Tema I – Introducción

El estudio de las Estructuras de Datos es fundamental en la Ingeniería del Software, pues sienta las bases para diseñar soluciones computacionales eficientes, modulares y elegantes. Para construir sobre cimientos sólidos, iniciaremos nuestro recorrido explorando tres pilares conceptuales esenciales. En primer lugar, abordaremos el concepto de abstracción, el proceso mental que nos permite modelar la complejidad del mundo real identificando y aislando las características esenciales de un problema, ignorando los detalles superfluos. A continuación, veremos cómo esta abstracción se materializa en el código a través de los tipos de datos y su especificación formal, donde definiremos el "contrato" que gobierna el comportamiento de nuestras estructuras. Finalmente, estudiaremos el mecanismo que dota de flexibilidad a estas construcciones: la gestión de memoria dinámica mediante punteros, la herramienta que nos permitirá crear estructuras de tamaño variable y relaciones complejas en tiempo de ejecución. Dominar estos tres elementos no es un mero ejercicio teórico; es la preparación indispensable para comprender, diseñar e implementar las estructuras de datos clásicas —como listas, pilas, colas y árboles— que constituyen el esqueleto de cualquier sistema software robusto y bien diseñado.

1. La abstracción

El diseño y la implementación de software eficiente y fácil de mantener comienza con un concepto fundamental: la abstracción. Podemos definir abstracción como el proceso mental mediante el cual el ser humano extrae las características esenciales de un objeto, un sistema o una idea, ignorando deliberadamente los detalles superfluos que no son relevantes para el propósito inmediato. Este mecanismo es primordial para poder modelar la complejidad del mundo real en un programa de ordenador sin vernos abrumados por una infinidad de pormenores.

Imaginemos, por ejemplo, un mapa de metro. Este mapa no intenta reproducir con exactitud topográfica las curvas, distancias o pendientes de las vías; en cambio, abstrae la información crucial: las líneas, las estaciones y sus interconexiones. Esta simplificación permite al usuario planificar un viaje de manera eficiente, una tarea mucho más difícil con un plano a escala real.

Ilustración 1. Plano de metro (vista parcial)

La abstracción es un proceso mental cotidiano que aplicamos constantemente, incluso cuando dibujamos. Imagina que tienes que representar un gato. ¿Cómo lo harías? La respuesta depende completamente de para qué necesitas el dibujo y qué información es esencial comunicar. La abstracción es precisamente esto: el arte de seleccionar lo fundamental y eliminar lo superfluo, dependiendo del contexto y objetivo. Desde un esquema minimalista (formas geométrica básicas

como triángulos y líneas simples, sin detalles anatómicos precisos), pasando por un dibujo más estilizado (formas simplificadas con detalles seleccionados que comunican carácter), hasta una representación detallada con mayor fidelidad anatómica, texturas e incluso sombras que buscarían un mayor realismo, cada nivel de abstracción sirve a un propósito distinto, seleccionando qué información incluir y qué omitir según el contexto y el objetivo de la representación.

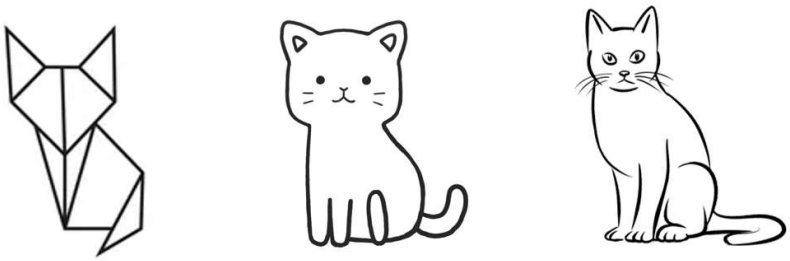


Ilustración 2. Diferentes niveles de abstracción en la representación de un gato

En el contexto de la Ingeniería del Software, la abstracción nos permite concebir sistemas complejos manejando solo los aspectos que nos interesan en un momento dado, ocultando la complejidad subyacente. En este contexto hay algunos elementos clave que materializan su utilidad en el desarrollo de software y que presentamos a continuación.

