Tema IV: Datos Abstractos

Introducción a los tipos de datos abstractos (TAD). Abstracción de datos. Concepto sobre tipos de datos. Módulos, interfaz e implementación. Encapsulamiento de datos. Diferencia entre tipo de dato y tipo abstracto de datos. Ventajas del uso de TAD. Formas de abstracción. Requerimiento y diseño de un TAD.

4.1 Introducción a los tipos de datos abstractos (TAD)

Un tipo abstracto de datos (TAD) es una colección de propiedades y de operaciones que se definen mediante una especificación que es independiente de cualquier representación. Se suele considerar que un tipo abstracto de datos es un tipo de datos construido por el programador para resolver una determinada situación.

Tipo abstracto de datos: (TAD): Un conjunto de valores y operaciones asociadas especificados de manera precisa e independiente de la implementación

Objetivos

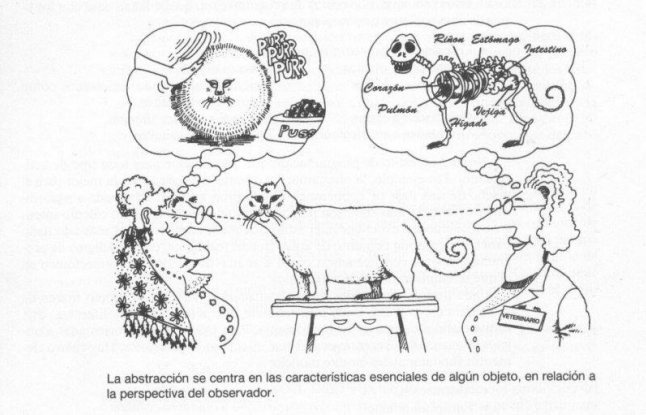
- Extender el concepto de tipo de dato definido por el usuario como una caracterización de elementos del mundo real, tendiendo al encapsulamiento de la representación y al comportamiento dentro de un tipo abstracto de datos (TAD).
- Enfatizar la importancia de la abstracción de los datos y de las operaciones para lograr la reutilización.

4.2 Abstracción de datos

Es una técnica o metodología que permite diseñar estructuras de datos. Consiste básicamente en representar bajo ciertos lineamientos de formato las características esenciales de una estructura de datos.

- Caracterizar el mundo real para poder resolver problemas concretos mediante el empleo de herramientas informáticas.
- Reconocer objetos del mundo real y abstraer sus aspectos fundamentales y su comportamiento de modo de poder representarlos en un ordenador.
- La utilidad de la abstracción y la modelización de objetos es la posibilidad de reusar soluciones.

El proceso de abstracción, debe convertirse en una habilidad para quien estudie una carrera relacionada con la computación. La capacidad de modelar una realidad por medio de herramientas computacionales requiere necesariamente de hacer continuas abstracciones, por lo que es vital conocer metodologías que desarrollen esta habilidad.



La abstracción se centra en las características esenciales de algún objeto, en relación a la perspectiva del observador.

Dos de los tipos más importantes de abstracción son los siguientes:

- División en partes: abstracción "Tiene-un"

Dividir un sistema complejo en sus partes, y dividir las partes en sus componentes puede considerar algunas de éstas, de forma aislada. Con la característica de la palabra "tiene-un" .

- División en especialización: abstracción "Es-Un"

La abstracción "Es-un" toma un sistema complejo, y lo ve como una instancia de una abstracción más general. Se caracteriza por las sentencias que tienen las palabras "Es-un".

