Funciones y punteros

Utilidad de una función

La función es un recurso que permite la modularización de un programa. A través de ella se puede descomponer un programa complejo en un conjunto de subprogramas (funciones) más simples, lo que facilita la construcción del programa.

Otras ventajas:

- Disminuir la cantidad de líneas de código, ya que la función se define una sola vez y se la puede invocar cuantas veces se necesite.
- Disminuir la cantidad de errores, ya que evita la duplicación de código. Una buena pista para detectar la necesidad de crear una nueva función, es justamente, encontrar duplicación de líneas de código en distintas partes del programa. La solución es extraer este código y "encerrarlo" dentro de una función. Luego, en reemplazo de este código se invoca a la función para que realice la tarea que antes realizaba el código extraído.
- Reducir la complejidad del programa ("divide y vencerás").
- Eliminar código duplicado.
- Limitar los efectos de los cambios (aislar aspectos concretos).
- Ocultar detalles de implementación (p.ej. algoritmos complejos).
- Promover la reutilización de código (p.ej. familias de productos).
- Mejorar la legibilidad del código.
- Facilitar la portabilidad del código.

Definición

Conjunto de instrucciones que realizan una tarea específica. En general toman valores de entrada, llamados parámetros y proporcionan un valor de salida o valor de retorno; aunque tanto unos como el otro pueden no existir.

Al igual que con las variables, las funciones pueden declararse y definirse. Una declaración es simplemente una presentación, una definición contiene las instrucciones con las que realizará su trabajo al función.

Una vez definida, una función puede invocarse. En general, la definición de una función se compone de las siguientes secciones:

```
<tipo de retorno> <nombre_función> ( <lista de parámetros formales> )
{
     <intrucciones>
     return <valor>
}
```

En general, la invocación de una función se compone de las siguientes secciones:

```
<variable receptora> = <nombre_función> (<lista de parámetros actuales>);
```

Si la función tiene valor de retorno, o :

```
<nombre_función> ( < lista de parámetros actuales > );
```

Si la función no tiene valor de retorno.

Una vez definida una función, la misma puede *invocarse* (usarse) en diferentes partes del programa, para que realice una determinada tarea. En cada invocación, es de esperar que la función arroje diferentes resultados, de acuerdo con los valores de los parámetros (actuales) usados. Por ejemplo:

`int suma(int a, int b)`
`{`
`  int respuesta;`
`  respuesta = a + b;`
`  return respuesta;`
`}`
	tipo de retorno: **int**
	
	nombre de función: **suma**
	
	lista de parámetros formales:
	int a
	int b
		instrucciones:
		int respuesta;
		respuesta = a + b;
		return respuesta;
		variables locales:
		int respuesta;
		valor de retorno: **respuesta**

Se ve la definición de la función suma, que justamente, suma el valor de dos números enteros.

`int main()`
`{`
`  int p, q, r;`
`  p = 5;`
`  q = 10;`
`  r = suma(p, q);`
`  ...`
`  ...`
`}`
	variable receptora: **r**
	
	nombre de función: **suma**
	
	lista de parámetros actuales:
	p, q
		La función suma recibe los datos 5 y 10 a través de las variables p y q, que además son los parámetros actuales.
		
		Luego de realizar los cálculos necesarios retorna el valor (15) que es almacenado en la variable r.

1. El **tipo del valor de retorno**, que puede ser "void", si no necesitamos valor de retorno. Si no se establece, por defecto será "int". Aunque en general se considera de mal gusto omitir el tipo de valor de retorno.
2. El **nombre de la función**. Es costumbre, poner nombres que indiquen, lo más claramente posible, qué es lo que hace la función, y que permitan interpretar qué hace el programa con sólo

leerlo. Cuando se precisen varias palabras para conseguir este efecto existen varias reglas aplicables de uso común. Una consiste en separar cada palabra con un "_", la otra, consiste en escribir la primera letra de cada palabra en mayúscula y el resto en minúsculas. Por ejemplo, si se hace una función que busque el número de teléfono de una persona en una base de datos, se puede llamar "busca_telefono" o "BuscaTelefono".

3. Una **lista de declaraciones de parámetros** entre paréntesis. Los parámetros de una función son los valores de entrada (y en ocasiones también de salida). Para la función se comportan exactamente igual que variables, y de hecho cada parámetro se declara igual que una variable. Una lista de parámetros es un conjunto de declaraciones de parámetros separados con comas. Puede tratarse de una lista vacía.

    a. **Parámetros formales:** están puestos en la definición de la función. Se declaran dentro de la lista de parámetros, estableciendo nombre y tipo para cada uno. En el momento de declararlos no contienen datos, están vacíos, y sirven a los fines de posibilitar la codificación del algoritmo de la función. Son la “puerta de entrada y a veces de salida” de los datos que necesita procesar la función.

    b. **Parámetros actuales:** se usan en la invocación de la función. Son variables del programa invocante (el que llama a la función) y se ubican reemplazando a los parámetros formales. Los parámetros actuales contienen datos (valores) que son usados por el cuerpo de la función para realizar los cálculos propios y generar (si hubiere) el valor de retorno.

    c. **Pasaje de parámetros:** existen dos maneras de usar los parámetros actuales, lo que determina el tipo de pasaje: pasaje por valor (o copia) y pasaje por referencia. En el primer caso (valor o copia) el valor original del parámetro actual “olvida” todo tipo de modificación que sufriera dentro de la función (a través del parámetro formal). En el primer caso, el valor original del parámetro actual puede “recordar” cualquier modificación realizada a través del parámetro formal. Ver el próximo apartado **Pasaje de parámetros.**

4. Un **cuerpo de función** que representa el código que será ejecutado cuando se llame a la función. El cuerpo de la función se encierra entre llaves "{}" Una función muy especial es la función "main". Se trata de la función de entrada, y debe existir siempre, será la que tome el control cuando se ejecute un programa en C.

Variables locales a una función

Una variable que se declara dentro de una función solamente es reconocida dentro de la misma, no existe fuera de los límites de la definición de la función.

Variables globales

Son aquellas que se declaran fuera de toda función. Son accesibles desde cualquier parte del programa. **NO DEBEN USARSE**. Las variables globales son una fuente errores difíciles de encontrar y corregir.

Punteros

Dirección de memoria

Si bien utilizamos nombres para identificar una variable y manipular datos, en realidad, a un nivel más cercano al hardware los datos se almacenan en direcciones de memoria. Las variables reemplazan a las direcciones de memoria y hacen el trabajo de la programación mucho más simple, dejando al compilador la traducción de “variables” a “direcciones de memoria”.
Toda variable tiene una dirección de memoria, y si bien es un valor que podemos conocer, no podemos asignarlo en forma arbitraria, sino a través de la invocación de funciones y operadores especiales.

Operador &

Este operador, aplicado a una variable retorna la dirección de memoria de la misma, siguiendo la siguiente sintaxis: `&idVariable` .

El valor retornado puede asignarse a otra variable (ver definición de puntero) o imprimirse por pantalla usando `%p` **.**

Definición de Puntero

Se denomina variable de tipo puntero a aquella que almacena como dato la dirección de memoria de otra variable, siguiendo la siguiente sintaxis: `tipo_dato * id_variable;`

La variable id_variable podrá almacenar la dirección de memoria de otra variable de tipo tipo_dato.
ejemplo:

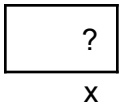
```
int * p;
```

Para poder acceder a la variable apuntada (variable cuya dirección de memoria almacena la variable de tipo puntero) se utiliza el mismo operador `*` .

Siguiendo el ejemplo anterior, `p` contiene la dirección de memoria de la variable `*p` (se dice “p apuntada”). Como se aclaró previamente, no se puede asignar a una variable puntero un valor arbitrario, o sea, las direcciones de memoria no pueden ser ingresadas por el programador en forma directa, no se pueden inventar. Cómo se hace entonces para obtener una dirección de memoria y así asignarla a una variable de tipo puntero?. Existen dos recursos:
1. usar el operador & (que retorna justamente la dirección de memoria de una variable existente).
2. usar la función malloc(), que busca una zona disponible de memoria, la reserva y retorna su dirección.

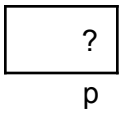
Utilizando el primer recurso, podemos realizar la siguiente composición:

int x;



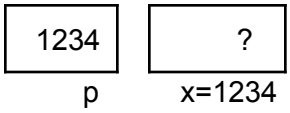
//se crea la variable entera **x**, cuyo primer valor no ha sido asignado.

int * p;



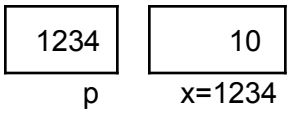
//se crea la variable de tipo “puntero a entero”, cuyo primer valor no ha sido asignado.

p = &x;



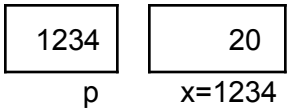
//se asigna a la variable puntero **p** la dirección de memoria de la variable **x.**

x = 10;



//se asigna a x el valor 10. Este valor puede ser accedido a través de la variable **p**, usando el operador ***p**.

***p = 20;**



//se asigna a la variable apuntada por **p** (en este caso **x**) el valor 20.

Pasaje de parámetros

Se realiza en el momento de la invocación de la función. Establece la forma en que el parámetro actual se vincula con el parámetro formal. Existen dos tipos:

Pasaje de parámetros por valor o copia (parámetros de entrada)

El parámetro formal copia el contenido del parámetro actual (pf = pa), por lo tanto cualquier modificación efectuada sobre el parámetro formal no se verá reflejada en el real. O sea, al terminar la función, las variables pasadas como parámetro recuperan sus valores originales.

Pasaje de parámetros por referencia (parámetros de entrada/salida)

Se utiliza una variable puntero como parámetro formal y el parámetro actual es una dirección de memoria. Lo que se copia ahora, es la dirección de memoria de la variable que oficia de parámetro actual. Por lo tanto, dentro de la función se puede acceder directamente al contenido de la variable a través del operador *. Se puede modificar el contenido de la dirección de memoria apuntada por el parámetro, pero no se puede modificar el parámetro.

Ejemplo de pasaje de parámetros por referencia:

```c
void intercambio(int *a, int *b)
{
     int aux;
     aux = *a;
     *a = *b;
     *b = aux;
}
```

```c
int main()
{
     int p = 10;
     int q = 20;
     intercambio(&p, &q);
     printf("p=%d, q=%d", p, q);
         // se muestra p=20, q=10;
}
```

De esta forma, se logra modificar el contenido de p y q, a través de *a y *b que son los contenidos de las memorias apuntadas por a y b.

No se modifican los valores de a y b, sino que modifican los contenidos de las variables apuntadas por a y b.

Cohesión

Medida del grado de identificación de un módulo con una función concreta.

Cohesión aceptable (fuerte)

Cohesión funcional (un módulo realiza una única acción).
Cohesión secuencial (un módulo contiene acciones que han de realizarse en un orden particular sobre unos datos concretos).
Cohesión de comunicación (un módulo contiene un conjunto de operaciones que se realizan sobre los mismos datos).
Cohesión temporal (las operaciones se incluyen en un módulo porque han de realizarse al mismo tiempo; p.ej. inicialización).

Cohesión inaceptable (débil)

Cohesión procedural (un módulo contiene operaciones que se realizan en un orden concreto aunque no tengan nada que ver entre sí).
Cohesión lógica (cuando un módulo contiene operaciones cuya ejecución depende de un parámetro: el flujo de control o “lógica” de la rutina es lo único que une a las operaciones que la forman).
Cohesión coincidental (cuando las operaciones de una rutina no guardan ninguna relación observable entre ellas).

Acoplamiento

Medida de la interacción de los módulos que constituyen un programa.

Niveles de acoplamiento (de mejor a peor):

Acoplamiento de datos (acoplamiento normal): Todo lo que comparten dos rutinas se especifica en la lista de parámetros de la rutina llamada.

Acoplamiento de control: Una rutina pasa datos que le indican a la otra rutina que hacer (la primera rutina tiene que conocer detalles internos de la segunda).

Acoplamiento externo: Cuando dos rutinas utilizan los mismos datos globales o dispositivos de E/S. Si los datos son de sólo lectura, el acoplamiento se puede considerar aceptable. En general, este tipo de acoplamiento no es deseable porque la conexión existente entre los módulos no es visible.

Acoplamiento patológico: Cuando una rutina utiliza el código de otra o altera sus datos locales ("acoplamiento de contenido").
La mayor parte de los lenguajes estructurados incluyen reglas para el ámbito de las variables que impiden este tipo de acoplamiento.

Objetivo

Reducir al máximo el acoplamiento y aumentar la cohesión de los módulos.

Pasos para escribir un subprograma

1. Definir el problema que el subprograma ha de resolver.
2. Darle un nombre no ambiguo al subprograma.
3. Decidir cómo se puede probar el funcionamiento del subprograma.
4. Escribir la declaración del subprograma (cabecera de la función).
5. Buscar el algoritmo más adecuado para resolver el problema.
6. Escribir los pasos principales del algoritmo como comentarios.
7. Rellenar el código correspondiente a cada comentario.
8. Revisar mentalmente cada fragmento de código.
9. Repetir los pasos anteriores hasta quedar completamente satisfecho.

