

# PROGRAMACIÓN

**Unidad 6 – Parte 2.** Punteros y funciones. Mecanismos de pasaje de parámetros por referencia. Asignación dinámica de memoria.

# Revisamos la tarea pendiente



## 1. ¿Qué es un puntero en C y para qué se utiliza?

Un puntero es una variable que almacena una dirección de memoria. Se utiliza para acceder y manipular datos en la memoria de manera directa, lo que permite una gestión eficiente de la memoria y la manipulación de estructuras de datos dinámicas.

## 2. ¿Cómo se declara un puntero en C? Proporciona ejemplos.

**tipo \*nombre-vble-puntero**

Ejemplo: `int *pnumero;`

## 3. ¿Cuál es la diferencia entre un puntero y una variable normal en C?

Una variable normal almacena un valor, mientras que un puntero almacena una dirección de memoria que apunta a un valor en la memoria.

## 4. ¿Qué es la dirección de memoria de una variable y cómo se obtiene?

La dirección de memoria de una variable se refiere a la ubicación en la memoria donde se almacena. Se puede obtener utilizando el operador de dirección &. Por ejemplo: `&variable`.

# Revisamos la tarea pendiente



## 5. ¿Cómo se asigna la dirección de memoria de una variable a un puntero?

La dirección de memoria de una variable se asigna a un puntero mediante la siguiente sentencia:

```
puntero = &variable;
```

## 6. ¿Qué es la indirección de punteros y cómo se realiza en C?

La indirección de punteros es acceder al valor almacenado en la dirección de memoria apuntada por un puntero. Se realiza mediante el operador de indirección `*`.

Ejemplo: `*puntero` accede al valor apuntado por puntero.

## 7. ¿Cómo se realiza la aritmética de punteros en C? Proporciona ejemplos.

La aritmética de punteros implica operaciones matemáticas en las direcciones de memoria. Por ejemplo, para avanzar un puntero a la siguiente ubicación en un arreglo de enteros: `puntero = puntero + 1`, o `puntero++`.

# Revisamos la tarea pendiente



8. ¿Cuál es la diferencia entre punteros y arreglos en C? ¿Cómo se relacionan?

En C, un nombre de arreglo actúa como un puntero al primer elemento del arreglo. Los punteros son más flexibles y versátiles que los arreglos, pero están estrechamente relacionados, especialmente al trabajar con arreglos dinámicos.

9. ¿Por qué son importantes los punteros al trabajar con cadenas de caracteres en C?

Los punteros son importantes al trabajar con cadenas de caracteres en C porque permiten un acceso eficiente, manipulación y recorrido de las cadenas, ahorran memoria al pasar cadenas a funciones.

# Pasaje de Parámetros

El lenguaje C transfiere variables entre funciones usando **dos**  
**mecanismos diferentes.**

Pero... de que  
dependerá el  
mecanismo que utilice?

- ✓ El método a emplear depende de si se quiera modificar o no los valores de las variables transferidas.
- ✓ De la cantidad de valores “devueltos” por la función.



# Revisemos algunos conceptos

```
#include<stdio.h>

int esPerfecto(int num);

int main()
{
    int indice, resultado;

    printf("\n*****NUMEROS PERFECTOS***** \n");
    for (indice=1;indice<=100;indice++)
    {
        resultado = esPerfecto(indice);
        if (resultado == 1)
            printf(" %d Es un numero perfecto \n",indice);
    }
    return 0;
}

int esPerfecto(int num)
{
    int i, suma=0, resto;

    for (i=1;i<num;i++)
    {
        resto=num%i;
        if (t==0)
            suma=suma+i;
    }
    if(suma == num)
        return 1;
    else
        return 0;
}
```

**Parámetros Actuales:** son las expresiones pasadas como argumentos en la llamada a una función.

**Función Emisora:** Función que emite valores para la transferencia de información.

**Parámetros Formales:** Los parámetros formales sólo se conocen dentro de la función.

**Función Receptora:** Función que recepciona valores transferidos.

# Pasaje de Parámetros



- Pasaje por valor
- Pasaje por Referencia

# Pasaje por Valor

Cuando pasamos parámetros por valor:

- ✓ La función que invoca recibe los valores de sus argumentos en **variables temporales y no en las originales.**
- ✓ La función que se invoca no puede alterar directamente una variable de la función que hace la llamada, solo puede modificar su copia temporal.