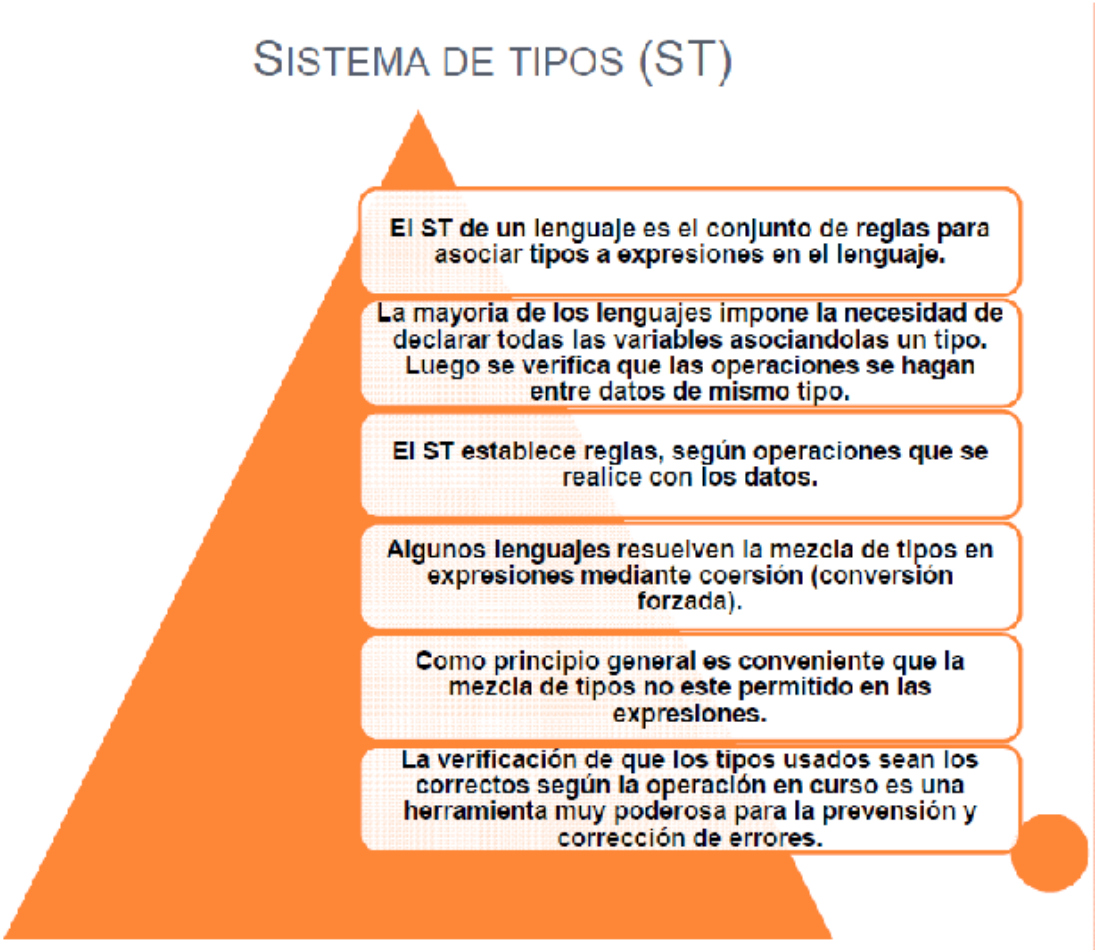
4.3 Concepto sobre tipos de datos.

En la memoria de datos de un ordenador no existen las “estructuras de datos”. Simplemente hay bits en 0 o 1.

Las nociones de tipos, estructuras de datos, variables y constantes son abstracciones para acercar la especificación de los datos de problemas concretos al mundo real.

Los leguajes y SO se encargan de manejar naturalmente las conversiones entre el ámbito propio del especialista en informática y la realidad del hardware de los ordenadores.

El concepto de tipo de datos es una necesidad de los lenguajes de programación que conduce a identificar valores y operaciones posibles para variables y expresiones.



4.4 Módulos, interfaz e implementación

Como se vio anteriormente, en general, el proceso de abstracción se entiende como la identificación de los conceptos esenciales, ignorando los detalles.

Se puede hacer referencia entonces:

Abstracción de procedimientos → módulos

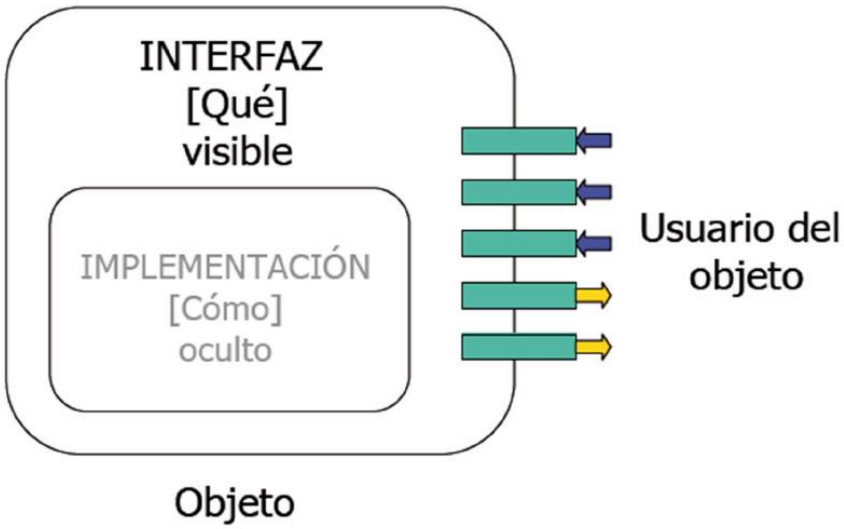
Abstracción de datos → TAD

¿Cómo relacionamos TAD con los módulos que utilizamos hasta el momento?

