

Unidad 2: Descripción de un programa

Estructura general de un programa en C

Un programa en C se compone de una o más funciones, donde si o si una de las funciones debe ser la función **main**. Las funciones son módulos que se identifican por un nombre propio y realizan una determinada tarea.

Un programa en C puede incluir:

- ✧ Directivas de preprocesador: Instrucciones que comienzan con # y se ejecutan en primer lugar, cuando se compila un código fuente C.
- ✧ Declaraciones globales: Variables o constantes declaradas fuera de cualquier función y **accesibles desde todo el programa**.
- ✧ La función main(): Es la función principal en un programa en C y desde ella se **convocan y coordinan** las demás funciones.
- ✧ Funciones definidas por el usuario: Bloques de código creados por el programador para realizar tareas específicas.
- ✧ Comentarios del programa: Es texto presentado entre /*...*/ esto no es interpretado ni traducido por el compilador. Se utiliza como documentación interna de un programa.

Directivas de preprocesador

Las directivas pueden definir macros, nombres de constantes, archivos fuente adicionales, etc.

La directiva #define se utiliza para definir una macro. Las macros proporcionan principalmente un mecanismo para la sustitución léxica. Una macro se define de la forma:

Sintaxis:

```
#define id secuencia
```

Cada ocurrencia de id en el fuente es sustituida por secuencia.

Un uso común de las directivas de preprocesador en C es la inclusión de archivos de cabecera mediante la instrucción #include. Estos archivos pueden ser estándar, como stdio.h, stdlib.h, math.h o string.h, los cuales contienen funciones y definiciones ampliamente utilizadas. También pueden ser archivos de cabecera creados por el propio usuario, que permiten organizar y reutilizar partes del código en distintos programas. Al utilizar #include, el compilador lee el archivo fuente especificado y lo inserta en el programa en el lugar donde aparece la directiva.

La sintaxis de la directiva es:

```
#include <nombre_de_biblioteca.h> para archivos estándar que se encuentran en el directorio del sistema.  
o #include "nombre_de_archivo.h" para archivos definidos por el usuario, que suelen estar en el mismo directorio del programa.
```

Declaraciones globales

Las declaraciones globales indican al compilador que las funciones definidas por el usuario o variables así declaradas son comunes a todas las funciones de su programa, es decir, que estas funciones y variables pueden ser utilizadas desde cualquier lugar del programa y desde cualquier función, lo cual torna complicado el control de errores en el programa. Esta es la razón por las que se busca minimizar su uso. Las declaraciones globales se sitúan antes de la función main().

Las declaraciones de función se denominan prototipos.

Función main()

Cada programa C tiene una función main() que es el punto de entrada al programa, un programa debe tener sólo una función main(). Además de la función main(), un programa C consta de una colección de funciones. Una función es un subprograma que posee una función establecida y retorna un valor. El retorno del valor lo realiza en el nombre de la función, convirtiéndose ese nombre en un lugar de memoria capaz de albergar datos del tipo asociado y declarado en el prototipo de la función. Las variables y constantes **globales** se declaran y definen fuera de la definición de las funciones, generalmente al inicio del programa, antes del main(), mientras que las variables y constantes **locales** se declaran y definen al inicio del cuerpo o bloque de la función principal, o al inicio de cualquier bloque.

Funciones definidas por el usuario

Un programa en C se construye a partir de la integración de funciones. Se debe diseñar cada función de manera que realice una única tarea, definida a través de una serie de sentencias contenidas en el bloque o cuerpo de la función. Todas las funciones deben tener un nombre y una lista de valores que reciben, llamados parámetros. Las funciones definidas por el usuario se invocan utilizando su nombre, junto con el listado de parámetros actuales que puedan requerir. En C, es necesario declarar las funciones mediante una declaración previa o prototipo dentro del programa. La palabra reservada void se utiliza para indicar que una función no retorna ningún valor.

Tipos de datos

Un tipo de dato es un patrón que determina el conjunto de valores que pueden tomar las variables asociadas a dicho tipo. Este patrón se refiere a una estructura o modelo que establece las reglas que debe seguir ese tipo de dato. Es decir, funciona como una plantilla que define qué valores puede tener una variable, cómo se representan internamente en la memoria y qué operaciones se pueden realizar sobre ellos. Además de establecer el rango de valores posibles, el tipo de dato también define la forma en que esos datos son interpretados por el sistema y las operaciones válidas para su uso.

En los lenguajes de programación, los tipos de datos numéricos se representan internamente mediante diversos métodos binarios que dependen de la cantidad de bits disponibles en la computadora. En el caso de los números enteros, existen varios métodos de representación: binario puro, módulo y signo, complemento a 1 y complemento a 2. El binario puro se usa para representar solo números positivos, por lo que si se dispone de n bits, el rango será de 0 hasta $2^n - 1$. Por ejemplo, con 16 bits, el rango sería de 0 a 65535. El método módulo y signo reserva un bit para indicar el signo (0 para positivo, 1 para negativo), y los restantes representan el valor absoluto del número. Esto genera un rango de representación simétrico, pero con doble representación del cero, ya que se puede escribir como positivo o negativo. El complemento a 1 también usa un bit de signo y representa los negativos invirtiendo los bits del número positivo, lo que también produce dos ceros distintos. Finalmente, el complemento a 2 representa a los negativos sumando 1 al complemento a 1 del número positivo, logrando un único cero y un rango ligeramente asimétrico: desde -2^{n-1} hasta $2^{n-1} - 1$.

Los números reales, también conocidos como números en punto flotante (porque pueden tener parte decimal), no se representan en binario de forma directa como los enteros. En su lugar, se utilizan en una forma similar a la notación científica, donde el número se expresa como:

$$\text{número} = \text{mantisa} \times \text{base}^{\text{(exponente)}}.$$

En las computadoras, la base siempre es 2, ya que trabajan en sistema binario. Por ejemplo, si queremos representar el número decimal 23.375, primero se convierte a binario: la parte entera 23 se convierte en 10111, y la parte decimal 0.375 en 0.011, por lo tanto, 23.375 en binario es 10111.011.

Para almacenarlo en formato de punto flotante, este número debe ser **normalizado**. Esto significa mover el punto binario (equivalente al punto decimal) hasta dejar un único dígito 1 a la izquierda. En este caso, se obtiene 1.0111011 * 2^4 , ya que el punto se movió cuatro posiciones hacia la izquierda.

La **mantisa** es la parte fraccionaria del número normalizado que sigue después del primer 1 (que no se guarda porque se da por hecho que siempre está presente). Por lo tanto, la computadora almacena solo los bits 0111011, y luego se completan con ceros hasta alcanzar los 23 bits requeridos por el formato de simple precisión.

El **exponente** indica cuántas posiciones se movió el punto. Puede ser positivo (cuando se corre a la izquierda, como en este caso) o negativo (cuando se corre a la derecha, como en números menores a 1). Sin embargo, para evitar el uso directo de números negativos —que es más complejo y requeriría más bits— se utiliza un método llamado **bias o sesgo**.

Este sesgo es un valor fijo que se suma al exponente real para convertirlo en un número siempre positivo. En el caso del formato de **simple precisión de 32 bits**, el bias es 127. Así, si el exponente real es 4, se suma el sesgo: $4 + 127 = 131$, y este valor se convierte a binario como 10000011, ocupando 8 bits.

Finalmente, el número completo se representa en 32 bits: **1 bit para el signo** (0 si es positivo, 1 si es negativo), **8 bits para el exponente ajustado con bias**, y **23 bits para la mantisa** (sin el 1 inicial). En el caso de 23.375, la representación completa en memoria sería:

0 10000011 01110110000000000000000000000000.

En resumen, un número real en binario se guarda usando tres partes: un bit de signo, un exponente ajustado con un sesgo fijo, y una mantisa normalizada sin incluir el 1 inicial. Este método permite representar una gran variedad de números positivos y negativos, con o sin parte decimal, de manera eficiente dentro de una cantidad limitada de bits.

Tipos de datos en C

Todos los tipos de datos simples o básicos de C son números. Los tres tipos de datos básicos son: enteros, números de coma flotante (reales) y caracteres. Estos tipos de datos son atómicos o simples, lo que significa que las variables asociadas a estos tipos de datos permiten contener un solo dato por vez.

Tipo	Significado	Tamaño en Bytes	Rango
char	Un carácter	1	[0, 255]
int	Entero	2 ó 4	short [-128,127] int[-32768, 32767] unsigned int[0, 65535] long [-2147483648, 2147483637]
float	Real de simple precisión	4	[-3.4 ⁻³⁸ , 3.4 ³⁸]
double	Real de doble precisión	8	[-1.7 ⁻³⁰⁸ , 1.7 ³⁰⁸]
void	Sin valor	0	Sin valor
Tipo de dato *	Puntero	1	Contiene la dirección de memoria de un dato de tipo "Tipo de dato"

El tipo void es utilizado como tipo de datos asociados a aquellas funciones que no retornan valor en su nombre, es decir, para crear procedimientos.

Variables

Una variable es una posición de memoria identificada por un nombre, a la cual se le asocia un tipo de dato. Esta posición almacena un valor que puede cambiar durante la ejecución del programa. Antes de utilizar una variable, es necesario declararla, lo que implica indicar su tipo y asociarla a una dirección en la memoria RAM. Las variables pueden ser globales, si se declaran fuera de cualquier función (por ejemplo, antes de main()), o locales, si se declaran dentro de una función o subprograma, incluyendo main().

En el lenguaje C, todas las declaraciones deben colocarse al inicio del bloque en el que se utilizan. El ámbito de una variable está limitado al bloque donde fue declarada. Por claridad y buena práctica, se recomienda evitar las variables globales y preferir el uso de variables locales a cada subprograma.

