

Fundamentos de Programación

Esaú Eliezer Escobar Juárez

MEMORIA DINÁMICA



Instituto Politécnico Nacional
La Técnica al Servicio de la Patria

Unidad Profesional Interdisciplinaria de
Ingeniería Campus Tlaxcala UPIIT



El lenguaje C y el manejo de la memoria

Todos los objetos tienen un tiempo de vida. En C, existen 3 tipos de duración: estática, automática y asignada.

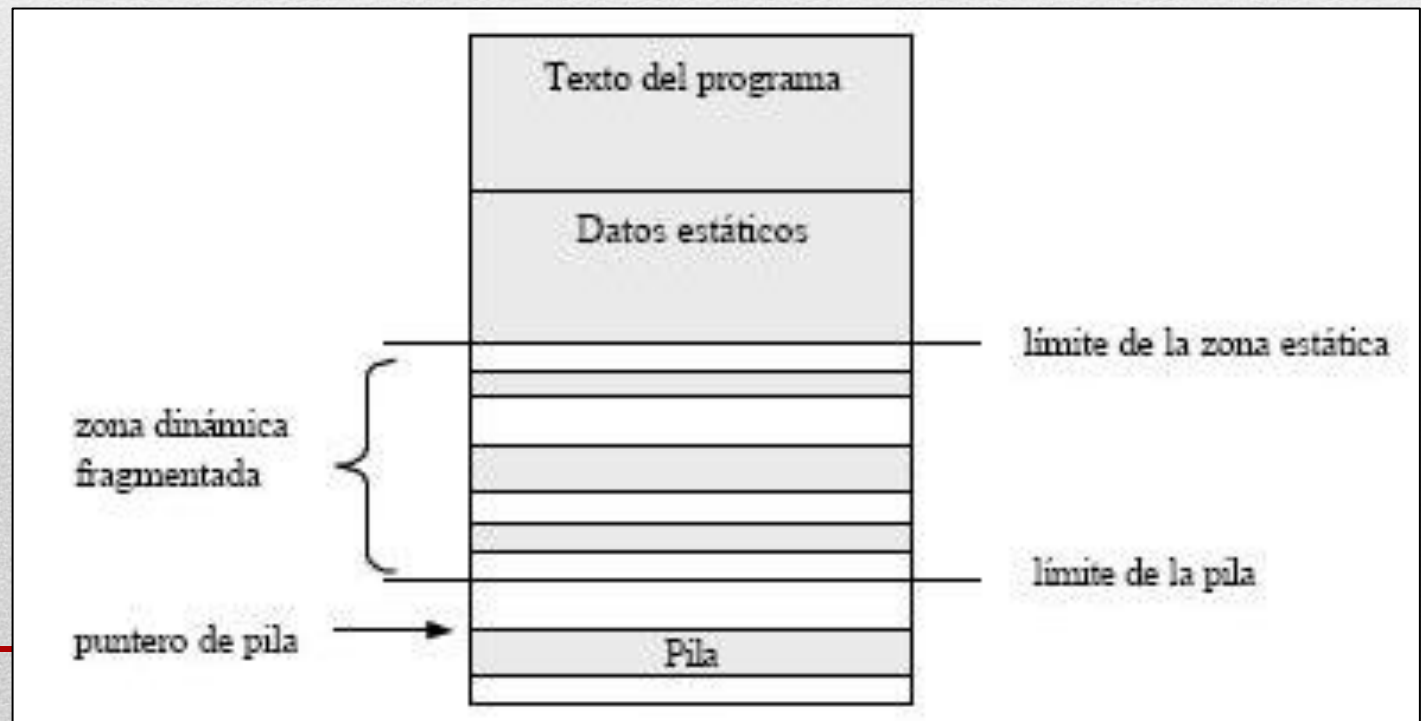
Las variables normales tienen duración automática. Se crean al entrar al bloque en el que fueron declaradas y se destruyen al salir de ese bloque.