Un **módulo** se puede ver como un bloque que tiene declaración de comunicación con otros módulos (interfaz) y una funcionalidad interna

La **interface** (del módulo) es una especificación de QUÉ puede hacer el módulo y puede tener declaraciones de tipos y variables que deban ser conocidas externamente

La funcionalidad interna (**implementación** del módulo) abarca el código de los procedimientos que concretan el CÓMO cumplir la función del módulo.



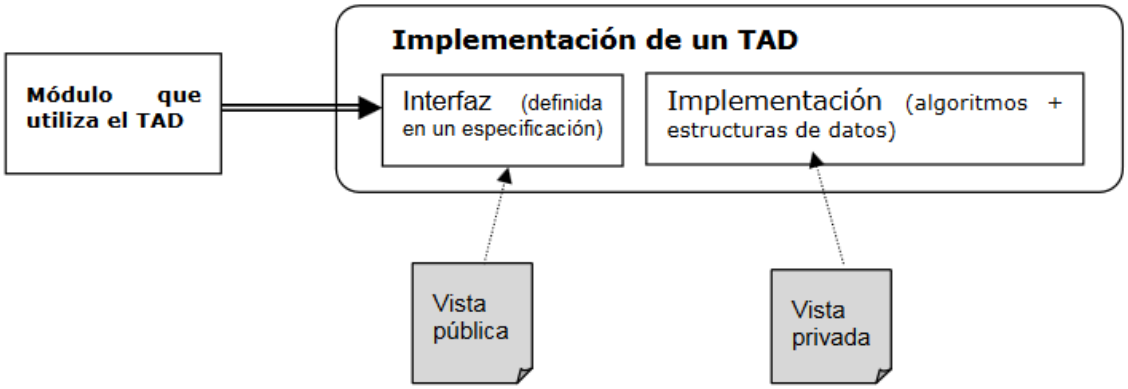
Dicho de otra manera….

Hemos visto la ventajas de descomponer (**modularizar**) un sistema de software en procedimientos y funciones. Esencialmente se logra abstraer las operaciones, de modo de descomponer funcionalmente un problema complejo.

Idealmente el modulo es una caja negra con una función interna y una interfaz de vinculación con otros módulos.

La **interfaz** es una especificación de las funcionalidades del módulo y puede contener las declaraciones de tipos y variables que deben ser conocidas externamente.
La **implementación** del módulo abarca el código de los procedimientos que concretan las funcionalidades internas.

La interfaz se conoce como la parte pública del módulo, mientras que la implementación es la parte privada

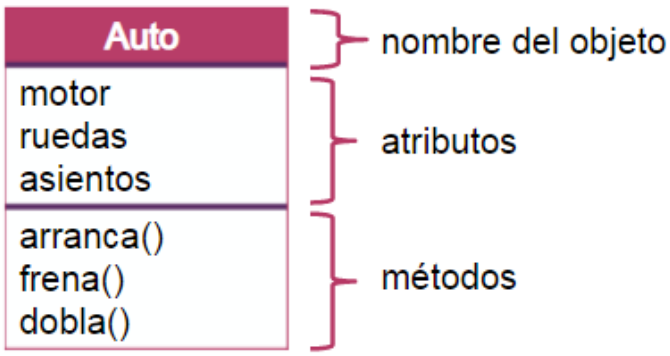


En un TDA existen dos elementos diferenciados:

- La Interface de utilización
- La implementación

4.5 Encapsulamiento de datos.

A partir de la creación de especificaciones abstractas verdaderas o abstracción de datos es lograr encapsulamiento (empaquetamiento) de los datos: se define un nuevo tipo y se integran en un módulo todas las operaciones que se pueden hacer con él, por ejemplo automóvil.
Este nuevo tipo tiene propiedades o atributos en común: poseen motor, ruedas y asientos, pero también se diferencian; es decir, tienen un comportamiento propio (funcionalidad, métodos): arrancar, frenar, etc.-



El encapsulamiento protege al tipo de datos de los usos indebidos o inapropiados. El automóvil utilizara otros mecanismos para llevar a cabo su interfaz, como, por ejemplo, arranca(), frena(), que no le permitirá usar a otros objetos.

