

Alojamiento en Memoria dinámica

Al programar con arreglos, se estiman sus tamaños en tiempo de programación; esto tiene dos desventajas en tiempo de ejecución:

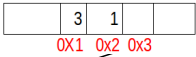

- 1) Si el tamaño es grande, no se usa todo y se desperdicia memoria cara.
- 2) Si el tamaño es pequeño, es excedido y usamos indebidamente áreas vecinas, y no somos advertidos, lo más probable es que los resultados sean errados, o que el programa aborte con el error **violación de segmento**.

Superamos estas dos dificultades, asignando el tamaño exacto de los arreglos en tiempo de ejecución, si es necesario podemos cambiarlo; a esto se le llama alojamiento de memoria dinámica; para lo cual requerimos de funciones contenidas en la librería:

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

Alojar memoria dinámica en tiempo de ejecución

Al momento de cargar el programa en la memoria RAM para ser ejecutado, conceptualmente sucede lo siguiente:

Programa fuente	Memoria RAM		
	Programa ejecutable	datos estáticos	Piscina de memoria dinámica
<pre>Int a = 3, b = 1, c; c = a + b; printf("%d\n", c);</pre>	<ol style="list-style-type: none"> 1) Ubica las variables a, b y c, definidas en el programa, en la memoria de datos estáticos. 2) Reemplaza las variables por las direcciones en el programa fuente, el ejecutable queda así: $*0x3 = *0x1 + *0x2;$ <pre>printf("%d\n", *0x3);</pre> 		

En tiempo de ejecución, podemos definir bloques de datos en la piscina de memoria dinámica de acuerdo a las necesidades del momento.

Al alojar memoria dinámica, se retorna un puntero al inicio de dicha memoria, nada más; el programador maneja el tamaño de dicha memoria en modo similar al control en arreglos.

En tiempo de ejecución, las variables de una función son alojadas en la RAM al momento de llamarla, al terminar su ejecución, son desalojados inmediatamente para ahorrar memoria, de este modo, los espacios alojados son **volátiles**; en cambio la memoria dinámica alojada es **persistente**, es decir no importa donde sea definida, esta memoria permanece en la RAM hasta que termine el programa o sea desalojada con la función **free()**.

Funciones para alojar memoria dinámica

Las funciones que nos permiten crear, modificar y eliminar los bloques de datos son:

Descripción	Sintaxis	Ejemplo
Aloja un bloque de memoria en bytes.	void * malloc(size_t num); Entrada: num de bytes a ser alojados Proceso: Aloja num bytes en la memoria. Salida: un puntero void *, el cual apunta al primer byte de la memoria alojada; si no es posible alojar se retorna la constante simbólica NULL	<pre>// Alojar memoria para 5 enteros int *ptr, i; i = 5* sizeof(int); ptr = (int *)malloc(i); // note el casting (int *) a puntero int if (ptr == NULL) { // if (!ptr) {...} printf("No hay suficiente memoria"); exit(EXIT_FAILURE); } ...</pre>
Aloja un bloque de memoria (en bytes) para datos de tipo específico.	void * calloc (size_t num, size_t tam); Entrada: num : de elementos a ser alojados tam : tamaño (en bytes) de cada elemento Proceso: Aloja num posiciones de tamaño tam (en bytes) e inicializa en cero los elementos alojados. Salida: un puntero (void *) que apunta al primer byte de la memoria alojada; si no es posible alojar se retorna la constante simbólica NULL	<pre>// Alojar memoria para 5 enteros int *ptr; ptr = (int *)calloc(5, sizeof(int)); // sizeof(int) = 4 if (ptr == NULL) { // if (!ptr) {...} printf("No hay suficiente memoria"); exit(EXIT_FAILURE); } ...</pre>
Puede alojar memoria, indistintamente, con malloc() o calloc()		
Cambia el tamaño de un bloque de memoria previamente asignado con malloc() o calloc()	void * realloc(void *ptr, size_t num); Entradas: ptr : puntero que apunta al bloque de datos a ser realojado num : número de bytes a ser alojados	<pre>// Existe el puntero ptr que apunta a un bloque que va a redimensionarse a 8 enteros. int i = 8* sizeof(int), *ptr=NULL; ptr = (int *)realloc(ptr, i); if (ptr == NULL) { // if (!ptr) {...}</pre>

