

Punteros y Memoria Dinámica

Ignacio Solar

ignacio.solar@usach.cl

30 de Marzo, 2024

En **C** un puntero es una variable cuyo **valor** es la **dirección** de memoria de otra variable.

El manejo de memoria toma una parte **importante** en el desarrollo en C.

Where do you live? (&)

- El address-of & operator, permite guardar la dirección de memoria de otra variable.

```
fscanf(file, "%d", &size);  
//al leer archivos, referencio una variable para asignarla
```

Operador de Desreferencia *

- El puntero * almacena el valor de la variable a la que se esta apuntando.

```
1  int variable = 25;
2  int *puntero = &variable;
3  printf("%p = %p\n", puntero, &variable); //%p da el puntero no el valor!
4  //000000E8627FF824 = 000000E8627FF824
5  //el puntero a la variable es lo mismo que acceder a su direccion
```

```
1  printf("%d = %d", *puntero, variable);    //25 = 25
```

Pointer Dereferencing (Acceso a los datos)

```
1 int variable = 150;
2 int *h = &variable;
3 int **f = &h;
4 printf("%p\n", *f); //0x00000076FBFFA44
5 printf("%p\n", &variable); //0x00000076FBFFA44
6 printf("%d\n", **f); //150
```

Modificar datos de una variable sin invocarla

```
1 int valor = 25;  
2 int *puntero = &valor;  
3 *puntero +=1;  
4 printf("%d", valor); //26
```

¿Por que Punteros?

Para poder realmente modificar una variable, se debe **referenciar**.

Mutabilidad de objetos

```
list1 = [1,3,2,4,5]
list1.sort()
print(list1) # [1, 2, 3, 4, 5]
```

```
def agregar_numero_lista(lst):
    return lst + [100]

mi_lista = [1, 2, 3]
agregar_numero_lista(mi_lista)
print(mi_lista) # [1,2,3]
mi_lista = agregar_numero_lista(mi_lista)
print(mi_lista) # [1,2,3,100]
```


Voy al peluquero y le entrego una copia mía

```
1 char persona;  
2 void cortar_pelo(char cliente){  
3     //corta el pelo  
4 }  
5 cortar_pelo(persona);  
6 char persona; // me cobraron y no me cortaron el pelo!
```

Va a cortar el pelo de una copia mía y no a mi!

```

1 char persona;
2 void cortar_pelo(*cliente){
3     /*corta el pelo
4  }
5 cortar_pelo(persona); //soy una persona distinta ahora!!

```

```

1 #include <stdio.h>
2 #include <assert.h>
3 int suma(int valor) {return valor +=1;}
4 void suma_bacan(int *valor){ *valor +=1;}
5 int main() {
6     int numero = 1;
7     int copia = numero;
8     suma(numero);
9     assert(copia == numero); //retorna verdadero y se continua ejecutando
10    suma_bacan(&numero); //entrego la direccion de mi variable
11    assert(copia == numero);
12    //Assertion failed: copia == valor, line 11
13    return 0;
14 }

```

- Referenciar permite hacer funciones que modifican direcciones directamente, en vez de copiar el valor.

Pass by Reference

Aritmetica de Punteros



Operadores Unarios a un Puntero

```
1 int arr[5] = {1,2,3,4,5};
2 int i;
3 for (i = 0; i < 5; i++){
4     printf("%d\n", i[arr]); // 🤪
5     printf("%d\n", (*(i + arr))); // wtf...
6     // arr retorna la direccion inicial de puntero
7     printf("%d\n", arr[i]);
8 }
```

Los 3 procesos son Equivalentes

```
1 int vector[] = {28, 41, 7};
2 int *pi = vector; // %p = 0x0000BF23DFFBAC
3 //pi existe en la primera posicion del arreglo
4 printf("%d\n", *pi); // 28
5 pi += 1; // pi: 0x00000BF23DFFBB0
6     printf("%d\n", *pi); // 41
7 pi += 1; // pi: 0x00000BF23DFFBB4
8 printf("%d\n", *pi); // 7
```

