.
4. *static int diaJulianoGregoriano*. Almacena el valor del día juliano en que se asume el comienzo del calendario gregoriano. Por defecto tiene el valor 2299161.

Para un correcto funcionamiento del módulo se debe cumplir, por tanto, el **invariante**

$$diaGregoriano/mesGregoriano/anioGregoriano(CG) \equiv diaJulianoGregoriano(DJ)$$

que queda garantizado en la función *FijarCambioGregoriano* que es la única que modifica estos valores.

3.8.4 Ejemplos de uso.

En esta sección podemos incluir algunos ejemplos de uso del tipo de dato. Por ejemplo, podemos mostrar:

- Un programa que imprime en la salida estándar varias fechas relacionadas con la pascua en el año 2001.
- Un programa que lee desde los argumentos de la línea de comandos dos números, el mes y el año. Imprime en la salida estándar todos los días de ese mes formateados en columnas. Además asume que el cambio al calendario Gregoriano se llevó a cabo el 14-Sep-1752 (Por ejemplo, Gran Bretaña y colonias).
- Un programa que calcula el tiempo necesario para que una cantidad depositada en un banco consiga una determinada rentabilidad, ofreciendo como resultado la fecha en que se puede retirar el dinero original más la rentabilidad deseada.

Véanse las páginas web de la asignatura para consultar los detalles de estos ejemplos.

3.8.5 Fuentes.

En esta sección se debe incluir los ficheros fuente que implementan el tipo de dato desarrollado (véanse en las páginas web de la asignatura).

3.9 Un ejemplo: El TDA Polinomio.

En esta sección vamos a construir el T.D.A. *Polinomio*, comenzando por la identificación de sus operaciones y realizando tanto la especificación de su funcionamiento como la implementación.

En primer lugar, debemos identificar el conjunto de operaciones que definiremos. Para ello, deberemos considerar cómo se va a usar el nuevo tipo. En nuestro caso, podemos pensar, por ejemplo, en

- Un método para asignar un valor a una variable del nuevo tipo. La forma más simple de hacerlo es construir una función que asigne un valor al coeficiente de un determinado monomio. La asignación de un ponomio completo se puede realizar con llamadas sucesivas a esa función.
- Un método para obtener el valor real que corresponde a un determinado monomio.
- Un método para obtener el grado que corresponde al polinomio almacenado.
- Un método para sumar dos polinomios.
- Un método para restar dos polinomios.
- etc.

Además, podemos considerar la necesidad de dos operaciones adicionales:

1. El tipo de dato abstracto puede requerir una estructura de datos compleja, que se tenga que inicializar antes de poder ser usada. Así debemos añadir una primitiva para reservar recursos si es necesario y asignar un valor inicial a una variable antes de ser usada. A esta función la podemos denominar **constructor**.
2. Lógicamente, si disponemos de una función que puede reservar recursos para almacenar un determinado valor, debemos añadir otra para liberarlos cuando ya no se vayan a usar más. A esta función la podemos denominar **destructor**.

Si analizamos detenidamente el conjunto de operaciones que podemos proponer para el T.D.A. polinomio, podemos ver que se pueden proponer las siguientes primitivas

1. CrearPolinomio: Obtiene los recursos necesarios y devuelve el polinomio nulo.
2. Grado: Devuelve el grado del polinomio.

3. Coeficiente: Devuelve un coeficiente del polinomio.

4. AsigCoeficiente: Asigna un coeficiente del polinomio.

5. DestruirPolinomio: Libera los recursos del tipo obtenido.

Como podemos observar, no podemos prescindir de ninguna de estas funciones (son un conjunto mínimo) a la vez que podemos llevar a cabo cualquier aplicación sobre el tipo polinomio sin necesidad de ninguna más (es un conjunto suficiente). Por tanto, con este conjunto de operaciones tenemos garantizada la abstracción. Sin embargo, existen otras operaciones que se pueden considerar interesantes para que el nuevo tipo de dato sea realmente útil. Como ejemplo podemos proponer[10]:

1. CopiarPolinomio: Crea un nuevo polinomio como copia de otro.

2. AnularPolinomio: Asigna el valor cero a un polinomio.

3. SumarPolinomio: Asigna a un polinomio el valor de la suma de otros dos.

4. RestarPolinomio: Asigna a un polinomio el valor de la resta de otros dos.

3.10 Especificación del TDA polinomio.

3.10.1 Definición.

Una instancia P del tipo de dato abstracto polinomio es un elemento del conjunto de polinomios en una variable x con coeficientes reales. El polinomio $P(x) = 0$ se llama *polinomio nulo*. Un polinomio no nulo se puede representar como

$P(x) = a_0 + a_1x^1 + a_2x^2 + ... + a_nx^n$ con $a_n \neq 0$

donde n es un número natural que denominamos grado, a_i $(0 \leq i \leq n)$ es un número real que denominamos coeficiente de grado i y cada sumando a_ix_i se denomina monomio.

Para poder usar el nuevo tipo de dato se debe incluir el fichero

polinomio.h

Una variable v del T.D.A. polinomio se declara con la sentencia

Polinomio v;

El espacio requerido para el almacenamiento es O(n), donde n es el grado del polinomio.

3.10.2 Operaciones del TDA polinomio.

1. *Polinomio CrearPolinomio (void)*. Constructor del tipo

   - *Devuelve*: polinomio nuevo inicializado a nulo
   - *Efecto*: Obtiene los recursos necesarios para almacenar un nuevo polinomio, lo inicializa con el polinomio nulo y lo devuelve como resultado. La operación se realiza en tiempo O(1).

2. *int Grado (const Polinomio p)*. Consulta del grado del polinomio

   - *Parámetro p*: polinomio del que se desea saber el grado. Está inicializado.
   - *Devuelve*: el grado de p
   - *Efecto*: Devuelve el grado del polinomio de entrada, es decir, el mayor grado de entre los monomios con coeficientes distintos de cero. La operación se realiza en tiempo O(1).

3. *float Coeficiente (const Polinomio p, const int i)*. Consulta de un coeficiente del polinomio

   - *Parámetro p*: polinomio que contiene el coeficiente que se desea conocer. Está inicializado.
   - *Parámetro i*: grado del coeficiente que se desea consultar. $i \geq 0$
   - *Devuelve*: coeficiente a_i, el coeficiente del monomio de grado i.

[10] Un tipo de dato polinomio contendrá, probablemente, más funciones que las aquí expuestas. Sin embargo, nuestra intención es ilustrar con un ejemplo simple los conceptos que se han desarrollado en el tema.

- *Efecto*: La operación se realiza en tiempo O(1).

4. *AsigCoeficiente (Polinomio *p, const int i, const float c)*. Modifica un coeficiente del polinomio.
   - *Parámetro p*: polinomio que contiene el coeficiente a modificar. Está inicializado.
   - *Parámetro i*: grado del coeficiente que se desea modificar. $i \geq 0$.
   - *Parámetro c*: nuevo valor a asignar
   - *Efecto:*: modifica el valor de a_i, de forma que el polinomio p pasa a contener el monomio cx^i. Nótese que el grado del polinomio puede verse modificado. La operación se realiza en tiempo O(1) si $i \leq grado(p)$ y O(i) si $i > grado(p)$

5. *DestruirPolinomio (Polinomio p)*. Destructor del tipo.
   - *Parámetro p*: el polinomio a ser destruido. Está inicializado.
   - *Efecto:* libera los recursos ocupados por *p*. El polinomio destruido deja de ser válido y debe ser creado antes de volver a usarse. La operación se realiza en tiempo O(1).

6. *Polinomio CopiarPolinomio(const Polinomio p)*. Constructor de copia.
   - *Parámetro p*: polinomio que se desea copiar.
   - *Devuelve*: polinomio nuevo inicializado al mismo valor que *p*
   - *Efecto:* Obtiene los recursos necesarios para almacenar un nuevo polinomio, lo iniciliaza al mismo valor que *p* y lo devuelve como resultado. La operación se realiza en un tiempo O(Grado(p)).

7. *void AnularPolinomio(Polinomio *p)*. Asigna el valor polinomio nulo.
   - *Parámetro p*: el polinomio a anular, pasado por referencia (puntero). **p* está inicializado.
   - *Efecto:* Asigna al polinomio **p* el valor nulo (0). La operación se realiza en un tiempo O(1).

8. *void SumarPolinomio (Polinomio *res, const Polinomio p1, const Polinomio p2);*. Suma dos polinomios.
   - *Parámetro res*: puntero al polinomio donde se obtendrá el resultado de la suma. Está inicializado.
   - *Parámetro p1*: primer sumando. Está inicializado.
   - *Parámetro p2*: segundo sumando. Está inicializado.
   - *Efecto*: Realiza la suma *p1+p2* y la almacena en el polinomio **res*. La operación se realiza en un tiempo O(Grado(p1)+Grado(p2))

9. *void RestarPolinomio (Polinomio *res, const Polinomio p1, const Polinomio p2);*. Resta dos polinomios.
   - *Parámetro res*: puntero al polinomio donde se obtendrá el resultado de la resta. Está inicializado.
   - *Parámetro p1*: primer operando. Está inicializado.
   - *Parámetro p2*: segundo operando. Está inicializado.
   - *Efecto*: Realiza la resta *p1-p2* y la almacena en el polinomio **res*. La operación se realiza en un tiempo O(Grado(p1)+Grado(p2))

3.11 Implementación del TDA polinomio.

El tipo de dato polinomio se representa como un puntero genérico. Internamente, se considera un puntero que apunta a

```
struct poli {
    float * coef;
    int MaxGrado;
    int grado;
};
```

donde la estructura tienes los campos

- *float *coef* (Puntero a los coeficientes). Apunta a una matriz dinámica que almacena los coeficientes, en la posición i, el coeficiente de grado i.
- *int MaxGrado* (Grado máximo que se puede almacenar). En cada momento, indica el grado máximo que se puede almacenar. Si el usuario accede a un coeficiente superior hay relocalización de memoria y actualización de esta variable.
- *int grado* (Grado del polinomio almacenado). Indica el grado del polinomio almacenado, es decir, el valor más alto del índice en la matriz *coef* que tiene un flotante distinto de cero.

La implementación de las operaciones se ha dividido en dos módulos, uno con las operaciones básicas que acceden a la implementación (*polinomio.c*) interna y otro con funciones adicionales que se implementan a partir del anterior (*utilpoli.c*).

3.11.1 Función de abstracción.

Un polinomio P del conjunto de valores en la representación (véase invariante de la representación), se aplica al valor del conjunto de valores abstractos

$$p->coef[0] + p->coef[1]X^1 + ... + p->coef[p->grado]X^{p->grado}$$

3.11.2 Invariante de la representación.

La condición para que un objeto representado sea válido es

$$(\forall P \neq 0,\ P->coef[p->grado] \neq 0)$$
$$\text{Y}$$
$$(P[i] = 0,\ \forall i \text{ tal que } p->grado < i \leq p->MaxGrado)$$

3.11.3 Funciones privadas.

1. *static void errorPolinomio (const char *cad)*. Función de gestión de errores.
   - *Parámetro cad*: Mensaje de error.
   - *Efecto:* Cuando ocurre un error en alguna función se llama a ésta con el mensaje de error correspondiente. La gestión es muy simple pues únicamente se ha construido esta función, en la que se imprime el error y se acaba el programa.