	Proceso: ver debajo de esta tabla Salida: puntero (void *) que apunta al primer byte de la nueva memoria alojada; si no es posible alojar se retorna la constante simbólica NULL	printf("No hay suficiente memoria"); exit(EXIT_FAILURE); } ...
Liberar memoria apuntada por ptr	void free (void *ptr); Entrada: ptr : apunta a la memoria a liberar Proceso: Libera la memoria apuntada por ptr .	// ya existe el puntero ptr apuntando a la memoria: free (ptr);

Proceso de realloc():

- 1) Si es posible ampliar la memoria apuntada por **ptr**, la amplía y retorna el mismo **ptr**.
- 2) Si NO hay suficiente espacio contiguo para ampliar el bloque actual, un nuevo bloque de **tam** (bytes) es asignado, los datos existentes se copian al nuevo bloque, el bloque antiguo es liberado y la función retorna un puntero al nuevo bloque.
- 3) Si **ptr** es NULL, la función actúa como malloc(), asigna un bloque de **tam** bytes y retorna un puntero a él.
- 4) Si la memoria es insuficiente para la nueva asignación, la función retorna NULL, el bloque original se mantiene apuntado por ptr

Arreglo de una dimensión utilizando punteros y alojamiento de memoria

En lugar de declarar y usar arreglos de una dimensión, trabajaremos con apuntadores.

```

/* 08_01.c: Ingresar temperaturas diarias una por una, luego de ingresar una temperatura, muestre todas las
temperaturas ingresadas anteriormente; termina cuando se ingresa temperatura 0.
*/
#include <stdio.h>
#include <stdlib.h>
int *leer (int *n, int *pt);    // Lee las temperaturas y las almacena en la memoria dinámica, termina con 0
void escribir(int n, int *pt); // Escribe las temperaturas desde la memoria dinámica.
void main(void) {
    int n = 0, *pt = NULL;      // se asigna NULL, porque realloc(pt ..) lo utilizará en leer()
    while(pt = leer(&n, pt)) escribir(n, pt); // pt es entrada y salida de leer()
    printf("Valores ya ingresados: ");
    while(pt < pmax) printf("\n%d", *pt++);
    free(pt);                  // libera la memoria
}
int *leer(int *n, int *pt){
    int t;
    printf("\nIngrese una temperatura, <para terminar: 0>: ");
    scanf("%d",&t);            // Lee una temperatura
    if(t==0) return NULL;
    (*n)++;
    pt = realloc(pt, *n*sizeof(int)); // aloja la memoria necesaria
    if(pt==NULL) {              // cancela el programa
        printf("No hay suficiente memoria");
        exit(EXIT_FAILURE);
    }
    *(pt+*n-1) = t;             // guarda la temperatura ingresada, puede usar pt[*n-1] = t;
    return pt;
}
void escribir(int n, int *pt){ // imprime la lista de temperaturas ingresadas
    int *pmax = pt+n;
    printf("Valores ya ingresados: ");
    while(pt < pmax) printf("\n%d", *pt++);
}

```

Salida:

Ingrese una temperatura, <para terminar: 0>: 1

Valores ya ingresados:

1

Ingrese una temperatura, <para terminar: 0>: 2

Valores ya ingresados:

1

2

Ingrese una temperatura, <para terminar: 0>: 3

Valores ya ingresados:

1
2
3

Ingrese una temperatura, <para terminar: 0>: 0

Ejercicio: Resuelve el mismo problema anterior, pero utilizando un arreglo. Será más fácil, pero no es flexible. El programador estima el tamaño **n** del arreglo para guardar las temperaturas y no debe sobrepasarlo:

- Si el número de temperaturas ingresadas por el usuario < **n**: se desperdicia memoria.
- Si el número de temperaturas ingresadas por el usuario > **n**: el programa podría ejecutar para pequeñas excesos, pero pronto dará resultados erróneos y/o un error de memoria sobrepasada (violación de segmento).
- Este problema se agrava si usa más de una dimensión y peor aún si utiliza funciones (a partir de 2 dimensiones); en cambio con punteros siempre se garantiza la flexibilidad (parametrización) aunque la programación sea un poquito más compleja.