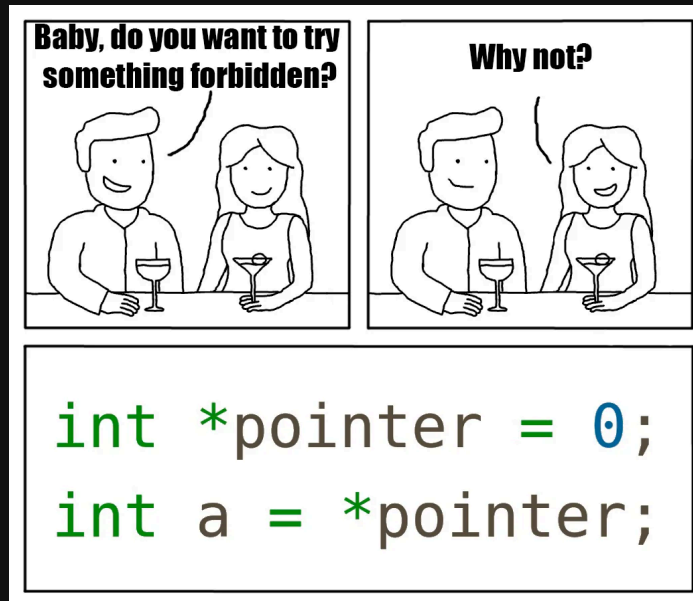
```
1 int l = 7; // %p = 0x7c35f4
2 int m = 13;
3 int* n = &l;
4 printf("%p\n", n); //7
5 printf("%p\n", n + 1);
6 //imprime 13 en gcc 8.1.0
7 //imprime 0x58FFFE4C en gcc 13.2
```

```
1 int g = 27;
2 int* h = &g;
3 int** i = &h;
4 printf("%d\n", g); //27
5 printf("%d\n", *h); //27
6 printf("%d\n", **i); //27
```

```
1 int vector[] = {28, 41, 7};
2 int *p0 = vector;
3 int *p1 = vector+1;
4 int *p2 = vector+2;
5 printf("p2>p0: %d\n", p2>p0); // p2>p0: 1
6 printf("p2<p0: %d\n", p2<p0); // p2<p0: 0
7 printf("p0>p1: %d\n", p0>p1); // p0>p1: 0
```

Uso de *NULL*

- Distintas definiciones:
 - `((void*)0)` tipo `void *`.
 - `(1+3-4)`, evaluado como 0, tipo `int`.
 - `NULL` es un estado específico en el cual NO tiene asignación.



Si `NULL` es 0, ¿cómo se diferencia del `int 0`?

```
00085 #ifndef NULL
00086 #define NULL      0
00087 #endif
```

Asignacion y declaracion

- Estar declarado pero no asignado **NO** implica que sea NULL

```
1 // declaracion de puntero
2 int* ptr;
3 if (ptr == NULL) {
4     printf("El puntero es null");
5 }
6 else {
7     printf("Valor del puntero %d", *ptr);
8 }
9 return 0;
```

Este código no genera ninguna salida

Es buena practica usar NULL

```
1 int* ptr = &variable; // memoria para mi puntero
2 // en el caso que falle...
3 if (!ptr) { // equivalente a -> prn != NULL
4     printf("Puntero no creado correctamente\n");
5     exit(0);
6 }
7 if(ptr){
8     printf("We gucci 😎😎\n");
9 }
```

```
if (ptr != NULL) {
    // Recien aqui es SEGURO dereferenciar el puntero
    // muestra tambien que solo lo queremos usar en otros estados
}
```

- Próximamente sera muy útil 🤖

```
Nodo *nodo_actual = L1->head; // aux nodo, se apodera de lista entrada
while (nodo_actual != NULL) {
    //¿Otra forma de ciclar?
}
```

¿Que imprime el programa?

```
1 #include <stdio.h>
2 #include <assert.h>
3 int main() {
4     int variable = 25;
5     int *puntero = &variable;
6     printf("%d = %d\n", *puntero, *&variable);
7     assert( x == *&x );
8     return 0;
9 }
```

