Punteros Dobles

Definición:

Un puntero en C es una variable que almacena la dirección de memoria de otro valor. Un puntero doble, por otro lado, almacena la dirección de memoria de otro puntero. Esto puede ser útil en situaciones en las que necesitamos trabajar con punteros o arreglos de punteros.

Acceso:

- Para acceder al valor (5), se usa **punteroDoble.
- Para acceder a la dirección de memoria de punteroDoble: *punteroDoble.

Ejemplo:

```
int main() {
   int num = 42;
   int *ptr1 = &num; // Puntero simple que apunta a 'num'
   int **ptr2 = &ptr1; // Puntero doble que apunta a 'ptr1'

   printf("Valor de num: %d\n", num);
   printf("Valor apuntado por ptr1: %d\n", *ptr1);
   printf("Valor apuntado por ptr2 (usando puntero doble): %d\n", **ptr2);

   return 0;
}
```

En el ejemplo:
Declaramos una variable num de tipo entero y le asignamos el valor 42.
Declaramos un puntero simple ptr1 que apunta a la dirección de memoria de num.
Declaramos un puntero doble ptr2 que apunta a la dirección de memoria de ptr1.

Luego, si imprimimos, la salida será:

```
Valor de num: 42
Valor apuntado por ptr1: 42
Valor apuntado por ptr2 (usando puntero doble): 42
```

En este ejemplo, ptr2 almacena la dirección de memoria de ptr1, que a su vez almacena la dirección de memoria de num. Al usar el puntero doble ptr2, podemos acceder al valor de num de manera indirecta.
Los punteros dobles son especialmente útiles cuando se trabaja con matrices de punteros, como matrices de cadenas de caracteres (arreglos de strings) o matrices de punteros a funciones.

Uso en funciones:

- Prototipo e implementación; **void función (tipo_dato ** pp);**
- Llamado desde el main; **tipo_dato * p = NULL;**
  función(&p);

Memoria dinámica

Problema con memoria estática: cuando trabajo con arreglos estáticos debo definir el tamaño que será reservado para almacenar datos en **tiempo de compilación**, y luego no existe posibilidad de cambiar la dimensión del arreglo durante la ejecución en caso de que el volumen de datos exceda el tamaño predefinido.

Solución: La memoria dinámica en C es una forma de asignar y liberar memoria **durante la ejecución** de un programa en lugar de hacerlo en tiempo de compilación. Esto permite la creación de estructuras de datos con tamaños variables y la gestión eficiente de recursos.

Funciones principales:

MALLOC; La función malloc se utiliza para asignar un bloque de memoria de tamaño especifico durante la ejecución del programa. Esta función devuelve un puntero void (puntero genérico) al inicio del bloque de memoria recién asignado, que debe ser convertido al tipo de dato deseado antes de su uso (casteo). Si ocurre algún error con la asignación de memoria retorna NULL.

```c
///Reserva memoria para un arreglo dinamico de enteros
int * generaArregloDinamico(int tamanio)
{
    int * arregloDinamico = (int *) malloc(tamanio * sizeof(int));

    if(arregloDinamico == NULL)
    {
        printf("Error al asignar memoria");
    }

    return arregloDinamico;
}
```

REALLOC; La función de realloc se utiliza para cambiar el tamaño de un bloque de memoria previamente asignado por malloc o realloc.

```c
int * redimensionarArregloDinamico(int * arregloDinamico, int * tamanio)
{
    *tamanio += *tamanio*2;
    arregloDinamico = (int*) realloc(arregloDinamico, (*tamanio) * sizeof(int));

    if(arregloDinamico == NULL)
    {
        printf("Error al asignar memoria");
    }

    return arregloDinamico;
}
```

Devuelve un puntero al bloque de memoria redimensionado, que puede ser igual a arregloDinámico o a una nueva dirección de memoria si la redimensión requiere reubicación. En el caso que ocurra un error devuelve NULL.

```c
int * cargaDatos( int * arregloDinamico, int * tamanio)
{
    int i = 0;
    int opc = 1;
    do{
        /** si i es igual a donde apunta tamaño estoy en situacion
            de realojar el arreglo ya que llegue a mi limite **/
        if(i == (*tamanio)){
            arregloDinamico = redimensionarArregloDinamico(arregloDinamico,tamanio);
        }

        printf("Ingrese un entero");
        scanf("%i",&arregloDinamico[i]);
        i++;

        printf("0 para salir");
        scanf("%i", &opc);

    }while(opc != 0);

    return arregloDinamico;
}
```

```c
int main()
{
    int * arregloDinamico;
    int tamani = 20;

    arregloDinamico = generaArregloDinamico(tamanio)

    arregloDinamico = cargaDatos(arregloDinamico, &tamanio);

    return 0;
}
```

Liberación de memoria: **free(arregloDinamico),** esto se puede utilizar por ejemplo cuando luego de cargar un arreglo dinámico con datos que luego se escriben en un archivo, en ese caso es útil y una buena práctica utilizar una función que libere la memoria.

