(II) Repaso Teoría C

Indice

- Memoria Local
  - Ventajas e Inconvenientes
  - El error debido a ‘&’
  - Paso de parámetros por referencia

- Listas Enlazadas
  - Función Push()

Ejemplo 1

```
int Square(int num) {
    int result;

    result = num * num;

    return result;
}
```

- Las variables locales están *ligadas* a la función donde están declaradas. Su existencia está ligada a la función.

- Cuando llamamos la función Square(), se reserva un espacio para las variables num y result. Una vez termina la ejecución, las variables desaparecen de la pila de ejecución del programa.

Ejemplo 2

```
void X(){
  int a=1;
  int b=2;
  //T1
  Y(a);
  //T3
  Y(b);
  //T5
}
```

```
void Y(int p){
  int q;
  q=p+2;
  //T2 y T4
}
```

Ejemplo 2

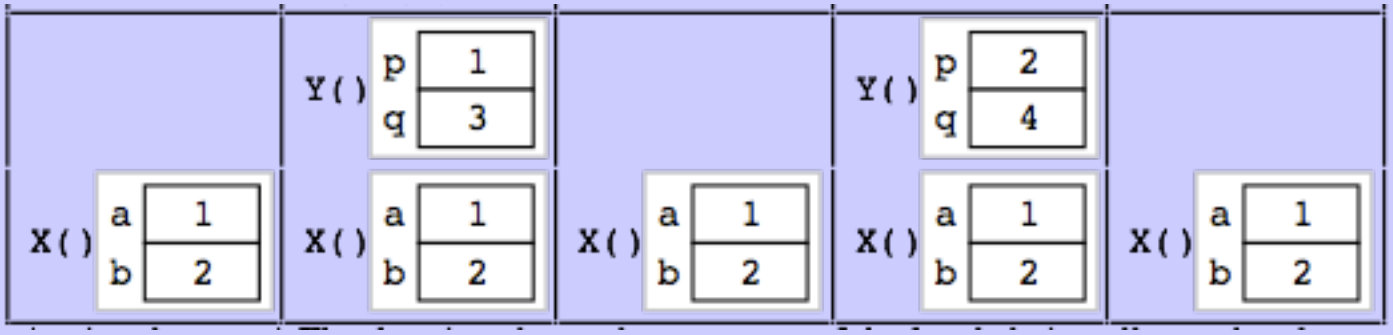
T1 - variables locales a X() son creadas y asignados valores

T2 - llamamos Y() con p=1, y creamos sus variables locales. Las v.a. locales de X() siguen asignadas

T3 - las variables locales de Y() son eliminadas de la pila. Sólo tenemos las variables de X().

T4 - llamamos Y() con p=2, y creamos sus variables locales por segunda vez. Las variables locales de X() siguen asignadas

T5 - las variables locales de Y() son eliminadas de la pila. Sólo tenemos las variables de X().



Ventajas de Memoria Local

- *Conveniente*: las funciones necesitan algo de memoria temporal que exista sólo mientras se ejecutan.
- *Eficiente*: respecto a otras técnicas de uso de memoria- utilizan y reciclan memoria.
- *Copias locales*: los parámetros pasados son **copias** de la información de la función llamante. La separación entre la función llamante y llamada es buena.

Desventajas de Memoria Local

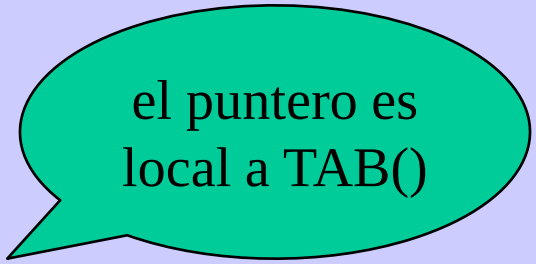
- *Vida de la variable*: algunas veces el programa necesita memoria que continúe accesible, incluso después de que la función que originalmente la ha creado haya concluido su ejecución. La solución-memoria dinámica.
- *Comunicación restringida*: puesto que los parámetros pasados son copias, *no existe comunicación de vuelta* entre la función llamada y la llamante. La solución - *paso de parámetros por referencia*.

El error debido a ‘&’

- ¿Podría decir qué funciona mal en el siguiente código, donde la función Victima() llama a la función TAB()?