Una variable puede ser inicializada (es decir, recibir un valor) en el momento de su declaración o posteriormente dentro del mismo bloque. Al declarar una variable, se le asignan principalmente tres atributos: nombre, tipo y dirección en memoria. Además, toda variable posee cinco atributos esenciales: nombre, dirección, tipo de dato, tiempo de vida y ámbito.

Cuando la declaración es de forma estática, el compilador reserva antes de la ejecución del programa la memoria necesaria para almacenar el tipo de dato correspondiente. Sin embargo, esa ubicación en memoria puede variar cada vez que se ejecute el programa, dependiendo de factores como la disponibilidad de memoria. El tamaño del espacio reservado dependerá del tipo de dato utilizado: por ejemplo, una variable int ocupará menos memoria que una double.

Entrada y salida de datos

El lenguaje C cuenta con una biblioteca estándar que facilita las operaciones de entrada y salida de datos, para lo cual todo programa debe incluir el archivo de cabecera stdio.h. Esta biblioteca contiene definiciones de macros, constantes, variables y funciones que permiten interactuar con el exterior del programa, es decir, con los dispositivos estándar de entrada y salida. Las funciones más comunes para realizar estas operaciones son printf() y scanf().

La función printf() se utiliza para mostrar información en pantalla. Su estructura incluye una cadena de control que especifica el formato en que deben presentarse los datos (por ejemplo, %d para enteros, %f para números con decimales), seguida por una lista de valores o variables que se van a imprimir.

Por su parte, scanf() permite leer datos ingresados por el usuario a través del teclado. También emplea una cadena de control que indica el tipo de datos esperados, seguida por las direcciones de memoria de las variables donde se almacenarán esos datos, las cuales se indican con el operador &. Esto permite a scanf() modificar directamente el valor de las variables.

La función printf() permite enviar información al usuario a través de la pantalla. Su sintaxis es:

```
printf(cadena-de-control, dato1, dato2, ...)
```

La cadena de control especifica cómo se deben mostrar los datos (por ejemplo, %d para enteros, %f para números con decimales, \n para salto de línea, etc.). Los valores a mostrar se escriben después de la cadena, separados por comas.

Por ejemplo, si declaramos int i = 3, j = 5; y usamos printf("%d \n %d", i, j);, se imprimirá el valor de i seguido de un salto de línea y luego el valor de j.

Por otro lado, la función scanf() se utiliza para leer datos desde el teclado y almacenarlos en variables. Su sintaxis es:

```
scanf(cadena-de-control, &var1, &var2, ...)
```

La cadena de control aquí también indica qué tipo de datos se espera leer (por ejemplo, %d para enteros, %lf para double, %f para float, etc.). Las variables **deben ser precedidas por** el operador & porque se pasa la dirección de memoria donde se almacenará el valor ingresado. Por ejemplo, si tenemos int n; double x; y ejecutamos scanf("%d %lf", &n, &x);, se leerán un número entero y un número decimal desde la entrada estándar y se almacenarán en n y x, respectivamente.

Estructuras de Control-Verificación

Las estructuras de control son componentes fundamentales de un lenguaje de programación que **permiten definir el flujo de ejecución de un programa**. Gracias a ellas, se puede decidir qué instrucciones se ejecutan, cuántas veces y en qué condiciones. Existen distintos tipos de estructuras de control: las **condicionales** (como if, else, switch) que permiten tomar decisiones basadas en condiciones lógicas; las **iterativas o bucles** (como for, while, do...while), que permiten repetir un bloque de instrucciones mientras se cumpla cierta condición; y **las de control de flujo** (como break, continue o return) que permiten modificar el curso normal de ejecución del programa. Estas estructuras son esenciales para resolver problemas que requieren decisiones o repeticiones de manera dinámica durante la ejecución.

Por otro lado, la verificación se refiere al proceso de comprobar que un programa funciona correctamente, tanto a nivel de lógica como de comportamiento. Implica **validar que los datos** ingresados por el usuario sean apropiados, asegurarse de que las condiciones utilizadas en las estructuras de control sean coherentes, y realizar pruebas que permitan identificar errores o fallos en el programa. También incluye la depuración del código, es decir, detectar y corregir errores de sintaxis o lógica. La verificación es una etapa clave en el desarrollo de programas robustos, ya que garantiza que los resultados producidos sean correctos y que el software responda adecuadamente ante distintas situaciones.

Unidad 3: Descomposición de problemas: modularización

Introducción

Para resolver problemas complejos y/o de gran tamaño, podemos utilizar la técnica "Divide y Vencerás", este consiste en, como su nombre lo dice, dividir el problema en problemas más pequeños, llamados subproblemas.

La resolución aplicando esta técnica comienza con una descomposición modular, y luego nuevas descomposiciones sucesivas de cada uno de estos módulos hasta que el problema original queda reducido a un conjunto de actividades básicas, que no se pueden o no conviene volver a descomponer. Luego, cada módulo se resuelve y las subsoluciones se fusionan en la solución del problema original :).

El lenguaje de programación C fue diseñado como un lenguaje de programación estructurado y modular, por lo que la solución de un problema en C comúnmente se expresa por medio de un programa formado por una serie de módulos, cada uno de los cuales realiza una tarea concreta que llevará a la solución de la tarea total.

Beneficios:

- ✧ Producir programas más fáciles de escribir y mantener.
- ✧ Crear módulos independientes por sus parámetros y realizar pruebas independientes.

Concepto

Un módulo es cada una de las partes de un programa que resuelve uno de los subproblemas en que se divide el problema original. Cada módulo tiene una tarea bien definida y algunos necesitan de otros para poder operar. Los módulos se comunican entre ellos mediante una interfaz de comunicación que también debe estar bien definida.

Tipos de módulos

Los módulos básicos **se clasifican** en procedimientos y funciones, la diferencia entre una función y un procedimiento es que, la función retorna un valor en el nombre de dicha función, mientras que el procedimiento no, es decir, el nombre de la función tiene asociado un espacio de memoria capaz de contener un dato, mientras que el nombre del procedimiento es solo una referencia a las sentencias vinculadas al módulo y no tiene la posibilidad de contener datos.

Funciones

- ✧ **Funciones de biblioteca:** Todas las versiones del lenguaje C ofrecen bibliotecas estándares de funciones en tiempo de ejecución que proporcionan soporte para operaciones utilizadas con más frecuencias. Las funciones estándar o predefinidas, como así se denominan a las funciones pertenecientes a la biblioteca estándar, se dividen en grupos; todas las funciones que pertenecen al mismo grupo se declaran en el mismo archivo de cabecera. Para utilizar una función o una macro, se debe conocer el número y tipo de datos de los argumentos y el tipo de dato de su valor de retorno. Esta información se proporciona, como dijimos, en el prototipo de la función.
- ✧ **Funciones definidas por el usuario:** Se debe tener en cuenta que la declaración de una función tiene que hacerse antes que sea llamada por otro modulo, por ello es conveniente realizar la declaración de los prototipos al inicio del programa, antes de la función main. Al invocarse una función, se ejecutan sentencias y retorna un valor en el nombre de la función. Es decir, que conceptualmente el nombre de la función tiene asignado un espacio de memoria con capacidad para representar un dato. El dato que representa debe ser compatible con el tipo de dato asociado al nombre de la función.

Es análogo representar una función matemática mediante una función de programación.

Las funciones pueden ser invocadas desde sentencias de asignación, sentencias de salida o como valores que conformen una expresión. Para llamar a una función con parámetros es importante respetar el orden y el tipo de los parámetros que ella recibe. El primer valor pasado corresponde al primer parámetro, el segundo valor al segundo parámetro; y así sucesivamente si hubiera más.

- ✧ **Declaración o prototipo:**

```
tipoDeRetorno nombreFunción(tddp1,tddp2,tdpd3. . .);
```

En la declaración del prototipo se puede definir en la lista de parámetros sólo el tipo de datos, sin incluir un identificador para el nombre de cada parámetro.

Los aspectos más sobresalientes en el diseño de una función son:

Elemento	Descripción
Tipo de retorno	Es el tipo de dato que devuelve la función en C, aparece antes del nombre de la función. En C, cada función devuelve valores de un único tipo. Si no se especifica ningún tipo, se asume un resultado entero. El tipo de retorno puede ser cualquier tipo excepto una función, un string, o un array, y debe coincidir siempre con el valor devuelto por la función.
Nombre de la función	Identificador o nombre de la función.
Lista de parámetros	Es una lista de parámetros tipificados (con tipos). Cada declaración de parámetro indica tanto el tipo del mismo como su identificador que se asociará a él dentro de la función.
Cuerpo de la función	Se encierra entre llaves de apertura y cierre {}.
Declaración local	Las constantes, tipos de datos y variables declaradas dentro de la función son locales a la misma y no perduran fuera de ella.
Valor devuelto por la función	Una función devuelve un único valor. Este valor debe coincidir con el tipo de retorno. El resultado se muestra con una sentencia return que generalmente se escribe al final de la función.

Procedimientos

Un procedimiento es un grupo de sentencias que realizan una tarea concreta y como resultado de ella puede modificar variables, presentar información en pantalla, generar nuevos valores, etc.

Para invocar un procedimiento basta con escribir su nombre en el cuerpo de otro procedimiento o en el programa principal. Al igual que con las funciones, la declaración de cada procedimiento tiene que hacerse antes que sea llamado por otro modulo, por ello es conveniente realizar la declaración con los prototipos al inicio del programa.

