Asignación dinámica de memoria

Memoria dinámica

Es memoria que se reserva en tiempo de ejecución. Su principal ventaja frente a la estática, es que su tamaño puede variar durante la ejecución del programa. (En C, el programador es encargado de liberar esta memoria cuando no la utilice más). El uso de memoria dinámica es necesario cuando de antemano no conocemos el número de datos/elementos a tratar.

Memoria estática

Es el espacio en memoria que se crea al declarar variables de cualquier tipo de dato (primitivas [int,char...] o compuestas [estructuras, matrices, apuntadores...]). La memoria que estas variables ocupan no puede cambiarse durante la ejecución y tampoco puede ser liberada manualmente.

Diferencias, ventajas y desventajas

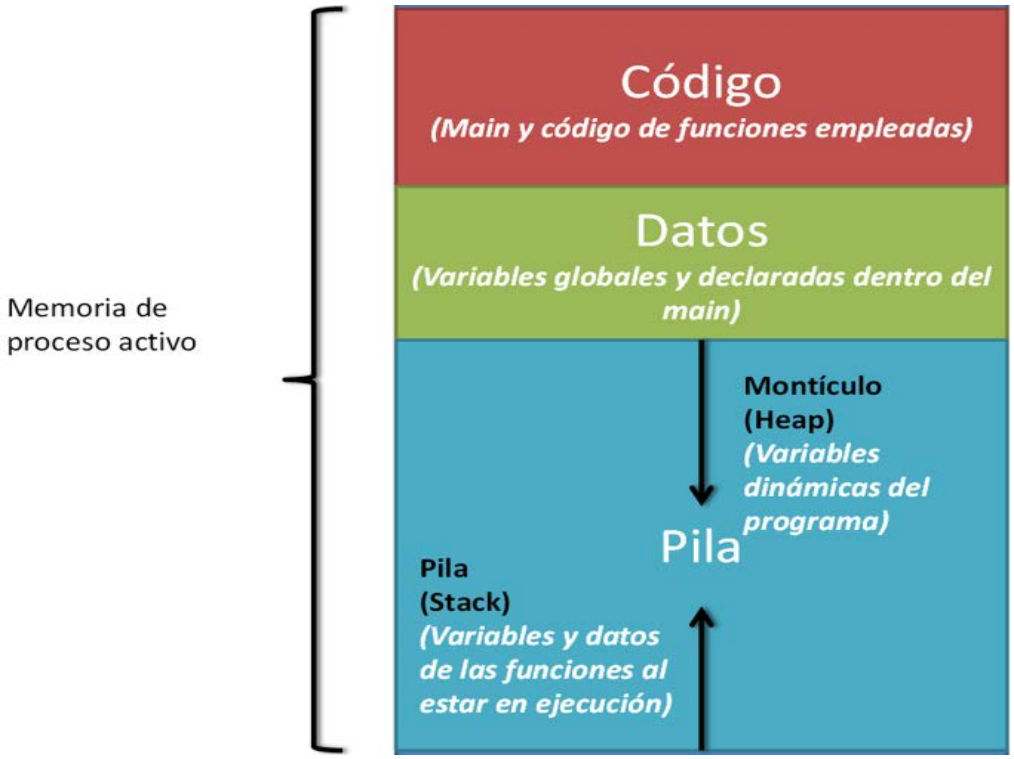
La memoria reservada de forma dinámica suele estar alojada en el heap o almacenamiento libre, y la memoria estática en el stack o pila. La pila generalmente es una zona muy limitada. El heap, en cambio, en principio podría estar limitado por la cantidad de memoria disponible durante la ejecución del programa y el máximo de memoria que el sistema operativo permita direccionar a un proceso. La pila puede crecer de forma dinámica, pero esto depende del sistema operativo. En cualquier caso, lo único que se puede asumir es que muy probablemente dispondremos de menor espacio en la pila que en el heap.

Otra ventaja de la memoria dinámica es que se puede ir incrementando durante la ejecución del programa. Esto permite, por ejemplo, trabajar con arreglos dinámicos. Aunque en C, a partir del estándar C99 se permite la creación de arreglos cuyo tamaño se determina en tiempo de ejecución, no todos los compiladores implementan este estándar. **Además, se sigue teniendo la limitante de que su tamaño no puede cambiar una vez que se especifica, cosa que sí se puede lograr asignando memoria de forma dinámica.**

Una desventaja de la memoria dinámica es que es más difícil de manejar. La memoria estática tiene una duración fija, que se reserva y libera de forma automática. En contraste, la memoria dinámica se reserva de forma explícita y continúa existiendo hasta que sea liberada, generalmente por parte del programador.