Una variable siendo referenciada y a la vez siendo
desreferenciada 🤔 🤔

```
1 #include <stdio.h>
2 #include <assert.h>
3 int main() {
4     int variable = 25;
5     int *puntero = &variable;
6     printf("%d = %d\n", *puntero, *&variable); //25
7     assert( x == *&x ); // son equivalentes!
8     return 0;
9 }
```

```
1 int roundup( void ); // declaracion de una funcion
2 int *proundup = roundup;
3 int *pround = &roundup;
4 assert( pround == proundup ); //son equivalentes!
```

```
1 #include <stdio.h>
2 int main() {
3     int a = 5, b = 10;
4     int *p1, *p2;
5     p1 = &a;
6     p2 = &b;
7     *p1 = 10;
8     p1 = p2;
9     *p1 = 20;
10    printf("a = %d\n", a);
11    printf("b = %d\n", b);
12    return 0;
13 }
```

```
1 #include <stdio.h>
2 int main() {
3     int a = 5, b = 10;
4     int *p1, *p2;
5     p1 = &a;
6     p2 = &b;
7     *p1 = 10;
8     p1 = p2;
9     *p1 = 20;
10    printf("a = %d\n", a); // a = 10
11    printf("b = %d\n", b); // b = 20
12    return 0;
13 }
```

Memoria Dinamica

Funciones nuevas

Todas las funciones pertenecen a `<stdlib.h>`

función	descripción
malloc	Agrega memoria del Heap al programa.
calloc	Agrega memoria del Heap y ademas la llena con ceros.
realloc	Realoca memoria ya creada, mayor o menor.
free	Libera un bloque de memoria del Heap

malloc

```
1 void* malloc(size_t)
```

- La función pide como argumento el tamaño (sizeof(type)) del tipo de dato que queremos usar y retorna un **puntero** al bloque de memoria creada.
 - Como es void, se puede asociar a cualquier tipo de dato.
 - asigna la memoria pero NO le inicializa un valor.

```
1 int *pi = malloc(sizeof(int)); //valido y correcto, se asocia a int
2 int *pi = (int*) malloc(sizeof(int)); //😎
```

- Se considera buena practica castear al tipo de dato (Portabilidad con otros compiladores y C++), aunque se debate realmente la necesidad.¹

Rellenar de 0s para luego utilizar

Si los espacios no se rellenan con un valor y se intentan manipular, existiran errores catastróficos.

```
1 int *vector = (int *)malloc(sizeof(int));  
2 printf("%d\n", vector[0]); //-2027859536 🤖
```

```
1 int *vector = (int *)malloc(sizeof(int));  
2 vector[0] = 0;  
3 printf("%d\n", vector[0]); // 0 😎
```

Buena practica revisar si la memoria fue asignada

```
1 int *pi = (int*) malloc(sizeof(int));
2 if(pi != NULL) {
3     // Puntero deberia estar OK y se puede usar
4 } else {
5     perror("Puntero no inicializado correctamente");
6     return(-1)
7     // perror permite a otros programar entender un estado de salida
8     // en vez de unicamente imprimir, muy util.
9 }
```

Generar espacio para mas de 1 elemento

- malloc(cantidad * type_de_dato)

```
1 int size;
2 printf("Ingrese la cantidad de elementos del arreglo:");
3 scanf("%d",&size);
4 int *arreglo = (int *)malloc(5 * sizeof(int));
5 //arreglo con espacio declarado para 5 enteros!
```

Ya que es un puntero a la primera posición, se puede
Desreferenciar y usar aritmética de punteros.

```
1 int *vector = malloc(2 * sizeof(int));
2 vector[0] = 125;
3 vector[1] = 10;
4 printf("%d\n", vector[0]); // 125
5 printf("%d\n", (*(vector+1))); // 10
```

calloc

```
void *calloc(size_t numElements, size_t elementSize);
```

- En vez de únicamente pedir el tamaño del tipo de dato, pide además la cantidad.
 - Aloja la memoria y además la "limpia", quedan todos los valores en un estado de asignación y además declaración.