El nombre de un subprograma
Procedimiento: Verbo seguido de un objeto.
Función: Descripción del valor devuelto por la función.
El nombre debe describir todo lo que hace el subprograma.
Se deben evitar nombres genéricos que no dicen nada (p.ej. calcular).

Se debe ser consistente en el uso de convenciones.

Los parámetros de un subprograma
Orden: (por valor, por referencia) == (entrada, entrada/salida, salida)

Si varias rutinas utilizan los mismos parámetros, éstos han de ponerse en el mismo orden (algo que la biblioteca estándar de C no hace).
No es aconsejable utilizar los parámetros de una rutina como si fuesen variables locales de la rutina.
Se han de documentar las suposiciones que se hagan acerca de los posibles valores de los parámetros.
Sólo se deben incluir los parámetros que realmente necesite la rutina para efectuar su labor.
Las dependencias existentes entre distintos módulos han de hacerse explícitas mediante el uso de parámetros.

Diseño Estructurado de Sistemas

El diseño estructurado de sistemas se ocupa de la identificación, selección y organización de los módulos y sus relaciones. Se comienza con la especificación resultante del proceso de análisis, se realiza una descomposición del sistema en módulos estructurados en jerarquías, con características tales que permitan la implementación de un sistema que no requiera elevados costos de mantenimiento.

La idea original del diseño estructurado fue presentada en la década de los '70, por Larry Constantine, y continuada posteriormente por otros autores: Myers, Yourdon y Stevens.

1. Introducción

El diseño estructurado es un enfoque disciplinado de la transformación de qué es necesario para el desarrollo de un sistema, a cómo deberá ser hecha la implementación.

La definición anterior implica que: el análisis de requerimientos del usuario (determinación del qué) debe preceder al diseño y que, al finalizar el diseño se tendrá medios para la implementación de las necesidades del usuario (el cómo), pero no se tendrá implementada la solución al problema. Cinco aspectos básicos pueden ser reconocidos:

1. Permitir que la forma del problema guíe a la forma de la solución. Un concepto básico del diseño de arquitecturas es: las formas siempre siguen funciones.

2. Intentar resolver la complejidad de los grandes sistemas a través de la segmentación de un sistema en cajas negras, y su organización en una jerarquía conveniente para la implementación.

3. Utilizar herramientas, especialmente gráficas, para realizar diseños de fácil comprensión. Un diseño estructurado usa diagramas de estructura (DE) en el diseño de la arquitectura de módulos del sistema y adiciona especificaciones de los módulos y cuplas (entradas y salidas de los módulos), en un Diccionario de Datos (DD).

4. Ofrecer un conjunto de estrategias para derivar el diseño de la solución, basándose en los resultados del proceso de análisis.

5. Ofrecer un conjunto de criterios para evaluar la calidad de un diseño con respecto al problema a ser resuelto, y las posibles alternativas de solución, en la búsqueda de la mejor de ellas. El diseño estructurado produce sistemas fáciles de entender y mantener, confiables, fácilmente desarrollados, eficientes y que funcionan.

2. Diagrama de Estructura

Los diagramas de estructura (DE) sirven para el modelamiento top-down de la estructura de control de un programa descripto a través de un árbol de invocación de módulos. Fueron presentados en la década de los 70 como la principal herramienta utilizada en diseños estructurados, por autores como Constantine, Myers, Stevens e Yourdon. La Fig. 1 muestra un ejemplo:

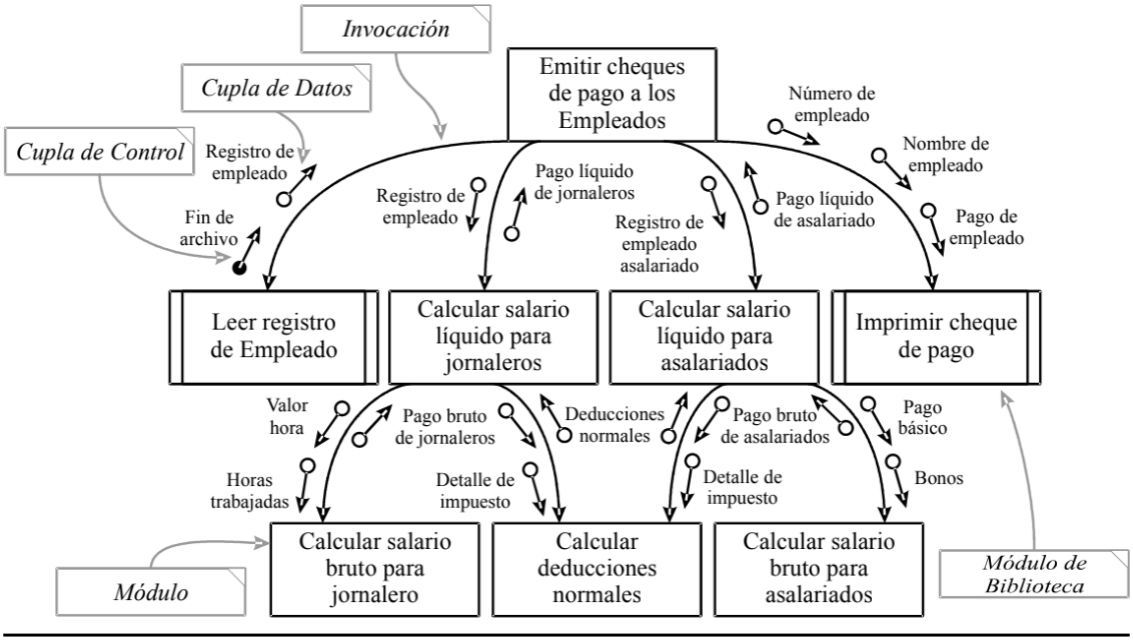


Fig. 1: Ejemplo de Diagrama de Estructura

Un diagrama de estructura permite modelar un programa como una jerarquía de módulos. Cada nivel de la jerarquía representa una descomposición más detallada del módulo del nivel superior. La notación usada se compone básicamente de tres símbolos:

- Módulos
- Invocaciones
- Cuplas

2.1 Módulos

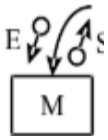
Un módulo es un conjunto de instrucciones que ejecutan alguna actividad, un procedimiento o función en PASCAL, una función en C o un parágrafo en COBOL. Tal vez, la definición más precisa es que un módulo es una caja negra, pero como será mostrado a continuación son cajas “casi” negras o grises.

Desde un punto de vista práctico, un módulo es una colección de instrucciones de un programa con cuatro características básicas:

1. *Entradas y Salidas*: lo que un módulo recibe en una invocación y lo que retorna como resultado.
2. *Función*: las actividades que un módulo hace con la entrada para producir la salida.
3. *Lógica Interna*: por la cual se ejecuta la función.
4. *Estado Interno*: su área de datos privada, datos para los cuales sólo el módulo hace referencia.

Las entradas y salidas son, respectivamente, datos que un módulo necesita y produce. Una función es la actividad que hace un módulo para producir la salida. Entradas, salidas y funciones proveen una visión externa del módulo. La lógica interna son los algoritmos que ejecutan una función, esto es, junto con su estado interno representan la visión interna del módulo.

Un módulo es diseñado como una caja, su función es representada por un nombre en el interior y las entradas y salidas de un módulo son representadas por pequeñas flechas que entran y salen del módulo.

Módulo		Un módulo es una caja negra conteniendo una colección de instrucciones y áreas de datos locales. Los módulos tienen una función, definida por el nombre contenido en el interior (M), datos de entrada y datos de salida generados por la aplicación de la función a los datos de entrada.

2.2 Relaciones entre Módulos (Invocaciones)

En la realidad, los módulos no son realmente cajas negras. Los diagramas de estructura muestran las invocaciones que un módulo hace a otros módulos. Estas invocaciones son diseñadas como una flecha que sale del módulo llamador y apunta al módulo llamado. La Fig. 2 muestra un ejemplo:

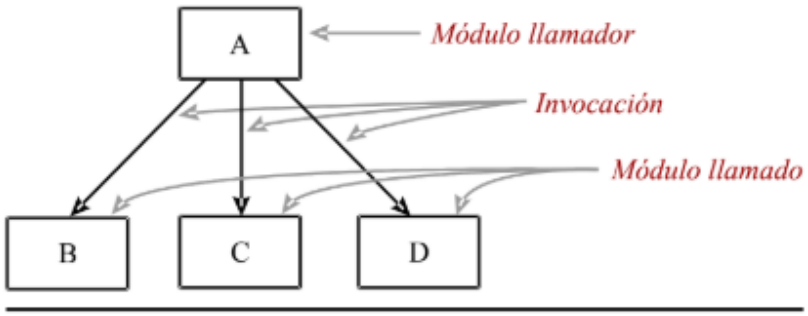
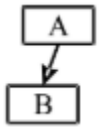


Fig. 2: Ejemplo de Invocación

Fig. 2: Ejemplo de Invocación En el ejemplo de la Fig. 2, el módulo **A** invoca (o llama) a los módulos **B**, **C** y **D**. La interpretación de las invocaciones provee información de la estructura interna del módulo llamador, que no concuerda con la idea de caja negra. Una caja negra no permite que se observe su interior y, las invocaciones que un módulo hace son componentes de su estructura interna. De todas formas, se dice que un módulo es *una caja casi negra o caja gris* porque ella permite que se observe solo las invocaciones.
Los diagramas de estructuras no tienen especificado el orden de invocación de los módulos invocados. El orden de dibujo de los módulos **B**, **C**, y **D** (de izquierda a derecha) no debe ser interpretado como el orden de invocaciones ejecutado por el módulo **A**. Ese orden puede ser cambiado, al dibujar, para evitar que se crucen flechas o se dupliquen módulos, como ocurre con el módulo *Calcular Deducciones Normales* en la Fig. 1. A pesar que el orden de invocación de los módulos del mismo nivel en un diagrama de estructura, no es especificado por el formalismo, se recomienda que siempre que fuese posible, se siga un orden de izquierda a derecha (si esto no produce que se crucen flechas) que se corresponde con el orden de invocación, y permitiendo un orden de lectura que es patrón en la mayoría de los idiomas.

Una invocación, representa la idea de llamada a funciones o procedimientos en los lenguajes de programación convencionales. A continuación se describe una invocación estándar:

Invocación estándar		El módulo A invoca al módulo B con la semántica de invocación de procedimientos o funciones en los lenguajes de programación convencionales (C, Pascal, etc.).

2.3 Comunicación entre Módulos (Cuplas)

Cuando una función o un procedimiento, en un lenguaje convencional, es invocado, comúnmente un conjunto de argumentos es comunicado y, en el caso de las funciones, también se espera que retorne un resultado. Estos datos comunicados en una invocación son modelados por medio de flechas, sobre el símbolo de invocación, llamadas *cuplas*.

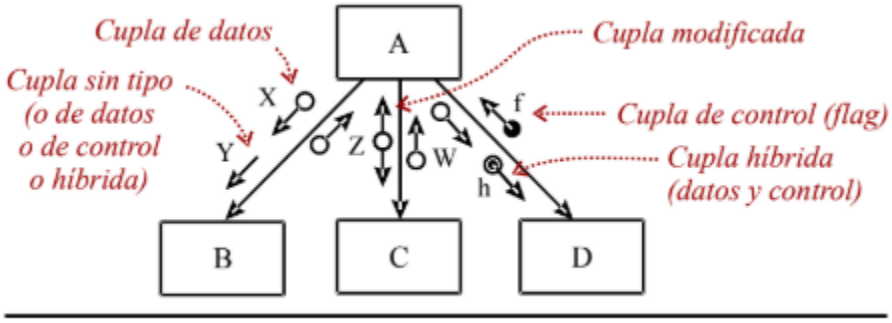


Fig. 3: Ejemplo de invocación con cuplas

Como se muestra en la Fig. 3, existen varios tipos de cuplas, basado en lo que ellas pueden producir en el módulo receptor, las cuales se describen a continuación:

Tipos de Cuplas		
Cupla de datos		Una cupla de datos transporta datos “puros” a un módulo. No es necesario conocer la lógica interna del módulo receptor, para determinar los valores válidos de la cupla (ej.: número de cuenta, saldo, tabla de movimientos).
Cupla modificada		Con una flecha doble (apuntando al módulo llamador y al módulo llamado) se especifica un argumento enviado a un módulo que deberá modificar su valor, fijando un nuevo valor disponible para el módulo llamador (en la implementación, se precisará que el lenguaje posea un mecanismo de pasaje de parámetros por referencia) (ej.: el buffer enviado a un módulo de lectura de un archivo).
Cupla de resultados		Existen módulos que retornan valores sin la necesidad de que estén inicializados en el momento que se invocan. Estos casos son dos: 1. Cuplas similares al tipo Modificada cuyos valores previos a la invocación del módulo NO se utilizan para calcular el valor de retorno. Si bien en Pascal se implementa como las “Cuplas Modificadas”, es conveniente documentarlas en el DE como “de Resultados” por cuestiones de claridad. 2. Si el Módulo en cuestión es una función (retorna un valor), se debe documentar este valor de retorno como “Cupla de Resultado” cuyo nombre se corresponderá con el nombre de la función.