La memoria dinámica puede afectar el rendimiento. Puesto que con la memoria estática el tamaño de las variables se conoce en tiempo de compilación, esta información está incluida en el código objeto generado, por lo cual el proceso es muy eficiente. Cuando se reserva memoria de manera dinámica, se tienen que llevar a cabo varias tareas, como buscar un bloque de memoria libre y almacenar la posición y tamaño de la memoria asignada, de manera que pueda ser liberada más adelante. Todo esto representa una carga adicional, aunque esto depende de la implementación y hay técnicas para reducir su impacto, además de que con el aumento de

velocidad de las computadoras cada vez más tiende a disminuirse el impacto de ésta técnica sobre la eficiencia global del sistema.



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.

1. Sizeof

El operador[1] **`sizeof(operando)`** entrega la cantidad de almacenamiento en bytes que ocupa el operando.

Este operador permite especificar el tamaño en bytes[2] de un dato independientemente del compilador o computadora empleada. Por ejemplo

```
sizeof(int)
```

entrega la cantidad de bytes que se requieren para almacenar un entero. El siguiente programa emplea sizeof() para determinar el tamaño de los diferentes tipos de datos simples:

[1] Estrictamente **`sizeof()`** no es una función, formalmente es un operador, tal como los operadores +,-,*,>,<, etc..
[2] La documentación oficial indica *"**`sizeof()`** generates the size of a variable or datatype, measured in the number of char size storage units required for the type"* que generalmente equivale a 1 byte por char

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

int main()
{
    printf("Tamanio de los diferentes tipos de datos:\n\n");
    printf("         char: %2d\n",sizeof(char));
    printf("   short int: %2d\n",sizeof(short int));
    printf("          int: %2d\n",sizeof(int));
    printf("    long int: %2d\n",sizeof(long));
    printf("       float: %2d\n",sizeof(float));
    printf("      double: %2d\n",sizeof(double));
    printf(" long double: %2d\n",sizeof(long double));
    printf("     pointer: %2d\n",sizeof(void *));
    return 0;
}
```

Resultado del programa para el gcc (tdm-1) 4.7.1 desde la plataforma codeblocks para windows

```
Tamanio de los diferentes tipos de datos:

         char:  1
   short int:  2
          int:  4
    long int:  4
       float:  4
      double:  8
 long double: 12
     pointer:  4

Process returned 0 (0x0)   execution time : 0.008 s
Press any key to continue.
```

Resultado del programa desde la plataforma Visual Studio Community 2013 para Windows

```
Tamanio de los diferentes tipos de datos:

         char:  1
   short int:  2
          int:  4
    long int:  4
       float:  4
      double:  8
 long double: 16
     pointer:  8

--------------------------------
```

Resultado del programa para el TDM-GCC 4.9.2 64 bits Realease desde la plataforma devc 5.11 para Windows

```
Tamanio de los diferentes tipos de datos:

         char:  1
   short int:  2
          int:  4
    long int:  4
       float:  4
      double:  8
 long double:  8
     pointer:  4
```

2. Malloc

La función malloc reserva un bloque de memoria y devuelve un apuntador **`void *`** al inicio de la misma.

```
void *malloc(size_t size);
```

Donde el parámetro **`size`** especifica el número de bytes a reservar. En caso de que no se pueda realizar la asignación, devuelve el valor nulo (definido en la macro NULL), lo que permite saber si hubo errores en la asignación de memoria.

Ejemplo 1:

```
int *i;
…
/* Reservamos la memoria suficiente para almacenar un int y asignamos su dirección a i */

i = malloc(sizeof(int));

/* Verificamos que la asignación se haya realizado correctamente */
if (i  == NULL) {
    /* Error al intentar reservar memoria */
}
```

Ejemplo 2: Uno de los usos más comunes de la memoria dinámica es la creación de vectores cuyo número de elementos se define en tiempo de ejecución:

```
int *vect1, n;
printf("N£mero de elementos del vector: ");
scanf("%d", &n);

/* reservar memoria para almacenar n enteros */
vect1 = malloc(n * sizeof(int));