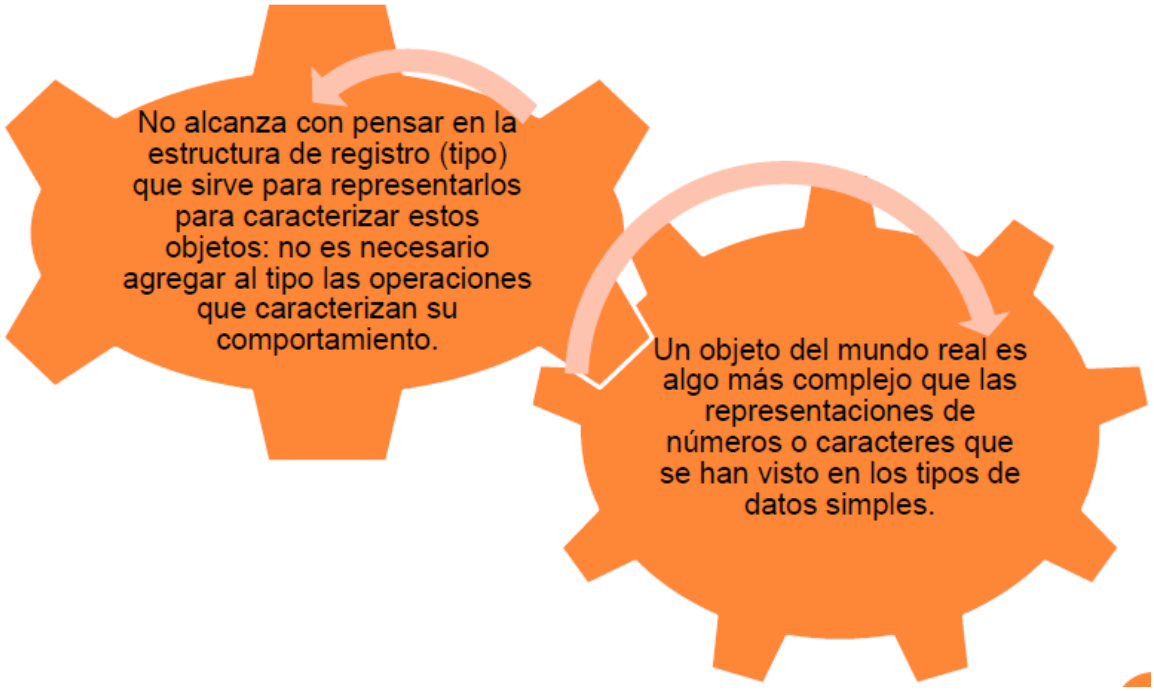
El comportamiento es exclusivo del tipo de datos, si bien algunos son iguales a simple vista, internamente son distintos.

Esto se conoce como encapsulamiento: los tipos de datos presentan la misma interfaz pero ocultan información de su comportamiento.

Si el lenguaje permite separar la parte visible (interfaz) de la implementación, se tendrá ocultamiento de datos (data hidding).

Y si se logra que la solución del cuerpo del módulo pueda modificar la representación del objeto – dato - , sin cambiar la parte visible del módulo, se tendrá independencia de la representación.

4.6 Diferencia entre tipo de dato y tipo abstracto de datos.



Entonces…

El concepto de tipo abstracto de dato es un tipo de datos definido por el programador que incluye:

- Especificación de la representación de los elementos del tipo
- Especificación de las operaciones permitidas con el tipo
- Encapsulamiento de todo lo anterior, de manera que el usuario no pueda manipular los datos del objeto, excepto por el uso de las operaciones definidas en el punto anterior.

- La forma de programación utilizada

Programa = Datos + Algoritmos

Se refina la ecuación anterior se puede mejorar:

Algoritmo = Algoritmo de datos + Algoritmo de control

Algoritmo de dato es la parte del algoritmo encargada de manipular las estructuras de datos del problema.

Algoritmos de control es la parte que representa el método de solución del problema, independiente de las estructuras de datos seleccionadas.