3.11.4 Ejemplos de uso.

En esta sección podemos incluir algunos ejemplos de uso del tipo de dato. Por ejemplo, podemos mostrar:

- un programa que lea de la entrada estándar un polinomio y escriba la derivada en la salida estándar.
- una función que transforme un polinomio al valor de su derivada. Desde la función *main* llamar a esta función para sumar un polinomio con su derivada leyendo y escribiendo desde la entrada estándar y a la salida estándar.

- una función para obtener el opuesto de un polinomio usando la función de restar polinomios. Para probarlo leeremos un polinomio, calculamos su inverso y visualizamos la suma.

Véanse las páginas web de la asignatura para consultar los detalles de estos ejemplos.

3.11.5 Fuentes.

En esta sección se debe incluir los ficheros fuente que implementan el tipo de dato desarrollado (véanse en las páginas web de la asignatura).

4 Tipos de datos abstractos en C++.

Aunque un lenguaje no soporte el concepto de tipos abstractos, es posible implementar un diseño basado en éste. Lógicamente, para poder realizarlo, será necesario un esfuerzo adicional ya que la falta de herramientas para ese soporte se debe de sustituir por el uso de las características disponibles y por una disciplina en el uso de los tipos de datos abstractos.

En esta sección se va a introducir al lector en el uso de C++ para implementar tipos de datos abstractos. Este lenguaje amplía las posibilidades del lenguaje C, incorporando multitud de herramientas que dan soporte al concepto de tipo de dato abstracto y por tanto, facilitando el desarrollo de programas. Como es sabido, es un lenguaje que soporta la programación dirigida a objetos (PDO), que podemos en general basarla en los tres principios:

1. Encapsulamiento.
2. Polimorfismo.
3. Herencia.

En esta sección sólo introducimos los conceptos de encapsulamiento y una forma de polimorfismo (sobrecarga de operadores), los cuales nos permitirán construir tipos de datos abstractos. Entrar en mayor profundidad en la PDO está fuera del objetivo de esta lección.

4.1 Conceptos previos.

4.1.1 Paso de parámetros y devolución de resultados en C

Supongamos una función *máximo* que toma dos valores enteros y devuelve como resultado otro entero. Consideremos un trozo de código que llama a esta función y en el que se declaran 3 variables enteras *max,ant,nue* de manera que en *max* almacenamos el resultado de multiplicar 5 por el máximo de las otra dos. En la figura 2 representamos esta situación.

Podemos observar la representación del código principal y la llamada a la función *máximo*. Además representamos:

- En la parte superior una zona de memoria con 3 localizaciones que almacenan las variables *max, ant, nue* (en las direcciones *0x20, 0x30, 0x40*.
- Encima de la fución, otra zona de memoria[11](en rojo), donde encontramos también 3 localizaciones de las variables locales que son necesarias para ejecutar la función. Esta zona empieza a existir cuando se comienza la ejecución de la función y deja de existir cuando se devuelve el control al trozo de código que la ha llamdo.
- Encima de la llamada a la función una localización temporal de memoria donde se almacena el valor devuelto por la función y que es recogido por el trozo de código que la ha llamado.
- Finalmente un conjunto de líneas de distinto trazo y que vienen descritas en la parte inferior de la figura.

[11]Esta zona se localiza en la pila.

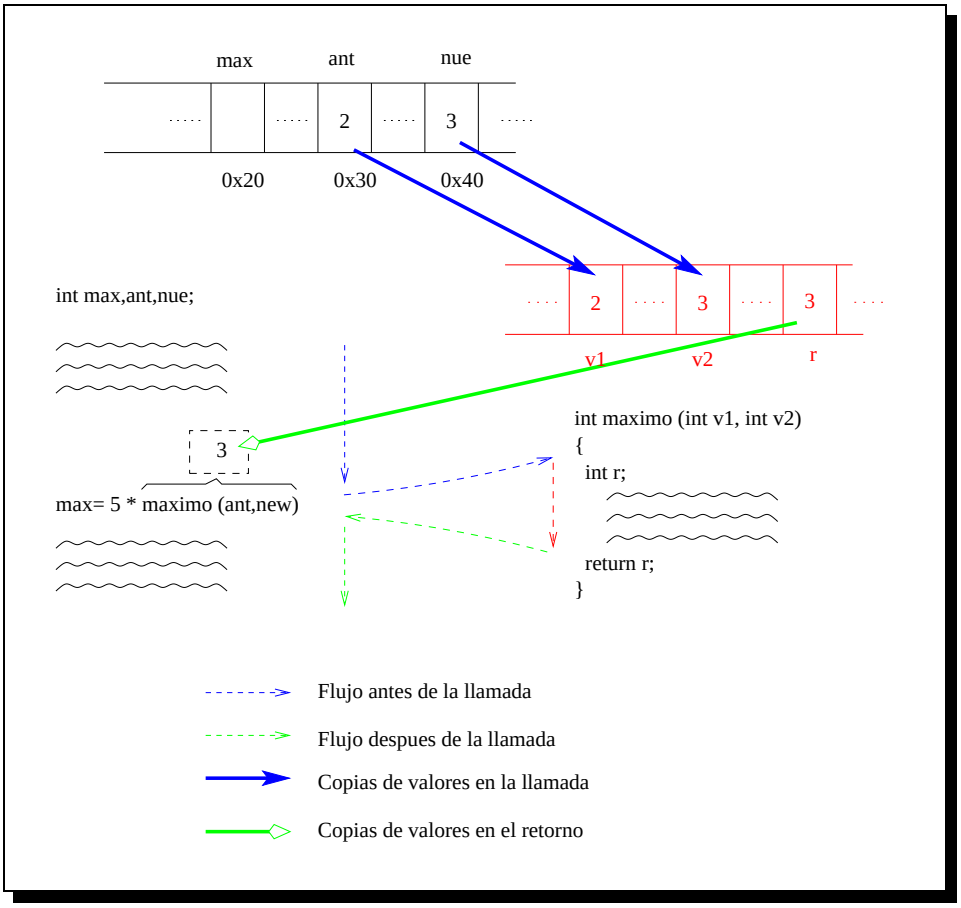

Figura 2: Paso por valor y retorno de valor.

En este ejemplo, se muestra el paso por valor y la devolución del resultado de una función tal como se conoce en *C*. Es interesante ver que en la llamada a la función se crea la zona de memoria para almacenar tanto los valores de los parámetros como las variables locales. Los valores se duplican y por tanto dentro de la función, la modificación de *v1,v2* no afectará a los valores de las variables *ant,nue* del trozo que llama.

A lo largo de la ejecución de la función, se carga el valor de la varible local *r*. Cuando termina la función, se devuelve como resultado su valor y se almacena en una zona temporal[12] desde donde la recoge el código que llama a la función. Nótese que se indica con una flecha de copia tras la ejecución. Después de esta copia la memoria local de la ejecución de la función deja de existir y por tanto los valores ahí almacenados dejan de poder usarse. Una vez finalizada, el valor temporal (3) se multiplica por 5 y se almacena en *max* (que acaba con el valor 15)

Por tanto, en la llamada como en la vuelta se ha realizado la copia de los **valores** que se pasan. Esta forma de comunicación es la que se denomina **"Paso por valor´´**.

Ahora bien, si se desea modificar uno de los parámetros actuales, es necesario buscar un mecanismo para que la función acceda a la variable original. En este segundo caso, mostramos el uso de punteros para poder modificar dichos valores. En la figura 3 se muestra esta situación.

Realmente, el paso sigue siendo por valor, pues ahora es un puntero lo que se transfiere[13]. Por tanto, en esta situación no debemos encontrar nada nuevo en lo que respecta a la forma en que se pasan los parámetros. Como vemos, en la llamada se calcula el valor de una expresión (el operador de obtención de dirección & aplicado a la variable *max*) que devuelve *0x20* (de tipo puntero a entero). Este valor es el que se pasa como primer parámetro y por tanto se copia a la zona donde se sitúa la variable local *res*. Es en la asignación, dentro de la función, cuando se modifica el valor de *max* ya que *res* es un puntero a dicha variable.

[12] Normalmente, también en la pila.

[13] Se puede decir que se realiza el paso por referencia de forma manual.

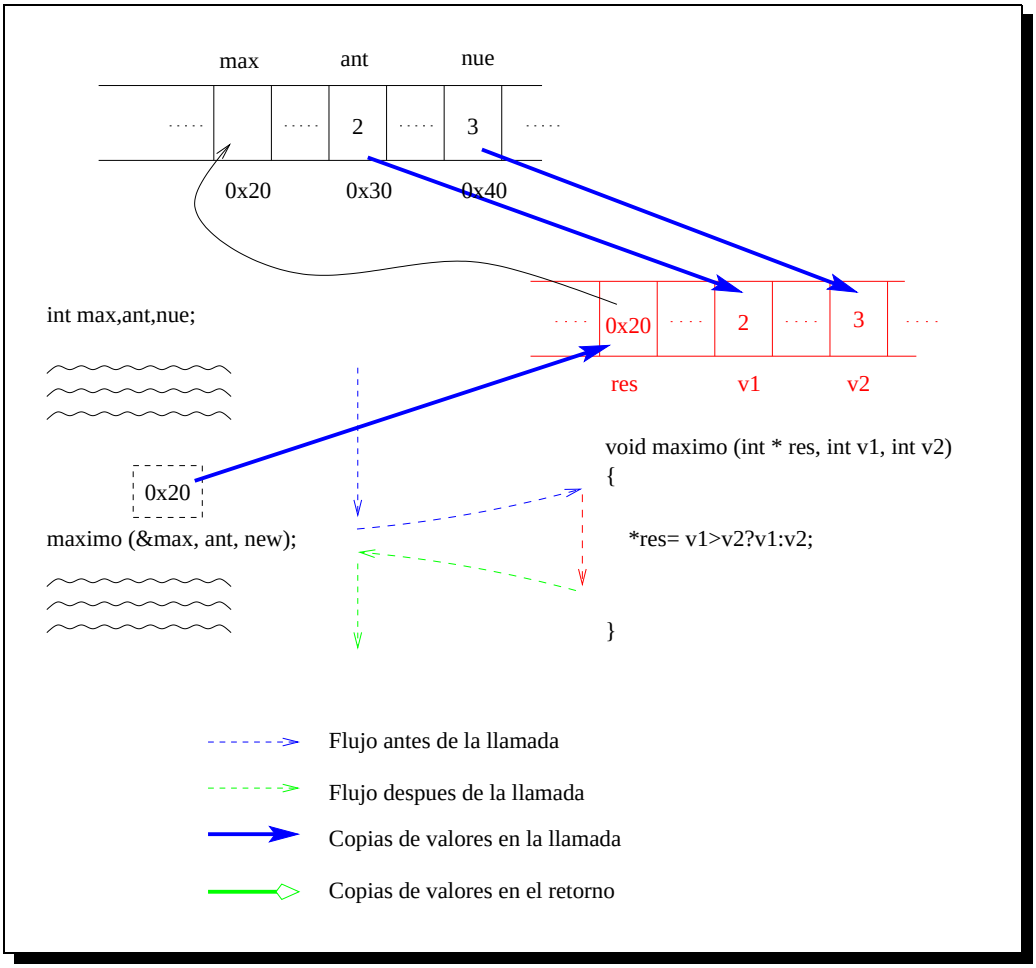

Figura 3: Paso de la dirección(puntero) de un enetero.