No existen los procedimientos en C, por lo que se simula la definición de un procedimiento utilizando como tipo de retorno el tipo de dato void.

✧ Declaración o prototipo:

```
void nombreProcedimiento(tddp1,tddp2,tddp3. . .);
```

Los aspectos más sobresalientes en el diseño de un procedimiento son:

Elemento	Descripción
Tipo de retorno	Los procedimientos no retornan resultado en su nombre, y como en C los módulos son funciones, se simula el comportamiento de los procedimientos declarando funciones de tipo void.
Nombre del procedimiento	Identificador o nombre del procedimiento.
Lista de parámetros	Es una lista de parámetros tipificados (con tipos). Cada declaración de parámetro indica tanto el tipo del mismo como su identificador que se asociará a él dentro del procedimiento.
Cuerpo del procedimiento	Se encierra entre llaves de apertura y cierre {}.
Declaración local	Las constantes, tipos de datos y variables declaradas dentro del módulo son locales a la misma y no perduran fuera de ella.
Valor devuelto por el procedimiento	Los procedimientos devuelven cero, uno o más valores y lo hacen a través de sus parámetros.

Declaración o prototipos

La declaración de un módulo se denomina prototipo. Los prototipos de un módulo son lo contienen la cabecera del módulo y terminan con un punto y coma.

Específicamente un prototipo consta de los siguientes elementos: nombre del módulo, una lista de parámetros, encerrados entre paréntesis y un punto y coma. Los prototipos de los módulos llamados en un programa se incluyen en la cabecera del programa, antes de la función main, para que sean reconocidos en todo el programa.

El compilador utiliza los prototipos para validar que el número y tipos de datos de los argumentos reales de la llamada a la función sean los mismos que el número y tipo de argumentos declarados en la función. Si se detecta una inconsistencia, se visualiza un mensaje de error.

Ámbito de identificadores

Cada identificador tiene un campo de acción(ámbito), solo dentro de este campo de acción es posible utilizarlo. El ámbito es la zona de un programa en la que es visible una variable.

En C existen cuatro tipos de ámbitos: **programa, archivo fuente, función y bloque**. En este curso se trabajará con los tres primeros. Cada variable puede ser declarada dentro de uno de estos ámbitos, lo que determina su visibilidad: una variable es invisible fuera del ámbito en el que fue declarada y solo puede accederse a ella desde allí. Además, C permite modificar el ámbito y la permanencia de una variable mediante modificadores de tipo o clases de almacenamiento, que son: **auto, extern, static y register**. Cada uno se corresponde con una palabra reservada del lenguaje. **Auto** es el valor por defecto para variables declaradas dentro de funciones o bloques.

- ✧ **auto** (automática o dinámica): Una variable se considera automática cuando, al ingresar a una función, se le asigna espacio en la memoria de forma automática, y este espacio se libera tan pronto como finaliza la ejecución de la función. Es el comportamiento predeterminado para variables locales.
- ✧ **extern** (externa): Se utiliza cuando se desea acceder a una variable que fue creada (es decir, definida) en otro módulo del programa. Esta definición puede encontrarse en una parte anterior del código o en un archivo fuente diferente. El modificador extern permite hacer referencia a dicha variable desde el módulo actual.
- ✧ **static** (estática): Cuando se antepone el modificador static a una variable local, esta deja de ser dinámica y pasa a ser estática. Esto significa que su valor se conserva en memoria incluso después de que la función donde fue definida termine su ejecución. Si la función se invoca nuevamente más tarde, la variable retiene el valor que tenía al finalizar la invocación anterior.
- ✧ **register** (registro): Indica al compilador que, si es posible, almacene la variable en un registro de la CPU, en lugar de en la memoria principal. Los registros son los lugares más rápidos para almacenar datos, por lo tanto, este modificador sugiere optimizar el acceso a la variable. Es ideal para variables que se usan frecuentemente en operaciones intensivas.

Variables locales y globales

- ✧ Las variables *locales* son aquellas que se declaran tanto dentro de la función principal (main) como en las demás funciones, su alcance está limitado solamente a la función en el cual está definida.
- ✧ Las variables *globales* están definidas fuera de cualquier función y pueden ser utilizadas por cualquier parte del programa. Cualquier módulo que forme parte de éste podrá utilizarla simplemente haciendo uso de su nombre.

En C, una variable global solo puede ser inicializada en el momento de su definición, y su valor inicial debe ser una constante, no una expresión. Si no se le asigna un valor, el compilador le otorga automáticamente el valor cero, asegurando así que siempre tenga un valor inicial. Además, es posible que una variable local tenga el mismo nombre que una global; en ese caso, dentro del módulo se da prioridad a la variable local, ocultando a la global.

Transferencia de información a y desde módulos

La comunicación entre módulos y el programa principal o bien entre los mismos módulos se lleva a cabo entre variables globales, parámetros por valor y parámetros por referencia.

Parámetros formales y actuales

- ✧ Parámetros *formales*: Estos se incluyen en la declaración de una función. El término "formal" hace mención a la forma que toman dichos parámetros, y el referirse a la forma indica la cantidad de parámetros, el tipo de datos asociado a cada uno de ellos, y el orden en que se presentan dichos parámetros.
- ✧ Parámetros *actuales*: Estos se incluyen en las sentencias de llamada a una función. El término "actual" se refiere a los valores o a las variables que intervienen como parámetros en cada llamada a la función. Estos parámetros actuales deben coincidir en cantidad, tipo de datos y orden con los parámetros formales.

Parámetros pasados por valor y por referencia

- ✧ Parámetros pasados por valor: Consiste en copiar el contenido del argumento actual a una nueva variable dentro del ámbito local de la *subrutina*. Esto implica que se crean dos valores duplicados e independientes, por lo que cualquier modificación realizada dentro de la función no afecta a la variable original. El parámetro actual puede ser una variable, una constante, una expresión o incluso una llamada a otra función, pero en todos los casos, si se modifica dentro de la función, el valor original permanece sin cambios.
- ✧ Parámetros pasados por *referencia*: Consiste en proporcionar al módulo llamado la dirección de memoria del dato original. De este modo, tanto el programa principal como la subrutina acceden al mismo valor a través de una única ubicación en memoria, lo que significa que cualquier modificación realizada dentro de la función afecta directamente a la variable original. En C, este mecanismo se implementa mediante punteros: se utiliza el operador '*' para acceder al valor apuntado (dirección) y el operador '&'amp; para obtener la dirección de memoria de una variable.

Funciones matemáticas (#include <math.h>)

Función	Prototipo	Descripción
ceil	double ceil (double x);	Redondea hacia arriba al entero más próximo (mayor o igual a x)
fabs	double fabs(double x);	Devuelve el valor absoluto de x.
floor	double floor(double x);	Redondea hacia abajo al entero más próximo (menor o igual a x).
fmod	double fmod(double x, double y);	Resto de x/y en coma flotante. Tiene el mismo signo que x.
pow	double pow(double x, double y);	Calcula x elevado a la potencia y. Puede haber errores de dominio.
sqrt	double sqrt(double x);	Raíz cuadrada de x. x debe ser mayor o igual a 0.
cos	double cos(double x);	Calcula el coseno del ángulo x (en radianes).
sin	double sin(double x);	Calcula el seno del ángulo x (en radianes).
tan	double tan(double x);	Devuelve la tangente del ángulo x (en radianes).
exp	double exp(double x);	Calcula la función exponencial de x.
log	double log(double x);	Calcula el logaritmo natural del argumento x

- ✧ Para pasar un ángulo expresado en grados a radianes, se multiplica los grados por pi y se divide por 180. ($\pi = 3.14159$).

Funciones aleatorias y matemáticas (#include <stdlib.h>)

Función	Prototipo	Descripción
rand	int rand(void);	Genera un número aleatorio entero entre 0 y RAND_MAX (por ej., 32767).
srand	◊ void srand(unsigned int semilla); ◊ srand(time (NULL)); /* Cada vez que se ejecute el programa, se usa una semilla de acuerdo al reloj de la PC */	Inicializa el generador de números aleatorios con una semilla. Usa el argumento como una semilla para una secuencia nueva de números pseudo aleatorios para ser retornados por llamadas posteriores a rand.
abs	int abs(int num);	Devuelve el valor absoluto de un número entero.
labs	long int labs(long int num);	Devuelve el valor absoluto de un número entero largo.

- ◊ Siempre que se necesite generar números aleatorios, se utiliza la función rand(). Se debe invocar al procedimiento srand() una vez por cada ejecución de programa para obtener un conjunto diferente de números aleatorios en cada corrida.
- ◊ Si se quiere obtener un número aleatorio entre un subrango determinado [R1...R2], se puede utilizar la función rand, y posteriormente modificar estos valores convenientemente y =rand()% (R2-R1+1)+ R1

Unidad 4: Tipos de datos estructurados y cadena

Datos estructurados

Los tipos de datos se pueden clasificar en simples y estructurados. Los tipos de datos simples se vinculan a variables que pueden contener un solo dato por vez, en cambio, las variables asociadas a tipos de datos estructurados permiten registrar varios datos a la vez.

Arreglos unidimensionales (vectores)

Un arreglo unidimensional es una colección *finita, homogénea y ordenada* de datos, en la que se hace referencia a cada elemento del arreglo por medio de un índice, que indica su ubicación en el arreglo.

- ✧ Finita: Tiene un número limitado de elementos.
- ✧ Homogénea: Todos los elementos del arreglo son del mismo tipo.
- ✧ Ordenada: Se determina cual es el primer elemento, el segundo y así sucesivamente.