/* Verificamos que la asignación se haya realizado correctamente */
if (vect1  == NULL) {
    /* Error al intentar reservar memoria */
}
```

3. calloc

La función calloc funciona de modo similar a malloc, pero además de reservar memoria, inicializa a 0 la memoria reservada. Se usa comúnmente para arreglos y matrices. Está definida de esta forma:

void *calloc(size_t nmemb, size_t size);

El parámetro nmemb indica el número de elementos a reservar, y size el tamaño de cada elemento.

Ejemplo 3:

```c
int *vect1, n;
printf("N£mero de elementos del vector: ");
scanf("%d", &n);

/* Reservar memoria para almacenar n enteros */
vect1 = calloc(n, sizeof(int));

/* Verificamos que la asignación se haya realizado correctamente */
if (vect1  == NULL) {
    /* Error al intentar reservar memoria */
}
```

4. realloc

La función realloc redimensiona el espacio asignado de forma dinámica anteriormente a un apuntador.

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

Donde ptr es el apuntador a redimensionar, y size el nuevo tamaño, en bytes, que tendrá. Si el apuntador que se le pasa tiene el valor nulo, esta función actúa como **malloc**. Si la reasignación no se pudo hacer con éxito, devuelve un apuntador nulo, dejando intacto el apuntador que se pasa por parámetro. Al usar **realloc**, se debería usar un apuntador temporal. De lo contrario, podríamos tener una fuga de memoria, si es que ocurriera un error en **realloc**.

Ejemplo 4:

```c
/* Reservamos 5 bytes */
void *ptr = malloc(5);
…
/* Redimensionamos el puntero (a 10 bytes) y lo asignamos a un puntero temporal */
void *tmp_ptr = realloc(ptr, 10);

if (tmp_ptr == NULL) {
    /* Error: tomar medidas necesarias */
}
else {
    /* Reasignación exitosa. Asignar memoria a ptr */
    ptr = tmp_ptr;
}
```

Cuando se redimensiona la memoria con **realloc**, si el nuevo tamaño (parámetro **size**) es mayor que el anterior, se conservan todos los valores originales, quedando los bytes restantes sin inicializar. Si el nuevo tamaño es menor, se conservan los valores de los primeros **size** bytes. Los restantes también se dejan intactos, pero no son parte del bloque regresado por la función, pudiéndose asignar a otras solicitudes de memoria.

5. free

La función **free** sirve para liberar memoria que se asignó dinámicamente. Si el apuntador es nulo, free no hace nada.

void free(void *ptr);

El parámetro **ptr** es el apuntador a la memoria que se desea liberar.

Ejemplo 5:

```
int *i = malloc(sizeof(int));
…
free(i);

/* Reutilizamos i, ahora para reservar memoria para dos enteros */
i = malloc(2 * sizeof(int));
…
/* Volvemos a liberar la memoria cuando ya no la necesitamos */
free(i);
```

Dado que el manejo de memoria es un tema complejo, y éste es un error muy común, se debe hacer énfasis en que cuando se trabaja con memoria dinámica, siempre se debe verificar que se incluya el archivo **stdlib.h**.

Tratar de utilizar un apuntador cuyo bloque de memoria ha sido liberado con **free** puede ser peligroso. El comportamiento del programa queda indefinido: puede terminar de forma inesperada, sobrescribir otros datos y provocar problemas de seguridad. Liberar un apuntador que ya ha sido liberado también es fuente de errores.

Para evitar estos problemas, se recomienda que después de liberar un apuntador siempre se establezca su valor a **NULL**.

```
int *i;
i = malloc(sizeof(int));
…
free(i);
i = NULL;
```

Ejemplo 6:

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

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

  c = (char *)malloc( sizeof(char) );
  entero = (int *)malloc( sizeof(int) );
  flotante = (float *)malloc( sizeof(float) );
  doble = (double *)malloc( sizeof(double) );

  *c = 'c';
  *entero = 2378;
  *flotante = 128.89172378;
  *doble = 18947282.48263;

  printf( "valores: caracter %c, entero %d, flotante
%f, doble %lf",
        *c, *entero, *flotante, *doble );
  free( c );
  free( entero );
  free( flotante );
  free( doble );
  return 0;
}
```

Ejemplo 7: Se quiere reservar espacio para un arreglo de 10 enteros,

```c
int *ptr;
ptr = (int *)malloc( 10 * sizeof(int) );
if( ptr == NULL){
  printf( "No hay memoria disponible…\n" ); //no utilizar ptr
  return; //fin del programa o realizar la acción conveniente
}
//utilizar ptr
```