3. Criterios de Validación de Calidad

Los diagramas de estructura son simplemente una herramienta para modelar los módulos de un sistema y sus relaciones y, junto con las especificaciones de funcionalidad de los módulos y las estructuras de datos de las cuplas, componen un diseño inicial que deberá ser analizado y mejorado.

Uno de los principios fundamentales del diseño estructurado es que un sistema grande debería particionarse en módulos más simples. Sin embargo, es vital que esa partición sea hecha de tal manera que los módulos sean tan independientes como sea posible y que cada módulo ejecute una única función. Para que los diseños tengan esas cualidades, son necesarios algunos criterios de medición que permitan clasificar y mejorar los diagramas de estructura. A continuación se describen los criterios utilizados para mejo- rar un diseño.

3.1 Acoplamiento

El acoplamiento entre módulos clasifica el grado de independencia entre pares de módulos de un DE. El objetivo es minimizar el acoplamiento, es decir, maximizar la in- dependencia entre módulos. A pesar de que el acoplamiento, es un criterio que clasifica características de *una invocación* (una relación existente entre dos módulos), será usado para clasificar un DE completo. Un DE se caracteriza por el *peor* acoplamiento existente entre pares de sus módulos, ya que ese es el problema que debe ser resuelto para mejorar la calidad del DE completo.

Un bajo acoplamiento indica un sistema bien particionado y puede obtenerse de tres maneras:

• *Eliminando relaciones innecesarias*: Por ejemplo, un módulo puede recibir algunos datos, innecesarios para él, porque debe enviarlos para un módulo subordinado.

• *Reduciendo el número de relaciones necesarias*: Cuanto menos conexiones existan entre módulos, menor será la posibilidad del efecto en cadena (un error en un módulo aparece como síntoma en otro).

• *Debilitando la dependencia de las relaciones necesarias*: Ningún módulo se tiene que preocupar por los detalles internos de implementación de cualquier otro. Lo único que tiene que conocer un módulo debe ser su función y las cuplas de entrada y salida (cajas negras).

3.2 Cohesión

Otro medio para evaluar la partición en módulos (además del acoplamiento) es observar cómo las actividades de un módulo están relacionadas unas con otras; este es el criterio de cohesión. Generalmente el tipo de cohesión de un módulo determina el nivel de acoplamiento que tendrá con otros módulos del sistema.

Cohesión es la medida de intensidad de asociación funcional de los elementos de un módulo. Por elemento debemos entender una instrucción, o un grupo de instrucciones o una llamada a otro módulo, o un conjunto de procedimientos o funciones empaquetados en el mismo módulo.

El objetivo del diseño estructurado es obtener módulos altamente cohesivos, cuyos elementos estén fuerte y genuinamente relacionados unos con otros. Por otro lado, los elementos de un módulo no deberían estar fuertemente relacionados con elementos de otros módulos, porque eso llevaría a un fuerte acoplamiento entre ellos.

3.3 Descomposición (Factoring)

La descomposición es la separación de una función contenida en un módulo, para un nuevo módulo. Puede ser hecha por cualquiera de las siguientes razones.

3.3.1 Reducir el tamaño del módulo

La descomposición es una manera eficiente de trabajar con módulos grandes. Un buen tamaño para un módulo es alrededor de media página (30 líneas). Ciertamente, toda codificación de un módulo debería ser visible en una página (60 líneas).

La cantidad de líneas no es un patrón rígido, otros criterios para determinar cuándo es conveniente terminar de realizar la descomposición, son los siguientes:

• *Funcionalidad*: Terminar de realizar la descomposición cuando no se pueda encontrar una función bien definida. No empaquetar líneas de código dispersas, de otros módulos, porque probablemente juntas podrán formar módulos con mala cohesión.

• *Complejidad de Interfaz*: Terminar de realizar la descomposición cuando la interfaz de un módulo *es tan compleja* como el propio módulo. Un módulo de mil líneas es muy confuso, más mil módulos de una línea son aún más confusos.

3.3.2 Hacer el sistema más claro

La descomposición no debería ser hecha de una manera arbitraria, los módulos resultantes de la descomposición de un módulo deben representar sub-funciones del módulo de más alto nivel en el DE.
En una descomposición no se debe preocupar por conceptos de programación. Si una sub-función, presentada como un módulo separado permite una mejor comprensión del diseño, puede ser subdividida, aún cuando, en una implementación, el código del módulo sea programado dentro del módulo jefe.

3.3.3 Minimizar la duplicación de código

Cuando se reconoce una función que puede ser reutilizada en otras partes del DE, lo más conveniente es convertirla en un módulo separado. Así, se puede localizar más fácilmente las funciones ya identificadas y evitar la duplicación del mismo código en el interior de otro módulo. De esta manera, los problemas de inconsistencia en el mantenimiento (si esa función debe ser modificada) pueden ser evitados, y se reduce el costo de implementación.

3.3.4 Separar el trabajo de la ‘administración’

Un administrador o gerente de una compañía bien organizada debería coordinar el trabajo de los subordinados en lugar de hacer el trabajo. Si un gerente hace el trabajo de los subordinados no tendrá tiempo suficiente para coordinar y organizar el trabajo de los subordinados y, por otro lado, si hace el trabajo los subordinados no serían necesarios. Lo mismo se puede aplicar al diseño del DE, relacionado a los módulos de Trabajo (edición, cálculo, etc.) y a los módulos de Gerencia (decisiones y llamadas para otros módulos).
El resultado de este tipo de organización es un sistema en el cual los módulos en los niveles medio y alto de un DE son fáciles de implementar, porque ellos obtienen el trabajo hecho por la manipulación de los módulos de los niveles inferiores.

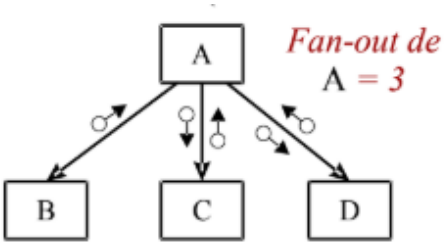
La separación del trabajo de la administración mejora la mantenibilidad del diseño. Una alteración en un sistema es: un cambio de control o un cambio de trabajo, pero raramente ambos.

3.3.5 Crear módulos más generales

Otra ventaja de la descomposición es que, frecuentemente, se pueden reconocer módulos más generales y, así, más útiles y reutilizables en el mismo sistema y, además, pueden ser generadas bibliotecas de módulos reutilizables en otros sistemas.

3.4 Fan-Out

El *fan-out* de un módulo es usado como una medida de *complejidad*. Es el número de subordinados inmediatos de un módulo (cantidad de módulos invocados).

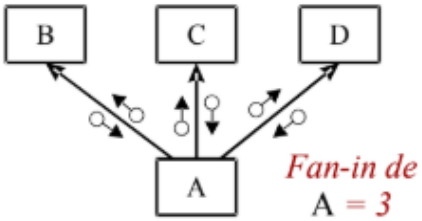


Si un módulo tiene un fan-out muy grande, entonces compone el trabajo de muchos módulos subordinados y, casi con certeza, tiene toda la funcionalidad no trivial representada por ese subárbol en el DE.

Para tener acotada la complejidad de los módulos se debe limitar el *fan-out* a no más de siete más o menos dos (7 ± 2). Un módulo con muchos subordinados puede fácilmente ser mejorado por descomposición.

3.5 Fan-In

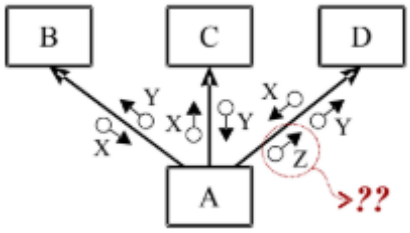
El *fan-in* de un módulo es usado como una medida de *reusabilidad*, es el número de superiores inmediatos de un módulo (la cantidad de módulos que lo invocan).



Un alto fan-in es el resultado de una descomposición inteligente. Durante la programación, tener una función llamada por muchos superiores evita la necesidad de codificar la misma función varias veces. Existen dos características fundamentales que deben ser garantizadas en módulos con un alto fan-in:

• *Buena Cohesión*: Los módulos con mucho fan-in deben tener alta cohesión, con lo cual es muy probable que tengan buen acoplamiento con sus llamadores.

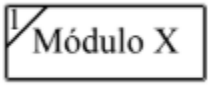
• *Interfaz Consistente*: Cada invocación para el mismo módulo debe tener el mismo número y tipo de parámetros. En el ejemplo que sigue, hay un error.



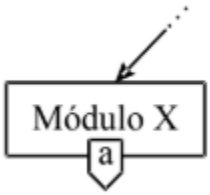
4. Algunas consideraciones adicionales

1. En primer término es importante destacar que bajo ningún aspecto es aceptable graficar una invocación de un módulo hacia otro que esté dibujado más arriba, es decir que no se puede dibujar ninguna flecha con sentido ascendente, a lo sumo se puede realizar una invocación hacia un módulo que está a la misma altura (hermano), en cuyo caso la flecha queda horizontal.

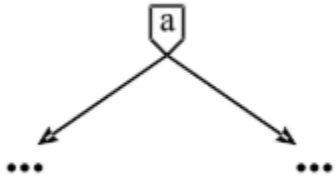
2. En los casos que exista una importante reutilización puede ser necesario dibujar el mismo módulo en distintos lugares del gráfico a fin de que las invocaciones existentes en el diagrama no se crucen. Cuando esto ocurre el módulo debe ser dibujado en cada ocurrencia con la incorporación de una línea diagonal en el extremo superior izquierdo y un número que lo identifica (todos los cuadros que se refieren al mismo módulo tie- nen el mismo nombre y el mismo número).
Estos módulos se grafican entonces de la siguiente manera:



3. Adicionalmente, si algunos de estos módulos u otros dentro de un diagrama de gran tamaño deben continuar la cadena de invocación se puede utilizar la siguiente sintaxis para desdoblar el diagrama de estructuras.
En el diagrama original se incorpora un pentágono rotulado (en este caso “a”) .



Se comienza un diagrama aparte indicando en el inicio el mismo pentágono con idéntico rótulo y continuando la gráfica de los módulos invocados por el módulo original.



Estructuras de Datos Estáticas y Dinámicas.

Temas:

- Algoritmos sobre arreglos: búsqueda del mínimo, máximo, ordenación por selección e inserción, etc.
- Arreglos paralelos. Algoritmos sobre arreglos paralelos.
- Concepto de tipo de dato definido por el programador (struct o registro).
- Arreglos de registros. Algoritmos sobre arreglos de registros.
- Arreglos dinámicos.

Búsqueda de la posición del menor elemento en un arreglo a partir de una posición dada.

Algoritmo:

1. Se almacena en una variable auxiliar el elemento del arreglo correspondiente a una posición dada (se lo presupone como el menor), y en otra variable la posición dada.
2. Se compara este valor con el resto de los valores del arreglo, y cada vez que se encuentra un elemento menor al guardado en la variable auxiliar, se actualiza este valor con el encontrado.

```
int posicionMenor(int a[], int cantVal, int pos){
    int menor = a[pos];
    int posMenor = pos;
    int index = pos + 1;
    while (index < cantVal){
        if(menor > a[index]){
            menor = a[index];
            posMenor = index;
        }
        index++;
    }
    return posMenor;
}
```

Ordenación por el método de selección

Algoritmo:

1. Se busca la posición del menor elemento de un arreglo comenzando desde la posición i, y se intercambia el menor elemento con el de la posición i.
2. Se repite el primer paso comenzando con i = 0 y terminando con i = N-1; siendo N el tamaño del arreglo.

19	**23**	**45**	**33**	**18**	**1**	**12**	**9**
0	1	2	3	4	5	6	7

i

1	23	45	33	18	**19**	12	9

i

1	23	45	33	18	**19**	12	9

i

1	**9**	45	33	18	**19**	12	**23**

i

!!

```
void ordenacionSeleccion(int a[ ], int cantVal){
    int posMenor;
    int i = 0;
    while(i< cantVal - 1){ /// llego hasta la anteúltima posición
        posMenor = posicionMenor(a, cantVal, i);
        aux = a[posMenor];
        a[posMenor] = a[i];
        a[i] = aux;
        i++;
    }
}
```

Las instrucciones

```
    aux = a[posMenor];
    a[posMenor] = a[i];
    a[i] = aux;
```

pueden cambiarse por una función que realice la tarea de intercambio.

```
void intercambio(int A[ ], int i, int j){
    int aux = A[i];
    A[i] = A[j];
    A[j] = aux;
}
```

```
void intercambio (int *a, int *b){
    int aux;
    aux = *a;
    *a = *b;
    *b = *aux;
}

void ordenacionSeleccion(int a[ ], int cantVal){
    int posMenor;
    int i = 0;
    while(i<cantVal - 1){ //llego hasta la anteúltima posición
        posMenor = posicionMenor(a, cantVal, i);
        intercambio(&a[porMenor], &a[i]);
        i++;
    }
}
```

Inserción de un elemento en un arreglo ordenado (manteniendo el orden)

Supongamos que debemos insertar un nuevo elemento en un arreglo ordenado. Se supone que el arreglo no ha sido usado en forma completa.