# Pasaje por Valor

**Ejemplo:** Función para calcular el peso de una persona en la luna.

```
FUNCION pesoenlaluna (peso): real >0 → real ≥ 0  
    P_luna ← peso/6  
    Retorna (P_luna )
```

**Función emisora:** transfiere el peso en la tierra.

**Función receptora:** recibe el valor del peso en la tierra, calcula y devuelve el valor en la luna, sin modificar el valor del peso en la tierra.

# Pasaje por Valor

- En el momento de la invocación, la función receptora toma, en forma temporal, un espacio de almacenamiento de características idénticas a las del Parámetro actual.
- Si se hacen cambios dentro de la función en el valor de la variable transferida, los cambios se reflejan en esa localidad de memoria temporal.
- Cuando la función termina su ejecución, el espacio de almacenamiento temporal que se usó para “la copia” queda liberado, perdiéndose los valores que había allí almacenado.

# Pasaje por Valor

```
#include <stdio.h>

float pesoEnLuna(float PesoTierra);

int main()
{
    float PesoTierra, PesoLuna;
    PesoTierra = 95.8;
    PesoLuna = pesoEnLuna(PesoTierra);

    printf("El peso en la luna de Homero es: %.2f \n", PesoLuna);
    printf("El peso en la tierra de Homero es: %.2f", PesoTierra);

    return 0;
}

float pesoEnLuna(float PesoTierra){
    float aux;
    aux= PesoTierra/6;
    return (aux);
}
```

Calcular el peso de Homero Simpson en la luna.



# Pasaje por Valor

- ❑ **Una desventaja fuerte de este mecanismo:**

Las funciones que lo usan pueden devolver a lo sumo un resultado asociado a la proposición return.

- ❑ **Ventaja:** Preserva los valores de las variables transferidas.

# Pasaje por Referencia

**Ejemplo: cambios de precios de varios productos comerciales.**

El precio de un producto **precioProd** se calcula como la suma de los precios de sus insumos **Insumo1** y **Insumo2** más una constante. Los precios de **Insumo1** y **Insumo2** se modifican periódicamente, por lo cual el precio de nuestro producto **precioProd** también se modifica. Calcular los incrementos de **Insumo1**, **Insumo2** y **precioProd** según los porcentajes de cambio indicados.

Porcentajes de Cambios: **Insumo1** se incrementa en un 3% y **Insumo2** en un 5%.

$$\text{Insumo1} \leftarrow \text{Insumo1} * 1.03$$

$$\text{Insumo2} \leftarrow \text{Insumo2} * 1.05$$

$$\text{precioProd} \leftarrow \text{Insumo1} + \text{Insumo2} + \text{cte}$$

# Pasaje por Referencia



- ❑ **Función emisora:** transfiere los precios de los productos (generalmente main).
- ❑ **Función receptora:** recibe los precios de los productos, los modifica y los devuelve modificados.

# Pasaje por Referencia

**Ejemplo:** Cambios de precios de productos comerciales.

El precio de un producto **precioProd**, se calcula como la suma de los precios de los insumos **Insumo1**, **Insumo2**. Los precios **Insumo1**, **Insumo2** se modifican periódicamente, por lo cual el precio de nuestro producto **precioProd** también se modifica. Calcular los incrementos de **Insumo1**, **Insumo2** y **precioProd** según los porcentajes de cambios indicados.