Dado que los TAD reúne en su definición la representación y el comportamiento de los objetos del mundo real se puede escribir la ecuación inicial como:

Programa = Datos + Algoritmos de Datos + Algoritmos de Control

Programa = TAD + Algoritmo de control

4.7 Ventajas del uso de TAD (con respecto a la programación convencional)

- Es posible el desarrollo de algoritmos sin utilizar tipos abstractos de datos.
- El TAD lleva en si mismo la representación y el comportamiento de sus objetos y la independencia que este posee del programa o módulo que lo utiliza permite desarrollar y verificar su código de manera aislada. Es decir es posible implementar y probar el nuevo TAD de una forma totalmente independiente del programa que lo va a utilizar.
- Esta independencia facilita la re – usabilidad del código.
- El modulo que referencia al TAD, lo utiliza como una caja negra de la que se obtienen los resultados a través de operaciones predefinidas. Esto permite que las modificaciones internas de los TAD no afectan a quienes lo utilizan (la interfaz no se debe modificar).

Con el uso de TAD, el programador diferencia dos etapas:

1. En el momento de diseñar y desarrollar el TAD no interesa conocer la aplicación que lo utilizara.
2. En el momento de utilizar el TAD no interesa saber cómo funcionara internamente, solo bastará con conocer las operaciones que permiten manejarlo.

En resumen ….

1. Permite uan mejor conceptualizacion y modelizacion del mundo real. Mejora la represetnacion y compresnibilidad. Clarifica los objetos basados en estructuras y comportamientos comunes.

2. Mejora la robustez del sistema. Si hay caracteristicas subyacentes en los lenguajes permiten la especificacion del tipo de cada variables, los tipos abstractos de datos permiten la comprobacion de tipos para evitar errores de tipo en tiempo de ejecucion.
3. Mejora el rendimiento (prestaciones). Para sistemas tipeados, el conocimiento de los objetos permite la optimizacion de tiempo de compilacion.
4. Separa la implementacion de la especificacion. Permite la modificacion y mejora la implementacion sin afectar la interfaz publica del TDA.
5. Permite la extensibilidad del sistema. Los componentes de software reutilizables son mas faciles de crear y mantener.
6. Recoge mejor la semantica del tipo. Los tipos abstractos de datos agrupan o localizan las operaciones y la representacion de atributos.

4.8 Formas de abstracción.

Abstracción de operaciones. Una serie de operaciones básicas se encapsulan para realizar una operación más compleja. En los lenguajes de programación este tipo de abstracción se logra mediante los *subprogramas o funciones o procedimientos.*

Abstracción de datos. Se encapsula la representación interna de un dato junto con las implementaciones de todas las operaciones que se pueden realizar con ese dato. Esta encapsulación implica que estos datos ocultos sólo pueden modificarse a través de la especificación (o interfaz de e/s). En los lenguajes de programación, la abstracción de datos se logra mediante los tipos de datos que suministra el lenguaje (enteros, reales, arrays, registros, …) y los subprogramas que van a implementar las operaciones permitidas, algunas de cuyas cabeceras formarán su especificación.

4.9 Requerimiento y diseño de un TAD

Disponer de un TAD posibilita tener código reusable.

Esto requiere:

- Poder encapsular dentro de un módulo del lenguaje
- Poder declarar tipos protegidos de modo que la representación interna este Oculta de la parte visible
- Poder heredar el TAD, es decir, crear instancias a partir de ese molde.

El diseño de un TAD lleva consigo selección de:

- Una representación interna, lo que implica conocer las estructuras de datos adecuadas para representar la información.

- Las operaciones a proveer para el nuevo tipo y el grado de parametrización de las mismas.

  Estas operaciones se clasifican en:

  - Operaciones para crear o inicializar objetos (Ej. Cree un pila vacía)
  - Operaciones para modificar los objetos del TAD que permiten quitar o agregar elementos del TAD (Ej. Push).
  - Operaciones que permiten analizar los elementos del TAD (Ej.: verificar que si la pila está vacía).

4.10 Ejemplo de aplicación

Crear tu propia biblioteca en C

Existen bibliotecas estándares en C que ya vienen incluida en la mayoría de los compiladores, como son *stdio.h, math.h, time.h.*

- La biblioteca, o también mal conocida como librería (del ingles *library*) nos permite el uso de estructuras y funciones, previamente definidas en un programa, sin la necesidad de escribir su código en nuestro programa. (TAD).
- Posteriormente, desde nuestro programa deberemos invocar dicha librería. En resumen, bastara con situar en la cabecera del programa el nombre de la biblioteca para poder utilizar todas las funciones y estructuras contenidas en la misma.