10	20	30	40				
			u				

u marca la posición de la última celda ocupada con información útil.
Y sea **dato** una variable con el valor a insertar.

Se recorre el arreglo desde la última posición a la primera buscando el lugar en donde insertar el nuevo dato. Mientras esto se hace, se realiza también un “corrimiento” de los elementos del arreglo, para crear un espacio en donde ubicar el nuevo valor.

Supongamos que se desea insertar el valor 25.-

10	20	30	40				
			u				

10	20	25	30	40			
				u+1			

```
void insertar(int a[ ], int u, int dato){
    int i= u; //ultima pos valida izq
    while (i >= 0 && dato < a[i]){
        a[i+1] = a[i];
        i--;
    }
    a[i+1] = dato;
}
// Se debe incrementar el valor de u para indicar que ha aumentado el
tamaño del arreglo.
```

Ordenación por inserción

Supongo un arreglo completo y desordenado.

```
void ordenacionInsercion(int a[ ], int cantVal){
    int u=0;
    while (u < cantVal - 1){
     //llega hasta la posición del anteúltimo elemento del arreglo.
        insertar(a, u, a[u+1]);
        u++;
    }
}
```

A medida que el algoritmo evoluciona, se marca en el arreglo una “zona ordenada” a la izquierda, separada de otra “zona desordenada” a la derecha. Se va tomando el siguiente elemento (a[u+1]) de la zona desordenada y se lo inserta dentro de la zona ordenada (se utiliza también el lugar u+1, para hacer crecer la zona ordenada).

Arreglos paralelos

Los arreglos paralelos (dos o más) son aquellos que están relacionados por un mismo índice, o sea, las celdas ubicadas en la misma posición (para distintos arreglos) tienen un vínculo entre sus datos. Se utilizan generalmente para asociar datos de distintos tipos y almacenarlos en arreglos de tipos de datos básicos.

En realidad no representa un nuevo tipo de datos, sino una forma en particular de trabajar con un grupo de arreglos.

Por ejemplo:

Almacenar los datos (número de legajo, nombre y año) de un grupo de alumnos. Se crean tres arreglos del mismo tamaño, cada uno del tipo correspondiente para que pueda almacenar uno de los datos.

legajos	121	122	123	124	125
nombres	Alvarez	Perez	Lopez	Garcia	Simpson
años	2	1	2	3	3
	0	1	2	3	4

Como es de suponer, los datos de igual número de celda de cada uno de los arreglos están vinculados, por pertenecer a una misma “entidad de datos”. Esto tiene que ver con el diseño, o sea con el significado que el programador le da a los datos (semántica).

En nuestro ejemplo tenemos almacenados los datos de cinco alumnos; en la celda número uno (subíndice 0) está almacenado el alumno Alvarez cuyo legajo es 121 y cursa el 2 año, y así con los demás.

Inserción ordenada en arreglos paralelos

```
void insertar(int id_windows[ ],int b[ ], int c[ ] int validos, int id, int
datob, int datoc){
    int i= validos-1; //ultima pos valida
    while (i >= 0 && id < id_windows[i]){
        id_windows[i+1] = id_windows[i];
        b[i+1] = b[i];
        c[i+1] = c[i];
        i--;
    }
    id_windows[i+1] = id;
    b[i+1] = datob;
    c[i+1] = datoc;
}

void carga(int id_windows[],int b[], int c[], int* validos)
{
…
do{
  printf(“ingrese id windows”);
  scanf(“%i”, &id);
  …
  insertar(id_windows,b,c,*validos,id,datob,datoc);
  (*validos)++; // *validos = *validos + 1;
}while(*validos < CANTMAX && continuarr == ‘s’);

}
// Se debe incrementar el valor de validos para indicar que ha aumentado el
tamaño del arreglo.
```

Ordenación sobre arreglos paralelos

Para ordenar arreglos paralelos se debe establecer uno de ellos como el arreglo que determinará el criterio de ordenación. Todos los arreglos deberán modificar la ubicación de sus elementos según la ordenación establecida por uno de ellos.

Por ejemplo, si se desea ordenar el conjunto de arreglos paralelos del ejemplo anterior, por orden de legajo, según el algoritmo de ordenación por selección, se tiene:

```
int posicionMenor(int leg[ ], int cantVal, int u){
    int menor = leg[u];
    int posMenor = pos;
    for(int index = u + 1; index < cantVal; index ++){
        if(menor < leg[index]){
            menor = leg[index];
            posMenor = index;
        }
    }
    return posMenor;
}
```

```c
void intercambioPalabra(char pal[ ][30], int i, int j){
    char aux[30];
    strcpy(aux, pal[i]);
    strcpy(pal[i], pal[j]);
    strcpy(pal[j], aux);
}
```

```c
void intercambioEntero(int numero[ ], int i, int j){
    int aux = numero[i];
    numero[i] = numero[j];
    numero[j] = aux;
}
```

```c
void ordenacionSeleccion(int leg[ ], char nombre[ ][30], int anio[ ],
int cantVal ) {
//los tres arreglos tienen la misma cantidad de elementos válidos
    int posMenor;
    int i=0;
    while(i<cantVal - 1){ /// llego hasta la anteúltima posición
        posMenor = posicionMenor(leg, i);
        intercambioPalabra(nombre, posMenor, i);
        intercambioEntero(leg, posMenor, i);
        intercambioEntero(anio, posMenor, i);
        i++;
    }
}
```

Queda para el alumno resolver el algoritmo de ordenación por inserción.

NOTA: como ya vimos en el apunte “Arreglos en C”, un **arreglo de palabras** se define igual que una matriz de caracteres, en donde el primer subíndice determina la cantidad de palabras del arreglo y el segundo subíndice determina la cantidad máxima de caracteres de cada palabra.

Para trabajar cada celda del arreglo de palabras simplemente se usa el nombre de la matriz de caracteres y solamente el primer subíndice, por ejemplo, sea un arreglo de 10 palabras de longitud 30 cada una:

```c
char palabras[10][30];
```

para acceder a la palabra de la celda i (0 <= i < 10) escribimos: **palabras[i]** dentro de la instrucción correspondiente.

Arreglos dinámicos

Se denomina arreglo dinámico a aquel que su tamaño es definido en tiempo de ejecución. En el arreglo estático el tamaño es definido en tiempo de compilación.

Para crear un arreglo en tiempo de ejecución se utiliza la instrucción **malloc**.
Sintaxis:

```
        tipo * variable;
        int cantidad = ...;
        variable = (tipo *) malloc (cantidad * (sizeof(tipo));
```

La instrucción sizeof(tipo) retorna la cantidad de bytes en memoria que ocupa un dato del tipo indicado. Ahora la cantidad de celdas del arreglo es un dato que se puede almacenar en una variable (cantidad). O sea que puede ser ingresada por el usuario por teclado o calculada.

Esta instrucción crea un arreglo que se puede utilizar de la misma manera que los arreglos estáticos, con la ventaja de posponer hasta el tiempo de ejecución la determinación de su tamaño.

La instrucción malloc busca en la memoria disponible una zona de tamaño igual a cantidad * sizeof(tipo), la reserva y retorna la dirección del comienzo de esta zona. Por lo tanto, la variable que determina el arreglo, en realidad es una variable de tipo puntero con la dirección de memoria de la primer celda del arreglo. Esto es válido para los arreglos estáticos también.

Ejemplo:

```
void main() {
    int * a;
    int cantidad;
    printf("ingrese el tamaño de su arreglo: ");
    scanf("%d", &cantidad);
    a = (int *) malloc (cantidad * sizeof(int));
    a[3] = ....
    …
}
```

Esto crea un arreglo llamado a, de tamaño (cantidad de celdas) igual al valor ingresado por el usuario en la variable cantidad.
Son muy útiles para ahorrar memoria.
Ver también las funciones calloc() y realloc().

RECURSIÓN

Definición

Un procedimiento o función se dice recursivo si durante su ejecución se invoca directa o indirectamente a sí mismo. Esta invocación depende al menos de una condición que actúa como condición de corte que provoca la finalización de la autoinvocación.

Definición

Repetición por autoreferencia, cuando una función se invoca a sí misma (puede ser en forma directa o indirecta).

Es la forma en la cual se especifica un proceso basado en su propia definición, pero con menor complejidad. De esta forma, en algún momento se llega a un proceso muy simple que puede resolverse facilmente (se llama solución trivial).

Parece un concepto nuevo, pero ya lo hemos visto en varias definiciones matemáticas…

Definiciones por inducción

- *El uno pertenece a los Naturales, y si N pertenece a los Naturales, N+1 también.*

- *Factorial(0) = 1* *(solución trivial)*

  *Factorial(N) = N * Factorial(N-1), para todo N > 0.*

- *0 ! = 1*

  *N ! = N * (N - 1) !*

Desarrollo de la función factorial

```
int factorial (int x)
{
   int rta;
   If ( x = = 0)
        rta = 1;
   else
        rta = x * factorial (x-1) ;
   return rta;
}
```

z =

```
factorial(3)
{  int rta;
   if (3 = = 0)  rta = 1;
   else
      rta= 3 * factorial(2)
               { int rta;
                 if (2 = = 0)  rta = 1;
                 else
                    rta= 2 * factorial(1)
                             { int rta;
                               if (1 = = 0)  rta = 1;
                               else
                                  rta= 1 *
                                                  
                             return rta;
                             }
                 return rta;
               }
   return rta;
}
```

Cómo funciona esto ??

En el momento de realizar una invocación recursiva, se suspende la ejecución de la función invocante hasta que se termina de resolver la función invocada.

Si bien es la misma función la que se invoca, se genera un nuevo espacio de memoria para la resolución de la misma.

Cada espacio tiene sus propias variables locales y parámetros, con valores propios.

Para poder resolver **factorial(3),** primero se debe resolver factorial(2), pero…

Para poder resolver **factorial(2),** primero se debe resolver factorial(1), pero…

Para poder resolver **factorial(1),** primero se debe resolver factorial(0), y…

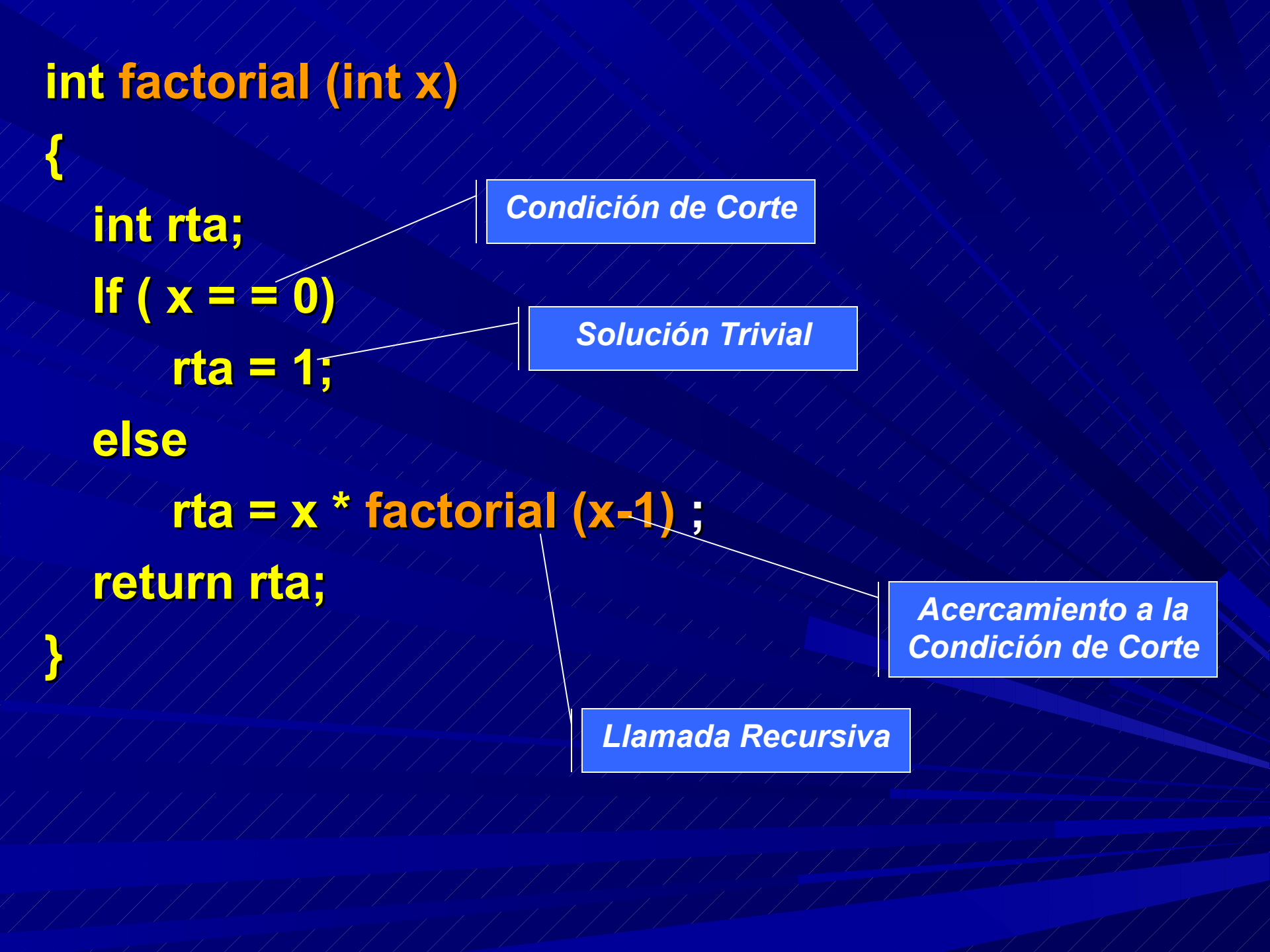
factorial(0) es 1

Reglas de la buena recursión

Toda función recursiva debe tener:

1. Al menos una Condición de Corte con su respectiva Solución Trivial.
2. Al menos una Llamada Recursiva (o Entrada en Recursión).
3. En cada Llamada Recursiva, se vuelve a invocar el mismo algoritmo, pero con un caso más simple de resolver. Se produce un acercamiento a la Condición de Corte.
4. Al llegar a la Solución Trivial, queda expresada la solución total (considerando toda la fórmula que quedó pendiente de resolver).

```
int factorial (int x)
{
   int rta;
   If ( x = = 0)
        rta = 1;
   else
        rta = x * factorial (x-1) ;
   return rta;
}
```



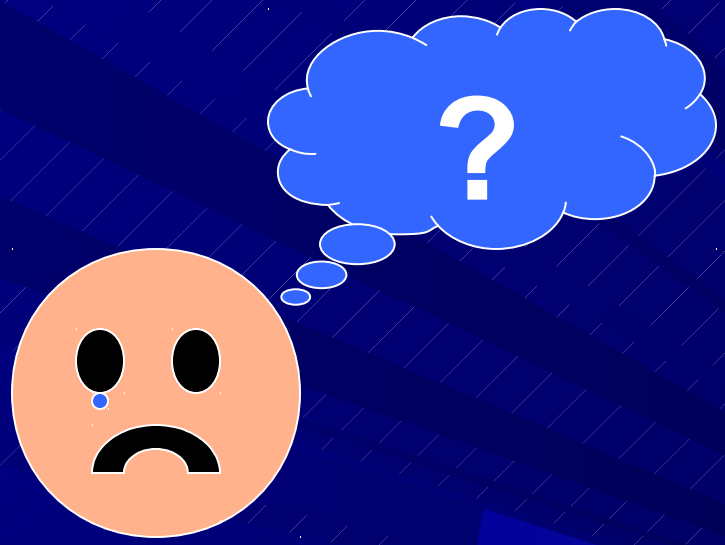
Usar recursión cuando:

1. La estructura de la función es recurrente y el algoritmo resulta más sencillo.
2. La estructura de datos es recursiva.

Definición de Lista

estructura de datos recursiva

El NULL es una Lista y además, un Nodo seguido de una Lista también es Lista.



Ejemplo: recorrer una lista vinculada

Se trata de una estructura de datos recursiva. Considerar la estructura que define al nodo.

```
void recorrer( Nodo * lista)
{
    if (lista != NULL)
    {
        printf("%d", lista->dato);
        recorrer(lista->siguiente);
    }
}
```

Condición de Corte
Corta cuando es falso

Llamada Recursiva

Acercamiento a la Condición de Corte

Solución Trivial: ausencia de ELSE. No hace nada al llegar a NULL

Qué hace el siguiente código?

```
void recorrer( Nodo * lista)
{
   if (lista != NULL)
   {
   recorrer(lista->siguiente);
   printf("%d", lista->dato);
   }
}
```

Tener en cuenta que para recorrer la lista NO hace falta usar while ni for. Ni ningún otra estructura de repetición. En vez de eso, se tiene que:

1. La recursión se encarga de repetir el algoritmo (printf en este caso).
2. La Llamada Recursiva con el acercamiento a la condición de corte hace que se avance en la lista.
3. La condición de corte finaliza el recorrido de la lista, justo cuando la lista se termina.
4. La solución trivial resuelve el algoritmo para la lista más simple, o sea la lista nula. Y la solución es no hacer nada, en este caso no imprimir.

Cómo se piensa en forma recursiva ?

1. **Encontrar la solución trivial.**
2. **Armar la expresión de la solución total del problema, contando con la solución al problema para una etapa "una vez mas cercana" a la solución trivial.**

Las reglas de la buena recursión completan todo lo que falta para que esta forma simple de razonamiento dé resultado.

Ejemplo: sumar el contenido de una lista de enteros.

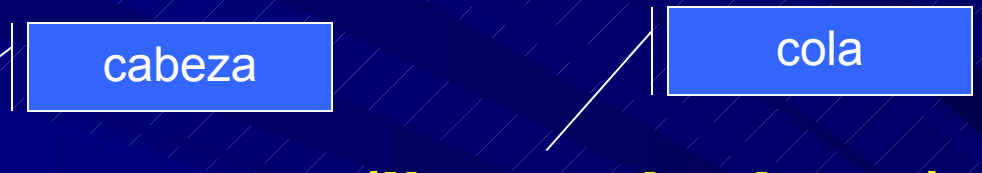
Para solucionar este problema planteo el siguiente algoritmo:

"La suma de una lista vacía es 0"

"La suma de los números de una lista será igual a la suma entre el primero y el resultado de la suma de la sublista siguiente."

En este caso particular del ejemplo de la lista, se separa el primer elemento y se *supone conocida la solución de los restantes*. También se le llama "separarlo en *cabeza y cola"*.

```
int suma(Nodo * lista)
{
    int rta;
    if(lista == NULL)
      //"La suma de una lista vacía es 0"
      rta = 0;
    else
      rta = lista->dato + suma(lista->siguiente);
      /*"La suma de los números de una lista será igual a la
    suma entre el primero y el resultado de la suma de la
    sublista siguiente."*/
    return rta;
}
```



Ejemplo: determinar si un arreglo es capicúa.

```
int capicua(int a[], int i, int j)
{     int rta;
      if(i<j)  //condición de corte
         if(a[i] == a[j]) //condición de corte
            rta = capicua(a, i+1, j -1); //recursión
         else
            rta = 0; //solución trivial
      else
         rta = 1; //solución trivial
      return rta;
}
```

Relación entre Recursión e Iteración

- *Tanto la iteración como la recursión:*
  - *se basan en una estructura de control: la iteración utiliza una estructura de repetición y la recursión una estructura de selección.*
  - *incluyen repeticion: la iteración utiliza una estructura de repetición de forma explícita; mientras que la recursión consigue la repetición mediante llamadas de función repetidas.*
  - *incluyen una prueba de terminación: la iteración termina cuando falla la condición de continuación del ciclo; la recursión termina cuando se reconoce el caso base.*
  - *pueden ocurrir de forma infinita: la iteración se puede quedar en ciclo infinito si la condición del ciclo nunca se hace falsa; la recursión se hará infinita si el paso de recursión no reduce el problema de tal forma que converja al caso base.*
- *La recursión invoca de manera repetida, y por lo tanto, puede sobrecargar las llamadas de función. Esto puede resultar costoso tanto en tiempo de procesador como en espacio de memoria.*

Recursión e Iteración

- Si un lenguaje de programación como C provee tanto recursión como iteración, cuál de los dos métodos debería ser usado para resolver un problema dado?
- En una solución recursiva el computador debe controlar y resolver cada una de las invocaciones manteniendo información acerca del resultado las mismas.
- Esto puede ser costoso en tiempo y espacio.
- En general, se recomienda el uso de una solución recursiva sólo en caso que:
  - la solución no puede ser fácilmente expresable iterativamente (sucede frecuentemente).
  - la eficiencia de la solución recursiva es satisfactoria.

Agregar nodos en una Lista con C AL FINAL con RECURSIVIDAD

Lic. Gabriel Chaldú

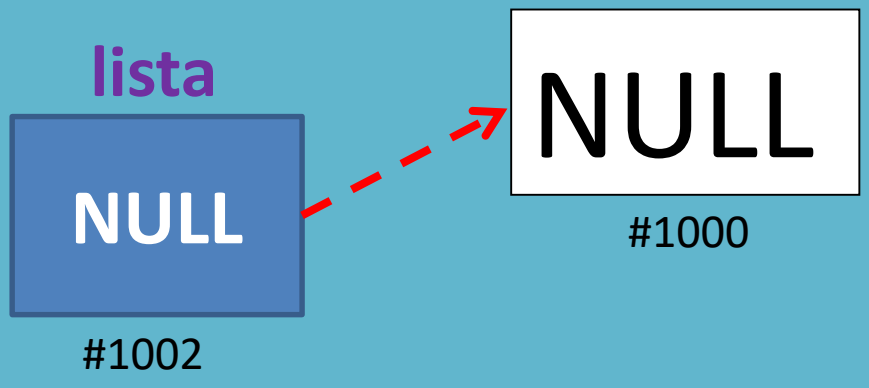
YOUTICS

Agregar nodos con RECURSIVIDAD

Dato=5
Siguiente = NULL

nuevoNodo
#1234

```
nodo * agregarFinalRecursivo(nodo * lista, nodo * nuevoNodo)
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = agregarFinalRecursivo(lista->siguiente, nuevoNodo);
    }
    return lista;
}
```



Agregar nodos con RECURSIVIDAD

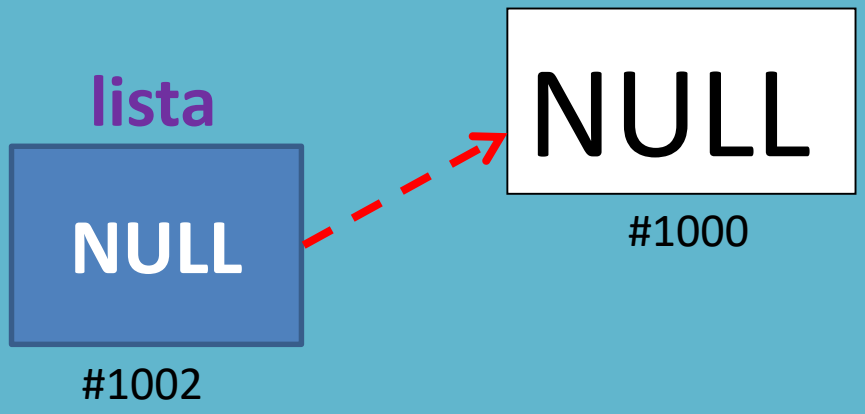
Dato=5
Siguiente = NULL

NuevoNodo
#1234

```
nodo * agregarFinalRecursivo(nodo * lista, nodo * nuevoNodo)
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = agregarFinalRecursivo(lista->siguiente, nuevoNodo);
    }
    return lista;
}
```

Lista ==NULL? **VERDADERO**

Lista SE VUELVE nuevoNodo



Agregar nodos con RECURSIVIDAD

Dato=5
Siguiente = NULL

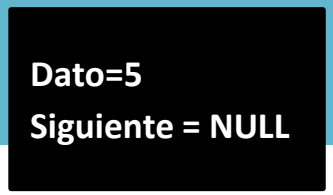
NuevoNodo
#1234

```
nodo * agregarFinalRecursivo(nodo * lista, nodo * nuevoNodo)
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = agregarFinalRecursivo(lista->siguiente, nuevoNodo);
    }
    return lista;
}
```

Lista ==NULL? **VERDADERO**

Lista SE VUELVE nuevoNodo

Agregar nodos con RECURSIVIDAD



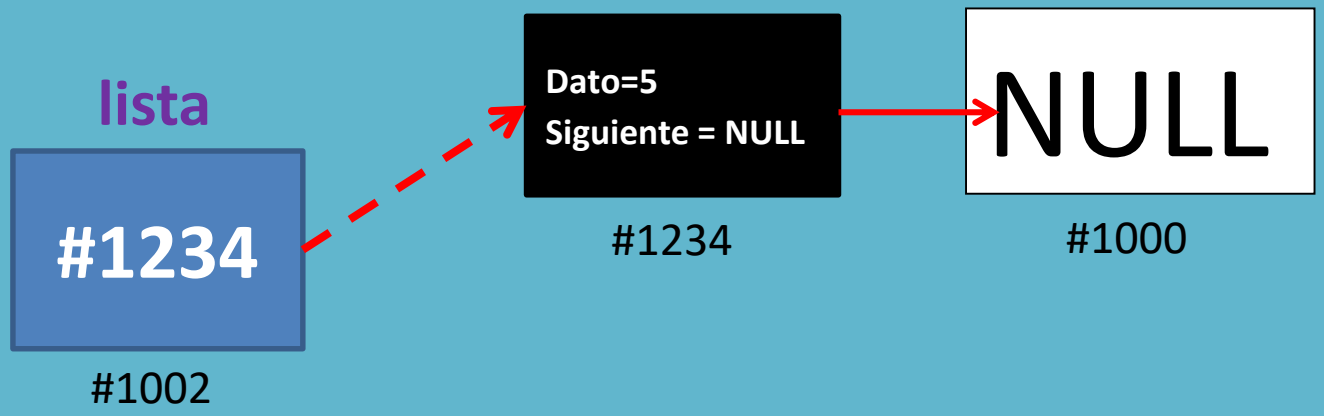
NuevoNodo
#1234

```
nodo * agregarFinalRecursivo(nodo * lista, nodo * nuevoNodo)
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = agregarFinalRecursivo(lista->siguiente, nuevoNodo);
    }
    return lista;
}
```