**Porcentajes de cambios:**

- **Insumo1** se incrementa un 3%
- **Insumo2** se incrementa un 5%

```
float modificarPrecio (float *insumo1, float *insumo2)
{
    float precioProd;

    *insumo1 = (*insumo1) * 1.03;
    *insumo2 = (*insumo2) * 1.05;
    precioProd = (*insumo1)+ (*insumo2);

    return(precioProd);
}
```

```

#include <stdio.h>

float modificarPrecio(float *pinsumo1, float *pinsumo2);

int main()
{
    float insumo1, insumo2, precioProd;

    printf("Ingrese el precio del insumo 1:");
    scanf("%f", &insumo1);

    printf("Ingrese el precio del insumo 2:");
    scanf("%f", &insumo2);

    precioProd = modificarPrecio(&insumo1, &insumo2);

    printf("El precio del producto es: %f \n", precioProd);
    printf("El insumo1 modificado es: %f \n", insumo1);
    printf("El insumo2 modificado es: %f \n", insumo2);

    return 0;
}

float modificarPrecio(float *pinsumo1, float *pinsumo2)
{
    float precioProd;

    *pinsumo1 = (*pinsumo1) * 1.03;
    *pinsumo2 = (*pinsumo2) * 1.05;
    precioProd = (*pinsumo1) + (*pinsumo2);

    return(precioProd);
}

```

# Pasaje por Referencia

La función receptora tendrá como Parámetros formales variables de tipo puntero.

En el momento de la invocación, la función receptora recibe como parámetros actuales las direcciones de memoria de variables de la función emisora.

Los cambios realizados adquieren el carácter de permanentes, es decir, mientras dure la ejecución del programa. Cuando la función termina su ejecución, todos los cambios realizados en esa locación de memoria quedan allí, por esto su carácter de permanentes.



# Pasaje por Referencia

## □ Ventajas:

- Bajo costo en términos de espacio de almacenamiento, No se realizan duplicaciones o copias de variables.
- La función puede “regresar” más de un valor.

## □ Desventaja:

- Usar con precaución, las variables quedan muy vulnerables.

# Pasaje por Referencia



Cada vez que se transfiere una variable por dirección, si la función receptora modifica la variable, ésta es modificada también en la función invocante.

# Pasaje de Parámetros

- ¿Cómo elegir que mecanismo usar?
  - No modificar los valores de las variables usadas como parámetros actuales: **POR VALOR**
  - Modificar valores de las variables usadas como parámetro actuales y obtener mas de un valor de retorno: **POR REFERENCIA**

# Pasaje por Referencia - Arreglos

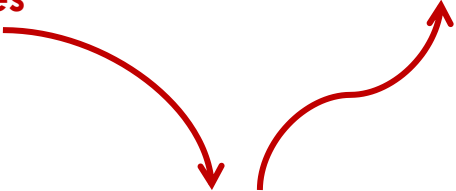


Cuando el nombre de un arreglo se emplea como argumento, el valor que se pasa a la función es la dirección de memoria de la 1<sup>ra</sup> componente del arreglo, se trabaja sobre la dirección original.

# Pasaje por Referencia - Arreglos

**Ingresa con sus  
valores iniciales**

**Sale modificado**



```
Void Inicializar(int arre[] , int n)
{
    int i;
    For(i=0; i<n; i++)
    {
        arre[i]=0;
    }
}
```

Cuando se llama la función de nombre “inicializar” se le envían 2 parámetros para que pueda proceder. Un entero que corresponde al tamaño del arreglo, (pasaje por valor), y también se envía un arreglo.

Cuando enviamos el nombre del arreglo, lo que enviamos en realidad es la dirección de memoria del primer elemento del arreglo. Es por este motivo que decimos que al pasar un arreglo a una función utilizamos pasaje por referencia.

# Asignación de memoria dinámica

Una característica esencial del lenguaje C es la capacidad de requerir bloques de memoria variable durante la ejecución del programa.

Hasta ahora hemos visto que se reserva espacio para variables en el momento que las definimos. Por ejemplo:

```
int x; // reservamos espacio para almacenar un entero
float z; //reservamos espacio para almacenar un valor real
char car; //reservamos espacio para almacenar un caracter
int arreglo[10]; //reservamos espacio para almacenar 10 enteros
int *puntero; //reservamos espacio para almacenar un puntero
```

# Asignación dinámica de memoria

- ❑ La memoria que C asigna para que utilice dinámicamente un programa se denomina la pila (heap).
- ❑ La pila se compone de la memoria sin utilizar que no ha sido asignada ni al sistema operativo ni, ni a ningún programa ni a ninguna variable de programa.
- ❑ Mientras un programa se ejecuta, el tamaño de la pila aumenta y disminuye en forma permanente.
- ❑ Si no se utilizara la asignación dinámica de memoria, un programa consumiría la misma cantidad de espacio de memoria durante toda su ejecución.