El concepto de **caja negra** implica estudiar un sistema centrándose exclusivamente en su comportamiento externo observable, obviando por completo su funcionamiento interno. Es como si el sistema estuviera encerrado en una caja opaca que no podemos abrir. Este enfoque tiene sus raíces en la ingeniería eléctrica de los años 40 (métodos de Cauer), donde se analizaban circuitos por la respuesta que producían a ciertas entradas, sin necesidad de conocer cada resistor o capacitor individual. El científico Ross Ashby generalizó posteriormente esta idea en el campo de la cibernética (1958), afirmando que todo sistema complejo puede ser estudiado como una máquina cuyo interior se ignora.

Ilustración 3. Caja negra

La **interfaz** es el conjunto de elementos (operaciones, métodos, funciones) mediante los cuales se puede interactuar con un objeto visto como una caja negra. Es el medio de comunicación entre componentes de software.

La interfaz de un ratón de ordenador define el modo en que se comunican el dispositivo y el usuario (o el sistema operativo). Físicamente, se compone del conector USB (o receptor inalámbrico), los botones primarios y secundarios, la rueda de desplazamiento y, en algunos modelos, botones laterales programables. A nivel lógico y de software, esta interfaz se traduce en un conjunto estandarizado de señales y protocolos que comunican eventos como "clic izquierdo", "movimiento delta-X/Y" o "scroll", sin revelar cómo se generan internamente. Gracias a esta interfaz bien definida y universal, cualquier sistema que la implemente—independientemente de su hardware interno, sistema operativo o marca—puede interpretar estas señales y hacer que el cursor responda de manera predecible. Esto permite la interoperabilidad total: el mismo ratón funciona en un PC con Windows, un Mac o una Raspberry Pi, porque todos "hablan" el lenguaje común de la interfaz definida para el ratón.

El **encapsulamiento** es el proceso que pone en práctica la ocultación de los detalles internos de una abstracción. En programación, es esencial para la reutilización y la robustez del código. Al ocultar *cómo* está hecho un módulo (su implementación) y solo exponer *qué* se puede hacer con él (su interfaz), logramos dos objetivos fundamentales:

- Reutilización**:** Podemos utilizar ese módulo en diferentes programas, confiando en que su comportamiento externo será constante.
- Protección**:** Impedimos que otros componentes del programa utilicen el objeto de modos no deseados o accedan directamente a su estado interno, lo que podría corromperlo. La interacción solo puede darse a través de las operaciones definidas, que actúan como guardianes.

El encapsulamiento se ejemplifica perfectamente con el ejemplo anterior del ratón de ordenador. Como usuario, solo necesito conocer su interfaz física (conector USB, los botones y la rueda de desplazamiento) para utilizarlo efectivamente. Su funcionamiento interno—el sensor óptico, los circuitos, el firmware—me está deliberadamente ocultado. Esto me permite usar el mismo ratón en cualquier ordenador con puerto USB, sin importar sus componentes internos. Si se rompe, puedo reemplazarlo por cualquier otro que respete la misma interfaz, y seguiré usándolo desde el primer día, demostrando cómo la ocultación de detalles (encapsulación) garantiza intercambiabilidad y reutilización.

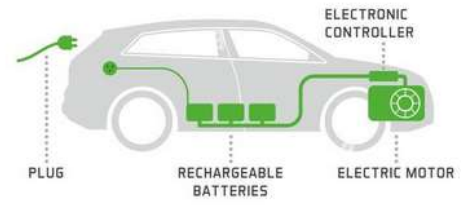
Vinculado a todo lo anterior interfaz está el concepto de **contrato**, una metáfora poderosa introducida por Bertrand Meyer. Este contrato establece las obligaciones y los beneficios mutuos y públicamente declarados entre el diseñador de un componente y su usuario. Así:

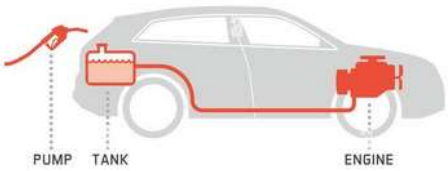
- El diseñador (o implementador) se compromete a *publicar* en la interfaz un conjunto de operaciones que garantizan un comportamiento específico. Oculta todo lo demás mediante encapsulación.
- El usuario (o cliente) se compromete a utilizar *únicamente* esas operaciones publicadas, confiando en que producirán los resultados esperados.