El error debido a ‘&’

```
//la función TAB devuelve un puntero a un entero
int* TAB() {
  int temp;
  return(&temp); //devuelve un puntero a una
                 //v.a.entera local
}

void Victima() {
  int* ptr;
  ptr = TAB();
  *ptr = 42; // ¡error de ejecución!
}
```



El error debido a ‘&’

- El problema es que el entero local sólo existe mientras se ejecuta TAB().
- Querríamos que la variable entera que crea TAB() exista después de su ejecución.

- No podemos utilizar ‘&’ para pasar un puntero de vuelta a la función llamante

Llamando a la función foo(6, x+1)

1. Los parámetros se evalúan en el contexto de la función llamante (tales como x+1).
2. Reservamos memoria (un bloque local de memoria) para las variables locales. Los parámetros tb. van en el bloque local.
3. Guarda la dirección actual del llamante (la dirección de vuelta) y conmuta a ejecutar foo().

Llamando a la función foo(6, x+1)

4. foo() se ejecuta convenientemente con su bloque de memoria situado al final del stack de ejecución.
5. Al finalizar foo(), libera haciendo *pop* de las v.a. locales del stack, y vuelve a la dirección almacenada desde la función llamante. Ahora, al final del stack están las variables locales del llamante, que puede finalizar su ejecución.

Notas sobre la memoria local

- Ejecución continuada de (1), (2) y (3), (1), (2) y (3), y nunca (4), provoca “Stack Overflow Error”.
- (2) sitúa el bloque local de memoria, pero el contenido inicial es aleatorio.
- Un sólo ciclo de CPU es suficiente para situar el bloque local de memoria de foo().
- En entorno *multi-hilo*, cada *hilo* tiene su propio stack de memoria para su ejecución.

Paso de parámetros por referencia

- La utilización de return() desde la función llamada a la llamante no siempre es posible.
- El paso de parámetros por referencia soluciona este problema.
- El "*valor de interés*" es el valor que desean comunicarse la función llamada y la llamante entre sí. El parámetro por referencia pasa un puntero al valor de interés, en lugar de hacer una copia de él

Ejemplo- por valor

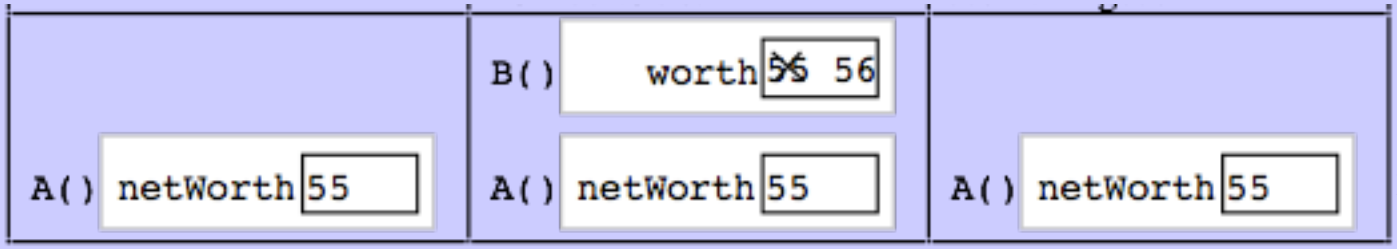
```
void B(int worth) {
   worth = worth + 1;
   // T2
}
void A() {
   int netWorth;
   netWorth = 55; // T1

   B(netWorth);
   // T3 -- B()
}
```

no cambió netWorth

T1- el valor de interés netWorth es local a A()

T2- netWorth es copiado a B(), worth. B() cambia worth de 55 a 56.

T3- B() termina y el valor de interés netWorth no cambia.