Lista ==NULL? **VERDADERO**

Lista SE VUELVE nuevoNodo

Lista apunta a #1234



Agregar 2º nodo
1º Etapa Recursiva

Dato=6
Siguiente = NULL

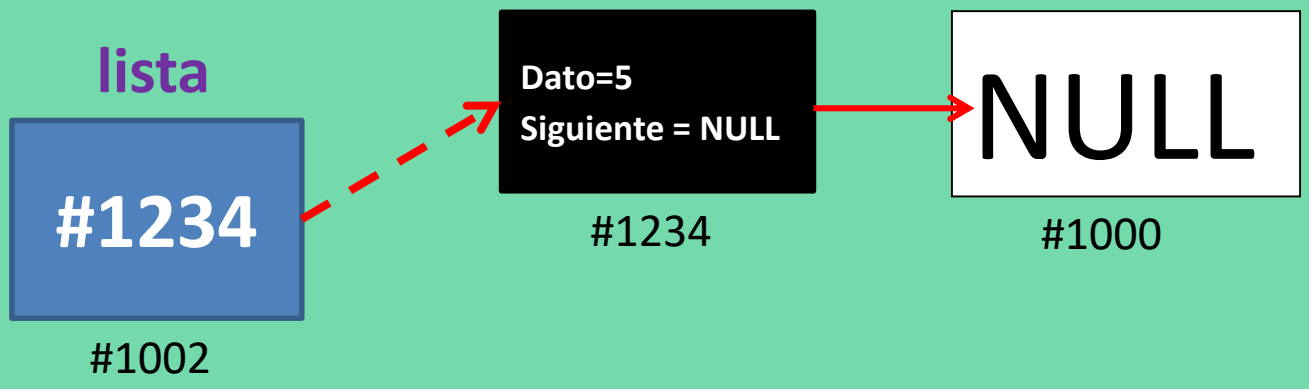
NuevoNodo

NOTA: EN ESTA ETAPA RECURSIVA
lista = #1234
lista->siguiente = #1000 (NULL)

```
nodo * agregarFinalRecursivo(nodo * lista, n
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = agregarFinalRecursivo(lista->siguiente, nuevoNodo);
    }
    return lista;
}
```

Lista ==NU

ENTRO EN RECURSIVIDAD



Agregar 2º nodo
1º Etapa Recursiva

Dato=6
Siguiente = NULL

NuevoNodo
#1001

```
nodo * agregarFinalRecursivo(nodo * lista, nodo * nuevoNodo)
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = agregarFinalRecursivo(lista->siguiente, nuevoNodo);
    }
    return lista;
}
```

Lista ==NULL? **FALSO**

Me muevo al siguiente elemento

ENTRO EN RECURSIVIDAD

1º INCOGNITA: Lista->siguiente = ?
(#1234)->siguiente

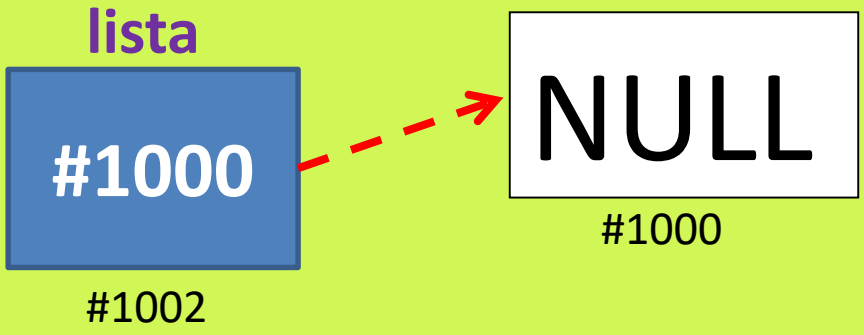
2º Llamada RECURSIVA

#1000

Dato=6
Siguiente = NULL

NuevoNodo
#1001

```
nodo * agregarFinalRecursivo(nodo * lista, nodo * nuevoNodo)
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = agregarFinalRecursivo(lista->siguiente, nuevoNodo);
    }
    return lista;
}
```



2º Llamada RECURSIVA

#1000

Dato=6
Siguiente = NULL

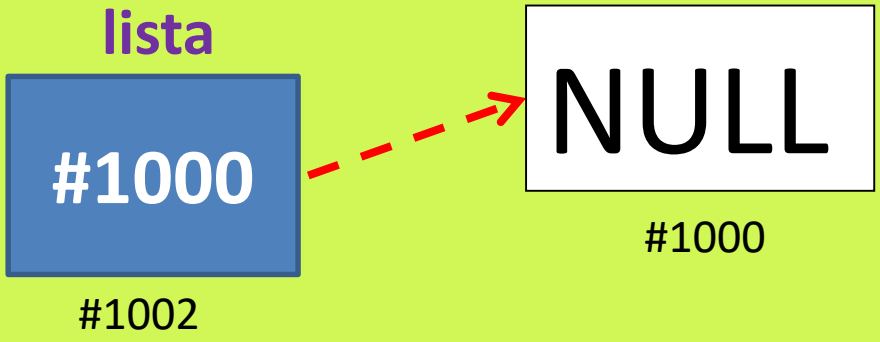
NuevoNodo
#1001

```
nodo * agregarFinalRecursivo(nodo * lista, nodo * nuevoNodo)
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = agregarFinalRecursivo(lista->siguiente, nuevoNodo);
    }
    return lista;
}
```

Lista ==NULL? VERDADERO

(#1000) ==NULL?

Lista SE VUELVE nuevoNodo



Agregar 2º nodo con RECURSIVIDAD

#1000

Dato=6
Siguiente = NULL

NuevoNodo
#1001

```
nodo * agregarFinalRecursivo(nodo * lista, nodo * nuevoNodo)
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = agregarFinalRecursivo(lista->siguiente, nuevoNodo);
    }
    return lista;
}
```

Lista ==NULL? VERDADERO

Lista SE VUELVE nuevoNodo

lista

#1001

#1002

Dato=6
Siguiente = NULL

#1001

NULL

#1000

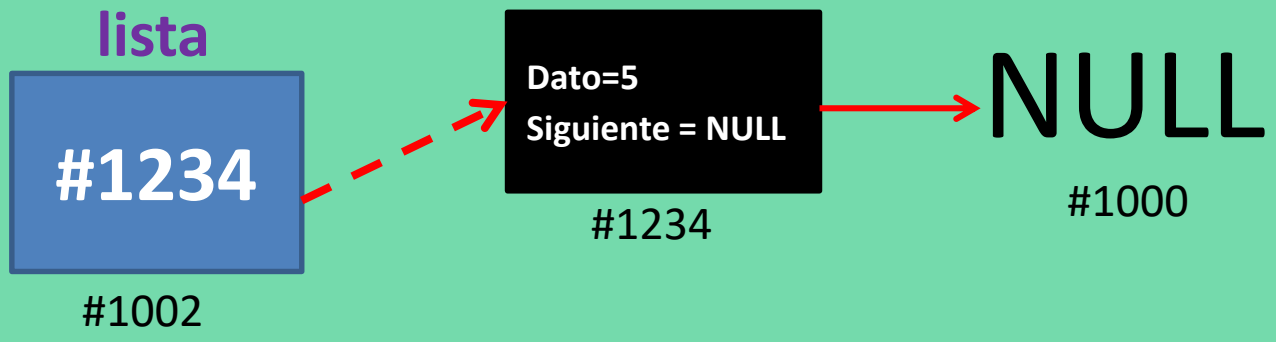
Retorno la lista #1001 a la etapa anterior

VUELVO A LA 1º Etapa Recursiva

Dato=6
Siguiente = NULL

NuevoNodo
#1001

```
nodo * agregarFinalRecursivo(nodo * lista, nodo * nuevoNodo)
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = agregarFinalRecursivo(lista->siguiente, nuevoNodo);
    }
    return lista;
}
```

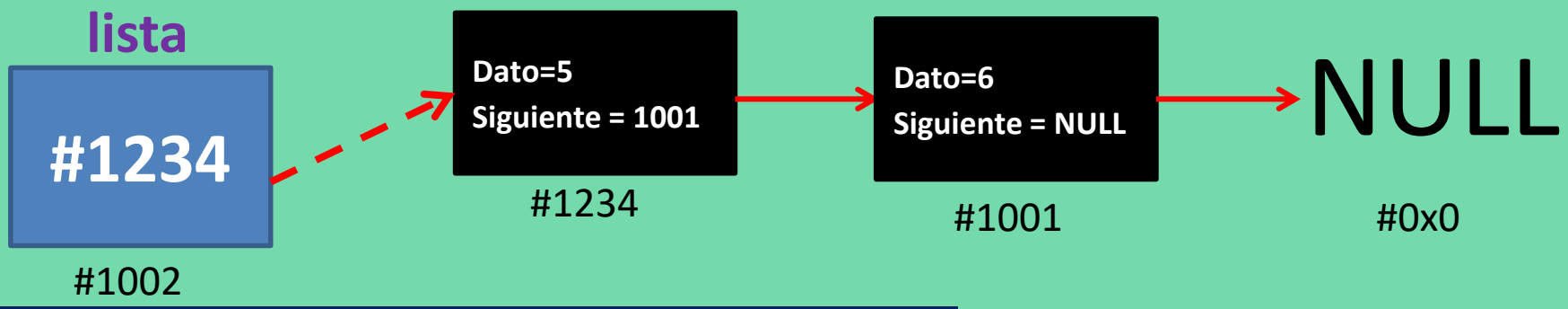


#1001

Dato=6
Siguiente = NULL

NuevoNodo
#1001

```
nodo * agregarFinalRecursivo(nodo * lista, nodo * nuevoNodo)
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = agregarFinalRecursivo(lista->siguiente, nuevoNodo);
    }
    return lista;
}
```



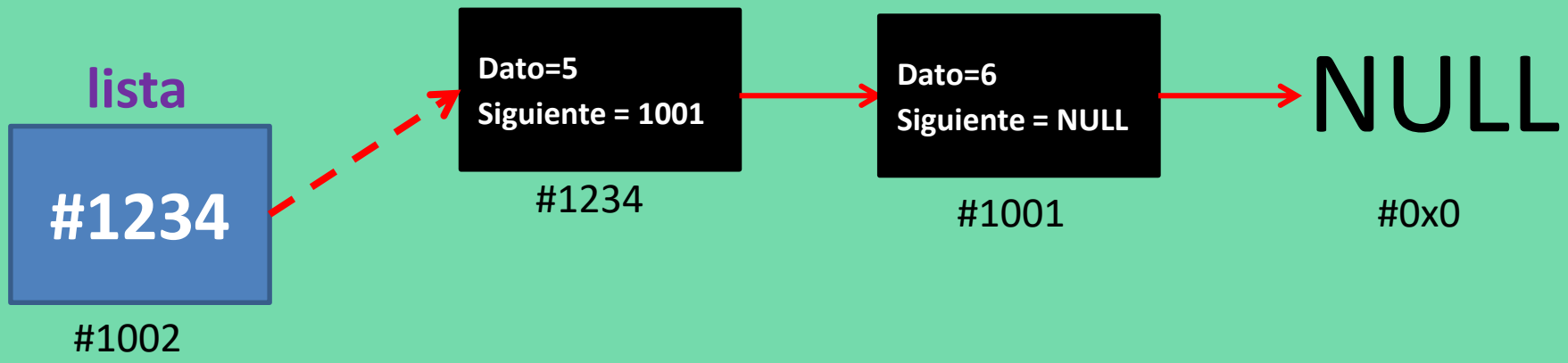
1º INCOGNITA: Lista->siguiente = #1001

(#1234)->siguiente

FINALIZADO

```
nodo * agregarFinalRecursivo(nodo * lista, nodo * nuevoNodo)
{
    if(lista == NULL)
    {
        lista = nuevoNodo;
    }
    else
    {
        lista->siguiente = ag                    do);
    }
    return lista;
}
```

FINALMENTE LA lista QUEDA EN SU POSICIÓN ORIGINAL #1234 QUE ES EL PUNTERO AL INICIO DE LA LISTA



Estructuras de datos dinámicas:

¿Qué es una estructura de datos?

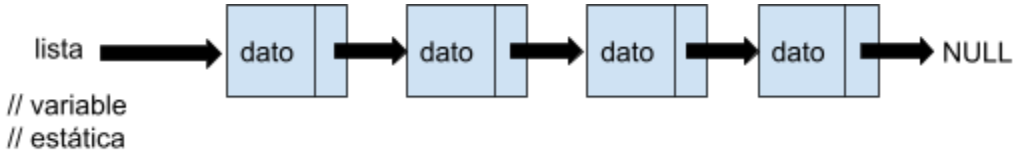
Se trata de un conjunto de variables de un determinado tipo agrupadas y organizadas de alguna manera para representar un comportamiento. Lo que se pretende con las estructuras de datos es facilitar un esquema lógico para manipular los datos en función del problema que haya que tratar y el algoritmo para resolverlo. En algunos casos la dificultad para resolver un problema radica en escoger la estructura de datos adecuada. Y, en general, la elección del algoritmo y de las estructuras de datos que manipulará estarán muy relacionadas.