En tercer lugar, podemos considerar una situación distinta, en la que uno de los parámetros es un vector de cinco enteros. Esta situación se representa en la figura 4

En este caso, no es buena idea realizar un paso por valor propiamente dicho, ya que sería necesario primero conocer cúantos componentes tiene el vector y después copiarlos en la llamada. En este caso, el compilador simplifica la operación pues cuando se pasa un *"array"*no realiza el paso por valor, sino que se pasa la dirección del primer elemento del *"array"*. Esto tiene dos interpretaciones equivalentes

- La estrecha relación que existe entre un *"array"*y un puntero a su primer elemento[14], permite al compilador interpretar el paso a la función como el paso de un puntero, copíandose el valor de la dirección del *"array"*al valor que almacena el *"array"*(o puntero) en la función. Podemos decir que en el paso de vectores el compilador opta por una interpretación como puntero y realmente el paso sigue siendo por valor.

- La dirección del parámetro actual se copia como la dirección donde se sitúa el parámetro formal. Esto significa que acceder al contenido del *"array"*en la función es acceder al mismo sitio que el *"array"*original. Esto es, el paso de matrices en C se realiza por **referencia**.

En cualquier caso, el lector deberá tener en cuenta que al pasar una matriz a una función implica que se usarán los valores originales y por tanto no existe copia del contenido de la matriz. Por otro lado, es importante insistir en que la interpretación del parámetro actual y formal debe ser idéntica para que el

[14]Nótese que esta relación no es más que la posibilidad de flexibilizar el comportamiento del programa por medio de una adecuada interpretación de la variable como un vector (por ejemplo, al acceder con el operador []) o como un puntero (por ejemplo, al sumar un entero al vector).

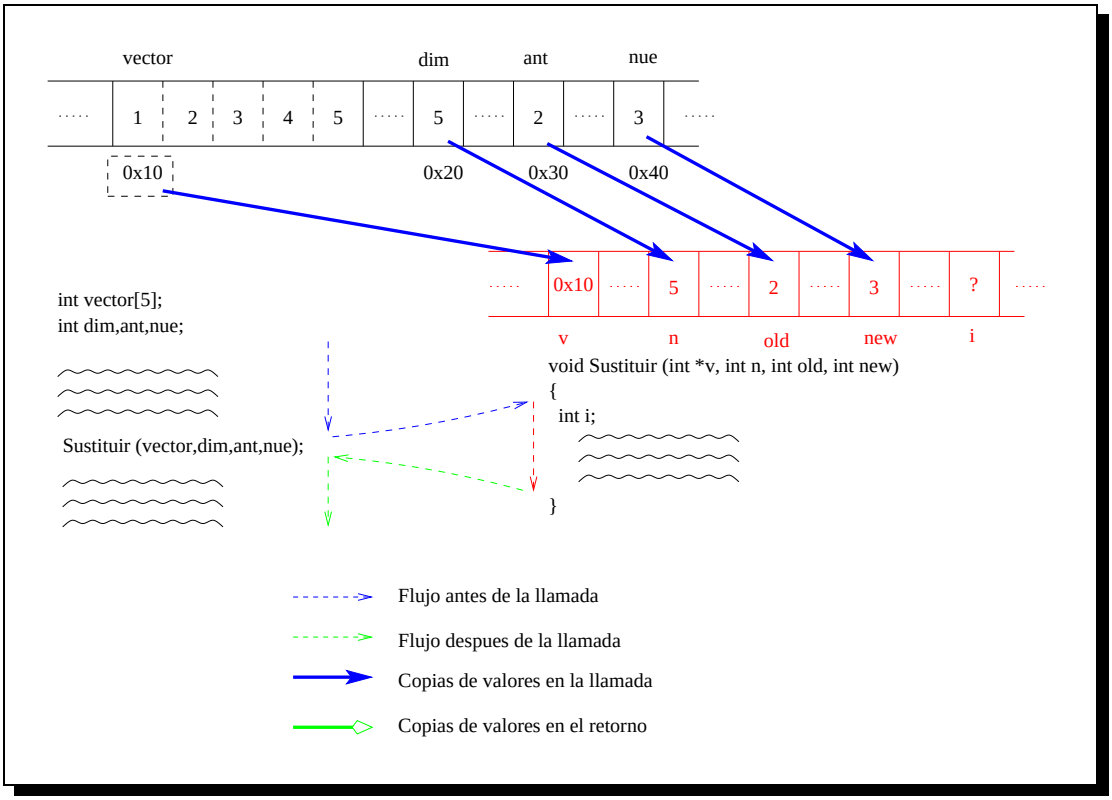

Figura 4: Paso de un vector de enteros.

programa sea correcto. Por ejemplo, no podemos pasar una matriz declarada como *int m[5][10]* como el parámetro formal *int *m[]*.

4.1.2 Paso de parámetros y devolución de resultados en C++. Tipo referencia.

Es muy importante entender el sistema que implementa el lenguaje C++ para pasar parámetros y devolver resultados. En primer lugar, el paso (y devolución) por valor que corresponde a C, se mantiene en C++ con un comportamiento idéntico. Por tanto, la sección anterior sigue siendo válida para este lenguaje. Veamos las nuevas capacidades.

En C++, aparece un nuevo tipo que nos va a permitir nuevas posibilidades en la transferencia de información. Es el **tipo referencia**. La declaración se lleva a cabo añadiendo el caracter & al tipo. Por ejemplo *int& ref* indica que *ref* es una referencia a un entero.

Cuando un programa se ejecuta, se almacenan "cosas" en distintas regiones de memoria. Así, al declarar una variable

```
int a;
```

indicamos al compilador que busque una zona de la memoria que almacenará un **objeto** de tipo entero. Además, el identificador *"a"* se **refiere** a esa zona de memoria (a ese objeto). Cuando escribimos líneas de código, utilizamos esas referencias a objetos para indicar un lugar de almacenamiento[15]. Por ejemplo, si escribimos:

```
a= 5;
```

el compilador entiende que se quiere almacenar el valor 5 en el objeto (lugar de almacenamiento) al

[15] En la literatura, el lector puede encontrar a menudo el concepto de *lvalue* como una expresión que se refiere a un objeto. El nombre proviene de "algo que se puede colocar a la izquierda de una asignación" aunque si declaramos una constante, en un *lvalue* pues se refiere a una zona de memoria pero no se puede poner a la izquierda de una asignación.

que se refiere *a*.

Cuando se declara una variable, el compilador se encarga de que exista un objeto(lugar contiguo de memoria) al que se refiere. Cuando se declara un tipo referencia, se indica que sólo se desea una nueva referencia, pero el compilador no debe encontrar ningún objeto al que se refiera. Eso significa que es similar al caso anterior, pero la variable referencia no nos sirve de nada hasta que no se le indica el lugar de la memoria al que se refiere. Somo nosotros los que, al inicializarla, hacemos que se refiera a una zona de memoria. Podemos decir que una referencia en un nombre alternativo a un objeto. Por ejemplo, si realizamos la siguiente declaración

```
int a=0;
int &ref= a;
```

declaramos un nuevo objeto de tipo entero (una zona que almacena un entero), donde se guarda el valor 0 e indicamos que *a* se refiere a esa zona. Cuando declaramos *ref* indicamos que será una referencia a un objeto entero. Cuando lo inicializamos con *a*, el compilador considera que es la misma referencia y por tanto ambas variables se pueden considerar la misma pues son dos referencias al mismo objeto (zona de memoria). Son dos nombres para un mismo objeto.

La aplicación de este tipo es en el paso de parámetros y devolución de resultados de funciones. En primer lugar, nos permite realizar un paso por variable a una función. Si uno de los parámetros formales de una función lo declaramos de tipo referencia, al llamar a la función, el parámetro actual que se pasa inicializa la referencia al mismo objeto y por tanto la utilización del nombre de la referencia dentro de la función será igual que si se usara dicho parámetro actual. Un ejemplo se muestra en la figura 5

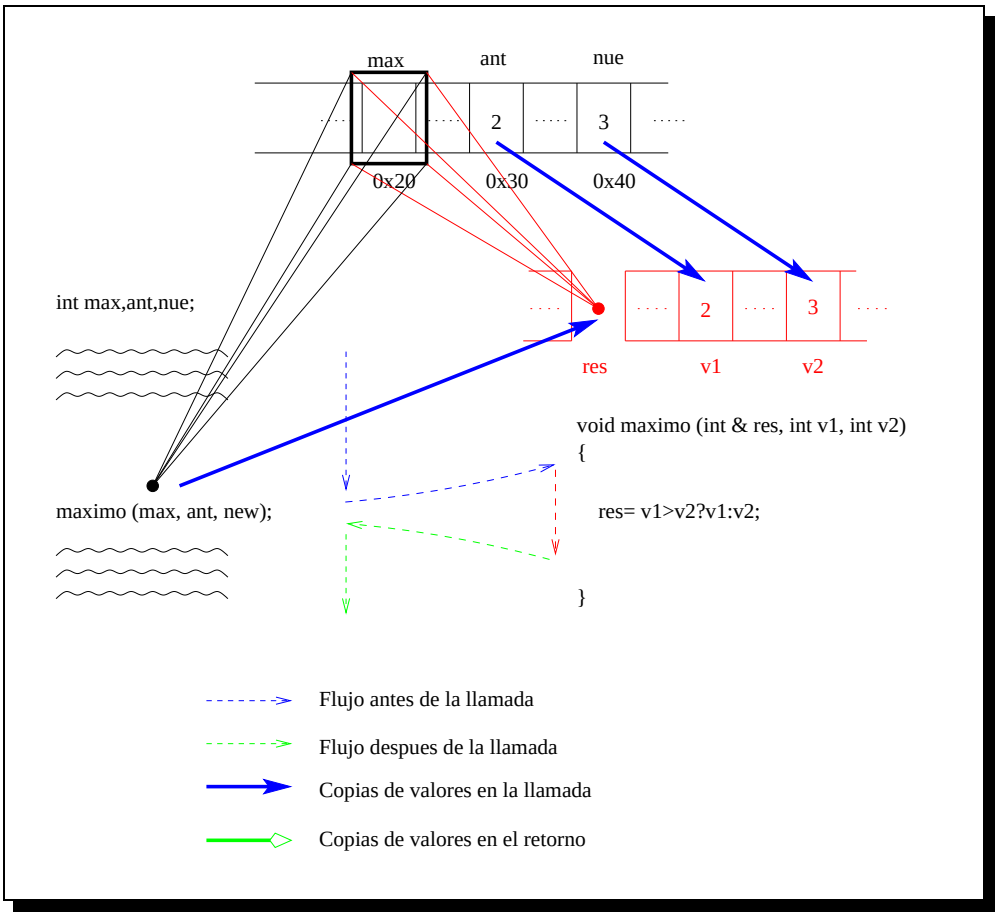

Figura 5: Paso de un entero por referencia.