Este "acuerdo" tiene un beneficio enorme: permite al diseñador cambiar completamente la implementación interna de un componente (por ejemplo, hacerlo más rápido o eficiente) sin que el usuario tenga que modificar ni una sola línea de su código, siempre y cuando se respete el contrato (la interfaz) establecido.

Un ejemplo práctico: el coche automático

Para ilustrar estos conceptos, consideremos un coche automático...





- Abstracción: La idea de "un vehículo para transportarse".
- Interfaz/Contrato: Las funciones principales que se ofrecen al conductor: *girar*, *acelerar* y *frenar*. Los mecanismos para ejecutarlas son el volante y los pedales.
- Implementación (detalles ocultos): El motor (diésel, eléctrico, híbrido), el sistema de frenos (de disco, ABS), el tipo de tracción (delantera, trasera, integral) o la dirección (eléctrica, hidráulica).
- Encapsulamiento: El capó del coche actúa literalmente como una barrera que oculta estos detalles complejos. Cualquier conductor,

sin conocer ingeniería mecánica, puede operar cualquier coche gracias a esta abstracción bien definida.

Un buen ejercicio para entrenar el pensamiento abstracto puede ser analizar objetos cotidianos y descomponerlos en sus conceptos clave:

- Interfaz: ¿Con qué interactúo? (botón, manija, superficie).
- Contrato: ¿Qué me promete hacer? (emitir luz, calentar, operar números).
- Posibles implementaciones: ¿De qué maneras distintas se podría construir? (linterna de dinamo vs. de pilas, placa de inducción vs. de gas, bombo físico vs. generador de números aleatorios).
- Nivel de generalidad: ¿Es específico (una cola para un banco) o genérico (una estructura de datos "Cola" que sirve para cualquier escenario)?

Piensa en una linterna, una placa de cocina, un bombo de lotería, un contador, un conjunto de elementos, una cola de personas o una tabla de récords y trata de encontrar respuesta a todos los interrogantes anteriores.

2. Tipos de datos

Una vez comprendida la necesidad de abstraer, debemos encontrar la manera de representar estas abstracciones dentro de un lenguaje de programación. Aquí es donde entran en juego los tipos de datos.

Los lenguajes de programación nos ofrecen un amplio catálogo de tipos de datos, que podemos clasificar de la siguiente manera:

A) Tipos predefinidos. Son de uso muy común y pueden clasificarse en:

- **Simples:** Modelan valores indivisibles. Algunos ejemplos en Pascal son `integer` (números enteros), `real` (números en coma flotante), `boolean` (valores lógicos `True`/`False`) y `char` (un carácter).
- **Estructurados:** Agrupan varios valores relacionados, por ejemplo el tipo `string` (cadena de caracteres) que es un tipo estructurado predefinido.

B) Tipos definidos por el usuario: Dado que los tipos básicos predefinidos no dan normalmente cobertura a todas las necesidades de modelado de la información que necesitan nuestros programas, los lenguajes incluyen mecanismos para que el programador pueda crear sus propios tipos. Así, en Pascal tendremos:

- **Simples:** `subrango` (ue permiten definir un rango de valores, ej: 1..10), `enumerado` (permiten definir una lista de identificadores, ej: `(rojo, verde, azul)`), o `puntero` (permiten referenciar a otra variable, como veremos más adelante).
- **Estructurados:** `array` (vector o matriz de elementos), `registro` (estructura que agrupa campos generalmente de distintos tipos), `conjunto` (set) o `archivo`.

Un tipo de dato puede y debe verse como una abstracción en sí mismo. Cada instancia de un tipo está sujeta a un "contrato" que define:

- Un **nombre** que lo identifica (`boolean`, `integer`).
- Un conjunto de **valores posibles** que puede tomar (su estado interno: para `boolean` son `true` y `false`; para `integer` es un rango muy amplio).
- Un conjunto de **operaciones** válidas que se pueden realizar con él (para `boolean`: `and`, `or`, `not`; para `integer`: `+`, `-`, `*`, `div`, etc.).
- Las **reglas de interacción** con otros tipos (ej: un `array` *de enteros* utiliza el tipo `integer` para sus elementos).