Formas para replicar el comportamiento de calloc

```
1 int i;
2 int *arreglo = (int*)malloc(3 * sizeof(int));
3 for(i = 0; i<3;i++){ arreglo[i]=0; }
4 for(i = 0; i<3;i++){ printf("%d",arreglo[i]); } //000
```

```
1 int *pi = malloc(5 * sizeof(int));
2 memset(pi, 0, 5* sizeof(int)); // no es parte del curso
```

```
1 int i;
2 int *arreglo = (int *)calloc(3, sizeof(int));
3 for(i = 0; i<3 ; i++){printf("%d", arreglo[i]);} //000
```

Calloc ahorra la asignacion para cada elemento.

- Mucho mas seguro para llegar y usar.
- Ya que ademas de asignar, declara, calloc es significativamente mas lento que malloc.

La asignacion de memoria es un tema gigante y muy importante en nuestro mundo digital, existen funciones mas recientes y con comportamiento definido. [Win32API](#)

realloc

```
1 void *realloc(void *ptr, size_t size);
```

Toma como parámetros un puntero a un bloque de memoria y además el tamaño nuevo.

- Solamente usar en espacios de memoria alocados por calloc o malloc
- Retorna un puntero al espacio nuevo asignado.
- Dependiendo del tamaño nuevo, devolverá la dirección original o una dirección totalmente nueva



```
1 ptr = malloc(sizeof(int));
2 ptr1 = realloc(ptr, count * sizeof(int));
3 if (ptr1 == NULL) // ptr1 realocado
4 {
5     printf("\nSaliendo!!");
6     free(ptr);
7     exit(0);
8 }
9 else
10 {
11     ptr = ptr1; // si no se vacio el original, se lo asignamos
12 }
```

Primer Parametro	Segundo parametro	Comportamiento
NULL	N/A	igual que malloc.
Not NULL	0	El bloque de memoria apuntado es liberado(free).
Not NULL	tamaño menor que el original	Se usa la misma direccion de memoria pero con menor tamaño alocado.
Not NULL	tamaño mayor que el original	Se crea una una nueva direccion de memoria en el Heap.

Memory leaks

Codigo con ciclo infinito, **⚠️ NO EJECUTAR ⚠️**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <windows.h>
4 #include <psapi.h>
5 int main() {
6     char *chunk;
7     PROCESS_MEMORY_COUNTERS pmc; //contador de bytes
8     // no es materia del curso
9     while (1) {
10         chunk = (char*)malloc(1000);
11         printf("Cantidad de bytes alocados:%lu bytes\n", pmc.WorkingSetSize);
12     }
13     return 0;
14 }
```

Name	Private bytes
 opera.exe	1.63 GB
 a.exe	1.29 GB

Luego de 5 segundos de ejecucion, llego a casi usar la misma cantidad de memoria que todo mi navegador.

Perdida de bloques de memoria

```
1 int *pi = (int*) malloc(sizeof(int));
2 *pi = 5;
3 //mas abajo en el codigo...
4 pi = (int*) malloc(sizeof(int));
```

El estado original se **PIERDE** y **NO** se puede recuperar.

Ahora esa memoria queda flotando en el programa.

```
1 char *name = (char*)malloc(strlen("Susan")+1);
2 strcpy(name, "Susan");
3 //strcpy copia un string en otra variable. <stdio.h>
4 while(*name != 0) {
5     printf("%c", *name); //Susan
6     name++;
7 }
```

Debido a la aritmética de punteros, al aumentarlo se mueve su dirección, saliendo del while se perdió la dirección inicial y **NO** se puede recuperar. (*Dangling pointers*)

free memoria

```
1 void free(void *ptr);
```

- La forma de evitar problemas es eliminarlos de memoria con `free(puntero)`.
- Solo usar con memoria asignada con `malloc` o `calloc`
 - El puntero seguira "apuntando" a esa direccion, pero los contenidos se consideran basura.
 - Se puede realocar este espacio de memoria liberado.