Según su comportamiento durante la ejecución del programa distinguimos estructuras de datos:

- Estáticas: su tamaño en memoria es fijo. Ejemplo: arrays.
- Dinámicas: su tamaño en memoria es variable. Ejemplo: listas enlazadas con punteros, ficheros, etc.

Listas vinculadas

Conjunto de datos de acceso secuencial.
La lista vinculada es la estructura dinámica más simple, consiste de un conjunto de variables anónimas (creadas con malloc, y que no tienen nombre) que se vinculan unas a otras a través de un puntero.
Gráficamente, una lista (de cualquier tipo de dato) sería algo así:



A cada uno de los elementos de la lista vinculada le vamos a llamar NODO. El acceso al primer nodo de la lista se hace a través de una variable (con nombre) de tipo puntero. El final de la lista se determina con NULL. Cada nodo de la lista contiene campos con información y uno con la dirección de memoria del siguiente nodo. El acceso a cada nodo de la lista se hace en forma secuencial, a través del campo que contiene la dirección del siguiente, así cada nodo, nos proporciona la dirección donde se encuentra el que le sigue. La estructura de datos que se necesita para crear un nodo (y con ello la lista de nodos) es la siguiente:

```
typedef struct {
     tipo campo1;  // campos de datos, pueden ser estructuras
     tipo campo2;
     ........
      tipo campoN;
      struct nodo * siguiente; // campo con la dirección de memoria
del siguiente nodo
} nodo ;

nodo * lista ; // variable estática y con nombre que contiene la
// dirección de memoria del
// primer nodo de la lista. Si la lista está vacía, entonces
// lista = NULL;
```

Operaciones básicas del manejo de listas: Para poder manipular las listas en forma adecuada, lo haremos a través de un conjunto de funciones que se explicarán a continuación.

NOTA: Para ejemplificar la explicación, usamos la siguiente estructura de datos:

```
typedef struct {
     persona dato;
     struct nodo * siguiente;
} nodo;
```

```
typedef struct {
     char nombre[20];
     int edad;
} persona;
```

```
nodo * lista; // variable estática definida en el main()
```

donde:

***dato** es una variable de tipo persona que contiene información.*

***siguiente** es un campo puntero, que contiene la dirección de memoria de otra estructura similar.*

***lista** es una variable estática (la declaramos a nivel de main) que contiene la dirección de memoria del primer nodo de la lista.*

Inicializar la lista

Inicializa el puntero al primer nodo de la lista con el valor NULL.

```
nodo * inicLista() {
   return NULL;
}
```

Crear un nuevo nodo para luego agregar a la lista

Función que recibe como parámetro los campos de información para la estructura, los agrega a la misma y retorna un puntero al NODO creado.

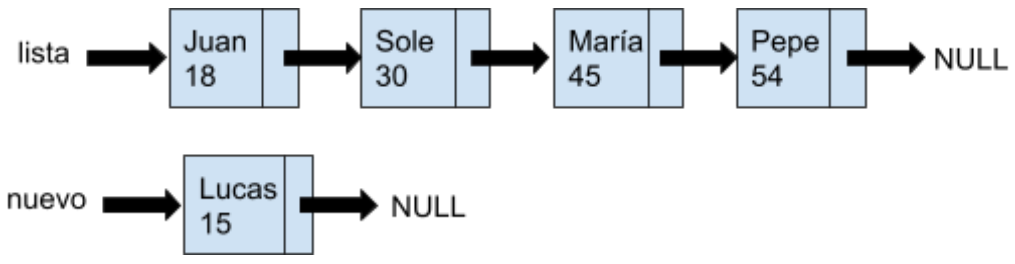
```
nodo * crearNodo (persona dato) {
     //crea un puntero de tipo nodo
     nodo * aux = (nodo*) malloc(sizeof(nodo));
     //asigna valores a sus campos de información
     aux->dato = dato;
     //asigna valor NULL al campo que contiene la dirección de memoria del
     //siguiente nodo
     aux->siguiente = NULL;
     //retorna la dirección de memoria del nuevo nodo, que deberá ser
     asignada a una variable de tipo "puntero a nodo".
     return aux;
}
```

Agregar un nodo ya creado al principio de la lista

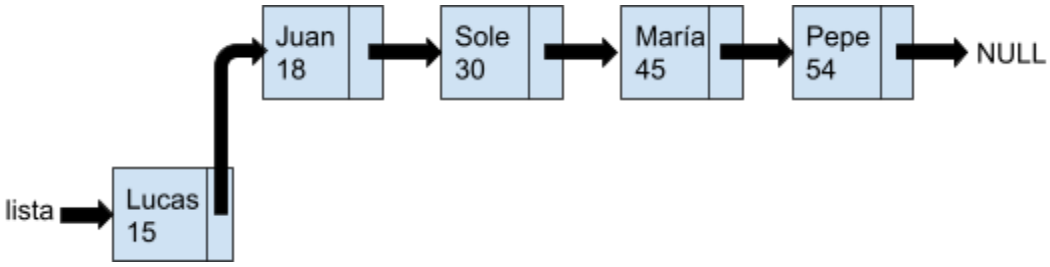
Función que recibe como parámetro una variable puntero al comienzo de la lista y otra variable puntero a un nuevo nodo. Agrega este nuevo nodo al comienzo de la lista y retorna la nueva posición de memoria.

```
 nodo * agregarPpio (nodo * lista, nodo * nuevoNodo) {
     //si la lista está vacía, ahora apuntará al nuevo nodo.
     if(lista == NULL) {
        lista = nuevoNodo;
     }else
     //si la lista no está vacía, inserta el nuevo nodo al comienzo de la
     //misma, y el viejo primer nodo pasa a ser el segundo de la lista.
     {
        nuevoNodo->siguiente = lista;
        lista = nuevoNodo;
     }
     return lista;
 }
```

Antes de la inserción:



Después de la inserción:



NOTA IMPORTANTE: debido al tipo de pasaje de parámetros (por valor), el contenido del parámetro lista (*un puntero al primer nodo*) no puede ser alterado una vez que la función ha terminado, o sea, el parámetro actual siempre conserva su valor. No confundirse, lo que sí se puede alterar es ***lista**.
Lo que hacíamos antes, era cambiar el contenido de una variable apuntada, y no el contenido de la variable apuntadora (que contiene una dirección de memoria).

Buscar el último nodo

Función que nos retorna la dirección de memoria del último nodo de la lista. Esta función solamente se invocará si la lista no está vacía (lista != NULL)

```
nodo * buscarUltimo(nodo * lista) {
   nodo * seg = lista;
   if(seg != NULL)
      while(seg->siguiente != NULL) {
         seg = seg->siguiente;
      }
    return seg;
 }
```

donde:

*la variable local **seg** contiene al principio, la dirección de memoria de primer nodo de la lista. Luego, al ir ejecutándose la instrucción **seg = seg->siguiente;** dentro del ciclo de repetición, **seg** va tomando la dirección del nodo siguiente. De esta forma se avanza en el recorrido secuencial de la lista.*

Utilizamos una variable local **seg** para recorrer la lista en vez del mismo parámetro **lista** para no alterar el contenido del mismo, aunque, debido al tipo de pasaje de parámetros (por valor) el contenido de **lista** no podría ser alterado nunca. Sí se puede alterar el contenido de ***lista.** Hacerlo de esta forma es una buena disciplina de programación que evita posibles errores.

Notar que en el ciclo de repetición, la condición es **(seg->siguiente != NULL)**, esto es porque debemos detener nuestro avance al llegar al último nodo, o sea, al nodo cuyo siguiente es NULL. Por este motivo evaluamos al siguiente y no al propio nodo.

Buscar un nodo según el valor de un campo

Función que retorna la dirección de memoria de un nodo que contiene el campo **dato.nombre** con el mismo valor que el parámetro **nombre**

```
nodo * buscarNodo(nodo * lista, char nombre[20]) {
   //busca un nodo por nombre y retorna su posición de memoria
   //si no lo encuentra retorna NULL.

   nodo * seg;  //apunta al nodo de la lista que está siendo procesado
   seg = lista; //con esto evito cambiar el valor de la variable
                //lista, que contiene un puntero al primer nodo de la
                //lista vinculada

   while ((seg != NULL) && ( strcmp(nombre, seg->dato.nombre)!=0 )) {
//busco mientras me quede lista por recorrer y no haya encontrado el nombre
      seg=seg->siguiente; //avanzo hacia el siguiente nodo.
   }
//en este punto puede haber fallado alguna de las dos condiciones
//del while. si falla la primera es debido a que no encontró lo
//que buscaba (seg es NULL), si falla la segunda es debido a que se
//encontró el nodo buscado.
   return seg;
}
```

Agregar un nuevo nodo al final de la lista

Esta función nos permite agregar al final de la lista un nuevo nodo.

```
nodo * agregarFinal(nodo * lista, nodo * nuevoNodo) {

   if(lista == NULL) {
      lista = nuevoNodo;
   } else {
      nodo * ultimo = buscarUltimo(lista);
      ultimo->siguiente = nuevoNodo;
   }
   return lista;
}
```

Notar que se invoca a la función **buscarUltimo(nodo *)** ya definida previamente.
La variable local **ultimo** contiene la dirección de memoria del último nodo de la lista, y a

través de la misma, accedo a su campo **siguiente**, actualizando su valor. Esto agrega un nodo más a la lista.
El retorno de la función siempre es el mismo, excepto la primera vez, cuando la lista está vacía y se le agrega el primer elemento.

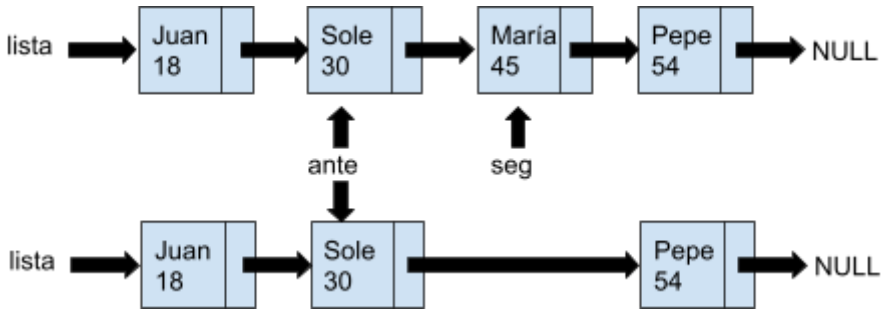
Borrar un nodo de la lista buscándolo por el valor de un campo

Función que nos permite borrar un nodo que está en el interior de la lista, buscándolo por el valor de uno de sus campos. Retorna un puntero al comienzo de la lista.
Generalmente el puntero al comienzo de la lista no se modifica, excepto cuando se elimina el primer nodo.

```
nodo * borrarNodo(nodo * lista, char nombre[20]) {
   nodo * seg;
   nodo * ante; //apunta al nodo anterior que seg.
   if((lista != NULL) && (strcmp(nombre, lista->dato.nombre)==0 )) {

      nodo * aux = lista;
      lista = lista->siguiente; //salteo el primer nodo.
      free(aux);                //elimino el primer nodo.
   }else {
      seg = lista;
      while((seg != NULL) && (strcmp(nombre, seg->dato.nombre)!=0 )) {
         ante = seg;           //adelanto una posición la variable ante.
         seg = seg->siguiente; //avanzo al siguiente nodo.
      }
      //en este punto tengo en la variable ante la dirección de
      //memoria del nodo anterior al buscado, y en la variable seg,
      //la dirección de memoria del nodo que quiero borrar.
      if(seg!=NULL) {
         ante->siguiente = seg->siguiente;
         //salteo el nodo que quiero eliminar.
         free(seg);
         //elimino el nodo.
      }
   }
   return lista;
}
```

ejemplo: eliminar el nodo “María”



Agregar un nodo nuevo manteniendo el orden (según un campo)

