APUNTES MEMORIA DINÁMICA

Resumen capítulo 12 del libro: "Programación en C, C++, JAVA y UML" de Luis Joyanes Aguilar e Ignacio Zahonero Martínez.

1. Introducción

Una de las caraterísticas más importantes de un programa es la cantidad de memoria que necesita para ejecutarse. Es importante que un programa no desperdicie memoria, que haga un uso eficiente de la misma. Esto plantea un serio problema cuando declaramos las variables, esencialmente los arreglos, ya que, si sólo usamos memoria estática, deberemos dimensionar el espacio de memoria para el peor caso posible.

Para evitar este problema existen funciones que permiten una gestión dinámica de memoria, es decir, permiten que un programa reserve memoria según se necesite, y la libere cuando deje de necesitarla. C dispone de las siguientes funciones para gestionar de forma dinámica la memoria, todas ellas definidas en la librería stdlib.h: `malloc(), calloc(), realloc(), free()`.

Las variables globales y locales se almacenan en la **pila (stack)** de memoria estática. Las variables locales existen con el tiempo de vida del programa (procedimiento, función) donde es declarada. Las variables globales existen a lo largo de la ejecución de todo el programa, aunque no es buen hábito hacer uso de variables globales si realmente no es necesario. En el momento de la compilación se reserva memoria para dichas variables.

Sin embargo, puede no conocerse cuántas variables o cuánta memoria será necesaria durante la ejecución del programa, no se sabe cuánta memoria habría que reservar. En C, se puede almacenar memoria dinámicamente, que se almacena en **memoria dinámica (heap)** y no en stack mediante las funciones: `malloc(), realloc(), calloc() y free()` que almacenan y liberan memoria en la heap.

Por ejemplo, usando memoria estática, hay situaciones donde en el momento de la declaración de un arreglo no se conocen sus dimensiones. Entonces, se recurre a declarar y usar un arreglo muy grande y se malgasta memoria. El método para evitar este inconveniente es usar punteros y asignación dinámica de memoria.

El espacio de la variable asignada dinámicamente se crea durante la ejecución del programa, al contrario que en el caso de una variable local cuyo espacio se asigna en tiempo de compilación. La asignación dinámica de memoria proporciona control directo sobre los requisitos de memoria de un programa. El programa puede crear o destruir la asignación dinámica en cualquier momento durante la ejecución. Se puede determinar la cantidad de memoria necesaria en el momento en que se haga la asignación.

El código del programa compilado se sitúa en segmentos de memoria denominados segmentos de código. Los datos del programa, tales como variables globales, se sitúan en un área denominada segmento de datos. Las variables locales y la información de control del programa se sitúan en un área

denominada pila. La memoria que queda se denomina memoria del montículo o almacén libre (heap o memoria dinámica). Cuando el programa solicita memoria para una variable dinámica, se asigna el espacio de memoria deseado desde la heap.

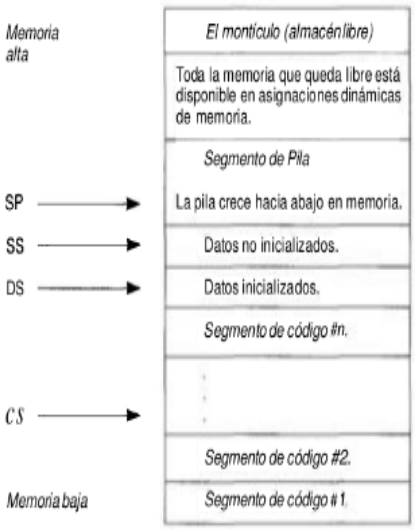

Figura 11.1. Mapa de memoria de un programa.

En C las funciones malloc() , realloc(), calloc() y free() requieren, generalmente, conversión de tipo (casting). Estas operaciones están declaradas en la librería `stdlib.h`, por lo que es necesario incluirla.

2. Funciones

2.1. malloc()

La forma más habitual de obtener bloques de memoria en C es con la función `malloc()`. Asigna un bloque de memoria que es el número de bytes pasados como argumento. `malloc()` devuelve un puntero, que es la dirección del bloque asignado de memoria heap.

Sintaxis:

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

El puntero retornado es del tipo void *, por lo que se suele hacer un casting. El casting es ocpional, pero sí se debe entender que aporta mayor portabilidad del código al lenguaje C++ y en resumen, es más seguro. Existen muchos debates sobre esto, uno interesante: link-debate-en-stack-oferflow

Por ejemplo:

```
puntero = malloc(tamaño en bytes); // sin casting:
puntero = (tipo*) malloc(tamaño en bytes); // con casting
```

En la sintaxis de llamada, puntero es el nombre de la variable puntero a la que se asigna la dirección del objeto (bloque de memoria en heap lo suficientemente grande para contener "tamaño" bytes, o NULL, si falla la operación de asignación de memoria). Más ejemplos:

```
int *pNum;
pNum = (int*) malloc(sizeof(int)); // aloca memoria para 1 entero. Con casting.
pNum = malloc(sizeof(int)); // aloca memoria para 1 entero. Sin casting.

int *vector;
vector = (int*) malloc(10*sizeof(int)); // aloca memoria para un arreglo de 10 enteros, con casting.
vector = malloc (10*sizeof(int)); // aloca memoria para un arreglo de 10 enteros, sin casting.
```

Si no hay memoria disponible la función `malloc()` retorna NULL (declarada en `stdlib.h`). Siempre habría que comprobar que `malloc()` no devuelva NULL:

