

**Ingeniería en Ciencias de la Computación**

**Introducción a la Programación**

Malloc y la asignación dinámica de memoria en C

Segundo Parcial

|  |  |
| --- | --- |
| **Equipo:** | **Nombre del Maestro:** |
| 6 | Ricardo Blanco Vega |
| **Matrícula:** | **Nombre del Alumno:** |
| 309694 | Miguel Angel Gardea Granados |
|  |  |
|  |  |
|  |  |

Fecha de entrega: **10/22/2019**

Tabla de Contenido

[Objetivo 3](#_Toc19798046)

[Desarrollo 3](#_Toc19798047)

[Conclusiones 3](#_Toc19798048)

[Recomendaciones 3](#_Toc19798049)

[Referencias 3](#_Toc19798050)

# Objetivo

Conocer las funciones que utilizan la memoria dinámica en el lenguaje de C, para poder saber cómo aplicarlas en los distintos proyectos a desarrollar y cómo aprovechar al máximo los recursos con los que se cuentan. Aprender sobre el manejo de la memoria dinámica, tanto sus ventajas como desventajas en un sistema informático.

# Desarrollo

La asignación dinámica de memoria, como su nombre lo dice, es una forma de conseguir espacio en memoria, dando más eficiencia y técnicamente hacer lo que requiera hacer con este. En programación se hace uso tanto de memoria estática como memoria dinámica, y vamos a definirlas de la siguiente manera:

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 no se sabe el número exacto de datos/elementos a tratar.

Memoria estática:

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

Manejo de memoria en el lenguaje C:

Todos los objetos tienen un tiempo de vida, es decir, el tiempo durante el cual se garantiza que el objeto exista. En C, existen 3 tipos de duración: estática, automática y asignada. Las variables globales y las variables locales declaradas con el especificador “static” tienen duración estática. Se crean antes de que el programa inicie su ejecución y se destruyen cuando el programa termina. Las variables locales no “static” tienen duración automática. Se crean al entrar al bloque en el que fueron declaradas y se destruyen al salir de ese bloque. La duración asignada se refiere a los objetos cuya memoria se reserva de forma dinámica. Como se explicó anteriormente, esta memoria se crea y se debe liberar de forma explícita.

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.

Funcion malloc:

La función malloc reserva un bloque de memoria y devuelve un puntero “void” al inicio de la misma. Tiene la siguiente definición:

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.

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("Numero 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 \*/

}

Funcion 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. El ejemplo anterior se podría reescribir con calloc de esta forma:

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 \*/

}

Funcion realloc:

La función realloc redimensiona el espacio asignado de forma dinámica anteriormente a un puntero. Tiene la siguiente definición:

void \*realloc(void \*ptr, size\_t size);

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

Ejemplo:

/\* 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.

Funcion Free o CFree:

La función free sirve para liberar memoria que se asignó dinámicamente. Si el puntero es nulo, free no hace nada. Además existe la función cfree, que sirve para liberar memoria de los elementos que han sido reservados con calloc(). Tienen la siguiente definición:

void free(void \*ptr); void cfree (void \*ptr);

El parámetro ptr es el puntero a la memoria que se desea liberar:

int \*i;

i = malloc(sizeof(int));

…

free(i);

Una vez liberada la memoria, si se quiere volver a utilizar el puntero, primero se debe reservar nueva memoria con malloc o calloc:

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);

En cualquier caso, dado que el manejo de memoria es un tema complejo, 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 puntero cuyo bloque de memoria ha sido liberado con free puede ser sumamente peligroso. El comportamiento del programa queda indefinido: puede terminar de forma inesperada, sobrescribir otros datos y provocar problemas de seguridad. Liberar un puntero que ya ha sido liberado también es fuente de errores.

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

int \*i;

i = malloc(sizeof(int));

…

free(i);

i = NULL;

# Conclusiones

La memoria dinamica puede resolver algunos problemas que tiene la memoria estatica, es decir, podemos aprovechar la longitud un vector reduciendo o aumentando sus elementos dependiendo de la aplicación y el tipo de datos que estamos manejando. Si tenemos algún arreglo donde se utilicen muy pocos datos, lo mas conveniente será utilizar la memoria estatica. Caso contrario, si utilizamos muchísimos datos que tienen a cambiar de valor constantemente, lo mejor será utilizar la memoria dinámica.

# Recomendaciones

# Referencias

<https://webs.um.es/iverdu/P00LibreriasANSIc.pdf>

<http://sopa.dis.ulpgc.es/fso/cpp/intro_c/introc75.htm>

<https://www.fing.edu.uy/inco/cursos/prog3/utils/Punteros.pdf>

https://es.wikipedia.org/wiki/Malloc