Ejemplo- por referencia

/*B() obtiene el valor del puntero que
le pasan como parámetro mediante '*'
*/

```
void B(int* worthRef) {
   *worthRef = *worthRef + 1;
   // T2
}

 void A() {
    int netWorth;
    netWorth = 55;

    B(&netWorth);

    // T3 -- B()
 }
```

//**T1**-el valor de interés es local a A()

//pasa el puntero al valor de interés
// en este caso con '&'

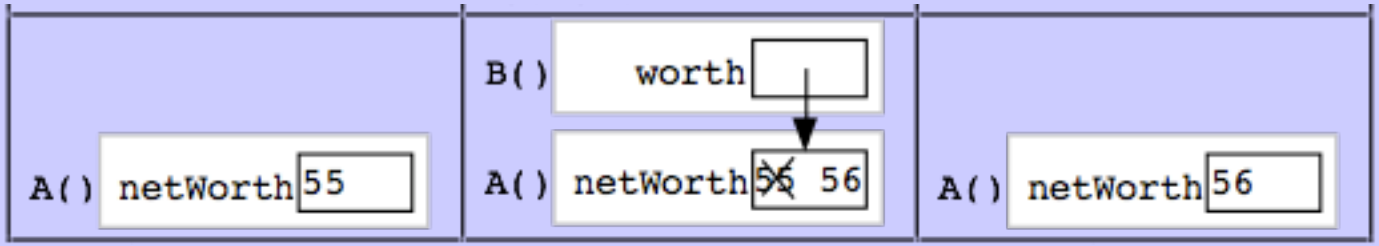
ha usado su puntero para cambiar el valor de interés

Ejemplo- por referencia

T1- el valor de interés netWorth es local a A()

T2- En lugar de una copia, B() recibe un puntero a netWorth. B() de-referencia y cambia el valor real de netWorth.

T3- B() termina y el valor de interés netWorth cambia.



Pasando por referencia

- Pasos a seguir en el código para realizar paso por referencia:
  - utilizar una copia, *master*, del valor de interés.
  - pasar punteros a ese valor a cualquier función que quiera ver o manipular ese valor.
  - las funciones pueden *de-referenciar* (con el operador‘*’) su puntero para cambiar el valor de interés.
  - las funciones no tienen su propia copia del valor; al cambiarlo, están cambiando el valor único.

17

No hacer copias

- es mejor no hacer copias,
  - Si el valor que deseamos pasar como parámetro es grande, p.ej. un array.
  - desde el punto de vista de diseño, manejar dos copias no es deseable, porque, tras algunas manipulaciones, podemos no saber cual es la copia *correcta*.

Ejemplo: Swap

```
void Swap(int* a, int* b) {
   int temp;

   temp = *a;
   *a = *b;
   *b = temp;
}

void SwapCaller2() {
   int scores[10];
   scores[0] = 1;
   scores[9[ = 2;
   Swap(&(scores[0]), &(scores[9]));
 }
```

¿otra forma de llamar a Swap()?

Swap(scores, scores + 9);

¿Es siempre necesario ‘&’?

- No siempre es necesario.
  - por ejemplo, cuando la función llamante ya trabaja con un puntero del valor de interés, podemos pasar como parámetro el puntero.

```
void C(int* worthRef) {
   *worthRef = *worthRef + 2;
}
void B(int* worthRef) {
   *worthRef = *worthRef + 1;

   C(worthRef);


}
```

¿Qué sucedía con este ejemplo?

```
//la función TAB devuelve un puntero a un entero
int* TAB() {
  int temp;
  return(&temp); //devuelve un puntero a una
                 //v.a.entera local
}

void Victima() {
  int* ptr;
  ptr = TAB();
  *ptr = 42; // ¡error de ejecución!
}
```

No siempre es válido pasar punteros con ‘&’