Declaración de un array:

Los arreglos ocupan espacio en memoria que se reserva en el momento de realizar la declaración del arreglo. Ej.: int A[10] define de un arreglo que contiene 10 números enteros.

Otra alternativa para la declaración de arreglos es definir un tipo de datos para luego asociar las variables unidimensionales al tipo de datos. Ej.: `typedef float TLISTA[20]; /*Definición del tipo de datos TLISTA como un arreglo de 20 número reales*/`

El vector se denomina indizado, ya que cada valor del índice hace referencia a una dirección de memoria distinta. Así V[1] ocupa un lugar en la memoria distinto de V[2] (aunque V[2] es contigua a V[1]). Esto permite utilizar los ciclos de control repetitivos, para recorrer los arreglos.

Los vectores son punteros a su primer elemento. Por lo tanto es un puntero constante a un tdd (dependiendo como este declarado el vector) C=&C[0].

Arrays como parametro:

Dado que un vector no tiene tamaño predefinido, pasar su contenido como un parámetro por valor es un costo que C no asume. Un vector completo se pasa a una función mediante un parámetro que contiene la primera dirección de memoria asignada a esta estructura, es decir que es un parámetro pasado por referencia. La referencia a un vector o la dirección inicial, se especifica mediante su nombre, sin corchetes ni subíndices.

En C los arreglos son pasados como parámetros por referencia. Esto es, el nombre del arreglo es la dirección del primer elemento del arreglo. Así mismo, un elemento cualquiera de un arreglo puede ser pasado a una función por valor o por referencia, tal y como se hace con una variable simple.

Cadenas (string)

En C una cadena se define como un array de caracteres (char) que termina en un carácter nulo ('\0').

Declaración de una cadena:

```
char <identificador> [<longitud máxima>];
```

Inicialización de una cadena:

Se puede inicializar una cadena de distintas formas al momento de su declaración, por ej.:

char a[5]="Hola";	a →	h o l a \0
	0 1 2 3 4	
char b[5]={'o','l','l','a','\0'};	b →	o l l a \0
	0 1 2 3 4	
char c[5]={97, 98, 99, 100, 0};	c →	a b c d \0
	0 1 2 3 4	

La representación gráfica de la memoria muestra que internamente los caracteres se almacenan en posiciones consecutivas de memoria. Así también se puede observar que el identificador de la cadena es la dirección de memoria de comienzo del array de caracteres. Entonces, al ser una cadena un array de caracteres, se pueden usar como parámetros por referencia **exclusivamente**.

Al momento de imprimir un string solo es necesario el printf con el código "%s". La verdadera dificultad radica al momento de leerlo o ingresarlos, ya que hay que tener en cuenta diversos factores, como el desbordamiento de buffer, los espacios, el salto de línea, etc. La siguiente lista contiene las diversas funciones que proporcionan las bibliotecas y razones por las cuales no deberían usarse.

Función	Descripción	Errores o problemas comunes
scanf("%s", cad)	Lee una palabra (hasta un espacio, tabulación o enter)	⌚ Solo lee hasta el primer espacio. ⌚ Deja el \n en el buffer, lo que puede afectar lecturas posteriores.
scanf("%c", &var)	Lee un solo carácter.	Puede leer el \n dejado por lecturas anteriores.
gets(cad)	Lee una línea completa (hasta \n).	Riesgo de desbordamiento de búfer, ya que no verifica el tamaño del arreglo.
fgets()	Permite limitar los caracteres leídos.	Más segura que gets() pero puede incluir el carácter \n en la cadena.
scanf("%[^\\n]", cad)	Lee hasta encontrar un \n, permitiendo espacios.	Captura frases completas pero deja el \n en el buffer, debe limpiarse manualmente si se usa luego gets().
fflush(stdin)	Limpia el buffer de entrada (input buffer).	Útil para evitar problemas con caracteres residuales pero no funciona en sistemas tipo Unix/Linux.
gets() + scanf()	Combinación usada para leer línea y luego valores escalares.	Puede causar efectos no deseados por mezcla de funciones y residuos en el buffer.
scanf() con %*c	Usa modificador para "consumir" caracteres no deseados del buffer.	Evita que \n quede en el buffer, lo malo es que puede ser confuso para principiantes.

LeeCad

```
void leeCad(tVec A, int TAM){
    int j;
    char c;

    j=0; //índice de la cadena → guarda el primer carácter ingresado por teclado
    c=getchar(); → controla de no pasar el tamaño máximo y reserva el último lugar(TAM) para el carácter nulo ('\0')

    while(j<TAM-1 &&c!=EOF && c!='\n'){
        A[j]=c; → controlan cuando finaliza la cadena
        j++; → A[j]=carácter guardado en c
        c=getchar(); } → avanza para guardar el próximo carácter
        A[j]='\0'; → asigna el carácter nulo después de que se corta la cadena

    while(c!='EOF'&&c!='\n') → en caso de que la cadena sobrepase TAM, mientras el usuario no finalice la
        c=getchar(); → cadena, los caracteres que ingrese el usuario se van a ir pisando
    }
```

Funciones de biblioteca para strings

Categoría	Función	Invocación	Biblioteca	Descripción	Que devuelve
Longitud	strlen	strlen(cad1);	string.h	Longitud de cad1 (sin '\0').	Devuelve un valor entero con la cantidad de caracteres de cad1.
Copia	strcpy	strcpy(destino, origen);	string.h	Copia en destino la cadena origen	Devuelve un puntero a la cadena destino cad1.
Copia	strncpy	strncpy(destino, origen, n);	string.h	Copia hasta n caracteres de origen en destino.	Devuelve un puntero a la cadena destino cad1.
Concatenación	strcat	strcat(destino, cad1);	string.h	Agrega cad1 al final de destino, finaliza destino con el carácter nulo '\0' y retorna un puntero a destino.	Devuelve un puntero a la cadena destino cad1.
Concatenación	strncat	strncat(destino, cad1, n);	string.h	Agrega hasta n caracteres de cad1.	Devuelve un puntero a la cadena destino cad1.
Comparación (alfabética)	strcmp	strcmp(cad1, cad2);	string.h	Compara cad1 y cad2	Devuelve 0 si son iguales, un valor negativo si cad1 es menor, positivo si cad1 es mayor que cad2.
Comparación (alfabetica)	strncmp	strncmp(cad1, cad2, n);	string.h	Compara hasta n caracteres de manera similar a strcmp.	Igual que strcmp, pero solo compara hasta n caracteres.
Búsqueda	strchr	strchr(cad1, 'a');	string.h	Primera aparición de 'a' en cad1.	Devuelve un puntero a la posición encontrada o NULL si no se encuentra.
Búsqueda	strrchr	strrchr(cad1, 'a');	string.h	Última aparición de 'a' en cad1. Retorna NULL si no lo encuentra.	Devuelve un puntero a la posición encontrada o NULL si no se encuentra.
Búsqueda	strstr	strstr(cad1, cad2);	string.h	Busca la subcadena cad2 dentro de cad1.	Devuelve un puntero al inicio de la subcadena encontrada o NULL si no está.
Búsqueda	strcspn	strcspn(cad1, cad2);	string.h	Cuenta hasta encontrar char de cad2.	devuelve un int
Búsqueda	strtok	strtok(cad1, ",\n");	string.h	Divide la cadena cad1 en tokens usando los caracteres de delimitadores como separadores.	Devuelve un puntero al siguiente token encontrado, o NULL si no hay más tokens.
Conversión caracteres	tolower	tolower(c);	ctype.h	Convierte c a minúscula si es letra.	Devuelve un entero que representa el carácter convertido.
Conversión caracteres	toupper	toupper(c);	ctype.h	Convierte c a mayúscula si es letra.	Devuelve un entero que representa el carácter convertido.
Conversión cadena	strlwr	strlwr(cad1);	string.h	Convierte cad1 a minúsculas.	Devuelve un puntero a la cadena modificada (cad1).
Conversión cadena	strupr	strupr(cad1);	string.h	Convierte cad1 a mayúsculas.	Devuelve un puntero a la cadena modificada (cad1).

Registros

Las estructuras, conocidas como **registros**, son un tipo de datos estructurado. Se usan para resolver problemas que involucran tipos de datos estructurados heterogéneos.

Una estructura es una colección de uno o más tipos de elementos denominados **campos**, cada uno de los cuales puede ser un tipo de dato diferente. Cada campo puede ser a la vez simple o estructurado.

Declaración de un registro:

```
typedef struct {  
    <td Campo> nombreCampo;  
    ...  
}<nombre del registro>;
```

Acceso a un registro:

Podemos acceder a una estructura mediante:

- ❖ Usando el operador punto (.)
- ❖ Usando el operador puntero (→)

Los registros pueden pasarse como parámetros pasados por valor o por referencia. Para pasar por referencia debe usarse el operador &.

En el caso en que la estructura sea pasada por **referencia**, el parámetro formal contiene la dirección de la estructura, es un puntero a la misma, por lo que, el acceso a cada campo se realiza mediante el operador **->**, como se muestra en el siguiente ejemplo:

```
void entrada (struct reg* t){  
    printf("Ingrese un entero ");  
    scanf("%d", &(t->campo1));  
}
```

Este operador es una **forma abreviada** de desreferenciar el puntero y luego acceder al campo. Es decir:

```
(*ptr).dni = 12345678; // Equivalente a ptr->dni
```

En cambio, si la estructura fuera pasada por **valor**, el parámetro formal contiene los datos de la estructura, por lo que, el acceso a cada campo se realiza mediante el operador **.** (punto), como se muestra en el siguiente ejemplo

```
void salida(struct reg t){  
    printf("El número entero es %d", t.campo1);  
}
```

Uniones

Las uniones representan también un tipo de datos estructurados. Son similares a las estructuras. Se distinguen fundamentalmente de las estructuras en que sus miembros comparten el mismo espacio de almacenamiento en la memoria. Son útiles para ahorrar memoria. Sin embargo, es necesario considerar que sólo pueden utilizarse en aquellas aplicaciones en que sus componentes no reciben valores al mismo tiempo. Es decir sólo uno de sus componentes puede recibir valor a la vez. El espacio de memoria reservado para una unión corresponde a la capacidad del campo de mayor tamaño. Definición: Una unión es una colección de elementos finita y heterogénea en la que sólo uno de sus componentes puede recibir valor a la vez.