¿Qué es un puntero void?

Un puntero void es un tipo de puntero genérico en C que puede apuntar a cualquier tipo de dato, pero no puede ser desreferenciado directamente sin conversión. Su utilidad principal se ve en funciones que deben ser flexibles respecto al tipo de datos que manejan, como malloc y realloc.

Un puntero void no tiene tipo asociado, por lo que el compilador no sabe cuántos bytes leer o escribir al desreferenciarlo. Por eso, hay que convertirlo al tipo correcto antes de usarlo.

Veamos como usarlo con un ejemplo:

```c
int main()
{
    char archivoAlumnos[] = "alumnos.bin";
    char archivoNotas[] = "notas.bin";

    Alumno a = cargarAlumno();
    escribirArchivoGenerico(archivoAlumnos, &a, sizeof(Alumno));
    mostrarArchivoAlumnos(archivoAlumnos);

    Nota n = cargarNota();
    escribirArchivoGenerico(archivoNotas, &n, sizeof(Nota));
    mostrarArchivoNotas(archivoNotas);

    return 0;
}
```

```c
/*
size_t es un entero sin signo o entero positivo, lo usamos para evitar que entren numeros
negativos por parametro. El puntero void puede apuntar a cualquier tipo de dato pero no
se puede desreferenciar. por esto se puede cargar de a un solo dato cada vez que la funcion
es invocada.
*/

void escribirArchivoGenerico(char nombreArchivo[], void* dato, size_t tamanioDato)
{
    FILE* buffer = fopen(nombreArchivo, "ab");
    if(buffer!=NULL)
    {
        fwrite(dato, tamanioDato, 1, buffer); ///el 1er parametro no lleva & porque la variable
        ///dato ya almacena una dir de memoria
        fclose(buffer);
    }
    else
    {
        printf("error");
    }
}
```

Aplicación de punteros dobles con memoria dinámica

¿Por qué usarlos juntos? las funciones que reservan o modifican memoria no necesitan retornar punteros.

Una pregunta que puede surgir al plantear el uso de punteros dobles en memoria dinámica para modificar el contenido del puntero simple y no tener que retornarlo es: ¿Por qué no se modifica la dirección de memoria del puntero simple si en definitiva estoy usando punteros? Hay una diferencia clave entre modificar el contenido que apunta un puntero vs. modificar la dirección de memoria que guarda un puntero.

```c
int main()
{
    int numero = 5;
    int * puntero = &numero;

    funcion(puntero);

    return 0;
}

void funcion (int * puntero)
{
    int numeroNuevo = 10;

    puntero = &numeroNuevo; ///este cambio no se refleja en el main
}
```

Variables simples; cuando pasas una variable simple por puntero puedes modificar su contenido, es decir estas modificando el **contenido** al que apunta el puntero, no la dirección de memoria.

```c
int main()
{
    int numero = 5;
    int * puntero = &numero;

    funcion(&puntero);

    return 0;
}

void funcion (int ** puntero) /// uso puntero doble
{
    int numeroNuevo = 10;

    puntero = &numeroNuevo; ///este cambio SI se refleja en el main
}
```

Sobre memoria dinámica; cuando pasas un puntero simple a una función, y dentro de esa función usas malloc o realloc **estas cambiando la dirección de memoria** que guarda el puntero (mi arreglo dinámico), pero como este fue pasado por puntero simple (por valor) el puntero original fuera de la función no cambia, es decir **no se entera de esa nueva dirección de memoria.**

```c
void reservarMemoria(int ** arregloDinamico, int tamanio)
{
    *arregloDinamico = (int *) malloc(tamanio * sizeof(int));

    if(*arregloDinamico == NULL)
    {
        printf("Error al asignar memoria");
    }
}
```

Funciones que Devuelven un Puntero:

Crear Arreglo Dinámico: Crear un arreglo dinámico de enteros y devolver un puntero a él.

```
int *crearArregloDinamico(int dim) {
   int *arre = (int *)malloc(dim * sizeof(int));
   return arre;
}

int main() {
   int n = 5;
   int *ptr = crearArregloDinamico(n);
   if (ptr == NULL) {
      printf("No se pudo asignar memoria (crear arreglo).\n");
      return 1;
   }
   for (int i = 0; i < n; i++) {
      ptr[i] = i * 2;
      printf("%d ", ptr[i]);
   }
   free(ptr);
   printf("\n");
   return 0;
}
```