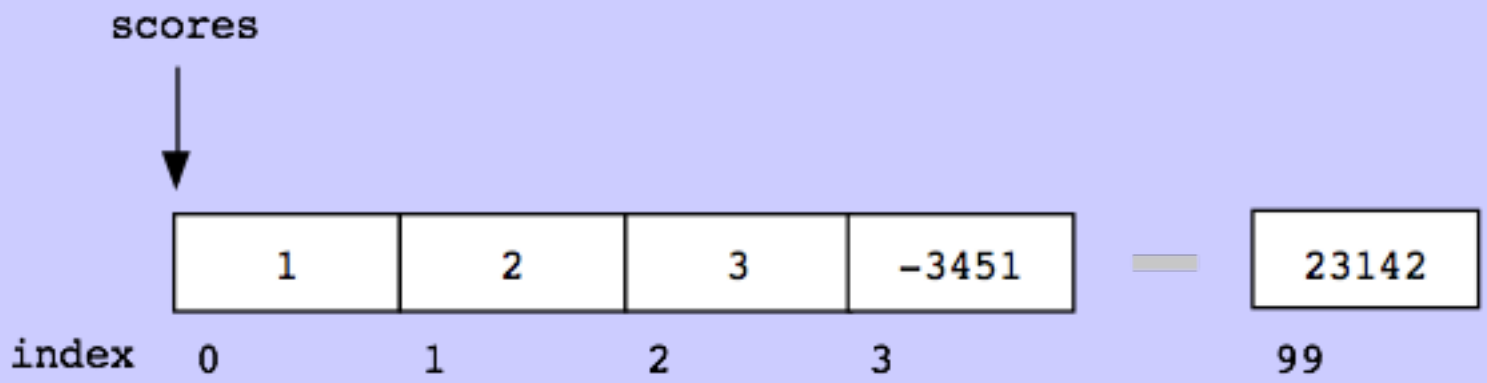
- La función llamada devuelve la dirección de un valor mediante el operador ‘&’, pero al devolver el control a la función llamante, la variable ya no está en la pila de ejecución.
- Al contrario, utilizar ‘&’ para pasar un puntero local de la función llamante a la llamada, es correcto, ya que las variables locales aún existen cuando se ejecuta la función llamada

¿y si el valor de interés es un puntero?

- Por ejemplo, int*, o struct fraction*
- ¿cambian las reglas de paso de parámetros por referencia? NO
- El parámetro por referencia es aún un puntero al valor de interés, que es un puntero.
- Por ejemplo, si el valor de interés es int*, el parámetro por referencia será int**.

Listas enlazadas: arrays

```
void ArrayTest() {
   int scores[100];
   //opera sobre los elementos del array scores...
   scores[0] = 1;
   scores[1] = 2;
   scores[2] = 3;
}
```



Listas enlazadas: arrays

- Las desventajas de los arrays son:
  1. Su tamaño es fijo; especificado en el momento de compilar. También podemos determinar el tamaño en tiempo de ejecución (memoria dinámica). Cambiar su tamaño requiere un esfuerzo extra (p.ej. utilizar realloc()).
  2. Por lo anterior, solemos especificar tamaños grandes de arrays en previsión del tamaño de lo que queremos almacenar. Con esto desperdiciamos memoria.

Listas enlazadas (LE)

- Las LEs complementan las debilidades de los arrays, es decir,
  - Mientras los arrays reservan memoria para todos sus elementos en un bloque de memoria,
  - las LEs reservan memoria para cada elemento de forma independiente o separada, y, sólo cuando sea necesario.
  - Cada elemento de la lista enlazada tiene su propia reserva de memoria, y se llama "nodo" o "elemento de lista enlazada"

Listas Enlazadas

- La estructura de LE la adquiere mediante punteros que conectan sus nodos, como enlaces de una cadena.
- Cada nodo contiene dos campos:
  - "data" donde almacenar cualquier tipo de elemento.
  - "next" que es un puntero encargado de enlazar un nodo con el siguiente nodo.
  - Cada nodo tiene una zona reservada de memoria dinámica (malloc()). Por lo tanto, la memoria reservada al nodo continua existiendo hasta que hacemos free().
  - También existe un puntero al primer elemento de la lista.

Listas Enlazadas

Esquema de la lista {1, 2, 3}

Stack

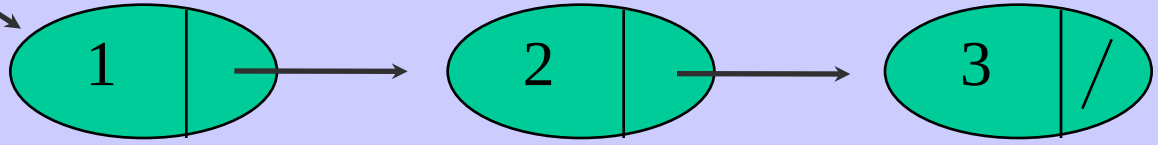
BuildOneTwoThree()

head

El puntero “head”, local a BuildOneTwoThree() mantiene toda la lista, almacenando un puntero al rpimer nodo

Memoria Dinámica

Toda la lista está construida conectando unos nodos con otros mediante el puntero “next”. Todos los nodos están en memoria dinámica



Cada nodo almacena un elemento “data” (un entero en este ejemplo)

Cada nodo almacena un puntero “next”

El campo next del últiimo nodo es NULL

Listas Enlazadas

- El coste de acceder a un elemento de la lista depende *linealmente* de la posición que ocupe dentro de la lista.
  - Operaciones con los *primeros* elementos son menos costosas que con otros nodos posteriores
- En un array [] el coste de acceso es constante
- En una lista vacía, el puntero *head* apunta a NULL.
  - el programador debe considerar este caso límite en su código.