Pasos para crear una biblioteca

1. Generar las funciones que interesan y escribirlas todas juntas (codigo y cabeceras) en un mismo archivo de texto (Se puede usar el editor de texto del compilador, el bloc de notas, igual da...).
2. El fichero creado anteriormente, guardarlo con extension *.h* en la carpeta *include* del compilador o se puede guardar el fichero en la misma carpeta del código que queramos compilar.
3. Llamar a la biblioteca en el programa. Se deberá colocar en la cabecera del programa, junto a los llamados de otras bibliotecas.

Utilizar librerías en el mismo directorio #include "libreria.h"

Crear una librería para operaciones aritméticas y se quiere incluir esta librería para probarla en otro archivo. Para esto seguimos los siguientes pasos.

1. Crear el archivo de cabeceras

Creamos un archivo con extensión ".h" en el mismo directorio del código principal, éste archivo debe tener todos los prototipos de funciones y definiciones de tipos de datos de tu librería.

```
#define _LIBRERIA
float suma(float A,float B);
float resta(float A,float B);
float multiplicacion(float A,float B);
#include ″libreria.c″
#endif
```

2. Crear el archivo del código de la librería

El archivo del código de la librería, contiene incluye el archivo de cabecera que creamos anteriormente y además contiene el código de todas las funciones que fueron escritas en el archivo de cabecera.

```
#include ″libreria.h″
float suma(float A,float B){
return A+B;
}
float resta(float A,float B){
return A-B;
}
float multiplicacion(float A,float B){
return A*B;
}
```

3. Llamar a la librería

Cuando estén terminados ambos archivos, nuestra librería está lista para ser usada. Creamos el archivo test.c dentro del mismo directorio, incluimos las librerías estándar y la nueva librería que creamos.

```
#include ″libreria.h″
void main(){
int A = 1;
int B = 2;
printf(″Resultado de %d+%d = %f\n″, A,B,suma(A,B));
printf(″Resultado de %d-%d = %f\n″, A,B,resta(A,B));
printf(″Resultado de %d*%d = %f\n″, A,B,multiplicacion(A,B));
return 0;
}
```

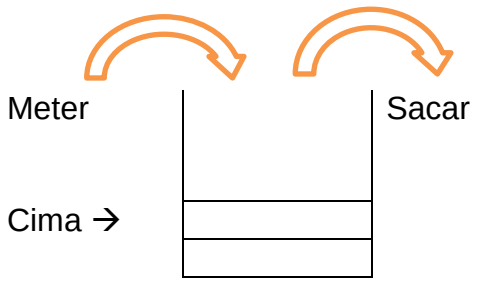
TDA Pila

Definición del tipo

Es una colección lineal, dinámica y homogénea en la que los elementos se insertan y se extraen por el mismo extremo. Estructura LIFO.

Operaciones:

CreaPila
Meter
Sacar
DestruirPila
Pila Vacia

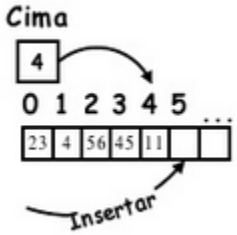


Representación:

Utilizaremos un arreglo para representar la pila.

Definiremos un tamaño máximo del arreglo (**MaxElemPila**)

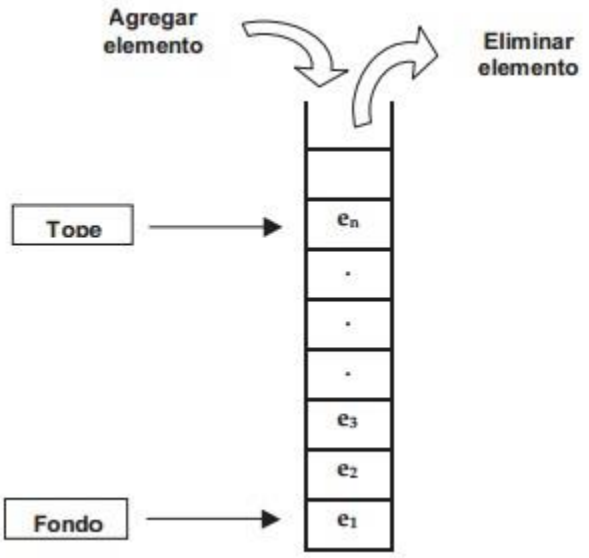
Definimos una variable **cima**, que indicara cual es el último elemento ocupado de la pila.




```
#define MaxElemPila
struct_TPilaEnteros{
     int elementos[MexElemPila];
     int cima;
     };
typedef struct_TPilaEnteros TPilaEnteros;
void creapila( TPilaEnteros *p){p->cima=0;};
int insertaPila (int nelem, TPilaEnteros *p)
{if p->cima ==MaxElemPila)
     { return 0; // no se ha podido insertar}
  else
     {p-> cima ++;
      p->elementos[p->cima]=nelem;
      return 1;};};
```