Unidad 5: Tipo Abstracto de Dato

Introducción

Para definir o conceptualizar un tad, se debe tener en cuenta los siguientes conceptos:

- ✧ **Tipo de dato:** Es un **atributo/propiedad** de los datos que impone ciertas restricciones sobre los mismos, como los valores que pueden tomar y el tipo de operaciones que se pueden realizar sobre ellos. El tipo de dato se asigna en función del propósito para el cual se va a utilizar.
- ✧ **Estructura de datos:** Es la forma de **organizar** un conjunto de datos elementales con el objetivo de facilitar su manipulación. Es como decidir cómo vincular o agrupar los datos según lo que convenga para la aplicación.
- ✧ **Tipo abstracto de dato:** Es un **modelo matemático compuesto por un tipo de dato y las operaciones definidas sobre dicho modelo.**

En resumen, un TAD va más allá de un tipo de dato tradicional. No solo describe la naturaleza de los datos, sino también las operaciones que los manipulan. Estas operaciones se mantienen ocultas (cerradas), lo que significa que solo pueden ser utilizadas a través de una interfaz específica. De este modo, el acceso y manipulación de los datos se realiza exclusivamente a través del TAD.

Definición de Vargas:

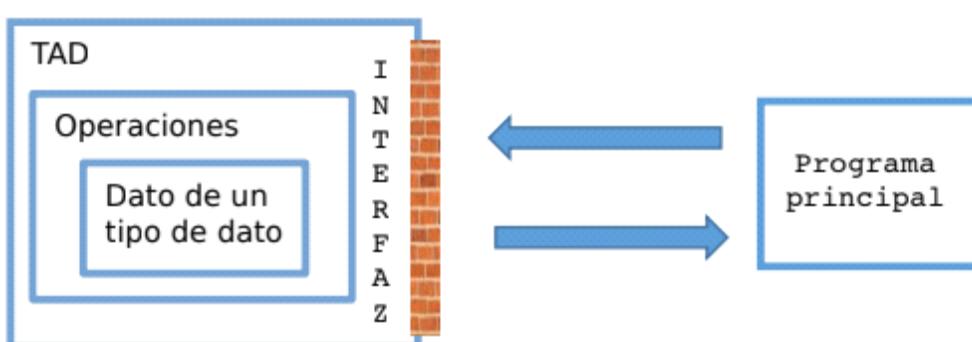
Un tad es una forma de estructurar los programas que venimos utilizando previamente. En sí, es un modelo de datos más las operaciones que manejan ese modelo. Todo ese conjunto encerrado dentro de una burbuja (encapsulamiento).

Para representar un modelo matemático básico de un TAD se utilizan estructuras de datos, que pueden ser de distintos tipos y conectarse de diversas formas. No existe una regla estricta sobre qué operaciones deben implementarse: eso depende del diseño del TAD. La comunicación entre el programa de aplicación (cliente) y la implementación del TAD se realiza exclusivamente a través de una interfaz definida, que expone únicamente las operaciones disponibles.

Características

- ✧ **Operaciones cerradas:** Dentro de un TAD, las operaciones son cerradas. Esto significa que solo se puede acceder a los datos mediante dichas operaciones; no es posible manipularlos directamente desde el exterior.
- ✧ **Comunicación mediante interfaz:** La comunicación entre el TAD y el programa que lo utiliza (por ejemplo, el programa principal) se realiza únicamente a través de una interfaz que expone las operaciones válidas definidas por el TAD.
- ✧ **Ocultamiento de la implementación:** La forma en que están almacenados los datos dentro del TAD no es visible ni accesible desde fuera; está completamente oculta.

El TAD encapsula tanto los tipos de datos como las operaciones, y proporciona una interfaz, que actúa como una barrera entre el programa y la implementación interna del TAD. El programa no interactúa directamente con los datos del TAD, sino a través de esta interfaz. Esta separación ayuda a reducir errores y facilita el mantenimiento. Los TAD están divididos en dos partes bien definidas, la parte pública o interfaz y la parte privada o implementación



Estos TADs se basan en dos conceptos fundamentales: La abstracción y el encapsulamiento.

Abstracción

Es el mecanismo que nos permite seleccionar las partes esenciales de una entidad, ignorando el resto. Nos permite filtrar los aspectos relevantes para resolver un problema determinado.

Tipos de abstracción

❖ Abstracción funcional:

Consiste en crear módulos (por ejemplo, funciones o procedimientos) que se invocan mediante un nombre. Lo importante es qué hace el módulo, sin necesidad de saber cómo lo hace internamente. El usuario sólo necesita conocer la especificación de la abstracción (el qué) y puede ignorar los detalles de su implementación (el cómo).

❖ Abstracción de datos:

Incluye varias formas de representar y manejar los datos en un programa:

- Tipos de datos predefinidos: proporcionados por los lenguajes de alto nivel. Su representación interna es invisible al programador, quien sólo puede usar las operaciones predefinidas para cada tipo.
- Tipos definidos por el programador: permiten crear tipos de datos más cercanos al problema que se quiere resolver. El programador define estructuras que representan mejor los elementos del dominio del problema.
- TAD (Tipos Abstractos de Datos): combinan una estructura de datos con las operaciones que se pueden realizar sobre ella, junto con las propiedades que deben cumplir. Representan una forma más completa de abstracción.
- Objetos: son una extensión de los TADs, a los que se agregan características como reutilización y herencia de código. Este tipo de abstracción no se estudia en esta materia.

Encapsulamiento

El encapsulamiento es un concepto fundamental que consiste en **ocultar el estado interno y los datos** de un Tipo Abstracto de Datos (TAD), de modo que sólo se puedan modificar o acceder a ellos mediante las **operaciones definidas** para ese TAD.

El usuario que utiliza un TAD **no necesita conocer cómo están implementadas internamente las operaciones ni cómo se representan los datos**; sólo debe saber **cómo usarlos correctamente** a través de su interfaz. Es decir, el TAD expone un conjunto de operaciones públicas, pero oculta los detalles de implementación.

Esta técnica permite “esconder” la complejidad interna de un módulo o componente, facilitando su uso y reduciendo el riesgo de errores derivados del acceso directo a sus datos.

Los TADs, por tanto, **ocultan su implementación interna** y sólo proporcionan a los programas que los utilizan una interfaz clara y bien definida, a través de la cual se puede interactuar con ellos.

Ventajas del encapsulamiento

- ❖ Facilita la modificación del código:
Como los detalles internos del TAD están ocultos, cualquier cambio en su implementación **no afecta** a los programas que lo utilizan, siempre y cuando la interfaz se mantenga igual. Esto hace que mantener o mejorar el código sea más sencillo.
- ❖ Fomenta la reutilización:
Los TADs bien encapsulados pueden ser usados en distintos programas sin necesidad de conocer cómo están construidos internamente. Esto promueve la reutilización de componentes.
- ❖ Reduce el impacto de los errores:
Como los errores dentro de un TAD no se propagan fácilmente al resto del sistema, se mejora la **integridad del sistema**. Esto se debe a que los datos están protegidos y sólo pueden ser manipulados de forma controlada.
- ❖ Simplifica el uso de los TADs:
Al ocultar detalles de implementación, el uso de un TAD se reduce a entender **qué operaciones puede realizar y cómo se deben utilizar**, lo que hace más simple el desarrollo de programas.
- ❖ Permite sustituir implementaciones:
Se puede reemplazar un TAD por otro **con la misma interfaz pero distinta implementación**, sin necesidad de modificar el resto del programa. Esto es especialmente útil para mejorar el rendimiento o adaptar el sistema a nuevas necesidades.

Interfaz e implementación

Las estructuras de los Tipos Abstractos de Datos (TAD) se componen de dos partes fundamentales: la **interfaz** y la **implementación**. La interfaz es la parte visible del TAD, la que se presenta al exterior, mientras que la implementación es su estructura interna. A nivel de archivos, suele representarse como *TAD.h* para la interfaz y *TAD.c* para la implementación.

La interfaz sirve como **cubierta** de la implementación y actúa como **límite entre dos entidades distintas**: el TAD y los programas que lo utilizan (programas clientes). En ella se **declaran las operaciones disponibles y los tipos de datos necesarios**, pero **nunca** se incluye el código fuente de las operaciones, ya que eso pertenece a la implementación. De este modo, la implementación queda oculta al usuario, quien sólo debe preocuparse por **cómo usar** el TAD y no por **cómo está hecho** internamente.

Esta separación se basa en el principio de **ocultación de información**, lo que proporciona protección frente a cambios futuros en el diseño del TAD. Así, la interfaz permanece constante mientras que la implementación puede **modificarse** sin afectar a los programas clientes. Por eso, la solidez de un TAD se apoya en que su implementación está escondida al usuario.

Una interfaz **bien diseñada** debe cumplir con ciertas características:

- **Unificación:** Las operaciones deben estar relacionadas temáticamente.
- **Simplicidad:** Debe haber pocos parámetros y con nombres adecuados.
- **Generalidad:** Debe resolver una buena cantidad de casos posibles.
- **Estabilidad:** Se pueden hacer cambios en la implementación sin modificar la interfaz.