# Asignación dinámica de memoria

Todos los usuarios y las tareas de los programas necesitan compartir la misma memoria, de modo que los programas no deberían reservar memoria hasta que la necesiten.

Por esto, el lenguaje C nos permite en tiempo de ejecución solicitar espacio de memoria mediante la función **malloc** (**m**emory **a**llocat**e** = Asignar memoria) y luego de usarla en forma obligada debemos devolverla llamando a la función **free**.

Ambas funciones se encuentra dentro de la librería: **<stdlib.h>**



# Asignación de memoria dinámica

Analicemos el siguiente ejemplo:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  int main()
6  {
7      int *puntero;
8
9      puntero = (int *) malloc(sizeof(int));
10
11     *puntero = 26;
12
13     printf("El valor alojado en la memoria asignada dinamicamente es: %d", *puntero);
14
15     free(puntero);
16     return 0;
17 }
```

malloc() retorna un puntero "vacío" de tipo indeterminado. Por lo tanto se necesita modificar su tipo de forma temporal con un cast.

malloc() necesita saber cuanta memoria de la pila necesita.

Luego de hacer uso de la memoria dinámica se "devuelve" a la pila, con la función free().

**Importante:** Si no invocamos la función free(), C recién liberará la memoria asignada y la devolverá a la pila cuando el programa finalice.

```

#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *puntero, tama, i;

    printf("Cuantos Elementos tendra el arreglo?:");
    scanf("%d", &tama);

    puntero = (int *)malloc(tama * sizeof(int));
    int *puntaux = puntero;

    for ( i = 0; i < tama; i++)
    {
        printf("Ingrese un elemento: ");
        scanf("%d", &puntero[i]);
    }

    printf("Contenido del arreglo dinamico es: \n");
    for ( i = 0; i < tama; i++)
    {
        printf("%d\n", *puntaux);
        puntaux++;
    }

    free(puntero);
    return 0;
}

```

# Asignación de memoria dinámica

La asignación dinámica, nos permite con un solo puntero asignar dinámica TODO un arreglo completo cuando se necesite.

## Trabajemos con el siguiente ejemplo:

Crear un arreglo en forma dinámica, cargar e imprimir sus datos.

# Asignación de memoria dinámica

Expliquemos el paso a paso:

- Para trabajar con la memoria dinámica en el lenguaje C es obligatorio trabajar con punteros. Lo primero que hacemos es definir un puntero a entero:

```
int *puntero;
```

- Solicitamos al usuario de nuestro programa que ingrese un entero que representa la cantidad de elementos que tendrá el arreglo:

```
printf("Cuantos elementos tendra el arreglo?: ");  
scanf("%d", &tama);
```

- No esta permitido indicar en el índice de un arreglo con una variable: ~~int arre[tam];~~
- Pero este problema lo podemos resolver solicitando espacio mediante la función malloc, debemos indicar el número de bytes a reservar:

```
puntero = (int *)malloc(tama*sizeof(int));
```

El operador sizeof cuando le pasamos como parámetro un tipo de dato lo que nos devuelve es la cantidad de bytes que se requieren para almacenar dicho tipo de dato.

# Asignación de memoria dinámica

- Una vez que hemos reservado espacio, cargamos el arreglo como siempre:

```
int *puntAux = puntero;

for ( i = 0; i < tama; i++)
{
    printf("Ingrese un elemento: ");
    scanf("%d", &puntero[i]);
}

printf("Contenido del arreglo dinamico es: \n");
for ( i = 0; i < tama; i++)
{
    printf("%d\n", *puntAux);
    puntAux++;
}
```

Podemos usar notación de índice para “mover” un puntero.

Necesitamos un puntero auxiliar en caso de recorrer el arreglo incrementando el puntero

- puntAux nos permite no perder el inicio de la reserva, de esta manera podremos liberar la memoria asignada de forma correcta. En caso de que incrementemos el puntero en el segunda estructura de iteración For(), al finalizar el bucle, el puntero quedará al final de la memoria usada y si ejecutamos la función free(puntero), no estaríamos liberando toda la memoria asignada.

# Asignación de memoria dinámica

- Otra forma de poder liberar correctamente la memoria podría ser decrementando el puntero al inicio.