Reserva de espacio para arreglos: Estática versus Dinámica

Reserva estática	Reserva dinámica y punteros
<pre>int arr[200][300]... ;</pre> <p>// El programador <u>estima</u>, por exceso, el número de elementos.</p>	<pre>int *parr;</pre> <p>No se define el número de elementos. el tamaño total del bloque es controlado y cambiado en tiempo de ejecución dependiendo de la dinámica (interacción con el usuario).</p>
El número total de elementos se guarda en otra variable	El número total de elementos se guarda en otra variable; el apuntador, solo apunta al primer elemento del bloque; no guarda el tamaño de la memoria reservada.
El área de memoria ocupada por un arreglo se libera al salir de la función que lo contiene o al terminar el programa.	El área de memoria apuntada por un puntero esta activa hasta que sea explícitamente liberada con free() o al terminar el programa.
<p>Un arreglo declarado dentro del cuerpo de una función:</p> <pre>#define TAM 10</pre> <pre>int arre[TAM]; // por ejemplo</pre> <p>No puede ser usado fuera de ella.</p>	<p>Un bloque declarado dentro del cuerpo de una función:</p> <pre>int *p; p = (int *) malloc(n * sizeof(int));</pre> <p>Si puede ser usado fuera de la función:</p> <p>Use el puntero p</p> <p>No libere la memoria apuntada por p.</p>
Es simple	Es un poquito más compleja

Arreglos multidimensionales utilizando punteros y alojamiento dinámico de memoria

Veamos un ejemplo para un arreglo de 2 dimensiones. Utilice funciones para leer e Imprimir una matriz de dos dimensiones, el programa debe ser paramétrico, cosa que no se puede, si utilizamos arreglos, ejemplo, considere la matriz:

```
10 20 30
40 50 60
```

```
// 08_02a.c: Leer, alojar en memoria dinámica e imprimir matrices utilizando puntero
#include<stdio.h>
#include <stdlib.h>
void leerMN(int *m, int *n);
int *leerP(int m, int n);
void imprimir(int m, int n, int *p);
void main(void){
    int m, n, *p ;
    printf("\nUso de punteros y funciones\n");
    leerMN(&m, &n);
    p = leerP(m, n);
    imprimir(m, n, p);
    free(p);
}
void leerMN(int *m, int *n){
    printf("\nIngrese el número de elementos de la primera dimensión: ");
    scanf("%d", m);
    printf("\nIngrese el número de elementos de la segunda dimensión: ");
    scanf("%d", n);
}
int *leerP(int m, int n){
    int i, j, *p, *pp;
    pp = p = malloc(m*n*sizeof(int));
    if(p==NULL) {printf("No se pudo alojar la matriz en memoria"); exit(EXIT_FAILURE);}
// Lectura de datos
    printf("\n");
```

```

    for (i=0; i < m; i++)
        for (j=0; j < n; j++){
            printf("Para la fila %d, ingrese el número %d: ", i+1, j+1);
            scanf("%d", pp++);
        }
    return p;
}

void imprimir(int m, int n, int *p){
    int i, j, *pp = p;
    printf("\nMatriz cargada:\n");
    for (i=0; i<m; i++){
        for (j=0; j<n; j++) printf(" %d\t", *pp++);
        printf("\n");
    }
}

```

Salida:

```

Matriz cargada: // datos ingresados
10 20 30
40 50 60

```

Arreglo multidimensional utilizando puntero a puntero y alojamiento dinámico de memoria

Podemos recorrer arreglos de diferentes formas equivalentes, se elige la que satisface criterios de optimidad: solución de problema, velocidad, manejo de memoria y claridad y simplicidad del programa.