Listas Enlazadas

- Antes de escribir código para construir una lista enlazada, necesitamos dos tipos de datos:
  - *Nodo*: cada elemento que forma la lista; está en memoria dinámica. Cada nodo contiene los datos y un puntero al siguiente nodo de la lista.

```
struct node {
   int             data;
   struct node*    next;
};
```

  - *Puntero a nodo*: El tipo de los punteros a los nodos, tanto del puntero *head*, como del resto dentro de cada nodo.

/*

Construye una lista {1,2,3} en memoria dinámica, y guarda el puntero head en el stack, como variable local. Devuelve el puntero head

*/

```
struct node* BuildOneTwoThree() {
   struct node* head = NULL;
   struct node* second = NULL;
   struct node* third = NULL;

   head = malloc(sizeof(struct node));  //guarda 3 nodos en mem. dinámica
   second = malloc(sizeof(struct node));
   third = malloc(sizeof(struct node));

   head->data = 1;     //ponemos el primer nodo
   head->next = second; //regla de asignación de punteros

   second->data = 2;   //ponemos el segundo nodo
   second->next = third;

   third->data = 3;    //ponemos el tercer enlace
   third->next = NULL;

   //En este punto la lista está referenciada por “head”

   return head;
}
```

Cálculo de la longitud de una LE

/*

Dado un puntero “head” de lista enlazada, calcula y devuelve la cantidad de nodos de la lista.

*/

```
int Length(struct node* head) {
   struct node* current = head;
   int count = 0;

   while (current != NULL) {
      count++;
      current = current->next;
   }

   return count;
}
```

Cálculo de la longitud de una LE

1. Pasamos la lista pasando el puntero “head”. El puntero es copiado en local, y con esto no se duplica la lista enlazada.

2. Iteramos sobre la lista con un puntero *local*

   - El puntero local current comienza apuntando a head. Cuando la función termina, current es *eliminada* ya que es una variable local, pero los nodos en memoria dinámica permanecen.

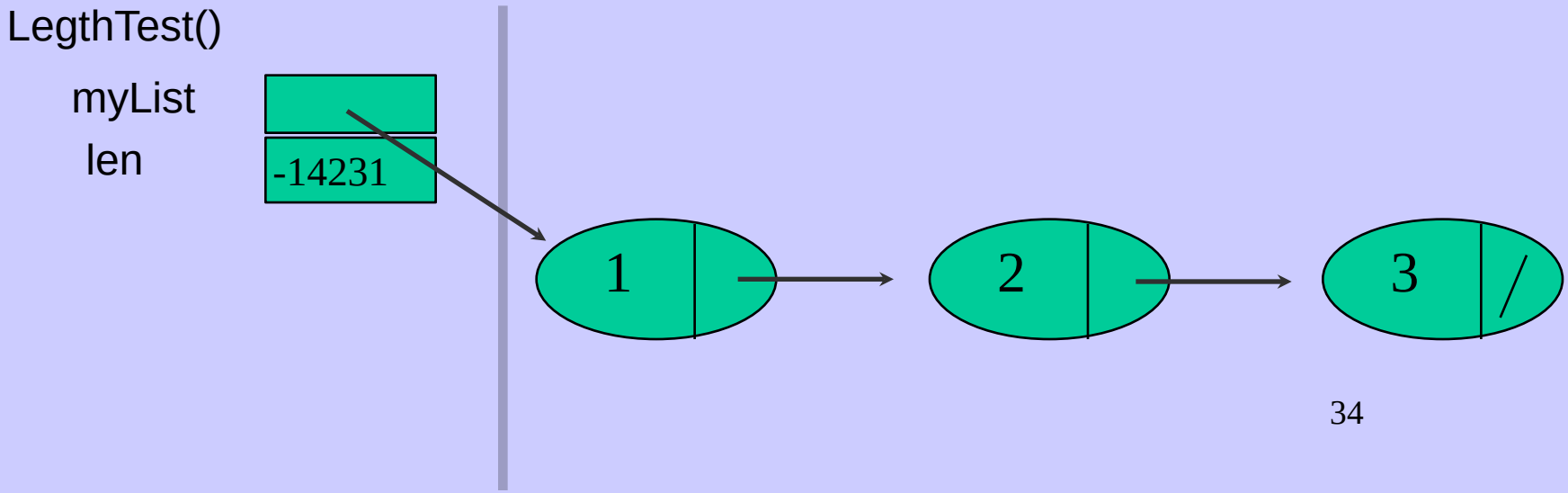
   - (current!=head) comprueba el caso de la lista vacía.