Como podemos observar, en la zona de memoria local que utiliza la función, se ha buscado espacio para almacenar las dos variables *v1, v2*, que se pasan por valor. En cambio, *res* es una referencia y por tanto no tiene localización asignada. Cuando se llama a la función, se pasa la variable *max* como

parámetro actual de la referencia y por tanto, la variable *ref* se inicializa con el objeto *max*, es decir, es una referencia al mismo lugar de memoria. Por consiguiente, tienen la misma dirección, y el paso del parámetro se ha realizado por **referencia**[16].

En la asignación interior a la función *máximo*, se indica el almacenamiento de un valor en *res* que se refiere al mismo objeto que *max*, y por tanto su modificación implica un cambio en el valor de *max*.

Por último, el concepto de tipo referencia que hemos explicado nos permite fácilmente implementar un mecanismo de devolución de valores por referencia. La sintaxis para esta devolución es muy simple pues sólo debemos de añadir el caracter & al tipo devuelto por la función.

La interpretación de esta operación es idéntica al caso de paso de un parámetro. De la misma forma que al pasar un parámetro el tipo referencia (parámetro formal) se inicializaba con la referencia del objeto que se pasaba como parámetro actual, cuando se devuelve un objeto, la referencia resultado se inicializa con la referencia de dicho objeto. Podríamos pensar, que el resultado de la función es una "variable" del tipo referenciado y que se localiza en el sitio que indique la sentencia de retorno en la función. En la figura 6 se muestra un ejemplo.

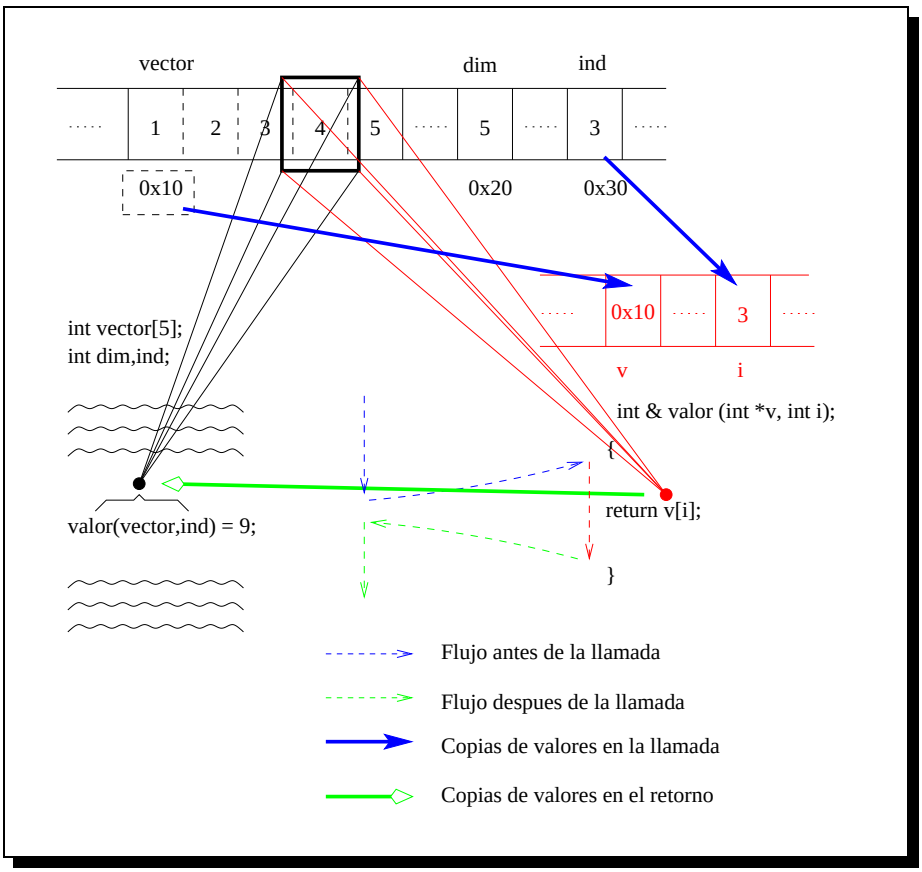

Figura 6: Devolución de un entero por referencia.

En esta figura se muestra una función que realiza una operación muy simple, la devolución de una referencia al elemento *i* de un vector de enteros. Es decir, la función

```
int &valor(int *v, int i)
{
  return v[i];
}
```

Como podemos ver, la llamada a la función se realiza tal como se ha mostrado en los apartados anteriores, mediante la copia de un entero y el paso por referencia del vector. Esto significa que el entero *v[i]* es directamente el valor original en el *"array"vector*.

[16] Nótese la similitud con el caso anterior de paso de un *"array"*a una función en C.

Cuando se ejecuta la orden *return* se asigna como resultado de la función una referencia que se inicializa con el objeto que se indica. Es decir, el resultado es una referencia al objeto que también referencia *v[i]*. Por tanto son dos nombres para una misma localización en memoria.

Tal como indicábamos antes, podemos interpretar que cuando escribimos, en el trozo de código que llama, *valor(vector,ind)* es equivalente a un nombre de variable que se localiza en el lugar donde indica la orden *return*, es decir, la cuarta posición del *"array"vector*.

4.1.3 Parámetros constantes.

Cuando se utiliza el paso por valor, el programador no se tiene que preocupar de que la parte interna pueda modificar el valor del parámetro actual ya que se trabaja con una copia. Sin embargo, cuando consideramos el paso por referencia, es necesario tener en cuenta si el objeto se modifica. Para poder controlar esta situación, se puede usar la palabra reservada *const*.

Cuando se utiliza la palabra *const* se indica al compilador que no se puede modificar el valor del objeto. Así, la más inmediata aplicación que podemos considerar es la declaración de constantes en un programa. Estas constantes se inicializan, y ya no vuelve a modificar su valor.

De la misma forma, podemos usarla en la declaración de un parámetro para indicar que no se puede modificar. Supongamos que se desea una función de búsqueda de un entero en un *"array"*ordenado de enteros. La cabecera se puede declarar como

```
int busqueda (int *v, int n, int el)
```

En este caso, sabemos que la modificación interna de los parámetros formales *n* y *el* no afectará al código que la llama, ya que se pasan por valor. En cambio, si la función modifica los valores a los que apunta *v*, ese código se verá afectado. La solución a este problema es indicar al compilador que los valores a los que apunta ese puntero son constantes y, por tanto, no pueden ser modificados. La forma correcta de declarar esta cabecera es

```
int busqueda (const int *v, int n, int el)
```

Así, por un lado el que la programa no puede equivocarse ya que el compilador avisará de una operación que viole esta modificación y por otro, el que la usa sabrá que la matriz queda intacta.

Adicionalmente, podemos considerar el uso de *const* para los dos parámetros restantes. En este caso, el que usa la función no encuentra ninguna ventaja (normalmente este tipo de parámetros se declaran sin especificar *const*, sin embargo, puede tener sentido si el que programa quiere garantizar que a lo largo de toda la función dichos parámetros no se modifican. La ventaja de esta declaración será por consiguiente que el usuario tiene garantizado en cualquier punto de la función que el valor que se almacena es exactamente el original que se pasó a la función. No puede cometer el error de modificar un parámetro en un punto de la función y usarlo posteriormente pensando que almacena el valor que originalmente se pasó.

Por otro lado, en C++ aparecen el tipo referencia y por tanto la posibilidad de modificar el parámetro actual. Inicialmente, podemos pensar que si alguien declara un paso por referencia es porque desea tener la posibilidad de modificar dicha variable, sin embargo, no es así. Es posible desear pasar una variable a una función, que no la modifique, pero que tampoco haga una copia a causa del paso por valor. Considere por ejemplo que tiene una variable de un tamaño considerable y quiere pasarla a una función que no la modificará, pero que perderá bastante tiempo para realizar la copia. En este caso es ideal el paso por referencia ya que será mucho más eficiente, pero nos encontramos con el mismo problema que antes, cuando pasamos un *"array"*que no deseamos modificar. De nuevo, añadir la palabra *const* indica al compilador que no se debe modificar, a pesar de que pasamos una referencia y trabajamos, por tanto, con el objeto original.

Por ejemplo, si tenemos el tipo *Gigante* que ocupa mucho espacio y deseamos una función para procesarlo sin que sea necesaria la copia, podemos usar una cabecera como

```
int Procesa (const Gigante& par)
```

en la que se pasa por referencia el parámetro pero se indica que no se modifica en el interior de la función.

Finalmente, indicar que *const* también se puede usar en los parámetros que se devuelven aunque es un uso más extraño que no necesitamos considerar en esta lección (ver por ejemplo Stroustrup[8] si se desean más detalles).

4.1.4 Inclusión de archivos cabecera.

La inclusión de archivos cabecera sigue las mismas normas que en el lenguaje C. Sin embargo, en el estándar se establece que los archivos de la biblioteca de C++ no usan la extensión *.h*. Por tanto, si consideramos un archivo estándar, por ejemplo, *iostream* la inclusión en el programa C++ se realizará mediante la sentencia

```
#include<iostream>
```

Por otro lado, y debido a que se mantiende la compatibilidad con los archivos cabecera de C, es posible usar éstos de la forma usual, por ejemplo

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

Sin embargo, a causa de algunos cambios para integrarlos en el entorno de C++, se han establecido nuevos archivos cabecera que, correspondiendo a los conocidos de C, se renombran con una *'c'* delante y pierden la extensión. Así, el ejemplo anterior se incluye como

```
#include<cstdio>
```

Finalmente, estos nombres correspondientes a los archivos estándar de C y C++ no establecen ninguna norma sobre los nombres de los archivos cabecera, sino que se han definido para poder reescribir los archivos estándar sin que el usuario se vea especialmente afectado. Así por ejemplo, el usuario puede seguir creando nuevos archivos cabecera con la extensión *'.h'*.

4.1.5 E/S básica en C++.

El lenguaje C++ soporta el sistema de E/S del lenguaje C. Sin embargo y a pesar de las posibilidades de este lenguaje, es necesario desarrollar un sistema de E/S dirigido a objetos. Como ventajes más importantes, podemos encontrar que

- El usuario puede extender este sistema para poder realizar operaciones de E/S con los tipos definidos por el usuario. Así, no es necesario construir una función con una sintaxis específica para cada nuevo tipo del usuario, sino que se puede utilizar como cualquier otro tipo del lenguaje.
- En C no es posible realizar una comprobación de tipos. Si usamos la función *scanf*, se recibe una lista de direcciones para las que no se comprueba la correspondencia en tipos. En C++, la operación se determina en función del tipo de dato.