Recorrido con arreglos	Recorrido con apuntadores		
	Equivalente a arreglos	Apuntador suele ser más rápida	Otras, por ejemplo: arreglos de apuntadores, apuntador a apuntador
scanf("%d", &p[i][j]);	scanf("%d", p+i*n+j);	scanf("%d", p++);	
printf(" %d\t", p[i][j]);	printf(" %d\t", *(p+i*n+j));	printf(" %d\t", *p++);	

Resolveremos el mismo problema anterior almacenando información en dos bloques de memoria:

- 1) Un apuntador a apuntador que apunta a un arreglo de apuntadores: las filas de una matriz
- 2) Un apuntador a cada bloque de datos por fila de matriz

Representación de una matriz (mat) por un arreglo de apuntadores (p):

Sea la matriz mat:

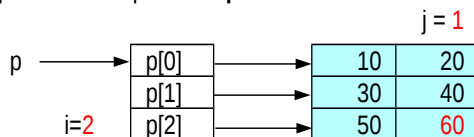
j = 1

10	20
30	40
50	60

i = 2

mat[2][1] = 60

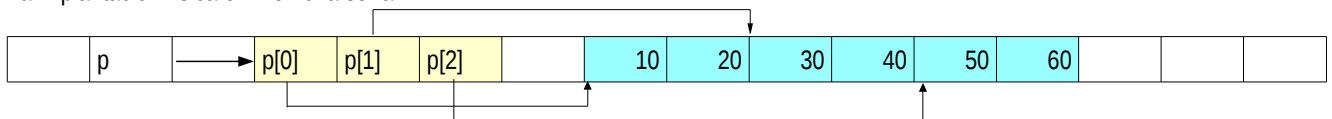
Apuntador de apuntador p:



p apunta al primer elemento de un bloque de apuntadores.

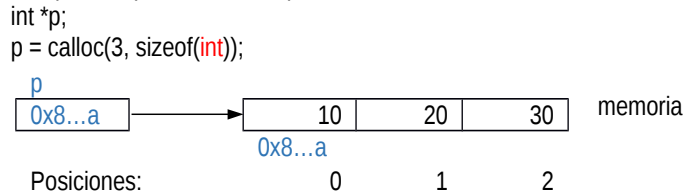
Cada elemento **p[i]** apunta al primer elemento de una fila i de la matriz (la cual es un bloque de 2 enteros).

La implantación física en memoria sería:

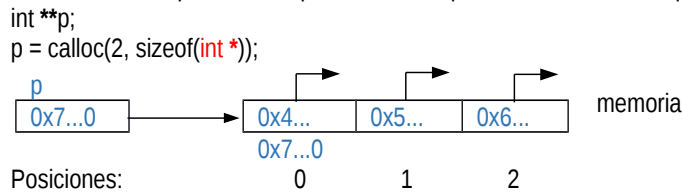


Atento a las definiciones de bloques de memoria:

Un apuntador puede apuntar a un bloque de datos reservado dinámicamente:



Lo mismo sucede con un apuntador a apuntador, sólo que en este caso el bloque reservado es de apuntadores:



Observe que cada posición del bloque anterior es del tipo `int *`, es decir, un apuntador a entero.

Instrucciones de programación a utilizar:

Instrucciones	Descripción
<code>int **p;</code>	Define un apuntador de apuntador: p .
<code>p = calloc(3, sizeof(int *));</code> // tamaño del arreglo = número de filas de la matriz (3).	Aloja memoria para p , el cual apuntará a un bloque de 3 (= número de filas) elementos de tipo apuntador .
<code>int i, j;</code> <code>for(i=0; i<3; i++) p[i] = calloc(2, sizeof(int));</code>	Aloja memoria para p[i] , cada elemento apunta a un bloque de 2 (= número de columnas) elementos de tipo int .
<code>for (i=0; i<3; i++)</code> <code>for (j=0; j<2; j++) scanf("%d", &p[i][j]);</code>	Lee datos para las posiciones de tipo int . Note la notación matricial

Liberar memorias reservadas:

```
for(i=0; i<3; i++) free(p[i]); // Libera cada elemento p[i].
free(p); // Libera p
```