```
1 int *pi = (int*) malloc(sizeof(int));  
2 //mas abajo en el codigo despues de usar el puntero  
3 free(pi);
```

- Se debe evitar manejar memoria de manera global, lo ideal es hacerlo todo dentro de un scope(funcion), Alocar y liberar dentro de la funcion.

¿NULL es liberado?

```
1 int *pi = (int*) malloc(sizeof(int));  
2 free(pi);  
3 pi = NULL;
```

Para evitar *dangling pointers* se pensaría que esto es válido.

- Si se intenta usar *pi* nuevamente, el código genera un runtime exception¹

Si se usa *free* en un puntero ya liberado, puede ocasionar estados no deseados o corrupciones.

[1]: runtime exception son errores que suceden dentro de la **ejecución** del programa, **NO** en compilación

El manejo de memoria es difícil

FEBRUARY 26, 2024

Press Release: Future Software Should Be Memory Safe



▶ [ONCD](#)

▶ [BRIEFING ROOM](#)

▶ [PRESS RELEASE](#)

**Leaders in Industry Support White House Call to Address Root Cause of
Many of the Worst Cyber Attacks**

White House Press Release

Matrices dinamicas

Crear una matriz de espacio dinámico con el tamaño como parámetros de entrada

```
1  int **crearMatriz(int filas, int columnas) {
2      int **matriz = (int **)malloc(filas * sizeof(int *));
3      for (int i = 0; i < filas; i++) {
4          matriz[i] = (int*)malloc(columnas * sizeof(int));
5      }
6      // doble malloc para asignar a cada fila memoria
7      // Arreglo de arreglos dinamicos!
8      /* rellenar matriz memoria con ceros */
9      for (int i = 0; i < filas; i++) {
10         for (int j = 0; j < columnas; j++) {
11             matriz[i][j] = 0;
12         }
13     }
14     return matriz;
15 }
16
17 int **Matrizuno = crearMatriz(filasA, columnasA);
18 //requiere asignacion!!!
```

- Ya que la matriz es una dirección de memoria doble, se retorna como doble puntero!

Modificar valor de una posición de la matriz por Referencia

```
1 void modificarPosicion(int***matriz, int i, int j, int valor){  
2     *matriz[i][j] = valor;  
3 } //cambio de valor por referencia puntero
```


Multiplicar 2 matrices

```
1 int **mult_matriz(int **Muno, int **Mdos, int filasA,
2                   int columnasA, int filasB, int columnasB){
3     if (columnasA != filasB) return NULL;
4     int **matriz = crearMatriz(filasA, columnasB);
5     for (int i = 0; i < filasA; i++)
6     {
7         for (int j = 0; j < columnasB; j++)
8         {
9             int suma = 0;
10            for (int k = 0; k < columnasA; k++)
11            {
12                suma = suma + (Muno[i][k] * Mdos[k][j]);
13            }
14            matriz[i][j] = suma;
15        }
16    }
17    return matriz;
18 }
19
20
21 int **matrizC = mult_matriz(Matrizuno, Matrizdos, filasA, columnasA, filasB, co
```

Arreglo dinamico simple

```
1 int *crear_arreglo(int n) {
2     int *arr = (int *)malloc(sizeof(int) * n);
3     // rellenar con ceros
4     for (int i = 0; i < n; i++) {
5         arr[i] = 0;
6     }
7     return arr;
8 }
9 fscanf(fp, "%d", &cantidad_elementos);
10 int *arreglo = crear_arreglo(cantidad_elementos);
11 // leo de un archivo (fp) el primer elemento y lo asocio...
```

```
1 void cambiar_datos(int **arr, int posicion, int cambio) {
2     (*arr)[posicion] = cambio;
3     //aritmética de punteros
4 }
```

Función en C que replique como funciona append en python