El estudio de este sistema en profundidad requeriría un espacio muy largo e incluso de varias lecciones y un conocimiento más profundo sobre polimorfismo y herencia en C++. Por tanto, en esta sección sólo se va a introducir al lector a las operaciones más simples que son necesarias y suficientes para entender la lección.

En ambos lenguajes, se opera con flujos (*stream*). Al igual que C, en C++ existem varios flujos predefinidos que se abren al comenzar la ejecución de un programa. En C, éstos eran *stdin, stdout, stderr* (el primero se abre para entrada y los otros para salida). En C++, cambian los tipos y los nombres. En lugar del tipo único *FILE ** para el que se tiene que definir el caracter de entrada o salida en la apertura, ahora consideraremos dos tipos básicos:

- Entrada. El tipo del flujo de entrada es **istream**. El flujo *stdin* es ahora **cin**.
- Salida. El tipo del flujo de salida es **ostream**. Los flujos *stdout* y *stderr* son ahora **cout** y **cerr**.

La forma de realizar una entrada o salida se simplifica considerablemente ya que no es necesario informar detalladamente al compilador (por medio de una plantilla de formato) de los tipos de datos que se manejan ya que es éste quien se encarga de realizar la operación que corresponde a cada tipo. La

forma de llevarlo a cabo es mediante los operadores '>>' y '<<' aplicados sobre los flujos de entrada y salida respectivamente.

Por ejemplo, si queremos realizar una lectura de un entero *a* y un real *b* desde la entrada estándar y escribir los valores de cada uno y de su suma a la salida estándar, se puede realizar con

```
int a;
float b;

cin ≫ a;
cin ≫ b;
cout ≪ ``La suma de '';
cout ≪ a;
cout ≪ `` y '';
cout ≪ b;
cout ≪ `` es '';
cout ≪ a+b;
cout ≪ endl;
```

en donde podemos ver que se han realizado entradas de distintos tipos (*int, float*) así como salidas también de distintos tipos (*int, float, char *, char*). En la última línea se escribe *endl*, que en C++ indica un salto de línea (equivalente a escribir '\n').

Sin embargo, puede resultar muy incómodo reescribir el flujo de entrada o salida muchas veces para una salida compleja. Afortunadamente, en C++ estos operadores sobre flujos se han definido de manera que se evalúan de izquierda a derecha devolviendo en cada paso, como resultado, el mismo flujo. Por tanto, al igual que en C se pueden encadenar asignaciones con el operador '=', en C++ se pueden encadenar operaciones de E/S. Así el ejemplo anterior se puede expresar como

```
int a;
float b;

cin ≫ a ≫ b;
cout ≪ ``La suma de '' ≪ a ≪ `` y '' ≪ b ≪ `` es '' ≪ a+b ≪ endl;
```

Estos flujos y operadores los tenemos disponibles incluyendo el fichero de cabecera **iostream**.

4.1.6 Gestión de errores.

La forma natural de gestión de errores en C++ es mediante el manejo de **excepciones**. Sin embargo, estudiarlas requiere un explicación muy extensa y, por tanto, no se presenta en esta lección.

Para poder llevar a cabo algún tipo de gestión de errores, consideraremos una solución simple, haciendo uso de la función **assert** de la biblioteca estándar de C (y por tanto, también disponible en C++).

La función tiene un único argumento, un valor booleano. Si el argumento es falso, la función escribe un mensaje indicando donde se ha producido el error y para el programa (llamando a la función *abort*).

Como es de esperar, el uso de la función *abort* es de difícil aceptación en una versión final de un programa[17].

Para facilitar el uso de la función *assert* el compilador nos permite definir **NDEBUG**. Cuando se define, el compilador no incluye las comprobaciones que se realicen con esta función.

El fichero cabecera donde se declara la función *assert* es **cassert** (en C, el fichero *assert.h*).

```
void buscar (const int v, int dim, int el)
{
   assert(dim>0);
   ...
}
```

[17] A pesar de ello se pude considerar el dejar algunas llamadas, en los casos en que se produzca un error fatal y se desea tener alguna información sobre algo que nunca debiera haber ocurrido.

4.2 Encapsulamiento.

Un concepto fundamental en C++, como se indicó anteriormente, es el de encapsulamiento. Las discusiones a lo largo de la lección sobre abstracción se han apoyado en gran parte sobre esta idea. La implementación de los T.D.A. en C que se ha mostrado es un claro ejemplo de su aplicación. Hemos encapsulado una estructura de datos como la representación de un nuevo tipo de dato y se han encapsulado en cada una de las operaciones el conjunto de pasos necesarios para operar con esa estructura.

Es interesante ver que a pesar de las nuevas posibilidades con la definición de T.D.A. en C, aún podemos reforzar este concepto ya que el T.D.A. es una representación junto con sus operaciones. Por tanto, es más conveniente encapsular tanto la representación como las operaciones en un mismo objeto.

Un ejemplo que puede ayudar a entender este paso es considerar cada uno de los objetos electrónicos de un hogar como un objeto independiente. Cada uno de ellos tiene un interface (unos mandos) y unos circuitos internos que se encargan de llevar a cabo la operación correspondiente. Cada objeto tiene encapsulado la parte interna y las operaciones que se pueden realizar con él. Si consideramos un situación como la desarrollada en los ejemplos anteriores en C, tendríamos por un lado los aparatos y por otro los interfaces para manejarlos. Es mucho más simple, y proporciona más independencia del resto, el considerar los dos partes como un único objeto. Una representación gráfica de esta idea se muestra en la figura 7.

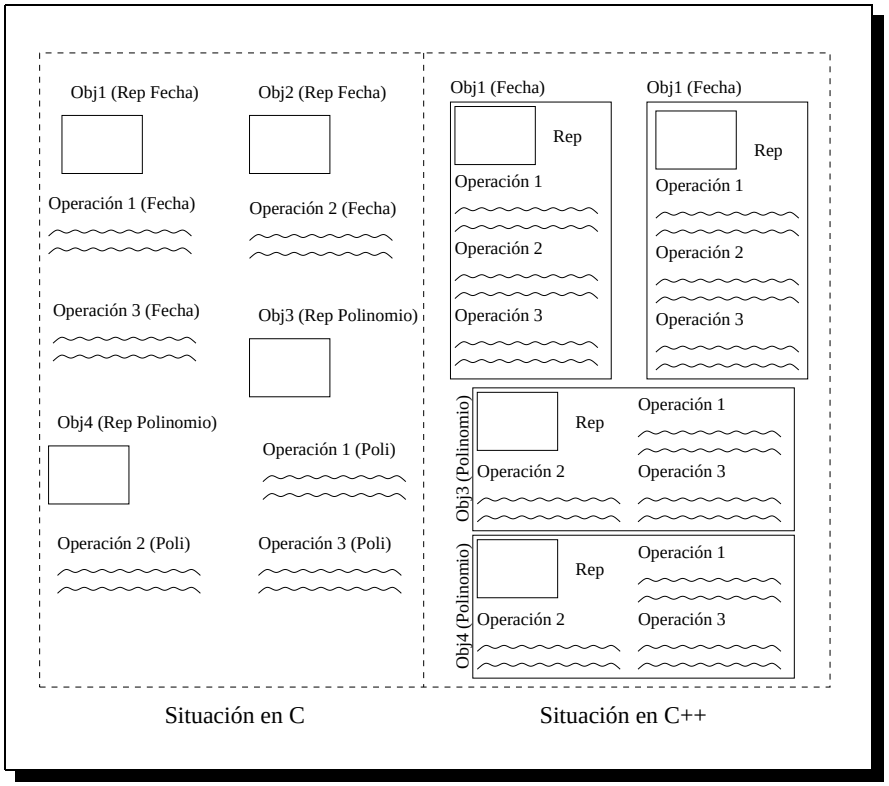

Figura 7: Encapsulamiento en C++.

Como el lector puede ver, los objetos encapsulan tanto los datos como las operaciones[18]. Este nuevo tipo definido por el usuario, se denomina **clase** y cada una de las instancias de una clase, **objeto**. Para poder implementarlo, el lenguaje permite la definición, dentro de una estructura, de datos y operaciones. A cada uno de los campos de la estructura se denomina **miembro**. Por ejemplo, si queremos definir el

[18] Aunque gráficamente aparecen varias instancias del código de cada operación (por cada objeto), la representación quiere mostrar la forma en que el usuario ve el interface. El compilador, como es de esperar, sólo incluirá una instancia de código por cada función.

tipo *Fecha* podemos hacer

```
struct Fecha {
  int dj;                                          // Representación interna como entero

  void AsigDJFecha(int DJ);                        // Asigna la fecha del día juliano DJ
  int DiaFecha();                                  // Devuelve el dia que corresponde a la fecha
  int NumeroSemanaFecha();                         // Devuelve el número de semana
  ...
};
```

o si queremos definir el tipo *Polinomio* podemos hacer

```
struct Polinomio {
  float *coef;                                     // Matriz interna con coeficientes
  int grado;                                       // campo que almacena grado
  int MaxGrado;                                    // Máximo espacio reservado

  void AsigCoeficiente (int i, float c);           // Asigna al coef i el valor c
  int Grado();                                     // Devuelve el grado
  Coeficiente(int i); Devuelve el coeficiente del grado i
  ...
};
```

Nótese como las funciones que aparecen son prácticamente como las que se han mostrado anteriormente en C, excepto que pierden un parámetro, el del objeto sobre el que se aplica la operación. La razón de ello es que los datos están junto con las operaciones y por tanto para llamar a una operación, se selecciona como campo de un objeto que existe (sobre el que se aplica la operación). Por ejemplo, para declarar objetos de estos tipos y llamar a operaciones se utiliza una sintaxis como la siguiente

```
Fecha f;
Polinomio p;
...
f.AsigDJFecha(2000000);
cout << ``Día:  '' << f.DiaFecha() << ``\n'';
...
p.AsigCoeficiente(p.Grado(),0);
...
for (i=0;i<=p.Grado();i++)
  if (p.Coeficiente(i)! =0)
    cout << ``Grado '' << i << `` coeficiente '' << p.Coeficiente(i);
...
```

Así, para cada llamada a una operación, se selecciona la función correspondiente del objeto sobre el que se quiere operar. Por tanto, no es necesario indicar una variable sobre la que aplicarla. Ahora bien, surgen tres cuestiones que se deben resolver:

- Se ha mostrado la declaración de las operaciones. ¿Cómo se definen?
- No es necesario pasar el objeto sobre el que se aplica la operación. ¿Cómo se hace referencia a él dentro de la definición?
- ¿Cómo se puede indicar que el objeto sobre el que se aplica la operación es *const* o no?

La forma de definir una operación es la misma que en C. La única cuestión a tener en cuenta es que la operación pertenece a la definición de una clase y por tanto el nombre debiera ser algo como "la función n que es miembro de la clase c" (distintas clases pueden tener un miembro con el mismo nombre). Para ello, se utiliza "::" para seleccionar el contexto o ámbito del identificador de función. Por ejemplo, en el ejemplo mencionado la función es *"c::n"*.