Ejemplo: Utilice funciones para leer e Imprimir una matriz de dos dimensiones (este es mismo problema anterior); pero utilizaremos puntero a puntero.

```
10 20 30
40 50 60
```

```
// 08_02b.c: Leer, alojar en memoria dinámica e imprimir matrices utilizando puntero a puntero
#include<stdio.h>
#include <stdlib.h>
void leerMN(int *m, int *n);
int **leerP(int m, int n);
void imprimir(int m, int n, int **p);
void liberar(int m, int **p);
void main(void){
    int m, n, **p;
    printf("\nUso de punteros y funciones\n");
    leerMN(&m, &n);
    p = leerP(m, n);
    imprimir(m, n, p);
    liberar(m, p);
}
void leerMN(int *m, int *n){
    printf("\nIngrese el número de elementos de la primera dimensión: ");
    scanf("%d", m); // m es una referencia a memoria
    printf("\nIngrese el número de elementos de la segunda dimensión: ");
    scanf("%d", n);
}
int **leerP(int m, int n){
    int i, j, **p;
    // Alojamiento de memoria
    p = malloc(m*sizeof(int *));
    if(p==NULL) {printf("No se pudo alojar la matriz en memoria"); exit(EXIT_FAILURE);}
    for (i=0; i < m; i++) {
```

```

    p[i] = malloc(n*sizeof(int));           // p[i] apunta a la memoria alojada
    if(p[i]==NULL) {printf("No se pudo alojar la matriz en memoria"); exit(EXIT_FAILURE);}
}
// lectura de datos
printf("\n");
for (i=0; i < m; i++){
    for (j=0; j < n; j++){
        printf("Para la fila %d, ingrese el número %d: ", i+1, j+1);
        scanf("%d", &p[i][j]);
    }
    return p;                             // se retorna el valor del puntero p
}
void imprimir(int m, int n, int **p){
    int i, j;
    printf("\nMatriz cargada:\n");
    for (i=0; i<m; i++) {
        for (j=0; j<n; j++) printf(" %d\t", p[i][j]);
        printf("\n");
    }
}
void liberar(int m, int **p){
    int i;
    for (i=0; i < m; i++) free(p[i]);
    free(p);
}

```

Resumen del uso de apuntadores y memoria dinámica:

Usted ya debe dominar todas las pequeñas diferencias al usar todos los tipos de variables y sus modificadores: *, **, &, y []; utilizadas en: la main(), las funciones, y los prototipos. **Esto no se aprende de memoria**; en el capítulo de funciones establecimos que se coordina el número, orden, tipo y modificadores de los datos; deben ser iguales en los 4 lugares de uso; supongamos la función: **float** miFun(**int** m), se debe coordinar en el orden siguiente:

- 1) En la función que llama, al definir variables : int **m**; float **n**; // m es de tipo int, n float
- 2) En la función que llama, al llamar : miFun(**m**); **n** = miFun(2); // el argumento es de tipo int, y el retorno es float
- 3) Al definir la función : float miFun(int m){ ... } // el argumento es de tipo int, y el retorno es float
- 4) Al definir el prototipo : float miFun(int m); // el argumento es de tipo int, y el retorno es float

Resumen del uso de apuntadores en el programa 08_02.c:

Operación de una matriz m x n		
Operación	Modo	
	Utilizando puntero: 08_02a.c	Utilizando puntero a puntero: 08_02b.c
Usando malloc() o calloc()	<pre> void main(void){ int m, n, *p; leerMN(&m, &n); p = leerP(m, n); imprimir(m, n, p); liberar(p); // libera memoria } </pre>	<pre> void main(void){ int m, n, **p; leerMN(&m, &n); p = leerP(m, n); imprimir(m, n, p); liberar(m, p); // libera memoria } </pre>
	<pre> int *leerP(int m, int n){ int i, j, *p; // Alojar memoria p = malloc(m * n * sizeof(int)); if(p==NULL) {... exit(...);} // Leer datos scanf("%d", p++); // int *pp = p; return p; } </pre>	<pre> int **leerP(int m, int n){ int i, j, **p; // Alojar memoria p = malloc(m * sizeof(int *)); if(p==NULL) {... exit(...);} for (i=0; i < m; i++){ p[i] = malloc(n * sizeof(int)); if(p[i]==NULL) {... exit(...);} } // Leer datos scanf("%d", &p[i][j]); return p; } </pre>
Usando realloc()	<pre> void main(){ int m, n, *p = NULL; leerMN(&m, &n); } </pre>	<pre> void main(void){ int m, n, **p = NULL; leerMN(&m, &n); } </pre>