Duplicar Cadena: Duplicar una cadena y devolver un puntero a la nueva cadena.

```
#include <string.h>
char *duplicarCadena(const char *str) {
   char *duplicado = (char *)malloc((strlen(str) + 1) * sizeof(char));
   strcpy(duplicado, str);
   return duplicado;
}
int main() {
   char original[] = "Hola, esto es una cadena de prueba.";
   char *copia = duplicarCadena(original);
   printf("Copia: %s\n", copia);
   free(copia);
 return 0;
}
```

Clonar Arreglo Dinámico: Clonar un arreglo dinámico de enteros y devolver un puntero al nuevo arreglo.

```
int *clonarArregloDinamico(int *arre, int dim) {
   int *clon = (int *)malloc(size * sizeof(int));
   for (int i = 0; i < dim; i++) {
      clon[i] = arre[i];
   }
   return clon;
}

int main() {
   int numeros[] = {2, 4, 6, 8, 10};
   int dim = sizeof(numeros) / sizeof(numeros[0]);
   int *copia = clonarArregloDinamico(numeros, dim);
   for (int i = 0; i < dim; i++) {
      printf("%d ", copia[i]);
   }
   printf("\n");
   free(copia);
   return 0;
}
```

Crear Cadena Dinámica: Crear una cadena dinámica y devolver un puntero a ella.

```
#include <string.h>

char *crearCadenaDinamica(const char *str) {
   char *cadena = (char *)malloc((strlen(str) + 1) * sizeof(char));
   strcpy(cadena, str);
   return cadena;
}

int main() {
   char texto[] = "Esta es una cadena dinamica.";
   char *cadena = crearCadenaDinamica(texto);
   printf("Cadena: %s\n", cadena);
   free(cadena);
   return 0;
}
```

Ejemplos punteros dobles

Ejemplo 1: Función para intercambiar dos valores usando punteros dobles

```
void swap_doble(int **a, int **b) {
   int temp = **a;
   **a = **b;
   **b = temp;
}
int main() {
   int num1 = 5, num2 = 10;
   int *ptr1 = &num1, *ptr2 = &num2;
   printf("Ejercicio 1: Intercambio de valores usando punteros dobles\n");
   printf("Antes del intercambio: num1 = %d, num2 = %d\n", num1, num2);
   swap_doble(&ptr1, &ptr2);
   printf("Después del intercambio: num1 = %d, num2 = %d\n", num1, num2);
}
```

Ejemplo 2: Uso de punteros dobles para acceder a un arreglo

```
   int arr[] = {1, 2, 3, 4, 5};
   int *arrPtr = arr;
   int i;
   printf("\nEjercicio 2: Acceso a un arreglo usando punteros dobles\n");
   for (i = 0; i < sizeof(arr) / sizeof(int); i++) {
      printf("Elemento %d: %d\n", i, *(*(&arrPtr) + i));
   }
```

Declara un puntero a un entero llamado arrPtr y lo inicializa con la dirección del primer elemento de arr. Es equivalente a int *arrPtr = &arr[0]

Esta línea imprime el índice del elemento (i) y el valor del elemento del array. La parte más compleja es *(*(&arrPtr) + i), que vamos a desglosar:

- &arrPtr: Obtiene la dirección de la variable arrPtr (que es un puntero).
- *(&arrPtr): Dereferencia la dirección de arrPtr, lo que nos da el valor de arrPtr (que es la dirección del primer elemento de arr).
- (*(&arrPtr) + i): Suma i a la dirección del primer elemento de arr. En aritmética de punteros, sumar un entero a un puntero incrementa la dirección en i veces el tamaño del tipo de dato al que apunta el puntero (en este caso, int). Por lo tanto, esto calcula la dirección del i-ésimo elemento de arr.

TUP
Programación I

Ejercicio 3: Pasar un arreglo a una función usando punteros dobles

```
void printArray(int **arrPtr, int size)
{
      for (i = 0; i < size; i++)
     {
         printf("Elemento %d: %d\n", i, *(*arrPtr + i));
      }
   }
   printf("\nEjercicio 3: Pasar un arreglo a una función usando punteros dobles\n");
   printArray(&arrPtr, sizeof(arr) / sizeof(int));
   return 0;
}
```

- int **arrPtr: Un puntero a un puntero a un entero. Esto está diseñado para recibir la *dirección* de un puntero a int.
- int size: El número de elementos en el array.

*arrPtr: Dereferencia arrPtr. Dado que arrPtr es un int **, *arrPtr nos da el int * al que apunta arrPtr.

(*arrPtr + i): Realiza aritmética de punteros. Suma i a la dirección almacenada en *arrPtr.