Por otro lado, dentro del código que define una función, es necesario poder hacer referencia al objeto sobre el que se aplica la operación. Esto se lleva a cabo usando un identificador predefinido, **this** que está definido como un puntero al objeto.

Como ejemplo, mostramos la posible definición para un par de funciones de los tipos que estamos discutiendo

```
int Fecha::DiaSemanaFecha()
{
  return this→dj%7;
}

int Polinomio::Coeficiente (int i)
{
  if (i>this→grado)
    return 0;
  else return this→coef[i];
}
```

A pesar de ello, el lenguaje nos permite prescindir del uso de *this*, ya que cuando se utiliza un identificador que no es local a la función, se busca en el alcance de la clase, y se entiende que es el miembro que corresponde al objeto. Así, en el ejemplo anterior se puede escribir equivalentemente

```
int Fecha::DiaSemanaFecha()
{
  return dj%7;
}

int Polinomio::Coeficiente (int i)
{
  if (i>grado)
    return 0;
  else return coef[i];
}
```

Finalmente, no hemos indicado el carácter constante del objeto sobre el que se aplica la operación. Por ejemplo, la función *DiaSemanaFecha* o *Coeficiente* recibían un objeto constante de tipo *Fecha* y *Polinomio* respectivamente. La forma de indicarlo ahora cambia ya que no aparece explícitamente el objeto en la cabecera de la función. Si deseamos indicar que el objeto no se modifica (es constante) deberemos añadir la palabra *const* después de la cabecera de la función, es decir, las definiciones anteriores se deben realizar

```
int Fecha::DiaSemanaFecha() const
{
  return dj%7;
}

int Polinomio::Coeficiente (int i) const
{
  if (i>grado)
    return 0;
  else return coef[i];
}
```

Nótese que las cabeceras también aparecen en la definición de la clase y por tanto también se deberá añadir en ese lugar.

4.2.1 Control de acceso. La palabra clave class.

Si queremos obtener una buena herramienta para construir un T.D.A. es necesario crear una forma de control de acceso a los miembros de una clase. Por ejemplo, en el caso del tipo *Polinomio* que hemos visto, el usuario del tipo podría acceder directamente a los campos *coef, grado, MaxGrado*, cuando esto debería considerarse incorrecto ya que corresponden a la parte privada, no al interface como vimos en la versión C del tipo. En C++, es posible indicar los miembros que se deben considerar privados y públicos.

Por defecto, todos los miembros de una clase declarada con la palabra *struct* se consideran públicos y por tanto en los ejemplos anteriores el usuario puede acceder a la parte interna. Se propone una nueva palabra clave, **class** que se utiliza como en el caso anterior y para la que por defecto todos los miembros son privados.

En la práctica, los programadores suelen utilizar la palabra clave *class* para definir nuevos tipos mientras que *struct* sigue usándose con el significado habitual en C[19].

Para indicar que un grupo de miembros es privado, se antepone **"private:"** y para indicar que son públicos **"public:"**. Por ejemplo, para el caso de *Polinomio* la declaración se podría realizar

```
class Polinomio {
  private:
    float *coef;                                  // Matriz interna con coeficientes
    int grado;                                    // campo que almacena grado
    int MaxGrado;                                 // Máximo espacio reservado

  public:
    void AsigCoeficiente (int i, float c);        // Asigna al coef i el valor c
    int Grado() const;                            // Devuelve el grado
    Coeficiente(int i) const; Devuelve el coeficiente del grado i
  ...
};
```

donde como hemos indicado, la palabra *private* es opcional pues se ha declarado la clase con *class*.

4.2.2 Constructores y destructores.

En el desarrollo de un T.D.A. juega un papel fundamental las operaciones de construcción y destrucción. Éstas resuelven fundamentalmente dos problemas

- Permiten inicializar el objeto con un dato válido. Con ello, garantizamos que se cumple el invariante y que todas las operaciones que se llamen correctamente se ejecutarán con éxito. Nótese como en los ejemplos que vimos en C, se insistía en que las operaciones se realizaran sobre un objeto inicializado, como era de esperar.

- Permiten reservar (constructores) y liberar (constructores) los recursos necesarios para manejar el tipo. Por ejemplo, en los polinomios era necesario solicitar memoria en la construcción y liberarla en la destrucción.

Además, su uso es bastante incómodo pues tenemos que tener cuidado en llamar al constructor cuando se declara una variable y al destructor cuando se deja de usar.

Para resolver este problema, el lenguaje C++ integra estas funciones en la definición de una clase. Así, considera que todas las clases tienen al menos un constructor (por defecto) y el destructor del tipo con unos nombres predeterminados. El hacerlo así, permite al compilador incluir el código para llamar a estas funciones de manera automática. Cuando se declara una variable, se llama automáticamente al constructor para preparar recursos e inicializarla y cuando esa variable deja de existir se llama automáticamente al destructor.

Los nombres predeterminados de estas funciones son

- Los constructores tienen el mismo nombre que la clase.

- Los destructores tienen el mismo nombre de la clase precedido por el símbolo ~

[19] Algunos autores defienden su uso cuando se desee que una clase tenga todos sus miembros públicos, siendo una forma sencilla de resaltar esta característica.

Por lo tanto, podemos añadir una función constructora y otra destructora a la clase polinomio. En el fichero cabecera aparece como

```
class Polinomio{
  ...
  public:
    Polinomio();
    ~Polinomio();
    void AsigCoeficinete (int i, float c);
    ...
};
```

mientras que en el fichero *.cpp* encontramos

```
Polinomio::Polinomio()
{
  int i;

  this→MaxGrado= 10;                          // Asignamos un tamaño por defecto
  this→coef= new float[p→MaxGrado+1];

  // Inicializamos a nulo
  for (i=0;i<=this→MaxGrado;i++)
    this→coef[i]= 0.0;
  this→grado=0;
}
Polinomio::~Polinomio()
{
  delete[] this→coef;
}
```

donde podemos observar que no se debe declarar ningún tipo devuelto por la función.

El hecho de indicar que el destructor es el de por defecto se debe a que en una clase se pueden definir varios constructores. Ya que se desea un mecanismo, no sólo para crear, sino también para inicializar cuando se declara una variable, podemos declarar varias funciones que teniendo el mismo nombre (el de la clase) tengan distintos parámetros. Todas ellas son constructoras y el compilador es el encargado de seleccionar la adecuada dependiendo del número y tipo de parámetros que usemos.

Por ejemplo, podemos desear otro constructor para el tipo *Polinomio* en el que se indique con un entero el máximo grado que alcanzará. Esto evitará que se tenga que relocalizar memoria cuando crezca. Ahora aparecen dos constructores, uno sin parámetros y otro con un entero.

El único que debe de existir de manera obligatoria es el constructor por defecto para poder permitir al compilador gestionar una declaración de variable. De hecho, si no se define, el compilador asigna uno por defecto[20].

[20] El constructor por defecto inicializa también por defecto cada uno de los miembros. El programador es el responsable de garantizar que el constructor es válido.

En el fichero cabecera añadimos el nuevo constructor, es decir, aparece como

```
class Polinomio{
  ...
  public:
    Polinomio();
    Polinomio(int MaxG);
    ~Polinomio();
    ...
};
```

mientras que en el fichero *.cpp* encontramos la nueva definición sustituyendo el valor que antes usábamos por defecto por el valor del parámetro. Podemos escribir

```
Polinomio::Polinomio()
{
...
}
Polinomio::Polinomio(int MaxG)
{
  int i;

  assert(MaxG>=0);
  this→MaxGrado= MaxG;                    // Asignamos un tamaño por defecto
  ...
}
```

donde hemos incluido un ejemplo de uso de la función *assert* que asegura el que el parámetro que se pasa el válido (seguramente tendremos como precondición que el parámetro no sea negativo).

Cuando declaramos una variable se puede indicar que se quiere inicializar usando el constructor con tamaño explícito.

```
Polinomio p;                    // El polinomio p no tiene tamaño (se creará con 10)
Polinomio q(20);                // Se crea con espacio suficiente para grado 20
```

Nótese como este tipo de declaración se puede usar para el caso del T.D.A. Fecha para inicializar un objeto con una fecha predeterminada. Para ello, se propone un constructor que recibe como parámetros tres enteros indicando día, mes, y año de forma que la fecha queda inicializada con esos valores. Por ejemplo, podemos declarar la fecha *navidad* con

```
Fecha navidad(25,12,anio);                    // Fecha 25-12 del año "anio"
```

4.2.3 Constructores de copias.

La intención final cuando se definen nuevos tipos de datos es que el comportamiento sea idéntico a los tipos predefinidos del lenguaje C++. Así, cuando usamos un tipo *Polinomio* deseamos que se comporte como un tipo *int* independientemente de su implementación.

Un caso claro donde no se da esta circunstancia es en el paso o devolución de parámetros. Si pasamos un tipo *Fecha* por valor a una función, el comportamiento es correcto ya que se hace una copia del objeto (se copian todos sus miembros), y por consiguiente dentro de dicha función no se tocará el valor del parámetro actual. Sin embargo, en el caso del tipo *Polinomio* la situación no es la misma. El paso por valor de un objeto de esto tipo implica la copia igual que antes, pero uno de sus campos es un puntero y por tanto el polinomio original y la copia comparten la misma zona de almacenamiento de coeficientes. Es decir, realmente no se ha sabido copiar el parámetro actual.

Para solucionar este problema, el programador del nuevo tipo debe enseñar mediante una función la forma de hacer copias. Esta se denomina, como es de esperar, el constructor de copias.

La forma de definir la función es mediante un nuevo constructor que toma como parámetro un objeto del mismo tipo. Como es un constructor, su nombre debe ser también el de la clase y como necesita un parámetro (el objeto a copiar) debemos declarar un parámetro que

- Pasa por referencia. Esto evita tener que hacer una copia (paso por valor). ¡Nótese que estamos definiendo el constructor de copias!. Además si el tipo es de un tamaño considerable será más eficiente.
- Se declara constante. El objeto que se copia no debe ser modificado. Al declararlo como referencia podríamos hacerlo, por tanto, indicamos que es constante para que el comportamiento sea correcto.

Por ejemplo, para el tipo *Polinomio*, añadimos una nueva función en la definición de la clase y la definimos en el fichero *.cpp* por ejemplo como

```
Polinomio::Polinomio(const Polinomio& orig)
{
  int i;

  this→MaxGrado= orig.grado;
  this→coef= new float[p→MaxGrado+1];

  // Copiamos los coeficientes
  for (i=0;i<=this→MaxGrado;i++)
    this→coef[i]= orig.coef[i];
  this→grado=orig.grado;
}
```

Nótese que el usuario no tiene que realizar las llamadas a la función cuando se paso un objeto por valor, sino que es el compilador en que se encarga de realizarlo de manera automática. Para el usuario, simplemente, el comportamiento es idéntico al de los tipos simples de C++.