Ejemplo completo

- Contenedor de objetos que se insertan y se eliminan siguiendo el principio 'Ultimo en entrar, primero en salir' (L.I.F.O.= 'Last In, First Out').
- Se puede considerar una pila como una estructura ideal e infinita, o bien como una estructura finita.



- Operaciones: cuenta con 2 operaciones imprescindibles: apilar y desapilar, a las que en las implementaciones modernas de las pilas se suelen añadir más de uso habitual.
  1. Crear: se crea la pila vacía. (constructor)
  2. Tamaño: regresa el numero de elementos de la pila. (size)
  3. Apilar: se añade un elemento a la pila.(push)
  4. Desapilar: se elimina el elemento frontal de la pila.(pop)
  5. Cima: devuelve el elemento que esta en la cima de la pila. (top o peek)
  6. Vacía: devuelve cierto si la pila está vacía o falso en caso contrario (empty).

- Implementación en C:

```
#include …
#include …
#define MAX_ITEMS 5

void menu();
void imprimePila();
void init();
void push(int x);
int pop();
```

```
bool isFull();
bool isEmpty();

int peek, size, pila[MAX_ITEMS];

int main() /*function main begins program execution*/
{
/* Declarar variables: */
int x,opcion, temporal;
size=MAX_ITEMS;
peek=0;
/**/
while (opcion!=6) {
imprimePila();
menu();
printf("\n Escoge una opcion:");
scanf("%d",&opcion);
switch (opcion) {
case 1:
init();
printf("\nPila inicializada a 0\n");
break;
case 2:
if (!isFull())
{
printf("\n Ingresa el numero a guardar:");
scanf("%d",&x);
push(x);
}
else
printf("\n La pila esta llena");
break;
case 3:
if (!isEmpty())
printf("\n la pila no esta vacia\n");
else
printf("\n la pila esta vacia\n");
break;
case 4:
if (!isFull())
printf("\n la pila no esta llena\n");
else
printf("\n La pila esta llena");
break;
case 5:
if (!isEmpty()) {
temporal=pop();
printf("\nLa posicion %d de la pila se ha borrado\n",peek);
printf("\nEl valor en la posicion %d de la pila era:%d\n",peek,temporal);
}
else
printf("\n La posición pila esta vacía");

break;
default:
printf("\n Error, escoge otra opción\n");
} //end case
```

```
} //end while
getchar();
return 0; /*indicate that the program ended succesfully*/
}

/* Declaracion de funciones*/

void menu()
{
printf("\nPrograma para ejemplificar una pila\n");
printf("\n MENU:");
printf("\n 1. Inicializar");
printf("\n 2. Push (poner)");
printf("\n 3. pila vacia?");
printf("\n 4. pila llena?");
printf("\n 5. Pop (sacar)");
printf("\n 6. Salir");

}
void imprimePila()
{
int j;
printf("\n*************************************************");
printf("\nEl peek es: %d",peek);
printf("\nPeek Pila\n");
for (j=size-1; j>=0; j-)
printf("%d %d\n",j, pila[j]);
}

void init()
{
int j;
peek=0;
for (j=0; j<size; j++)
pila[j]=0;

}

void push(int x)
{
if (!isFull()) {
pila[peek]=x;
peek++;
}
}

bool isFull()
{
if (peek==MAX_ITEMS)
return true;
else
return false;
}
bool isEmpty()
{
if (peek==0)
return true;
```

```
else
return false;
}

int pop()
{
int temp;
if (!isEmpty()) {

temp=pila[peek-1];
pila[peek-1]=0;
peek-;
}
return temp;
}
```