Las static tienen duración estática. Se crean antes de que el programa inicie su ejecución y se destruyen cuando el programa termina.

Duración asignada se refiere a los objetos cuya memoria se reserva de forma dinámica.

Para trabajar con datos dinámicos necesitamos dos cosas:

1. Subprogramas predefinidos en el lenguaje que nos permitan gestionar la memoria de forma dinámica (asignación y liberación).
2. Algún tipo de dato con el que podamos acceder a esos datos dinámicos.





Espacio de almacenamiento que se solicita en tiempo de ejecución. A medida que el proceso va necesitando espacio va solicitando más memoria al S.O.

Este tipo de datos se crean y se destruyen mientras se ejecuta el programa, la estructura de datos se va dimensionando de forma precisa a los requerimientos del programa,

Evita perder datos o desperdiciar memoria.

Cuando se crea un programa en el que es necesario manejar memoria dinámica el S.O. divide el programa en cuatro partes: texto, datos (estáticos), pila y una zona libre o heap.

En el momento de la ejecución si no se liberan las partes utilizadas de la memoria y que han quedado inservibles es posible que se “agote” la zona libre.

También la pila cambia su tamaño dinámicamente, pero esto no depende del programador sino del sistema operativo.

La biblioteca estándar de C proporciona las funciones malloc, calloc, realloc y free para el manejo de memoria dinámica. Estas funciones están definidas en el archivo de cabecera `stdlib.h`.

Uso de malloc, sizeof y free

- **malloc()** es la forma más habitual de obtener memoria dinámica. La función asigna un bloque de memoria de acuerdo al número de bytes que se pasen como argumento
 - **malloc()** devuelve un apuntador **void*** al bloque de memoria asignado, hay que realizar un **cast** al tipo requerido.
-

El prototipo de **malloc()** sería:

```
void*=malloc(size_t bytes);
```

Donde:

- **void***: es el apuntador que almacenará la referencia al bloque de memoria asignado.
 - **bytes**: es el tamaño que se quiere reservar.
-

La forma de llamar a la función **malloc()** es:

```
puntero = malloc(tamaño en bytes);
```

Generalmente se hará una conversión al tipo del puntero:

```
tipo *puntero;  
puntero = (tipo *)malloc(tamaño en bytes);
```

Por ejemplo:

```
int *p;  
p = (int *)malloc(4);
```

Al llamar a `malloc()` puede ser que no haya suficiente memoria disponible, entonces, `malloc()` devolverá `NULL` en la operación, por lo tanto, siempre es conveniente preguntar después de la operación si se asignó el bloque de memoria.

```
int *ptr;
ptr = (int *)malloc( 10 * sizeof(int) );
if( ptr == NULL){
    printf("No hay memoria disponible...\n"); //No usar ptr
    return; //Fin del programa o realizar la acción onveniente
}
//Utilizar ptr
```

free():

Libera la memoria reservada.

Prototipo:

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

Donde:

- ***ptr** es el apuntador el bloque asignado.
 - Si **ptr** es un apuntador mal referenciado, **free** probablemente destruya el mecanismo de gestión de memoria y provocará un fallo en el programa.
-

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

int main(void){
    char *c;
    int *entero;
    float *flotante;
    double *doble;

    //Uso de malloc para generar variables sencillas
    c=(char *)malloc(1);
    entero=(int *)malloc(4);
    flotante=(float *)malloc(4);
    doble=(double *)malloc(8);

    *c = 'a';
    *entero = 10;
    *flotante = 3.8917;
    *doble = 1.48365;

    printf("Valores: car %c, entero %d, flotante %f, doble %lf \n\n",
           *c, *entero, *flotante, *doble);