4.2.4 Encapsulamiento de variables globales.

Cuando se define una nueva clase, tal vez es necesario declarar datos que no corresponden a un determinado objeto, sino que son globales a la clase y por tanto, todos los objetos los comparten. Un caso claro es el T.D.A. Fecha que hemos presentado. En éste, existen tres variables internas y que almacenan el primer día que se considera el calendario gregoriano.

En lenguaje C, estas variables que se declaran en el fichero *.c* como *static* eran locales al fichero y no se conocían fuera de éste. En C++, se podría utilizar el mismo método (declarándolas en el fichero *.cpp* que, aunque funcionaría, resultaría una solución incorrecta ya que esas variables están en el ámbito de la clase y el declararlas fuera oscurece el código y van en contra de nuestro objetivo de encapsulamiento y programación modular. Además, resulta más conveniente otro método si queremos que se conozcan fuera del fichero *.cpp*.

La solución es declararlas como miembros de la clase. El problema es que se considerarían tres campos independientes para cada uno de los objetos de la clase. Para indicar que son globales se les añade la palabra *static*.

Por tanto, un miembro *static* de una clase es global a ésta y sólo existe una ocurrencia para todos los

objetos. En nuestro caso, la declaración sería

```
class Fecha {
  private:
    static int diaGregoriano, mesGregoriano, anioGregoriano;
    static int diaJulianoGregoriano;
    int dj;                                    // Representación interna como entero
  public:
    Fecha();
    Fecha(int d, int m, int a);
  ...
};
```

que necesitaría de una inicialización que no se puede hacer en la definición de la clase. Por tanto en algún otro sitio (usualmente en el fichero *.cpp* que define funciones de la clase se añaden líneas como

```
int Fecha::diaGregoriano=15;
int Fecha::mesGregoriano=10;
int Fecha::anioGregoriano=1582;
int Fecha::diaJulianoGregoriano=2299161;
```

donde podemos observar como ha sido necesario anteponer *Fecha::* al nombre de cada función ya que pertenecen a ese ámbito de resolución. Nótese además que son miembros privados y por tanto no podrán ser modificados si no es por alguna función de la clase (o función que tiene acceso a la parte privada como se ve en el siguiente apartado). Como vemos el encapsulamiento es a nivel de clase y no de objeto ya que algo que corresponde a la parte privada de la clase puede ser modificado por cualquier función de ésta[21].

4.2.5 Funciones auxiliares.

A pesar de nuestro interés por conseguir encapsular un T.D.A. en una clase, es posible encontrar funciones que no correspondan a una operación sobre un determinado objeto, o incluso que aunque trabajen sobre algún objeto no necesitan acceder a la parte privada (el definirlas fuera de la clase nos permite simplificar su interface y facilitar el mantenimiento si se modifica la representación).

Estas funciones pueden denominarse auxiliares ya que no pertenecen a la clase pero se definen para trabajar con ella. Un ejemplo del T.D.A. *Fecha* lo constituye por ejemplo el definir una función que nos devuelva si un año es bisiesto (es independiente de los objetos de la clase pero es conveniente incluirla como función del T.D.A.).

Entre este conjunto de funciones auxiliares, es posible encontrar algunas que necesiten tener acceso a la parte privada de la clase. Esto nos plantea un problema ya que si no corresponden a miembros no pueden acceder a la parte interna. La solución viene indicando que dicha función es *amiga* de la clase, mediante la declaración de la cabecera dentro de la clase y anteponiendo la palabra **friend** .

Por ejemplo, en el tipo *Fecha* nos encontramos con la función *FijarCambioGregoriano* que no se aplica sobre ningún objeto. La podemos definir, al igual que la función bisiesto, como una función auxiliar, pero necesita tener acceso a los miembros privados (las variables globales) pues los modifica. Para poder

[21] Por ejemplo, si llamamos a una función sobre un objeto, puede modificar los miembros privados de un objeto distinto.

realizarlo se añade la cabecera a la clase indicando que es una función *amiga* de la siguiente forma

```
class Fecha {
  private:
    static int diaGregoriano, mesGregoriano, anioGregoriano;
    static int diaJulianoGregoriano;
    int dj;
  public:
    Fecha();
    Fecha(int d, int m, int a);
    friend void FijarCambioGregoriano(int d, int m, int a);
  ...
};
```

Así por ejemplo, podrá acceder a la variable *Fecha::diaGregoriano* sin ningún problema.

Finalmente, nótese que otra solución para este problema podría haber sido declarar la función en el alcance de la clase como *static*, es decir, global a la clase (no corresponde a ningún objeto) y a la que se podría invocar como *Fecha::FijarCambioGregoriano*.

Otros ejemplos de funciones amigas se muestran en la sección de sobrecarga de operadores.

4.2.6 Agrupación: espacios de nombres.

Como acabamos de comprobar es posible encontrarnos con un grupo de definiciones de variables, funciones, tipos, etc. que corresponden a un grupo de herramientas estrechamente relacionadas entre sí. Así por ejemplo, en el caso anterior nos encontramos con la clase *Fecha*, un grupo de funciones auxiliares, otras posibles definiciones de clases (por ejemplo, la clase *tiempo*), etc.

Es interesante en la programación modular que puediéramos establecer un mecanismo por el que indicar que todas estas herramientas están relacionadas como un módulo que se ocupa de resolver los problemas de fechas y tiempo.

El lenguaje C++ nos ofrece la posibilidad de definir un espacio de nombres, es decir de agruparlos como partes de un entorno al que asignamos un nombre. La forma de realizarlo es con la palabra clave **namespace**[22]. Por ejemplo, en el caso de la fecha y el tiempo, podemos declararlo

```
namespace FechaTiempo {
  class Fecha {
    ...
  };

  bool bisiesto (int a);

  class Tiempo {
    ...
  };
  ...
}
```

Lo que implica que para acceder a cada uno de ellos debemos siempre anteponer *FechaTiempo::* delante (de la misma manera que un miembro de una clase se especifica con su nombre). Por ejemplo,

```
FechaTiempo::Fecha f;

...
cout ≪ ‘‘El año :’’≪f.AnioFecha()≪ ‘‘ es ’’
     ≪ FechaTiempo::bisiesto(f.AnioFecha()) ≪’\n’;
```

Para evitar que el código sea demasiado engorroso, es posible indicar al compilador que se va a usar

[22]Esta herramienta nos permite por tanto modularizar nuestro código desde un punto de vista lógico independiente de su distribución en ficheros.

un espacio de nombres para que, desde ese momento, no sea necesario. Por ejemplo,

```
using namespace FechaTiempo;

Fecha f;

...
cout ≪ ‘‘El año :’’≪f.AnioFecha()≪ ‘‘ es ’’
     ≪ bisiesto(f.AnioFecha()) ≪’\n’;
```

aunque no es muy recomendable incluir cada uno de los espacios de nombres en el espacio global[23].

Un ejemplo de espacio de nombres es *"std"* dentro del que se incluye la biblioteca estándar de C++. Eso significa que para usar alguno de sus componentes es necesario anteponer *std::* (por ejemplo, delante del identificador *cout*). Nosotros supondremos que aparece la cláusula *using*, ya que nuestros ejemplos son muy simples y es más claro presentarlos de forma simplificada.

4.2.7 Encapsulamiento de otros componentes.

Como hemos visto en la sección anterior, es posible realizar el encapsulamientos de múltiples componentes dentro de un espacio de nombres. Así, para acceder a ellos es necesario especificar este espacio.

En el caso de las clases podemos incluir, asímismo, otros componentes distintos a variables y funciones. Así podemos declarar un nuevo tipo (con *typedef*), definir el tipo de una estructura, o incluso definir una clase completa. Estos componentes se conocen dentro de la clase, aunque si se quieren utilizar fuera tendremos que especificar (como en las variables y funciones globales) en nombre de la clase seguido por '::' (lógicamente, no pueden utilizarse fuera si no son públicos).

Para ver un ejemplo, véase más adelante la clase *Lista*.

4.2.8 Funciones *inline*.

El obligar a que el acceso a un T.D.A. se realice a través de su interface nos puede llevar a un código ejecutable menos eficiente. Los casos más claros resultan de operaciones muy simples que pueden realizarse multitud de veces. Por ejemplo, podemos usar un bucle que recorra todos los coeficientes de un polinomio, y para el que comprobamos en cada iteración si hemos llegado al grado de éste. Para consultar el grado llamamos a la función miembro *grado* que simplemente devuelve el campo correspondiente del objeto. Podríamos pensar que el definir esta clase ha empeorado el tiempo de ejecución pues es necesario dedicar tiempo a la llamada y retorno de la función.

El lenguaje C++ nos ofrece un mecanismo para resolver este problema que consiste en declarar la funciones **inline**. Se realiza anteponiendo la palabra *inline* a la cabecera de la función. El efecto es que cuando aparece una llamada a la función, el compilador no la realiza sino que incluye directamente el código de la función[24]. Esto implica un ahorro de tiempo aunque aumenta el tamaño del código objeto.

Para que el compilador pueda hacerlo, es necesario que disponga del código de la función para expandirlo en la llamada y por tanto, debe conocer la definición de la función antes de la llamada. Por ejemplo, si definimos una función inline en un fichero *.cpp*, no es posible expandirla si compilamos otro distinto que tiene una llamada a ésta a no ser que se haya repetido la definición en este otro. Por tanto, es posible encontrar definiciones de funciones *inline* en ficheros cabecera. Por ejemplo, para hacer que la función *grado* del tipo *Polinomio* sea *inline* se puede incluir su definición en el fichero cabecera de la

[23] Podemos encontrar nombres que coinciden o puede ser más claro dejar explícito el módulo al que pertenece cierto identificador

[24] Realmente es un consejo al compilador. Es posible que el compilador no pueda hacerla inline en cuyo caso ignora ésta característica.

siguiente forma

```
class Polinomio {
 ...
  public:
  int Grado() const;
  ...
};

inline int Polinomio::Grado() const
{
  return grado;
}
```

Sin embargo, para el caso de las funciones miembro, existe la posibilidad de hacerlo de otra forma. Se considerará una función *inline* si la función se define en la misma clase. Así, el ejemplo anterior es equivalente a

```
class Polinomio {
 ...
  public:
  int Grado() const { return grado; }
  ...
};
```

Nótese finalmente que las funciones *inline* nos ofrecen una alternativa a las macros con la directiva del compilador *"define"*, alternativa que resulta más conveniente.

4.3 Sobrecarga de operadores.

El lenguaje C++ permite usar un conjunto de operadores con los tipos predefinidos, de manera que es fácil y resulta un código más legible esta notación. Por ejemplo, una expresión compleja como

$$a + \frac{b \cdot c}{d \cdot (e + f)}$$

resulta muy simple escribirla como

```
a+(b*c)/(c*(e+f))
```

y además, fácil de leer. Nótese que cada uno de los operadores se puede considerar una función que toma dos parámetros y devuelve un resultado. Si no disponemos de dichos operadores y tenemos que escribir dichas funciones, podría resultar algo como

```
Sumar(a,Dividir(Producto(b,c),Producto(c,Sumar(e+f))))
```

que resulta bastante más engorroso de escribir y de entender.
Otro ejemplo puede ser

```
a=b=c=d=e=f=0
```

En el que aparece 6 veces el operador de asignación. Éste se evalúa de derecha a izquierda y devuelve como resultado la variable a su izquierda. Así, en el ejemplo, primero se lleva a cabo la asignación *f=0* que da como resultado em f, después la operación de asignación a *e* del valor resultado de la operación anterior (*f*), dando como resultado *e*, etc. Como podemos ver el resultado de evaluar esta expresión es *a* quedando todas las variables con el valor 0. Este operador por tanto nos permite una asignación múltiple en una línea muy simple y fácil de entender.