   - current->next; avanza el puntero local al siguiente nodo de la lista. Cuando no hay más nodos, el puntero queda a NULL. Olvidar este punto puede ser causa de bucles infinitos.

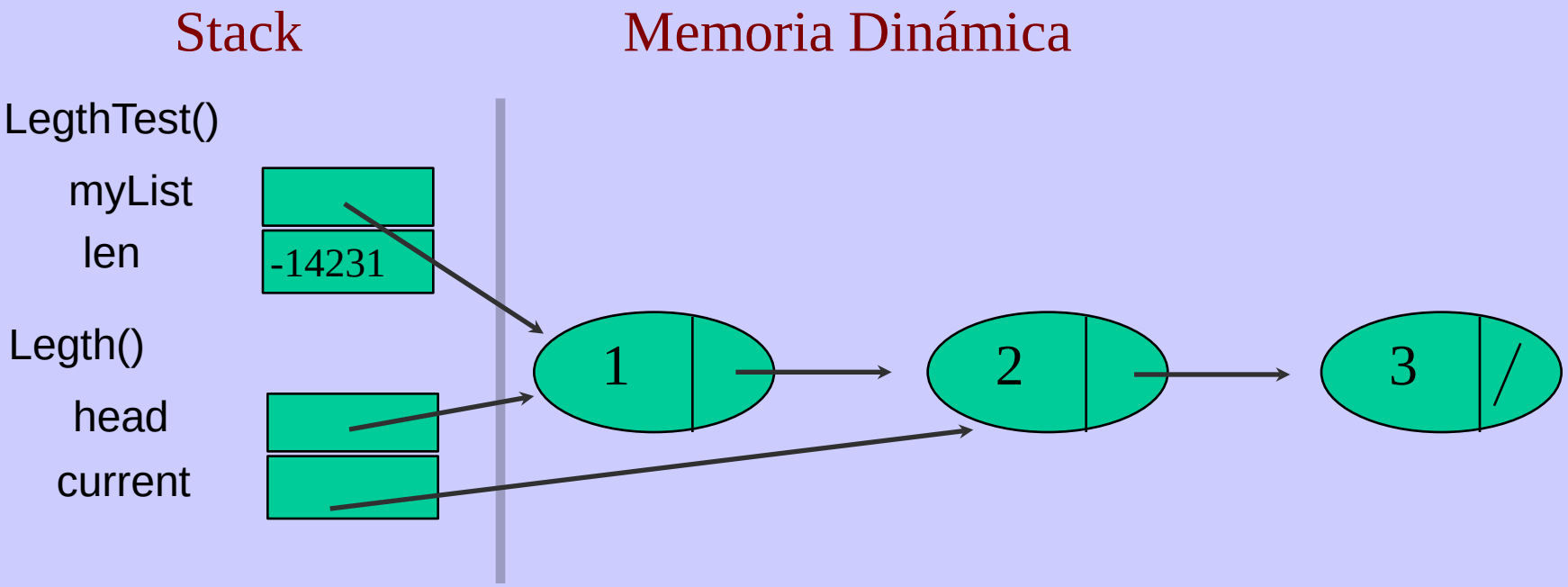
Llamando a Length()

```
void LengthTest() {
   struct node* myList = BuildOneTwoThree();
   int len = Length(myList);
}
```

Esquema de memoria antes de llamar a Length(myList);



Esquema de memoria a mitad de ejecutarse Length(myList);



Función para añadir un nodo a una LE
Push()

- Con BuildOneTwoThree() formamos una lista de 3 nodos. Queremos tener una función que cada vez que la llamemos cree un nodo en una LE. Identificamos 3 pasos siguientes:

Los 3-pasos son:

1. Reserva de memoria dinámica para el nodo y guardar los datos.

```
struct node* newNode;
newNode = malloc(sizeof(struct node));
newNode->data = data_client_wants_stored;
```

2. Apuntar el nodo nuevo hacia el actual primer nodo. “Asignar un puntero a otro puntero los hace apuntar a lo mismo”

```
newNode->next = head;
```

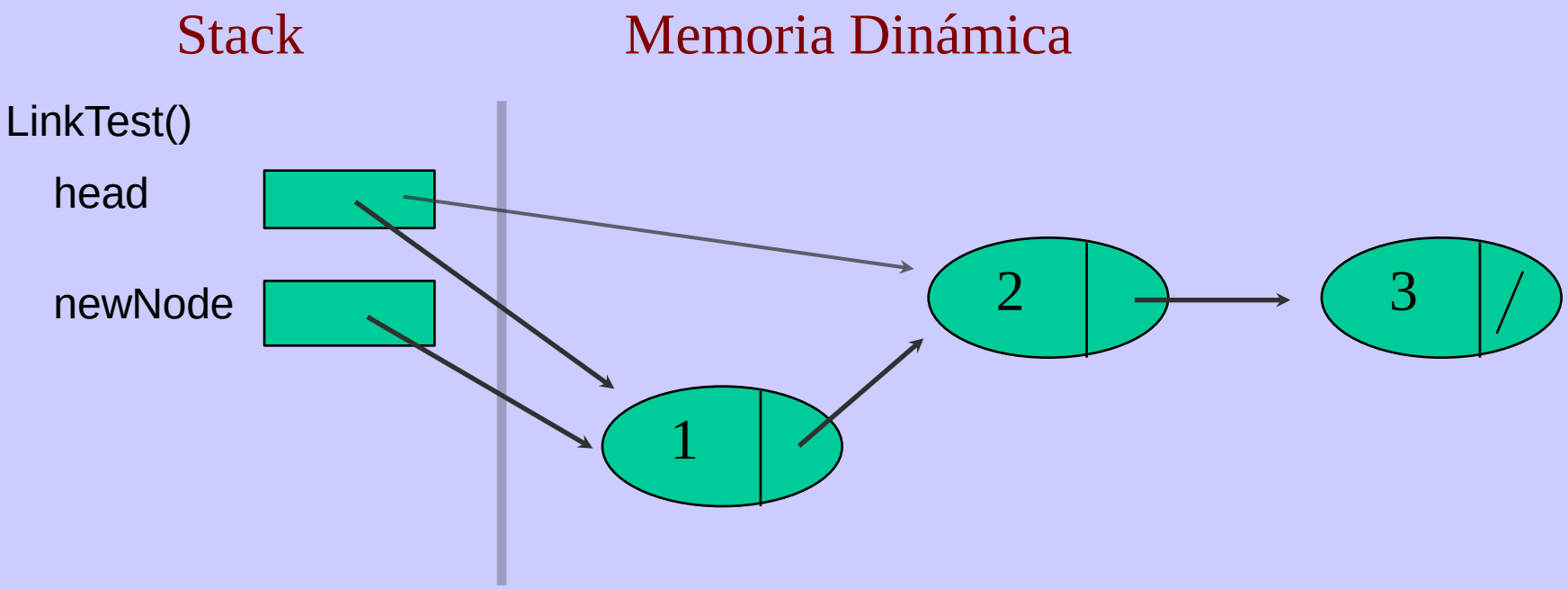
3. Cambiar el puntero *head* para que apunte al nuevo nodo, que ahora es el primer nodo de la lista enlazada.

```
head = newNode;
```

Ejemplo

```
void LinkTest() {
   struct node* head = BuildTwoThree(); //construye lista {2,3}
   struct node* newNode;

   newNode= malloc(sizeof(struct node));
   newNode->data = 1;

   newNode->next = head;

   head = newNode;

  //ahora head apunta a la lista {1,2,3}
}
```

Esquema de memoria



Función Push() errónea

- Podemos escribir una función que integre los 3 pasos para insertar un nuevo nodo a una LE, y una función que la llame con los parámetros adecuados.
- Los parámetros serían un puntero a la LE, y, el dato que queramos añadir.