```
1 int *append(int *arreglo, int *n, int numero_a_agregar) {
2     int i;
3     int *nuevo_arreglo = (int *)malloc(sizeof(int) * (*n + 1));
4     if (nuevo_arreglo == NULL) {
5         return NULL;
6     }
7     for (i = 0; i < *n; i++) {
8         nuevo_arreglo[i] = arreglo[i];
9     }
10    nuevo_arreglo[i] = numero_a_agregar;
11    *n = *n + 1;
12    if (arreglo != NULL) {
13        free(arreglo);
14    }
15    return nuevo_arreglo;
16 }
```

[1]:Para el ejercicio 3 usar un arreglo secundario completamente aleatorio, quizá `crear_arreglo_random(int*arreglo)` usando `srand()`, es parte de `stdlib.h`

Ejercicio para practicar

Dado un archivo de solo ints, donde su primer valor es el tamaño, crear las siguientes funciones dinámicas y usando punteros:

1. `crear_arreglo(int size)` con `calloc`
2. `reverse_arreglo(int*arr)` usando aritmética de punteros
3. `combinación_arreglos(intarr1, intarr2)` combinar 2 arreglos y dejar los elementos intercalos.¹

```
1 arr[] = {1,3,5,7};
2 arr2[] = {2,4,6,8};
3 arr3 = combinacion_arreglos(arr,arr2);
4 arr3 //{1,2,3,4,5,6,7,8}
```

Ejercicios PSEUDO CODIGO

Dado 2 arreglos A y B, escribir un algoritmo que verifique si el arreglo B es igual al inverso de A.

En caso de no serlo debe entregar la cantidad de posiciones donde NO coinciden los elementos

```
1  int inverso(int A[], int B[])
2      int n = largoArreglo(A[])
3      int i = 0
4      int j = n-1
5      for (i = 0 to n-1)
6          if (A[j] != B[i])
7              contador++ //contador = contador + 1
8          end if
9          j--
10     end for
11     if (contador == 0)
12         print("son inversos")
13         return 0
14     else
15         print("Los arreglos no son inversos")
16         return contador
17     end if
```


Encontrar la mayoría simple de un arreglo y retornarlo si es que existe.

Mayoría simple = elemento que se repite mas de $n//2$ veces

```

1  int mayoriaSimple(int arr[], int size)
2      int maxCount = 0
3      int index = -1 //ningun elemento
4      for(int i = 0; i < size; i++)
5          int count = 0
6          for(j to size-1)
7              if(arr[i] == arr[j])
8                  count++
9              end if
10         end for
11         if(count > maxCount)
12             maxCount = count
13             index = i
14         end if
15     end for
16
17     if (maxCount > size/2)
18         return arr[index]
19     else

```

Orden(n^2)

```

1  int mayoriaSimple(int arr[], int size)
2      int mayor = 0 // candidato a mayorSimple
3      int contador = 0 // contador del candidato
4      //buscar candidato a mayor
5      for (i = 0 to size-1)
6          if (contador == 0)
7              mayor = arr[i]
8              contador = 1
9          else if (mayor == arr[i])
10             contador++
11         else
12             contador--
13     end if
14 end for
15 //opcional, se verifica que en verdad sea el candidato
16 for (i = 0 to size -1)
17     if (arr[i] == mayor)
18         contador++
19     end if

```

Si no se hace la verificacion, queda de Orden(n)!

Siguiente ayudantia

- Falta solamente ver structs para el lab 1
 - Lo mas seguro es que sea pre-grabada.

Actividad 1 Laboratorio	Semana 4 (Lunes 8/Abr)	Unidad I
-------------------------	------------------------	----------

- Subiré otro vídeo de un ejercicio que abarca todo antes del lab 1.
- El concepto de punteros se aplicara el resto del semestre, ahora es el mejor momento para entenderlo.

Referencias

- [open pubs functions](#)
- [glibc wiki](#)
- [cprogramming](#)
- [Understanding and Using C Pointers](#) (recomendado 100)
- [gcc wiki](#)
- [C Programming Notes, Chapter 11](#)
- [Learning C++ Pointers for REAL Dummies](#)
- [Intro a C](#)
- [C FAQ](#)