Ejemplo 8: Se quiere reservar espacio para un arreglo de enteros pero no se conoce el tamaño, sino hasta que el usuario lo indique:

```c
int tam;
int *ptr;
printf( "Ingresa el tamaño del arreglo " );
scanf( "%d", &tam; );
ptr = (int *)malloc( tam * sizeof(int) );
```

Arreglos bidimensionales

Para generar un arreglo bidimensional utilizando memoria dinámica se hacen dos pasos:

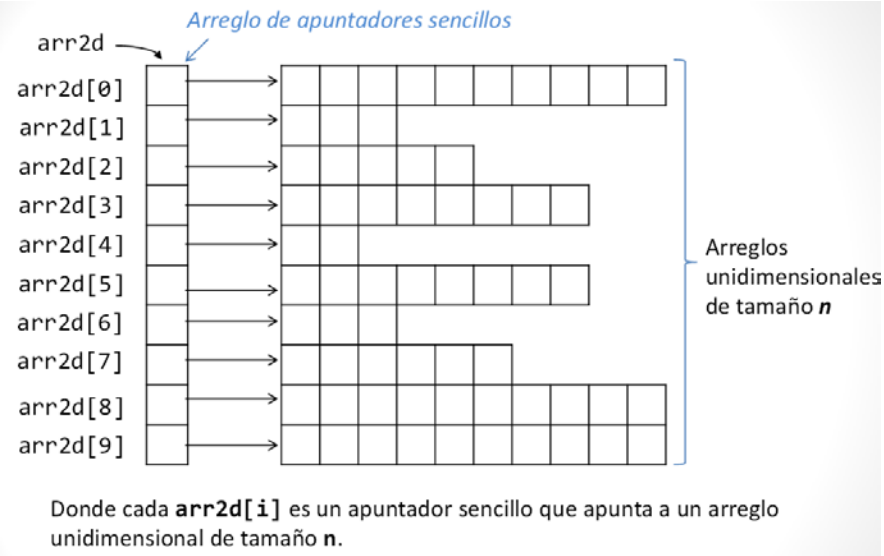
1. Se solicita la memoria para crear un arreglo de apuntadores que van a apuntar a cada renglón del arreglo.

   int **arr2d = (int *) malloc(sizeof(int *) *num_renglones);

2. Se solicita memoria para almacenar el número de elementos que tendrá cada renglón (recuerda que ese renglón tendrá un arreglo unidimensional).

   arr2d[i] = (int *) malloc (sizeof(int) * num_columnas);

   en el caso de matrices el número de columnas es el mismo para todos los renglones, por lo que sólo se pregunta una vez el número de columnas. También tome en cuenta que si el arreglo no es una matriz entonces es necesario recordar de alguna manera de qué tamaño es cada renglón (por ejemplo, por medio un vector adicional que guarde el tamaño de cada renglón o por medio de un indicador de fin de renglón).



Ejemplo 9: Arreglo bidimensional cuadrado (Matriz cuadrada)

main.c

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

int main() {
    int **arr2d;
    int r,c,k,j;

    do {
        printf("Numero renglones? ");
        scanf("%d",&r);
    } while (r<=0);

    arr2d = (int**) malloc(r*sizeof(int *));

    if (arr2d ==  NULL) {
        printf("Ocurrio un error durante la creación de la matriz!\n");
        exit(0);
    }

    do {
        printf("Numero columnas? ");
        scanf("%d",&c);
    } while (c<=0);

    for (k=0; k<r; k++) {
        arr2d[k] = (int *)malloc(c*sizeof(int));
        if (arr2d[k]==NULL) {
            printf("Ocurrio un error durante la creación de la matriz!\n");
            exit(0);
        }
    }

    for (k=0; k<r; k++) {
        for (j=0; j<c; j++) {
            arr2d[k][j] = rand()%100;
        }
    }

    printf("\n\nMatriz:\n");
    for (k=0; k<r; k++) {
        for (j=0; j<c; j++) {
            printf(" %3d ",arr2d[k][j]);
        }
        printf("\n");
    }

    for (k=0;k<r;k++) {
        free(arr2d[k]);
        arr2d[k]=NULL;
    }
    free(arr2d);
    arr2d=NULL;

    return 0;
}
```