    //Importante liberar la memoria ya no requerida
    free(c);
    free(entero);
    free(flotante);
    free(doble);

    return 0;
}
```

```
//Otra manera de reservar la memoria dinámica
c=(char *)malloc(sizeof(char));
entero=(int *)malloc(sizeof(int));
flotante=(float *)malloc(sizeof(float));
doble=(double *)malloc(sizeof(double));
```

La función sizeof se utiliza con frecuencia para referirnos al tamaño de memoria que se va a generar.

Si queremos reservar un bloque para un arreglo de 10 enteros:

```
int *ptr;
ptr = (int *)malloc(10*sizeof(int));
```

```

#include <stdio.h>
#include <stdlib.h>
int leercantidad();
void leer(int, int *);
void imprimir(int, int *);

main()
{
    int cantidad,*ptr;
    cantidad=leercantidad();
    ptr=(int *)malloc(cantidad*sizeof(int));
    leer(cantidad,ptr);
    imprimir(cantidad,ptr);
}

int leercantidad()
{
    int n;
    printf("Cuantos datos quiere ingresar: ");
    scanf("%d",&n);
    return n;
}

void leer(int n, int *ptr)
{
    for(int i=0;i<n;i++){
        printf("dato [%d]\t",i+1);
        //scanf("%d",&ptr[i]);
        /*scanf("%d",ptr)
        ptr++;*/ Mueve el puntero usando la
aritmetica
        scanf("%d",(ptr+i));//Mueve el puntero
referenciado
    }
}

```