```
printf("Contenido del arreglo dinamico es: \n");
for (i = 0; i < tama; i++)
{
    printf("%d\n", *puntero);
    puntero++; // Avanza el puntero al siguiente elemento
}

// Restablece el puntero al inicio del bloque de memoria asignado
puntero -= tama;

free(puntero);
```

# Función realloc

La función `realloc` en el lenguaje de programación C se utiliza para redimensionar (aumentar o disminuir) la memoria asignada previamente con `malloc`. En otras palabras, se utiliza para cambiar el tamaño de un bloque de memoria dinámica.

Analicemos la siguiente línea de código:

```
nuevo_puntero= (int *)realloc(puntero_inicial, tama_nuevo * sizeof(int));
```

Nuevo puntero, del mismo tipo base que el puntero inicial

Función `realloc` que está disponible en la biblioteca `stdlib.h`

Puntero en el que se hizo la asignación dinámica con `malloc()`

El nuevo tamaño por el cual vamos a redefinir el espacio asignado inicialmente

## Ejemplo: uso de la función realloc()

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int *puntero, capacidad = 5, longitud = 0, numero;

    puntero = (int *)malloc(capacidad * sizeof(int));

    printf("Ingrese enteros (0 para finalizar):\n");
    scanf("%d", &numero);

    while (numero != 0){
        if (longitud >= capacidad)
        {
            capacidad *= 2; // Duplicar la capacidad
            int *nuevo_puntero = (int *)realloc(puntero, capacidad * sizeof(int));
            puntero = nuevo_puntero;
        }
        puntero[longitud] = numero;
        longitud++;

        scanf("%d", &numero);
    }
    printf("Elementos ingresados:\n");
    for (int i = 0; i < longitud; i++) {
        printf("%d ", puntero[i]);
    }

    free(puntero);
    return 0;
}
```

# Bonus Track

## Una función puede retornar un puntero?

La respuesta es Si, una función en C puede retornar un puntero.

Para declarar una función que retorna un puntero, la sintaxis es similar a la declaración de una función que retorna otros tipos de datos, pero debes especificar el tipo de datos al que apuntará el puntero. Por ejemplo:

```
int *crearArregloDinamico(int tamano);
```



# Bonus Track

```
#include <stdio.h>
#include <stdlib.h>

int *crearArreglo(int tamano);
int main() {
    int *puntero = crearArreglo(5);

    for (int i = 0; i < 5; i++) {
        puntero[i] = i * 2;
    }

    for (int i = 0; i < 5; i++) {
        printf("%d ", puntero[i]);
    }

    free(puntero);

    return 0;
}

int *crearArreglo(int tamano) {
    int *punteroArreglo = (int *)malloc(tamano * sizeof(int));
    return punteroArreglo;
}
```

Veamos el  
ejemplo completo

# Tarea para la casa



1. ¿Qué significa pasar un parámetro por valor en C?
2. ¿Los cambios en un parámetro pasado por valor en una función afectan a la variable original fuera de la función?
3. ¿Cómo se pasa un parámetro por referencia en C?
4. ¿Cuál es la ventaja de pasar parámetros por referencia en comparación con por valor?
5. ¿Puedes dar un ejemplo de una función que acepte un parámetro por referencia en C?
6. ¿Qué es la asignación dinámica de memoria en C y por qué es importante?
7. ¿Cuáles son las funciones en C que se utilizan para asignar memoria dinámicamente?
8. ¿Cuál es la diferencia entre malloc() y realloc()?
9. ¿Por qué es necesario liberar la memoria asignada dinámicamente con free()?

**Lo revisamos la próxima clase.**

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
    printf("Hasta la próxima clase!!\n");
```

```
    return 0;
```

```
}
```