Algorithm 1: Comprobación de alocación de memoria correcta.

```
1  ptr-lista = malloc(1000*sizeof(double));
2  if (ptr-lista == NULL)
3  {
4      printf ("Error en la asignación de memoria") ;
5      return -1;
6  }
```

Al principio se había planteado el problema de no conocer hasta momento de ejecución el tamaño de un vector. Entonces, ahora, se puede, por ejemplo, pedir al usuario el tamaño y luego utilizando `malloc()` se puede almacenar la cantidad justa para dicho vector:

Algorithm 2: Alocación de memoria conociendo el tamaño en tiempo de ejemcución.

```
1  double *vector;
2  int n;
3  printf ("Número de elementos del array: ") ;
4  scanf ("%d", &n);
5  vector = malloc(n*sizeof(double));
6  if (vector == NULL) //comprobacion memoria alocada correctamente
7  ...
```

En funciones donde se use repetidas veces la alocación de memoria, se sugiere calcular el tamaño de los arreglos, almacenar dicho cálculo en una variable y luego usar esa variable para alocar memoria. Por ejemplo:

```
int size = 10 * sizeof(int) ;
int* vector = malloc(size);
```

Mejor aún, usar constantes en vez de 10, por ejemplo.

2.2. calloc()

La función `calloc()` también se usa para almacenar memoria dinámica, es muy similar a `malloc()` pero tiene 2 diferencias fundamentales: recibe 2 parámetros indicando cantidad de datos a almacenar y tamaño en bytes de cada uno de ellos e inicializa los valores a 0.

Sintaxis:

```
void* calloc(cantidad elementos, tamaño del elemento);
```

el primer argumento es la cantidad de elmentos para los cuales se va a reservar memoria y el segundo es el tamaño de cada elemento (se sugiere usar `sizeof(tipo)`).

Por ejemplo:

```
arr = (int*) calloc(5, sizeof(int));
```

Aloca memoria para 5 enteros y los inicializa a 0. Lo que sería similar a:

```
arr = (int*) malloc(5 * sizeof(int));
memset(arr, 0, sizeof(int)*5);
```

2.3. realloc()

La función `realloc()` sirve para ampliar el tamaño de un bloque reservado anteriormente.

Sintaxis:

```
void* realloc(puntero a bloque, tamaño total del nuevo bloque);
```

Ejemplo de uso:

```
int size = 10 * sizeof(int) ;
int *arreglo = (int*) malloc(size));
arreglo = realloc (arreglo, size*2); // duplica el tamaño de arreglo y queda apun-
```
tado por el puntero `arreglo`

El tamaño del bloque se expresa en bytes. El puntero a bloque referencia a un bloque de memoria reservado previamente con `malloc()`, `calloc()` o la propia función `realloc()`.

2.4. free()

La memoria alocada usando las funciones `malloc(), calloc(), realloc()` no liberan la memoria por sí mismas. Explicitamente se debe usar la función `free()` para liberar el espacio.

Sintaxis:

```
free(puntero);
```

Se libera el espacio almacendo para puntero.

Algorithm 3: Ejemplo de uso de la función `free()`.

```
1   int size = DIM * sizeof(double);
2   double *lista = malloc(size));
3   if (lista == NULL)
4   {
5       printf (”Error en la asignación de memoria") ;
6       exit(-1);
7   }
8   ...
9   //uso de lista
10  ...
11  free(lista);
```

3. Resumen y reglas

Arreglos dinámicos vs arreglos estáticos: simplemente, `int m[10]` reserva memoria en tiempo de compilación y se mantiene asignada durante la ejecución del programa. En cambio `int *m; m = (int)malloc(10 * sizof(int));` asigna el espacio para 10 enteros en tiempo de ejecución, cuando se ejecuta esa instrucción y se libera en el momento de un `free()`.

3.1. Reglas de uso de memoria dinámica

- El prototipo de las funciones está en la librería `stdlib()`. Agregar `#include <stdlib.h>`.
- Las funciones `malloc(), calloc() y realloc()` retornan un tipo void* lo cual a veces es conveniente una reconversión al tipo del puntero. Por ejemplo: `m = (int*)malloc(10*sizeof(int));`
- Las funciones de asignación tienen como parámetro el número de bytes a reservar.
- El operador `sizeof()` permite calcular el tamaño de un tipo para el que se está almacenando memoria.
- La función `realloc()` permite expandir memoria reservada.

- Las funciones de memoria retornan NULL si no han podido reservar la memoria requerida. Es **siempre conveniente** verificar que las funciones hayan podido alocar memoria de forma correcta.
- Se pueden utilizar las funciones de alocación de memoria para reservar espacio para objetos más complejos, tales como estructuras, arreglos, en el almacenamiento heap. Por ejemplo: `Pz = (struct complejo*)calloc(n, sizeof(struct complejo));`
- Se pueden crear arreglos multidimensionales de objetos. Para un array de n x m, se asigna en primer lugar el espacio para el arreglo de punteros (de n elementos) y después se asigna memoria para cada fila (m elementos), con un bucle de n-1:

```
int** matriz;
matriz = malloc (n * sizeof (int*)); // array de punteros a enteros
for (i=0; i<size; i++)
... matriz[i] = (int*)malloc (m * sizeof(int)); // fila de m elementos
```

- Toda memoria reservada se puede liberar con la función free(). Para mat:

```
int** matriz;
//liberación de la memoria
for (i=0; i<size; i++)
... free matriz[i];
free(matriz);   // fila de m elementos
```