Por ejemplo el tipo Boolean está definido mediante los siguientes datos:

- Nombre: `Boolean`
- Valores posibles: `True`, `False`.
- Operaciones: `AND`, `OR`, `NOT`.
- Uso de otros tipos: Normalmente, solo opera consigo mismo.

Especificación formal

La especificación de un tipo de datos es la definición formal de su contrato. Podemos definirlo como todo aquello que el cliente (quien lo va a usar) necesita saber para utilizarlo correctamente, sin revelar *cómo* se implementa. Una especificación consta de dos partes:

- Signatura: Define la "forma" de las operaciones.
  - Nombre del tipo (ej: `TipoCola`).
  - Nombres de las operaciones (ej: `Encolar`, `Desencolar`, `EstaVacia`).
  - Parámetros de entrada y tipo de resultado de cada operación.

- Axioma: Define la "semántica" o comportamiento de las operaciones. Describe cómo interactúan entre sí, a menudo mediante reglas o pre/postcondiciones. Por ejemplo, un axioma para una cola podría ser: "`Desencolar` aplicado a una cola resultante de `Encolar(X, C)` debe devolver `X` y una cola equivalente a `C`".

Esta separación entre *qué hace* (especificación) y *cómo lo hace* (implementación) es el núcleo de la programación abstracta y permite diseñar programas más flexibles y fáciles de mantener, un objetivo fundamental dentro de la ingeniería del software.

3. Memoria dinámica

Hasta ahora, hemos trabajado principalmente con **estructuras de datos estáticas**, como los arrays. Estas estructuras, aunque útiles, presentan rigideces importantes:

- Tamaño fijo: Su capacidad debe conocerse en tiempo de compilación, algo que no siempre es posible. No se puede redimensionar un array estándar durante la ejecución del programa.
- Reorganización costosa: Modificar el orden de los elementos (como mover el último al principio) puede requerir desplazar muchos elementos, una operación ineficiente.

Para superar estas limitaciones y poder crear estructuras que se adapten a necesidades desconocidas antes de la ejecución, necesitamos gestionar la memoria de forma dinámica. La herramienta fundamental para esto, en lenguaje Pascal, es el puntero.

Peroi antes de entrar en detalles, y para entender los punteros, creemos que es útil visualizar cómo se organiza la memoria de un programa en ejecución (proceso). Esta memoria generalmente, se divide en cuatro segmentos:

- **Segmento de Código (`Code`):** Contiene las instrucciones del programa. Asignación automática.
- **Segmento de Datos Globales (`Global/BSS`):** Almacena variables globales y `static`. Asignación automática.
- **Pila (`Stack`):** Gestiona la memoria local de las funciones. Cada llamada a función reserva un "marco de pila" para sus parámetros y variables locales. Es de tamaño limitado y su gestión es automática (LIFO: Last-In, First-Out).

- **Montículo (`Heap`):** Un bloque grande de memoria para asignación manual. Aquí es donde se reserva la memoria dinámica. El programador es responsable de pedir (`new`) y liberar (`dispose`) espacio aquí.

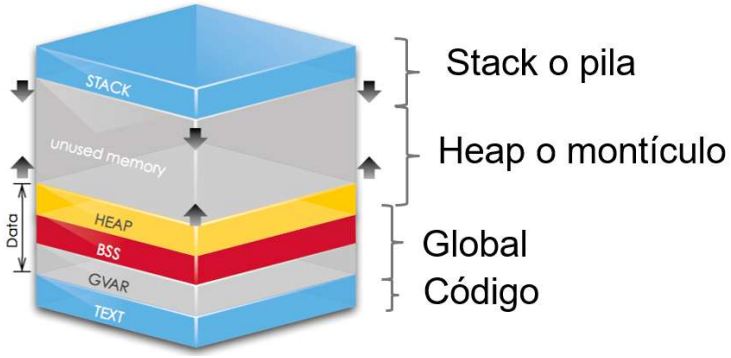


Ilustración 4. Esquema de bloques de memoria de un programa en ejecución

Un ejemplo de código que utiliza la pila sería el siguiente, donde en la declaración de la variable global "precio" se almacenan datos en el segmento global, en la llamada a la función Calculo Iva y en el retorno de l amisma se utiliza la pila y finalmente en la propia llamada (dentro del writeln en el programa principal) se crea una pila para la llamada donde se "deja pendiente" la ejecución del resto del programa principal y se activa la ejecución de la función:

```
PROGRAM Memoria;
VAR
  precio: real; // Se almacena en el segmento global

FUNCTION CalculoIVA(p: real): real; // p y el valor de retorno usan la pila

BEGIN
  CalculoIVA := p * 0.21;
END;

BEGIN
  precio := 200.25;
  writeln('El IVA es: ', CalculoIVA(precio)); // Una pila para la llamada
END.
```