```

void imprimir(int n, int *ptr){
    for(int i=0;i<n;i++){
        printf("dato [%d] %d\n",i+1,*(ptr+i)+1);
    }
}

```

Veamos un ejemplo en el cual el usuario ingresa una cantidad variable de números. Primero el sistema le pide que especifique la cantidad de números que va a ingresar, y luego de ingresarlos, los imprime, incrementando en 1 cada número.

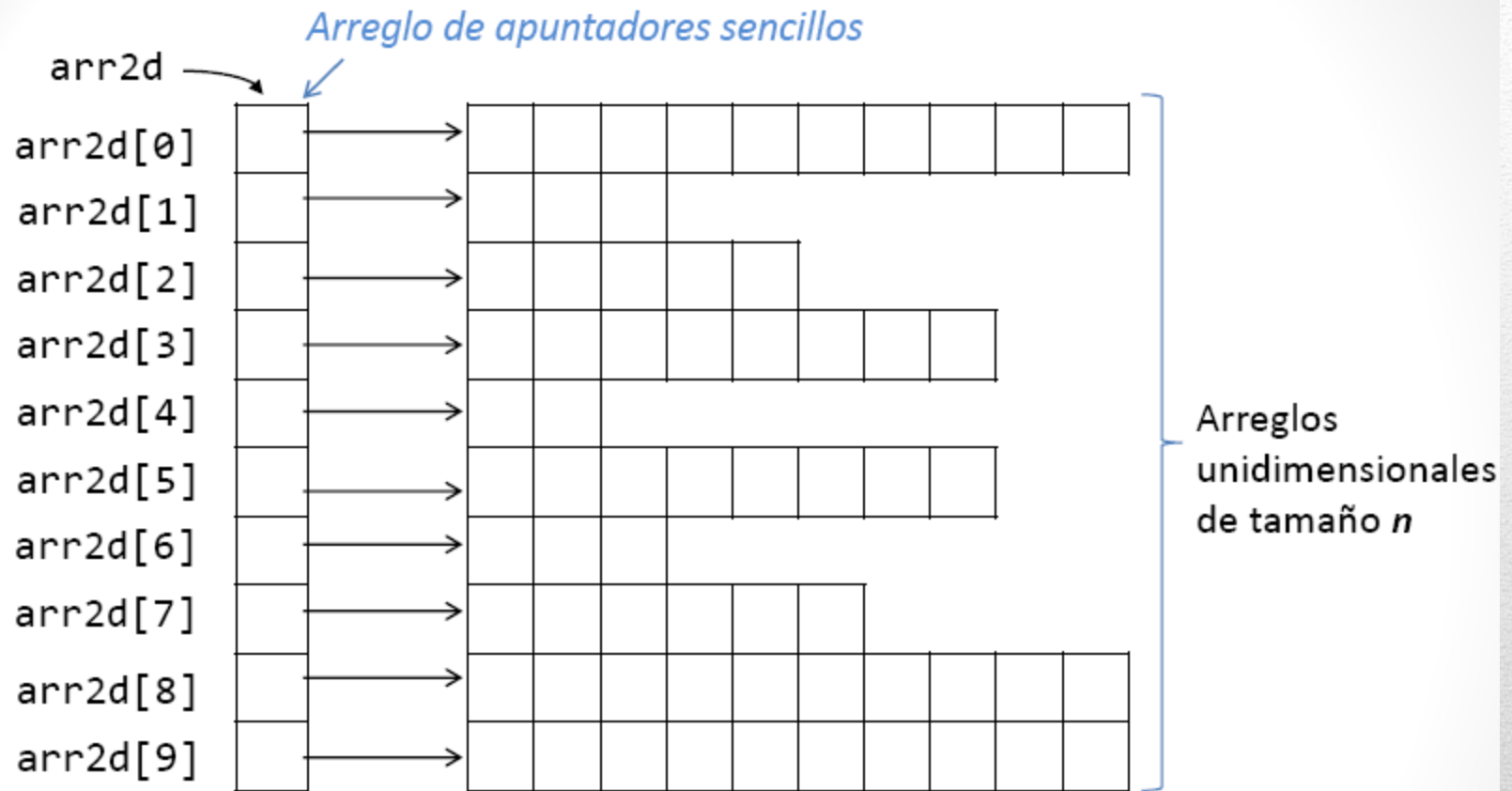
Es importante observar que en este ejemplo se crea un arreglo dinámico.

Importante:

Los apuntadores que se utilizan para hacer referencia al bloque de memoria asignado por malloc() son de tipo dinámico y NO es conveniente que dichos apuntadores apunten a otro lugar antes de liberar el bloque de memoria asignado ya que se estará perdiendo la referencia al bloque de memoria y no abra forma de recuperar la referencia.

Se pueden crear arreglos bidimensional. Al ser el nombre de un array unidimensional un puntero constante, un array bidimensional será un puntero a puntero constante (tipo **).

Para asignar memoria a un array multidimensional, se indica cada dimensión del array de igual forma que se declara un array unidimensional.



Donde cada `arr2d[i]` es un apuntador sencillo que apunta a un arreglo unidimensional de tamaño n .

Para generar al arreglo bidimensional utilizando memoria dinámica se hace en dos pasos:

1. Se solicita la memoria para crear un arreglo de apuntadores que van a apuntar a cada fila del arreglo.

```
int **arr2d = (int **)malloc(filas*sizeof(int*));
```

2. Se solicita memoria para almacenar el número de elementos que va a formar cada fila o arreglo unidimensional.

```
for (i=0;i<filas;i++)  
    arr2d[i] = (int*)malloc(columnas*sizeof(int));
```

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

int main(void) {
    int filas = 2;
    int columnas = 3;
    int **x;
    int i; // Recorre filas
    int j; // Recorre columnas

    // Reserva de Memoria
    x = (int **)malloc(filas*sizeof(int*));
    for (i=0;i<filas;i++)
        x[i] = (int*)malloc(columnas*sizeof(int));

    // Damos Valores a la Matriz
    x[0][0] = 1;
    x[0][1] = 2;
    x[0][2] = 3;

    x[1][0] = 4;
    x[1][1] = 5;
    x[1][2] = 6;

    // Dibujamos la Matriz en pantalla
    for (i=0; i<filas; i++) {
        printf("\n");
        for (j=0; j<columnas; j++)
            printf("\t%d", x[i][j] );
    }
    return 0;
}
```