Muchos de los tipos de datos que aparecen en un programa son definidos por el usuario. En muchos lenguajes es necesario definir funciones y procedimientos para manejar el tipo, sin embargo, sería interesante poder definir operaciones con los operadores habituales que hemos visto. Esta posibilidad resulta

mucho más atractiva para nuevos tipos tales como *Complejo*, *Matriz*, *Racional*, etc. para los que son necesarios si queremos obtener un tipo como los predefinidos en el lenguaje.

Esta posibilidad se cubre en C++ mediante la sobrecarga de operadores. Es interesante destacar cómo el lenguaje tiene una misma notación para distintas operaciones. Por ejemplo, cuando usamos el signo '/' entre dos enteros, el lenguaje considera la operación entre enteros, mientras que si lo hacemos entre dos flotantes, la operación es distinta. Por tanto, podemos decir que este operador está sobrecargado al tener distintos significados dependiendo de los operandos. Cuando desarrollamos un nuevo tipo de dato, tenemos la posibilidad de sobrecargar los operadores añadiendo un nuevo significado si se aplica sobre un objeto de nuestro tipo.

Para realizarlo, C++ considera que los operadores tienen un nombre de función asociado. Cuando definimos dicha función para nuestro tipo añadimos un nuevo significado al operador correspondiente. Los nombres son muy simples de recordar ya que se construyen añadiendo a la palabra **operator** el operador correspondiente.

Por ejemplo, consideremos un nuevo tipo *Racional* y queremos definir el operador de suma. La función que corresponde es *operator+* que podemos añadir como un miembro más de la clase. Así, podemos definir

```
class Racional {
  private:
    float numerador,denominador;
    void simplifica();                                   // Obtiene irreducible
  public:
    Racional() {numerador= 0; denominador= 1;}
    Racional operator+ (Racional r);
    ...
};
```

De esta forma, cuando el compilador encuentra *a+b* interpreta que la llamada es *a.operator+(b)* (de hecho, si escribimos esto, funciona correctamente), que como vemos, llama a la función miembro de suma sobre el objeto *a* con el parámetro *b*.

La definición de la función se podría realizar como

```
Racional Racional::operator+ (Racional r)
{
  Racional res;

  res.numerador= numerador*r.denominador+denominador*r.numerador;
  res.denominador= denominador*r.denominador;
  res.simplifica()

  return res;
}
```

4.3.1 Operador de asignación.

Un operador que es muy importante es el de asignación (=), ya que es muy probable que el usuario del tipo lo utilice. En C++ tiene un valor predefinido (llama a la asignación de cada uno de los miembros de la clase). Sin embargo, es probable que deseemos definirlo nosotros.

El primer aspecto a tener en cuenta es que no debe confundirse con el constructor de copias. Es importante ver que la asignación da un valor a un objeto que ya estaba construido, mientras que el constructor de copias da un valor a un objeto que está por construir.

La cabecera de la función, para una clase *X*, de este operador la podemos escribir como

```
X& operator= (const X& orig);
```

donde podemos ver que recibe un parámetro del tipo de la clase que se pasa por referencia y no puede ser modificado y devuelve una referencia a un objeto de la clase.

El hecho de considerar un paso por referencia hace que la llamada no sea costosa si el tamaño del objeto es grande. Además, añadimos que sea constante de manera que no podemos modificar el objeto dentro de la función. Esto no es una limitación pues es de esperar que la asignación de un valor no lo modifique.

La referencia que devuelve es la del objeto que recibe la petición de asignación, es decir, **this*. Esto permite usar la operación de asignación dentro de expresiones, ya que el resultado es el objeto a la izquierda. Por ejemplo, se pueden encadenar asignaciones, comparar el resultado de una asignación con cierto valor, etc.

Un ejemplo ilustrativo de definición de la función de asignación es el caso del tipo *Polinomio*. Como hemos visto, la asignación miembro a miembro no es correcta pues compartirían la zona de los coeficientes. Por tanto, es necesario redefinir la función.

Podríamos hacerlo de la siguiente forma

```
Polinomio& Polinomio::operator=(const Polinomio& orig)
{
  int i;

  if (&orig! =this) {
    delete[] this→coef;
    this→MaxGrado= orig.grado;
    this→coef= new float[p→MaxGrado+1];

    // Copiamos los coeficientes
    for (i=0;i<=this→MaxGrado;i++)
      this→coef[i]= orig.coef[i];
    this→grado=orig.grado;
  }
  return ∗this;
}
```

En esta función es interesante fijarse en que

- Comprobamos si el objeto que se asigna es igual al asignado. Por ejemplo, si hacemos *p=p* la función no haría nada. Si lleváramos a cabo las operaciones de asignación que se incluyen dentro del bloque de la sentencia condicional, provocaríamos un error.
- El objeto al que se asigna el valor *orig* ya está creado. Por lo tanto debemos liberar la memoria que tiene asignada antes de volver a asignarle un bloque nuevo (compárese con el constructor de copias).
- Finalmente devolvemos una referencia al objeto al que se asigna *orig*, para que pueda ser usado dentro de otra expresión.

4.3.2 Operadores como funciones miembro o auxiliares.

Un tema especialmente interesante en la sobrecarga de operadores, es la posiblidad de llevarla a cabo como funciones miembro o como funciones auxiliares. Por ejemplo, podemos plantear la sobrecarga del producto de dos polinomios como una función miembro

```
class Polinomio {
  ...
  public:
  ...
  Polinomio operator∗ (const Polinomio& p);
  ...
};
```

de forma que hacer *p*q* es equivalente a *p.operator*(q)*.

Sin embargo, tenemos la alternativa de declarar la sobrecarga del operador como una función auxiliar. En este caso no aparece como un miembro sino como una función que tomas dos parámetros y devuelve un polinomio. En este caso, si queremos acceder a la parte privada de la clase, debemos de indicar que la función es amiga

```
class Polinomio {
  ...
  public:
  ...
  friend Polinomio operator* (const Polinomio p1, const Polinomio& p2);
  ...
};
```

y definir la función en algún lugar como

```
Polinomio operator* (const Polinomio p1, const Polinomio& p2)
{
  ...
}
```

Nótese que no se ha indicado *Polinomio::* ya que ahora no es miembro de la clase sino una función auxiliar.

Podemos pensar que ambas soluciones son equivalentes. Sin embargo no es así, ya que el considerar una función miembro de la clase, obliga a que sea llamada con un objeto de dicha clase. Un ejemplo muy simple para entenderlo es la multiplicación en la clase polinomio. Si queremos multiplicar dos polinomios *p* y *q*, podemos escribir *p*q*, entendiéndose como *p.operator*(q)* si es miembro de la clase. Esto nos obliga a que *p* debe ser una instancia de la clase polinomio.

Consideremos, por ejemplo, que deseamos sobrecargar el operador producto para que podamos multiplicar un flotante por un polinomio. En este caso, si queremos multiplicar *f* (por ejemplo, *float*) por el polinomio *p*, escribimos *f*p* que no puede ser interpretado como *f.operator*(p)* ya que esto es considerar un miembro de la clase *float* que sobrecarga el producto. Esa clase ya está en el lenguaje, nosotros estamos desarrollando la clase *Polinomio*.

Por tanto es imposible un miembro de la clase para solucionar este problema. Sin embargo, podemos hacerlo sin problemas como una función auxiliar que, al no ser miembro, no necesita que el primer parámetro sea de la clase *Polinomio*. La cebera de esta función es

```
Polinomio operator* (float f, const Polinomio& p)
```

que también se añade a la clase indicando que es una función amiga en el caso de que necesitemos que acceda a la parte privada.

4.3.3 Operadores de E/S.

Un caso particular de la sobrecarga de operadores es el caso de los operadores >> y << aplicados sobre los tipos *istream* y *ostream* respectivamente.

Como sabemos, para obtener en la salida estándar el valor de un entero, podemos llamar la operador << sobre un *ostream* (por ejemplo, *cout*) con el parámetro entero correspondiente. Es decir, la sentencia *cout << a;* es equivalente a *cout.operator<<(a)*, dando como resultado el mismo *ostream* (*cout*) para poder encadenar más veces el operador con otros parámetros. Como vemos, no es más que un caso similar a los que hemos comentado en la sección anterior y por tanto, podemos sobrecargar estos operadores para permitir que nuestro nuevo tipo pueda disponer de las operaciones de E/S que C++ ofrece a los tipos predefinidos.

Por ejemplo, supongamos que queremos añadir operaciones de E/S al tipo *Fecha*. En primer lugar tenemos que notar que el operador se aplica sobre el tipo *istream* u *ostream*, es decir, tenemos que

sobrecargar los operadores definiendo una función auxiliar. Para el caso de *Fecha* serían

```
ostream& operator≪ (ostream& s, const Fecha& f)
{
  cout ≪ f.DiaFecha() ≪ '/' ≪ f.MesFecha() ≪ '/' ≪ f.AnioFecha();
  return s;
}
istream& operator≫ (istream& s, Fecha& f)
{
  int d,m,a;
  char c;

  cin ≫ d ≫ c ≫ m ≫ c ≫ a;

  Fecha g(d,m,a);
  f= g;
  return s;
}
```

que como podemos ver, no necesitan ser declaradas como funciones amigas ya que no acceden a la parte privada de la clase.

```
Fecha f;

cout ≪ ``Introduzca una fecha'' ≪ endl;
cin ≫ f;
cout ≪ ``La fecha '' ≪ f ≪ `` corresponde a la semana ''
      ≪ f.NumeroSemanaFecha() ≪ endl;
```

Referencias

[1] Aho, Hopcroft y Ullman. *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.

[2] Carrano, Helman y Veroff. *Data abstraction and problem solving with C++. Walls and Mirrors*. Addison-Wesley, 1998.

[3] Cortijo, Cubero Talavera y Pons Capote. *Metodología de la programación*. Proyecto Sur. 1993.

[4] Fdez-Valdivia, Garrido y Gcia-Silvente. *Estructuras de datos. Un enfoque práctico usando C*. Edición de los autores. 1998.

[5] B. Liskov, John Guttag. *Abstraction and Specification in Program Development*. The MIT Press, 1987.

[6] Peña Marí, *Diseño de Programas, Formalismo y Abstracción*. Prentice Hall, 1997.

[7] Pressman, *Ingeniería del software. Un enfoque práctico*. MacGraw Hill, 1993.

[8] B. Stroustrup, *The C++ Programming Languaje. Third edition*. Addison-Wesley, 1997.