Punteros en Pascal: Concepto y declaración

Un puntero es una variable cuyo valor no es un dato "normal" (como un número o un carácter), sino una dirección de memoria. Es decir, en lugar de almacenar un dato entero, real o booleano, almacena la dirección de memoria de otro dato por lo que se dice que "apunta" a la ubicación donde realmente se almacena el dato de interés.

Se puede declarar de dos formas:

a) Tipo anónimo (directo): `VAR miPuntero: ^integer;`

b) Tipo definido por el usuario (recomendado para claridad):

```
TYPE
  TPunteroEntero = ^integer; // Define un tipo "puntero a entero"
VAR
  pEntero: TPunteroEntero;    // Declara una variable de ese tipo
```

donde el símbolo ^ (acento circunflejo) se lee como "puntero a". El tipo base (en este caso `integer`) indica qué tipo de dato se espera encontrar en la dirección apuntada.

Los punteros son una herramienta fundamental en la construcción de estructuras de datos avanzadas (listas enlazadas, árboles, grafos). Permiten:

- Crear estructuras de tamaño variable en tiempo de ejecución.
- Construir relaciones complejas entre datos (enlaces, nodos).
- Gestionar la memoria de forma eficiente.

Es esencial a la hora de manejarlos el distinguir siempre entre **`puntero`**(la variable que guarda una dirección) y **`puntero^`** (el valor almacenado en esa dirección).

Operaciones con punteros

A continuación se describen las operaciones más importantes con punteros en Pascal:

El **operador** `@` (referencia) devuelve la dirección de memoria de una variable:

```
VAR
  precioPan: integer;
  pEntero: ^integer;

BEGIN
  precioPan := 85;
  pEntero := @precioPan; // pEntero ahora guarda la dirección de 'precioPan'
```

El **operador** `^` (desreferenciación) accede al valor almacenado en la dirección a la que apunta un puntero:

```
pEntero^ := 100;     // Modifica el valor en la dirección apuntada
writeln(precioPan);  // Imprime 100, porque precioPan y pEntero^ son sinónimos
```

Los punteros permiten llevar a cabo la gestión de memoria dinámica mediante las instrucciones `new` y `dispose`

La instrucción **`new`** reserva (asigna) un nuevo bloque de memoria en el *montículo* con el tamaño suficiente para almacenar un valor del tipo base del puntero. Como resultado de su ejecución la variable puntero pasada como argumento queda apuntando a ese nuevo bloque de memoria**.** Téngase en cuenta que el contenido de este bloque es inicialmente indefinido ("basura").

```
new(pEntero); // Reserva espacio en memoria para un entero en el montículo y
              // pone pEntero a apuntar allí
pEntero^ := 50; // Asignamos un valor a ese espacio
```

Por el contrario, la instrucción **`dispose`** libera (devuelve al sistema) el bloque de memoria del *heap* que está siendo apuntado por `puntero`. Es crítico llamar a `dispose` para cada bloque reservado con new cuando ya no se necesite, pues no hacerlo causa un efecto indeseado denominado fugas de memoria **(***memory leaks***)**, que consiste en que el montículo se va llenando de memoria inaccesible pero que no se puede reutilizar al estar marcada como en uso, lo cual puede llevar al agotamiento de la memoria y al fallo del programa o del sistema.

Por último, `NIL` es una constante especial que representa un puntero que no apunta a ninguna dirección de memoria válida. Es una práctica excelente asignar `NIL` a un puntero tras liberar su memoria con `dispose`, para evitar su uso accidental (*dangling pointer*).

```
dispose(pEntero);
pEntero := NIL; // Buen hábito: lo marcamos como "no válido"
```