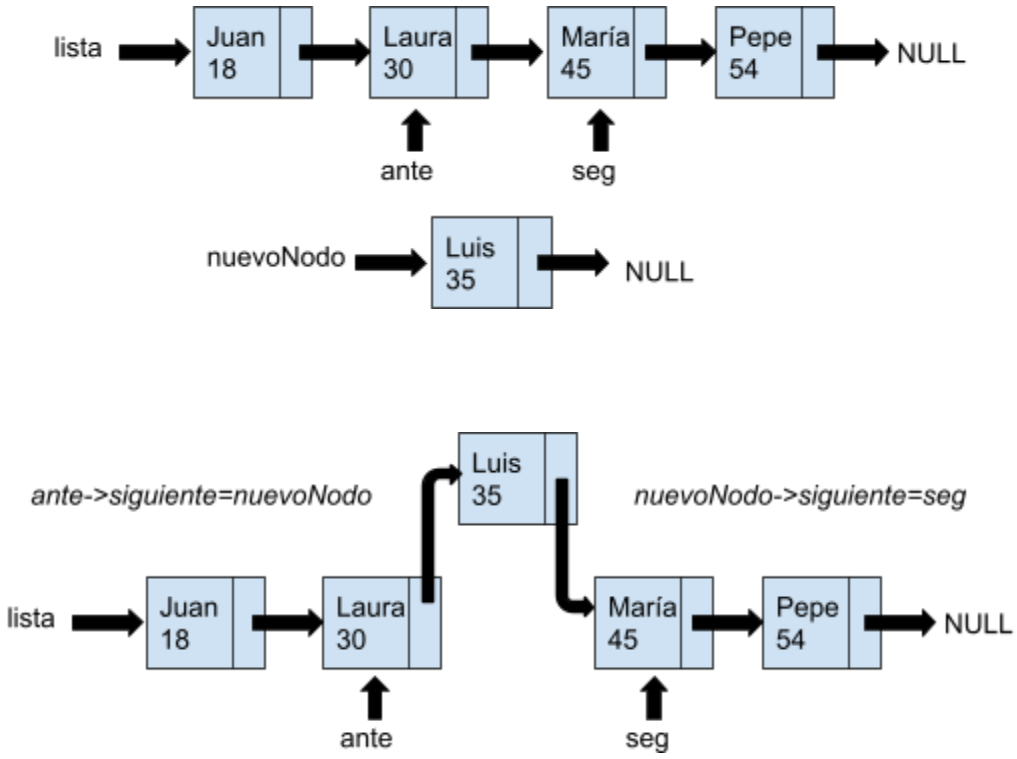
Esta función agrega un nodo nuevo a la lista, insertándolo en el lugar correspondiente según un orden preestablecido por un campo. Por ejemplo: insertar un nuevo nodo en la lista ordenada por el campo nombre. Retorna un puntero al primer nodo de la lista, que se modifica solamente cuando el nuevo nodo se inserta en la primera posición.

```
nodo * agregarEnOrden(nodo * lista, nodo * nuevoNodo) {
   // agrega un nuevo nodo a la lista manteniendo el orden.

   //si la lista está vacía agrego el primer elemento.
   if(lista == NULL) {
      lista = nuevoNodo;
   }else {
      //si el nuevo elemento es menor que el primero de la lista,
      //agrego al principio

      if(strcmp(nuevoNodo->dato.nombre,lista->dato.nombre)<0){
         lista = agregarPpio(lista, nuevoNodo);
      } else {
         //busco el lugar en donde insertar el nuevo elemento.
         //necesito mantener la dirección de memoria del nodo anterior
         //al nodo que tiene un nombre mayor al del nuevo nodo.
         nodo * ante = lista;
         nodo * seg = lista->siguiente;
         while((seg != NULL)
                  &&(strcmp(nuevoNodo->dato.nombre,seg->dato.nombre)>0)) {
            ante = seg;
            seg = seg->siguiente;
         }
         // inserto el nuevo nodo en el lugar indicado.
         nuevoNodo->siguiente = seg;
         ante->siguiente = nuevoNodo;
      }
   }
   return lista;
}
```

ejemplo: agregar el nodo “Luis”



Borrar toda la lista y liberar la memoria ocupada

Esta función borra toda la lista entera y libera todas las direcciones de memoria ocupada por sus nodos, retornando NULL.

```
nodo * borrarTodaLaLista(nodo * lista) {
   nodo * proximo;
   nodo * seg;
   seg = lista;
   while(seg != NULL) {
      proximo = seg->siguiente;  //tomo la dir del siguiente.
      free(seg);                 //borro el actual.
      seg = proximo;             //actualizo el actual con la dir del
                                 //siguiente, para avanzar.
   }
   return seg; // retorna NULL a la variable lista del main()
}
```

Sumar el contenido de cada nodo (el campo edad)

```
int sumarEdadesLista(nodo * lista) {
//recorro la lista y sumo las edades de los socios.
   int suma = 0;
   nodo * seg = lista;
   while (seg != NULL) {
      suma = suma + seg->dato.edad;
      seg = seg->siguiente;
   }
   return suma;
}
```

Eliminar el primer nodo de una lista

Queda como ejercicio para el alumno.

Eliminar el último nodo de una lista

Queda como ejercicio para el alumno.

Otras funciones

Para armar un sistema, usaremos también las siguientes funciones, que usan a las funciones anteriores.

```
void mostrarUnNodo(nodo * aux);
void recorrerYmostrar(nodo * lista);
nodo * subprogramaIngresarDatosAlFinal(nodo * lista);
nodo * subprogramaingresarDatosAlPpio(nodo * lista);
nodo * subprogramaAgregarUnNodoEnOrden(nodo * lista);
nodo * subprogramaCrearListaOrdenada(nodo * lista);
void subprogramaBusquedaDeUnNodo(nodo * lista);
nodo * subprogramaBorrarNodo(nodo * lista);
void menu();
```

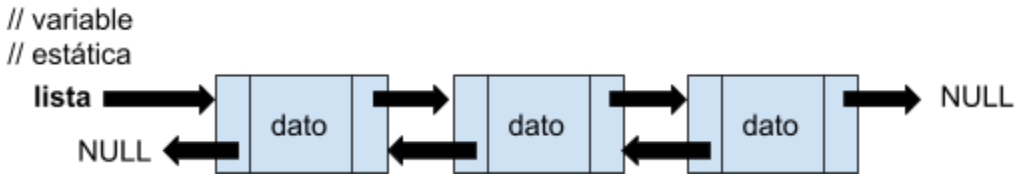
Estas funciones son muy simples, por lo cual no se dará una explicación detallada. Se recomienda analizar el código de las mismas.

Listas doblemente vinculadas

Estructura de datos que consiste en un conjunto de nodos enlazados secuencialmente. Cada nodo contiene dos campos, llamados enlaces, que son referencias al nodo siguiente y al anterior en la secuencia de nodos. El enlace al nodo anterior del primer nodo y el enlace al nodo siguiente del último nodo, apuntan a un tipo de nodo que marca el final de la lista, normalmente un puntero null, para facilitar el recorrido de la lista.

El doble enlace de los nodos permite recorrer la lista en cualquier dirección. Mientras que agregar o eliminar un nodo en una lista doblemente enlazada requiere cambiar más enlaces que en estas mismas operaciones en una lista simplemente enlazada, las operaciones son más simples porque no hay necesidad de mantener guardado el nodo anterior durante el recorrido, ni necesidad de recorrer la lista para hallar el nodo anterior, la referencia al nodo que se quiere eliminar o insertar es lo único necesario.

Gráficamente, una lista doblemente enlazada (de cualquier tipo de dato) sería algo así:



Si se administra de una forma especial (con una estructura), el primer y el último nodo de una lista doblemente enlazada son accesibles inmediatamente (o sea, accesibles sin necesidad de recorrer la lista, y usualmente llamados cabeza y cola) y esto permite recorrerla desde el inicio o el final de la lista, respectivamente. Cualquier nodo de la lista doblemente enlazada, una vez obtenido, puede ser usado para empezar un nuevo recorrido de la lista, en cualquier dirección (hacia el principio o el final), desde el nodo dado.

Los enlaces que contiene un nodo de este tipo de lista son frecuentemente llamados anterior y siguiente o antecesor y sucesor.

La estructura de datos que se necesita para crear un nodo (y con ello la lista de nodos) es la siguiente:

```
typedef struct {
     tipo campo1;  // campos de datos, pueden ser estructuras
     tipo campo2;
     ........
     tipo campoN;
     struct nodoDoble * anterior; // dir de memoria nodo anterior
     struct nodoDoble * siguiente; // dir de memoria nodo siguiente
} nodoDoble ;
```

```
nodoDoble * listaDoble ; // variable estática y con nombre que
contiene la    // dirección de memoria del
// primer nodo de la lista. Si la lista está vacía, entonces

listaDoble = NULL;
```

Operaciones básicas del manejo de listas doblemente enlazada

Para el funcionamiento de esta colección de datos, debemos codificar las mismas funciones desarrolladas para la Lista Simple, adaptando el código al tratamiento de los dos enlaces que posee cada nodo, teniendo en cuenta las distintas posibilidades en cuanto a la posición del mismo (inicio de la lista, medio o final).

NOTA: Para ejemplificar la explicación, usamos la siguiente estructura de datos:

```
typedef struct {
     persona dato;
     struct nodoDoble * anterior;
     struct nodoDoble * siguiente;
} nodoDoble;
```

```
typedef struct {
     char nombre[20];
     int edad;
} persona;
```

```
nodoDoble * listaDoble; // variable estática definida en el main()
```

donde:

***dato** es una variable de tipo persona que contiene información.*

***anterior** es un campo puntero, que contiene la dirección de memoria de otra estructura similar, anterior a la actual. Si es el nodo inicial, esta apunta a NULL.*

***siguiente** es un campo puntero, que contiene la dirección de memoria de otra estructura similar, siguiente a la actual. Si es el nodo final, esta apunta a NULL.*

***listaDoble** es una variable estática (la declaramos a nivel de main) que contiene la dirección de memoria del primer nodo de la lista.*

A modo de ejemplo, a continuación se desarrollan algunas de las mencionadas funciones, es tarea del alumno completar el resto.

Inicializar la lista

Inicializa el puntero al primer nodo de la lista con el valor NULL.

```
nodoDoble * inicListaDoble() {
   return NULL;
}
```

Crear un nuevo nodo para luego agregar a la lista

Función que recibe como parámetro los campos de información para la estructura, los agrega a la misma y retorna un puntero al NODO creado.

```
nodoDoble * crearNodoDoble (persona dato) {

    nodoDoble* aux = (nodoDoble*) malloc(sizeof(nodoDoble));
    aux->dato = dato;
    //asigna valor NULL a los campos que contienen la dirección de memoria
    //de los nodos anterior y siguiente
    aux->anterior = NULL;
    aux->siguiente = NULL;

    return aux;
}
```

Agregar un nodo ya creado al principio de la lista

Función que recibe como parámetro una variable puntero al comienzo de la lista y otra variable puntero a un nuevo nodo. Agrega este nuevo nodo al comienzo de la lista y retorna la nueva posición de memoria.

```
nodoDoble * agregarPpioDoble (nodoDoble * lista, nodoDoble * nuevo) {
    nuevo->siguiente = lista;
    if(lista != NULL)
       lista->anterior=nuevo;
    return nuevo;
}
```

Buscar el último nodo (recursiva)

Función que nos retorna la dirección de memoria del último nodo de la lista. Se comparte este ejemplo recursivo, pensar cómo hacerlo de forma iterativa (es similar a la lista simple)

```
nodoDoble * buscarUltimoDobleRecursivo (nodoDoble  * lista) {
   nodoDoble * rta;
   if (lista==NULL)
      rta=NULL;
   else
      if(lista->siguiente==NULL)
         rta=lista;
      else
         rta=buscarUltimoDobleRecursivo(lista->siguiente);
   return rta;
 }
```

Agregar un nuevo nodo al final de la lista

Esta función nos permite agregar al final de la lista un nuevo nodo.

```
nodoDoble * agregarFinalDoble(nodoDoble  * lista, nodoDoble * nuevo) {

   if(lista == NULL) {
      lista = nuevo;
   } else {
      nodoDoble * ultimo = buscarUltimo(lista);
      ultimo->siguiente = nuevo;
      nuevo->ante = ultimo;
   }
   return lista;
}
```

Agregar un nodo nuevo manteniendo el orden (según un campo)

Esta función agrega un nodo nuevo a la lista, insertándolo en el lugar correspondiente según un orden preestablecido por un campo. Por ejemplo: insertar un nuevo nodo en la lista ordenada por el campo nombre. Retorna un puntero al primer nodo de la lista, que se modifica solamente cuando el nuevo nodo se inserta en la primera posición.

```
nodoDoble * agregarEnOrdenDoble (nodoDoble  * lista, nodoDoble * nuevo) {

   if(lista == NULL) {
      lista = nuevo;
   }else {

      if(strcmp(nuevo->dato.nombre,lista->dato.nombre)<0){
         lista = agregarPpioDoble(lista, nuevo);
```

```
        } else {
           // se puede recorrer la lista utilizando un solo puntero??
           nodoDoble * ante = lista;
           nodoDoble * seg = lista->siguiente;
           while((seg != NULL)
                    &&(strcmp(nuevo->dato.nombre,seg->dato.nombre)>0)) {
              ante = seg;
              seg = seg->siguiente;
           }
           ante->siguiente = nuevo;
           nuevo->anterior = ante;
           nuevo->siguiente = seg;
           if (seg!=NULL)
               seg->ante=nuevo;
        }
     }
     return lista;
 }
```

Las siguientes funciones, completan el TDA Lista doblemente vinculada, quedan como ejercicio para el alumno:

- Borrar un nodo de la lista buscándolo por el valor de un campo (se puede recorre la lista utilizando un solo puntero)
- Buscar un nodo
- Eliminar el primer nodo de una lista
- Eliminar el último nodo de una lista
- Mostrar un nodo
- Recorrer y mostrar la lista (usar la función anterior)

Subprogramas:

- Menú del sistema
- Cargar lista al principio
- Cargar lista al final
- Cargar lista en orden