Función Push() errónea

```
void WrongPush(struct node* head, int data) {
   struct node* newNode = malloc(sizeof(struct node));
   newNode->data = data;
   newNode->next = head;
   head = newNode;
}

void WrongPushTest() {
struct node*List head = BuildTwoThree();

   WrongPush(head, 1);
}
```

//esta línea no funciona

//esta línea no funciona

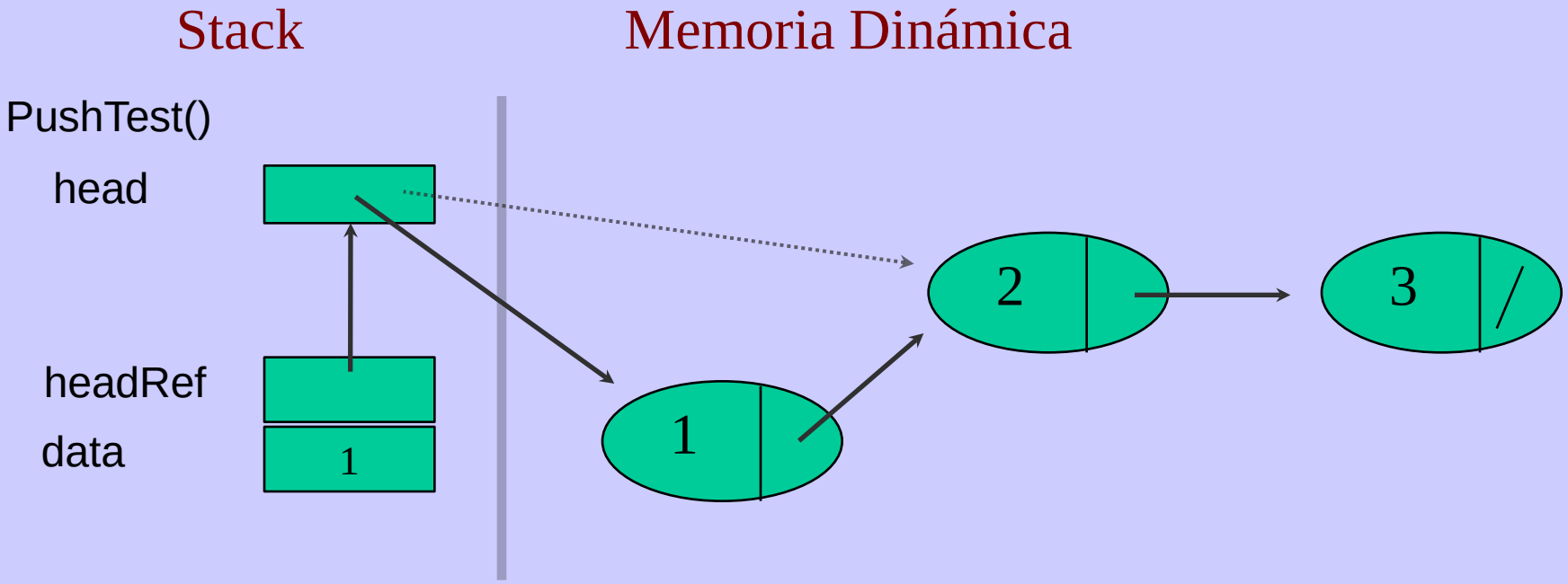
- Son dos head distintos.
  - Necesitamos que Push() pueda cambiar la memoria del llamante - “paso parámetro por referencia”
  - Queremos cambiar struct node*, luego hay que pasar struct node**

Función Push() correcta

- *Para cambiar la memoria de la función llamante, debemos pasar un puntero a esa memoria.*

```
void Push(struct node** headRef, int data) {
   struct node* newNode = malloc(sizeof(struct node));

   newNode->data = data;
   newNode->next = *headRef;    //El '*' obtiene el valor real de head
   *headRef = newNode;
}


void PushTest() {
   struct node* head = BuildTwoThree();
   Push(&head, 1);          //utilizamos '&' para pasar la dirección de head
   Push(&head, 13);
     //ahora head es la lista {13,1,2,3}
   }
```

Esquema de memoria



headRef es un parámetro para Push() que no es el head real de la LE.
Es un puntero al head real que está en el stack de la función llamante