En una interfaz se pueden incluir:

- Prototipos de funciones.
- Declaraciones de constantes.
- Declaraciones de nuevos tipos de datos.

Nunca se debe incluir la implementación en la interfaz.

Librerías/Bibliotecas

En el lenguaje C, se utilizan **bibliotecas o librerías** para cumplir con los principios de **encapsulamiento y ocultamiento de la información**. Una biblioteca es un conjunto de implementaciones con una **interfaz bien definida**, que indica cómo invocar su comportamiento. No están pensadas para ejecutarse por sí solas, sino para ser **utilizadas por otros programas** de forma independiente y simultánea.

La idea central es que una biblioteca actúe como un **módulo de software preconstruido**, al que se puede acceder sin necesidad de conocer los detalles internos de su funcionamiento. Basta con saber **qué funciones ofrece y cómo utilizarlas**.

Para facilitar esto, junto con los archivos de biblioteca propiamente dichos (.lib, .a, .dll, etc.), se incluye normalmente un **archivo de cabecera (h)**. Este archivo se incorpora al programa durante la fase de **preprocesamiento** y contiene las **declaraciones de funciones, macros y constantes** necesarias. En C, es obligatorio declarar una función antes de usarla, por lo que simplemente **incluyendo el archivo .h correspondiente**, el programador puede acceder a todos los recursos ofrecidos por la biblioteca sin preocuparse por su implementación.

Tipos de bibliotecas

Existen dos tipos principales de bibliotecas en C: **estáticas y dinámicas**.

Las bibliotecas estáticas **se incluyen dentro del archivo ejecutable** en el momento de la compilación. Esto hace que el ejecutable resultante sea más grande, pero totalmente autónomo, ya que no depende de archivos externos al momento de ejecutarse. En cambio, las bibliotecas dinámicas (como las DLL en Windows) **son archivos externos que el programa utiliza durante la ejecución**. Solo se necesita una copia de la biblioteca en el sistema, y puede ser compartida por múltiples programas. Además, las dinámicas pueden cargarse y descargarse cuando se necesiten, lo que brinda mayor flexibilidad. Sin embargo, requieren estar disponibles en el sistema al momento de correr el programa, y suelen ser más complejas de depurar y mantener. En cuanto al enlace, las estáticas dependen del compilador, mientras que las dinámicas lo hacen del sistema operativo.

Ventajas del uso de TAD

Los tipos abstractos de datos proporcionan numerosos beneficios al programador, que se puede resumir en los siguientes:

- ❖ Mejor conceptualización y modelado del mundo real: Los TAD permiten representar de forma más clara y natural los elementos del problema. Ayudan a pensar en los objetos como entidades completas, con estructura (datos) y comportamiento (operaciones).
- ❖ Mayor robustez del sistema: Los TAD hacen posible verificar los tipos de datos en tiempo de compilación, es decir, antes de ejecutar el programa. Esto permite detectar errores más temprano, como usar mal un dato o pasar un tipo incorrecto, reduciendo errores en tiempo de ejecución.
- ❖ Mejora del rendimiento: Al definir un TAD, el compilador puede realizar ciertas optimizaciones en tiempo de compilación. Como el tipo está bien definido y cerrado, se puede generar código más eficiente. Esto es especialmente útil en sistemas más grandes o con muchas estructuras de datos.
- ❖ Separación entre especificación e implementación: La especificación es lo que se ve desde afuera (la interfaz), mientras que la implementación es cómo está hecho por dentro. Gracias a esta separación, se pueden modificar, mejorar o corregir internamente los TAD sin que eso afecte a los programas que los usan, siempre que la interfaz no cambie.
- ❖ Facilita la extensibilidad y reutilización del sistema: Un TAD bien diseñado puede reutilizarse en distintos programas. Por ejemplo, un TAD Cola o Lista puede servir en varios proyectos. También facilita agregar nuevas funcionalidades al sistema sin necesidad de reescribir todo.
- ❖ Captura mejor la semántica del tipo: Un TAD agrupa tanto los datos como las operaciones que se pueden hacer sobre esos datos en un solo lugar. Esto hace que el diseño del programa sea más ordenado y que el comportamiento del tipo sea coherente y predecible, ya que todo lo que se puede hacer con él está centralizado.

Unidad 6: Recursividad

Definición

La recursividad es una técnica de programación que puede ser utilizada en lugar de la iteración. Los algoritmos recursivos se caracterizan por la posibilidad de invocarse a sí mismos para resolver una versión más "pequeña" o "sencilla" del problema original que se le fue encomendado. Estos subproblemas se resuelven independientemente y después se combinan sus soluciones parciales para obtener la solución final del problema original.

Composición

Los algoritmos recursivos están compuestos por:

- ❖ Llamada recursiva: Es la invocación a sí mismo, en donde por lo menos uno de los parámetros debe cambiar en cada llamado.
- ❖ Caso base: Es la condición de fin o de salida, la cual no provoca una nueva auto invocación, con la que finaliza y evita una ejecución infinita.

En las sucesivas llamadas recursivas los argumentos deben aproximarse a los casos base.

Eficiencia de la recursividad

La recursión permite resolver problemas de alta complejidad con soluciones "elegantes y simples", y que generalmente son estructuradas y modulares. Sin embargo, posee desventajas desde el punto de la eficiencia. Muchas veces un algoritmo iterativo es más eficiente que su correspondiente recursivo, la razón principal es la sobrecarga asociada con las llamadas a sub-algoritmos; una simple llamada inicial a un subalgoritmo puede generar un gran número de llamadas recursivas.

Uso de memoria

En general, en la pila se almacena el entorno asociado a las distintas funciones que se van activando. En particular, en un módulo recursivo, cada llamada recursiva genera una nueva zona de memoria en la pila, independiente del resto de llamadas.

Tipos de recursividad

Directa	Indirecta		
Una función A se llama a sí misma una o más veces directamente.	Una función A llama a otra función B, que a su vez vuelve a llamar a la función A.		
Múltiple	Lineal	Final	
Una función A se invoca a sí misma, más de una vez dentro de una misma instancia. Ej.: Fibonacci.	Una función A se invoca a sí misma como mucho una vez dentro de una misma instancia, pero esta no es la última acción que realiza la función. Ej.: factorial de n.	Una función A se invoca a sí misma como mucho una vez dentro de una misma instancia, y es la última acción que efectúa como parte de la función.	Ejemplo: Una función esPar(n) llama a otra función eslmpar(n-1) y eslmpar(n) vuelve a llamar a esPar(n-1). Se alternan hasta llegar a una condición base.

// Ejemplo recursión final: Una función invertir(n, invertido) se llama a sí misma agregando el último dígito a una variable acumuladora. La última operación siempre es la llamada recursiva, por ejemplo:
invertir(n/10, invertido*10 + n%10).

Cuando n llega a cero, se retorna el número invertido directamente, sin hacer más cálculos después de la llamada

QSort

Quicksort es un algoritmo eficiente para ordenar una lista de elementos. Se basa en elegir un "pivot", o punto de referencia, y luego dividir la lista en partes más pequeñas, organizando cada una de ellas.

1. **Se elige un elemento del arreglo a ordenar**, al que se le llama *pivot* (piv). En el algoritmo que vamos a trabajar, el pivot siempre será el primer elemento de la lista (o sublistas) a ordenar.
2. **Luego se acomodan los demás elementos de la lista a cada lado del pivot, de manera que a un lado queden todos los menores que él, y al otro los mayores.** Los elementos iguales al pivot pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. Como resultado de este proceso, al pivot se asigna una posición del arreglo, esta es la posición que le corresponde en la lista ordenada. Para lograrlo, el algoritmo usa dos índices (izq y der), que recorren la lista desde ambos extremos hasta encontrar elementos que no cumplan con el criterio:
 - ✧ izq avanza de izquierda a derecha hasta encontrar una persona con edad mayor o igual al pivot.
 - ✧ der retrocede de derecha a izquierda hasta encontrar una persona con edad menor al pivot.

Cuando izq y der encuentran elementos en el lado equivocado, se intercambian. Así, los menores que el pivot se acumulan a la izquierda y los mayores a la derecha.

3. Al terminar de recorrer la lista, se coloca el pivot en su posición final, intercambiándolo con el elemento en a[der]. Esto asegura que el pivot queda en el lugar correcto, con todos los menores a su izquierda y los mayores a su derecha.
4. **Sobre cada sublistas se repite este proceso de forma recursiva para cada sublistas**, mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos quedarán ordenados.

Código:

```
void qSort (tVec a, int ini, int fin){  
    int piv,izq,der,mid;  
    if(ini<fin) {           //caso base  
        piv=a[ini];          /*Se resguarda el valor a[ini]*/  
        izq=ini;  
        der=fin;  
        while(izq<der){      //Se determina posición de “piv” en el arreglo  
  
            while(izq<der&&a[der]>piv)  
                der--;           // Se recorre parte derecha para encontrar un  
            if(izq<der){           elemento que deba estar en la parte izquierda  
                a[izq]=a[der];  
                izq++;  
            }                      // Se asigna a a[izq] el valor de a[der]  
            while(izq<der&&a[izq]<piv)  
                izq++;  
            if(izq<der){           // Se recorre parte izquierda para encontrar un  
                a[der]=a[izq];  
                der--;  
            }                      // Se asigna a a[der] el valor de a[izq]  
        }  
        a[der]=piv;  
        mid=der;               // Se asigna en la posición a[der] el valor de “piv”  
        qSort(a,ini,mid-1);  
        qSort(a,mid+1,fin);  
    }                         // Invocación recursiva de la función  
}
```

¿Por qué nunca se pisa un dato importante en la partición de Quicksort?