	<pre> p = leerP(m, n, p); } </pre>	<pre> p = leerP(m, n, p); } </pre>
	<pre> int *leerP(int m, int n, int *p){ p = realloc(p, m * n * sizeof(int)); // Leer datos scanf("%d", pp++); // int *pp = p; return p; } </pre>	<pre> int **leerP(int m, int n, int **p){ p = realloc(p, m*sizeof(int *)); for (i=0; i < m; i++) { p[i] = NULL; p[i] = realloc(p[i], n*sizeof(int)); if(p[i]==NULL) {... exit(...);} } // Leer datos scanf("%d", &p[i][j]); return p; } </pre>
Mostar datos	<pre> void imprimir(int m, int n, int *p){ int i,j; printf(" %d\t", *pp++); } </pre>	<pre> void imprimir(int m, int n, int **p){ int i,j; printf(" %d\t", p[i][j]); } </pre>
Liberar memoria	<pre> void liberar(int *p){ free(p); } </pre>	<pre> void liberar(int m, int **p){ int i; for (i=0; i < *m; i++) free(p[i]); free(p); } </pre>

```

// 08_03.c Leer una cadena de caracteres de cualquier longitud
#include<stdio.h>
#include<stdlib.h>
char *ingresar(int *n);
void main(void) {
    char *p;
    int n;
    printf("ingrese una cadena de caracteres de cualquier longitud: ");
    p = ingresar(&n);                // n = numero de caracteres leídos
    printf("%d caracteres: %s\n", n, p);
}
char *ingresar(int *n){
    // *n = número de caracteres leídos; *p = apuntador a la cadena
    *n = 0;
    int ch;
    char *p;
    p = malloc(1);
    while((ch=getchar())!=10) {
        *(p+*n)= ch;
        (*n)++;
        p = realloc(p, *n);
        exit(EXIT_FAILURE);}
    *(p+*n) = '\0';
    return p;
}

```

Ejercicios

- 1) Utilice alojamiento dinámico. Lea del teclado una matriz A[m][n] y defina otra B[n][m], que será transpuesta de A, e imprímalas a las dos.

Sugerencia: supongamos: A[2][3] = {1,2,3,4,5,6}

NO se definirá ni A[2][3] ni B[3][2] porque estamos en memoria dinámica; se debe hacer lo siguiente:

int *Pa, *pB; // definir los apuntadores

Alojar 6 espacios int de memoria para pA y 6 para pB.

Asignar valores a A que se alojarán así: 1 2 3 4 5 6 // puede usar las notaciones *pA++ ó pA[i*n+j]

Asignar los valores a B que se alojarán así: 1 4 2 5 3 6 // puede usar la notaciones:
// *pA++ ó pA[i*n+j] *pB++ ó pB[i*n+j]

- 2) Escriba un programa que pida al usuario ingresar temperaturas diarias una por una, termina cuando temperatura = 0, luego de ingresar una temperatura se muestran todas las temperaturas ingresadas anteriormente, ordenadas ascendentemente.
- 3) Utilice alojamiento dinámico y un solo apuntador (*p) para leer del stdio una matriz a[m][n][k] y luego imprímala.
- 4) Utilice alojamiento dinámico y un puntero a puntero a puntero (**p) para leer del stdio una matriz a[m][n][k] y luego imprímala.
- 5) Utilice alojamiento dinámico y un puntero (*p) y luego otro puntero de puntero (**q) para leer del stdio una matriz a[m][n][k] y luego imprímala.