Durante el proceso de partición en el algoritmo de Quicksort, se reorganiza el arreglo en torno a un pivote, de forma tal que todos los elementos menores o iguales al pivote quedan a su izquierda, y los mayores a su derecha. Para lograrlo, se utilizan dos índices (izq y der) que se mueven desde los extremos hacia el centro, intercambiando elementos que no están en la posición correcta.

Un detalle clave que asegura la seguridad del algoritmo es que nunca se pisa un dato importante, y eso se garantiza por varios motivos:

El pivote se guarda aparte, en una variable auxiliar (piv), antes de comenzar el ciclo de partición. Por lo tanto, la posición que originalmente lo contenía ($a[izq]$) queda disponible para ser sobrescrita sin riesgo de perder el valor del pivote.

En cada iteración del ciclo, los datos se copian hacia posiciones que ya fueron leídas. Es decir, no se pisan datos no evaluados; se sobrescriben aquellos que ya fueron descartados o movidos. Esto evita la necesidad de usar una variable auxiliar para cada intercambio, ahorrando memoria y manteniendo el código más simple.

Caso especial: ¿Qué pasa si no se copia ningún $a[der]$?

Puede ocurrir que ningún valor a la derecha del pivote cumpla la condición para ser movido a la izquierda (por ejemplo, si todos son mayores que el pivote). En ese caso, el primer movimiento igualmente es seguro, ya que se copia el valor de $a[der]$ sobre $a[izq]$, y $a[izq]$ contenía el pivote, que ya fue resguardado en la variable piv.

Por eso:

El primer "pisado" siempre es sobre $a[izq]$, y como ahí estaba el pivote que ya fue guardado, no se pierde nada.

Si no se encuentra ningún valor válido para intercambiar, no se ejecuta el if y no se realiza ningún pisado innecesario.

El ciclo de partición continúa hasta que izq y der se cruzan o se igualan. En ese momento, queda una posición vacía que será ocupada por el pivote.

¿Por qué el pivote se coloca en $a[der]$ al final?

Cuando el ciclo termina, los índices izq y der se encuentran o se cruzan. El último movimiento que se realiza es copiar un valor desde $a[izq]$ hacia $a[der]$ y der--;, lo que deja vacía la posición $a[der]$. Por lo tanto, esa es la posición segura para colocar el pivote.

Si el pivote se colocara en $a[izq]$, se perdería el valor recién copiado desde $a[der]$, porque esa posición ya contiene un nuevo dato válido. En cambio:

Al asignar el pivote en $a[der]$, se asegura que no se pisa ningún dato válido.

Se cumple la condición de partición, dejando los menores o iguales a la izquierda del pivote y los mayores a la derecha.

El resultado es una partición correcta sin pérdida de información, y sin necesidad de estructuras auxiliares.

MergeSort

Merge Sort es un algoritmo eficiente para ordenar una lista de elementos. Se basa en **dividir** repetidamente la lista en partes más pequeñas hasta tener listas de un solo elemento, y luego **mezclar** (merge) estas listas ordenadas hasta recomponer la lista original completamente ordenada.

1. El primer paso consiste en dividir el arreglo:

La lista original se divide **en dos mitades**. Se calcula el punto medio (mid) y se parte el arreglo en dos sublistas:

- La primera sublista es desde la posición inicial (ini) hasta el medio (mid).
- La segunda sublista es desde el medio + 1 (mid + 1) hasta el final (fin).

Esto se hace de forma recursiva: cada mitad se sigue dividiendo hasta que cada sublista tenga **solo un elemento**, que es un caso base trivialmente ordenado.

2. Luego se procede a mezclar las listas:

Una vez que las sublistas tienen un solo elemento, comienza el proceso de **mezcla (merge)**.

- Se toman dos sublistas **ordenadas** (izquierda y derecha) y se **combinan en una sola lista ordenada**.
- Se compara el primer elemento de cada sublista:
 - Se elige el **menor de los dos** y se lo coloca en el arreglo auxiliar aux.
 - Se avanza el puntero correspondiente (izq o der).
- Este proceso se repite hasta que uno de los punteros llega al final de su sublista.

Cuando se acaban los elementos de una de las sublistas:

- Se copian los elementos sobrantes de la otra sublista directamente al arreglo aux.

3. Finalmente se transfiere el resultado al arreglo original:

Una vez mezcladas las dos sublistas en el arreglo auxiliar, se **copia todo el bloque ordenado al arreglo original**, desde la posición ini hasta fin. Esto asegura que la parte procesada del arreglo quede correctamente ordenada.

4. Se repite recursivamente sobre cada sublista:

El proceso se repite recursivamente para cada mitad del arreglo, dividiendo y mezclando, hasta que todo el arreglo esté ordenado.

Una de las principales ventajas de **Merge Sort** frente a otros algoritmos como **QuickSort** es que siempre garantiza un rendimiento eficiente. Mientras que QuickSort en el mejor de los casos puede ser muy rápido, en situaciones desfavorables —como cuando los datos están muy desordenados o casi ordenados en el peor orden posible— su rendimiento puede degradarse a $O(n^2)$, lo que significa que la cantidad de operaciones crece de manera cuadrática con respecto al tamaño de los datos. En cambio, **Merge Sort** siempre mantiene un comportamiento predecible y estable de $O(n \log n)$, sin importar cómo estén organizados los datos inicialmente.

Esto significa que Merge Sort asegura una eficiencia óptima incluso con grandes volúmenes de información, realizando divisiones y mezclas en tiempos controlados. Además, es un algoritmo **estable**, ya que respeta el orden relativo de los elementos iguales, característica importante en ciertos casos prácticos. Por estas razones, Merge Sort es considerado una opción más segura y consistente cuando se requiere eficiencia garantizada, especialmente al trabajar con listas muy grandes.

Código:

```
void merge (tVec a, int ini, int mid, int fin){ //recibe dos sublistas
    tVec aux;    //vector q va a usar para armar el intervalo ordenado
    int izq,der,ia,k;

    ia=0; //ia=índice para el arreglo auxiliar
    izq=ini;
    der=mid+1;

    while(izq<=mid&&der<=fin){ //controla que los índices no se desborden
        ia++;
        if(a[izq]<a[der]){ //analiza los dos primeros de cada lista y coloca el menor
            en la posición correcta del arreglo
            aux[ia]=a[izq];
            izq++;
        }
        else{
            aux[ia]=a[der];
            der++;
        }
    }
    for(k=izq;k<=mid;k++){ //cuando una sublista termina, copia directamente el resto de la
    sublista que quedó sin terminar de copiarse
        ia++;
        aux[ia]=a[k];
    }
    for(k=der; k<=fin; k++){
        ia++;
        aux[ia]=a[k];

        for(k=1;k<=ia;k++)
            a[ini+k-1]=aux[k]; //modifica la sublista ya ordenada en el vector original
    }
}

void mSort(tVec a,int ini, int fin){
    int mid;
    if(ini<fin){ //caso base: que el vector tenga 1 elemento
        mid=(ini+fin)/2;
        mSort(a,ini,mid);
        mSort(a,mid+1,fin);
        merge(a,ini,mid,fin);
    }
}
```

Unidad 7: Tipos de datos dinámicos: Punteros

Definición

Un **puntero** en C es una variable especial que **almacena una dirección de memoria**, es decir, indica en qué lugar de la memoria se encuentra almacenado un dato en un momento determinado de la ejecución de un programa. Su función principal es señalar la ubicación de los datos en memoria, permitiendo acceder y manipular información de forma indirecta.

Al igual que cualquier otra variable, un puntero tiene un **nombre**, una **dirección propia en memoria**, un **valor almacenado**, un **tipo de dato asociado**, un **tiempo de vida** y un **ámbito**. La principal diferencia respecto a las variables tradicionales es que su **valor almacenado no es un dato directo**, sino una dirección de memoria donde efectivamente se encuentran los datos a los que hace referencia.

El comportamiento de los punteros se resume en dos reglas fundamentales:

- ✧ Un puntero es una variable como cualquier otra, pero su contenido es una dirección.
- ✧ La dirección almacenada en un puntero apunta a otra posición de memoria, y es en esa posición donde realmente están almacenados los datos.

Estas características permiten que los punteros sean utilizados para acceder a datos, trabajar con estructuras dinámicas y manejar recursos de memoria de manera flexible durante la ejecución de los programas.

Las principales operaciones asociadas al uso de punteros son:

- ✧ Operador &: Lo utilizamos para averiguar la dirección de memoria asociada a una variable.
- ✧ Operador *: Lo utilizamos para indireccionar un puntero, es decir, para obtener el valor almacenado en la dirección de memoria a la que apunta el puntero.

Declaración e inicialización

Cuando se declara un puntero en C, lo que se hace es reservar una variable destinada a almacenar una dirección de memoria, es decir, no contiene directamente un dato, sino la ubicación donde se encuentra almacenado ese dato. Es obligatorio especificar el tipo de dato al cual apunta el puntero, ya que cada tipo ocupa una cantidad distinta de memoria y el compilador necesita saber cómo interpretar la información a la que se accederá. Por ejemplo, un puntero a int generalmente apunta a direcciones donde se almacenan 4 bytes, un puntero a char accede a direcciones de 1 byte, y un puntero a float normalmente apunta a direcciones de 4 bytes. Por lo tanto, la dirección almacenada por el puntero debe corresponder al tipo de dato especificado, para asegurar un acceso correcto a la memoria.

Es fundamental tener en cuenta que, al ser declarado, un puntero no apunta automáticamente a una dirección válida, sino que su contenido inicial es indefinido o basura de memoria, lo que puede provocar errores si se accede a esa dirección sin inicialización previa. Por esta razón, siempre se recomienda inicializar los punteros, ya sea asignándoles la dirección de una variable utilizando el operador &, o bien asignándoles el valor especial NULL, indicando así que el puntero no apunta momentáneamente a ningún dato válido.

Asignación de una dirección de memoria valida a una variable puntero

Existen dos maneras de realizar la asignación de una dirección de memoria adecuada a un puntero.

1. Apuntarlo a una dirección de memoria ya reservada para el programa.
Para realizar esto basta asignar a la variable puntero la dirección de cualquier variable **estática** del mismo tipo que tengamos declarada en nuestro programa o igualarlo a otro puntero que ya esté apuntando a una dirección adecuada.
 - ✧ Malloc: En caso de haber memoria disponible retorna un puntero que apunta a un bloque de memoria asignado, de lo contrario retorna nulo o cero.
 - ✧ Free(ptr*): El espacio apuntado por ptr*es desasignado, esto es, se pone disponible para otra asignación.
2. Reservar una zona de memoria específica para la variable puntero y hacer que apunte a ella. Para esto se utilizaran las funciones malloc() y free() de la biblioteca stdlib.

Asignación de Memoria

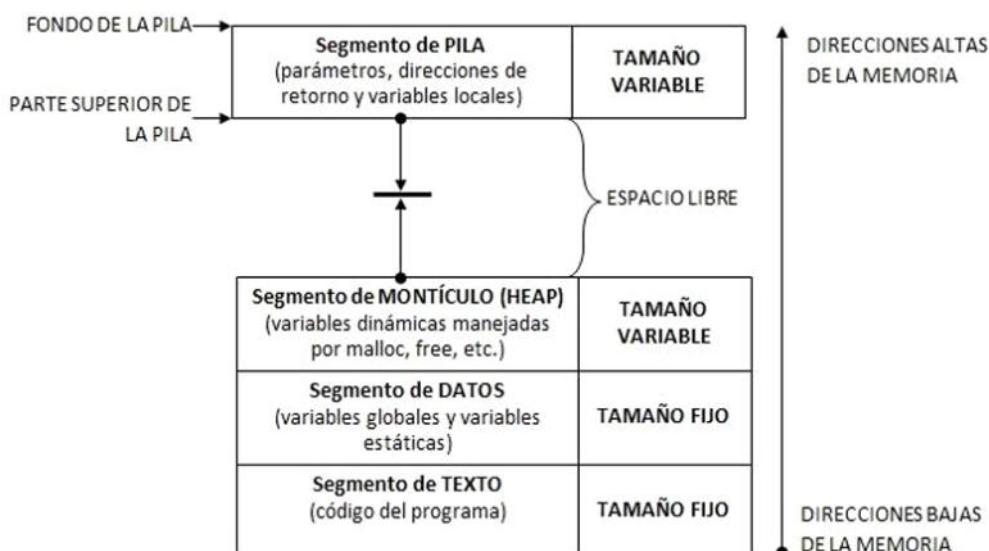
Existen dos formas de asignar una dirección de memoria válida a un puntero:

- ✧ Asignación estática de memoria: En este caso, se utiliza la dirección de una variable preexistente en el programa mediante el operador &, asignando así su dirección a un puntero. Esta variable recibe su dirección de memoria en el momento de la compilación, dentro de la zona de memoria reservada para el programa. La memoria asignada permanece ocupada durante toda la ejecución del programa y es liberada únicamente cuando el programa finaliza. El programador no tiene control sobre cuándo se asigna o libera dicha memoria ni sobre la ubicación exacta dentro del sistema. Este es el tipo de asignación que se utiliza de manera habitual al trabajar con variables comunes.
- ✧ Asignación dinámica de memoria: A través de las funciones malloc() y free(), el programador puede gestionar dinámicamente la memoria durante la ejecución del programa. Con malloc(), se puede asignar memoria en cualquier momento según la necesidad del programa, mientras que con free(), es posible liberar esa memoria cuando ya no se requiere. Este enfoque brinda mayor flexibilidad y control sobre el uso de los recursos de memoria.

Descomposición de la memoria ram

La memoria de un programa se organiza en varias zonas con funciones distintas:

- ✧ Pila (Stack): Es una zona de memoria que se asigna de forma exclusiva a la aplicación, su tamaño es estático, es decir que una vez asignada esta porción de memoria a la aplicación, ésta no crece ni decrece durante la ejecución de la aplicación. Como su nombre lo indica, funciona como una pila o stack, donde los datos se apilan. La pila crece hacia abajo, es decir, hacia direcciones de memoria más bajas. Si la pila crece en exceso y alcanza la zona reservada para el montículo, ocurre una colisión entre pila y montículo, lo que suele provocar errores graves como el stack overflow.
- ✧ El código y los datos estáticos ocupan otra parte de la memoria asignada; el código es el programa en sí; la colección de instrucciones que hay que ejecutar y los datos estáticos son la información fija del Programa. Esta información suele ubicarse por encima de la pila, para evitar invasiones, o bien al otro extremo de la memoria, por idéntica razón.
- ✧ Montículo(Heap): Esta parte de la memoria es compartida y la administra el Sistema Operativo, que a petición de una aplicación, asigna de forma dinámica, por lo tanto la cantidad de memoria del montículo asignada a una aplicación varía durante la ejecución de la aplicación. Normalmente, la mayor parte de la memoria que utiliza una aplicación proviene del montículo, y es precisamente aquí donde se van a ir creando y destruyendo variables dinámicas durante la ejecución de la aplicación.



Las variables dinámicas se crean y destruyen a elección del programador. El **objetivo** que se persigue es racionalizar la utilización de un recurso escaso, la memoria RAM.

Punteros a estructuras

Un puntero también puede apuntar a una estructura. Se declara un puntero a estructura de la misma forma que se declara un puntero a cualquier otro tipo de dato, simplemente utilizando el asterisco * delante del nombre del puntero.

Cuando se accede a un miembro de una estructura usando una **variable estructura**, se utiliza el **punto .** para separar el nombre de la estructura y el campo, por ejemplo:

empleado.nombre

En cambio, cuando se accede a un miembro a través de un **puntero a estructura**, se utiliza el operador **flecha ->**, por ejemplo:

p->nombre

Este operador simplifica el acceso directo a los campos de una estructura apuntada por un puntero.

// La función sizeof() obtiene el espacio que ocupa aquello que se desee, si como argumento de esta función se pone int, como son 2 bytes, se asignan dos bytes de memoria.

Listas enlazadas

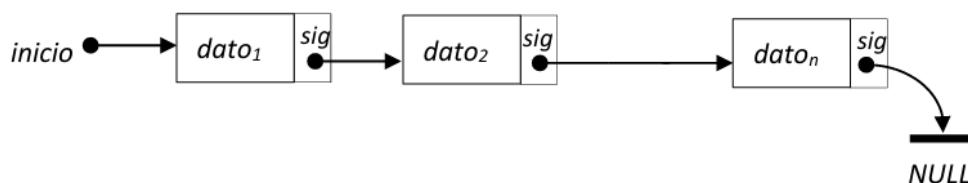
Las listas enlazadas son una estructura dinámica donde cada elemento (nodo) contiene información y un puntero que enlaza con el siguiente elemento. A diferencia de los vectores, las listas no requieren una reserva fija de memoria, ya que su tamaño se ajusta dinámicamente según la cantidad de datos que almacenan.

Listas simplemente enlazadas

Una lista simplemente enlazada es una secuencia de registros unidos por punteros. Cada nodo tiene al menos dos campos:

1. Dato: almacena la información (puede ser de cualquier tipo, por ejemplo, un int, un struct, etc.).
2. Siguiente: un puntero que apunta al siguiente nodo de la lista.

El último nodo siempre apunta a **NULL** para indicar el **final** de la lista. Para acceder a la lista se utiliza un puntero inicial (llamado **inicio**, **cab**, etc) que guarda la dirección del primer nodo.



Entonces, una lista simplemente enlazada es una secuencia de elementos dispuestos uno detrás de otro, en la que cada elemento se conecta al siguiente elemento por un “enlace” o puntero.

Si se quiere leer una lista, se comienza en la posición indicada por **INICIO** y se halla el primer dato junto con un puntero a la siguiente entrada. Si se sigue este puntero se accede a la segunda entrada, y así sucesivamente hasta el final de la lista. Para detectar el final de la lista se utiliza un puntero nulo (**NULL**).

Declaración de una lista enlazada

```
typedef struct Nodo{
```

 Un registro de tipo tNodo está formado por:

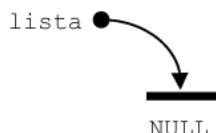
```
        tReg dato;  
        struct Nodo *sig;
```

- ✧ Un campo **dato**, que almacena una estructura **tReg**.
- ✧ Un campo **sig**, que es un puntero a otro nodo de la misma estructura.

```
    }tNodo;
```

Esta definición es un ejemplo de **estructura recursiva**, ya que dentro de la estructura **se incluye un puntero a un nodo del mismo tipo**. Gracias a esta autoreferencia, es posible crear cadenas o secuencias de nodos enlazados.

```
typedef tNodo *tPtr;
```



Para trabajar con la lista, se declara un puntero inicial, llamado por lo general **lista**, que guarda la dirección del primer nodo. Al inicio, la lista está vacía, por lo que este puntero se inicializa con **NULL**: **tPtr lista = NULL;**