|  |  |  |
| --- | --- | --- |
|  | **Carátula para entrega de prácticas** | |
| Facultad de Ingeniería | | Laboratorio de docencia |

Laboratorios de computación

salas A y B



|  |  |
| --- | --- |
| Profesor(a): | M.I. Marco Antonio Martinez Quintana. |
| Asignatura: | Estructura de Datos y Algoritmos I |
| Grupo: | 17 |
| No de Práctica(s): | 04 |
| Integrante(s): | Pineda González Rodrigo |
| No. de Equipo de cómputo empleado: | 26 |
| No. de Lista o Brigada: | 30 |
| Semestre: | 2020-II |
| Fecha de entrega: | 02/03/2020 |
| Observaciones: |  |

CALIFICACIÓN: \_\_\_\_\_\_\_\_\_

# Objetivo:

Utilizarás funciones en lenguaje C que permiten reservar y almacenar información de manera dinámica (en tiempo de ejecución).

# Introducción:

La memoria dinámica se refiere al espacio de almacenamiento que se reserva en tiempo de ejecución, pues su tamaño será definido mientra se ejecuta el programa. Dentro de la memoria RAM, la memoria reservada de forma dinámica está alojada en el heap o almacenamiento libre y la memoria estática (como los arreglos o las variables primitivas) en el stack o pila.

El heap es un medio de almacenamiento con más capacidad que la pila y que puede almacenar datos durante toda la ejecución. Las variables globales y estáticas viven en el heap mientras la aplicación se esté ejecutando. Para acceder a cualquier dato almacenado dentro del heap se debe tener una referencia o apuntador en la pila.

La memoria que se define de manera explícita (estática) tiene una duración fija, que se reserva (al iniciar el programa) y libera (al terminar la ejecución) de forma automática. La memoria dinámica se reserva y libera de forma explícita. Lenguaje C permite el almacenamiento de memoria en tiempo de ejecución a través de tres funciones:

* Malloc {(void \*malloc(size\_t size);}

Permite reservar un bloque de memoria de manera dinámica y devuelve un apuntador tipo void.

* Calloc {void \*calloc (size\_t nelem, size\_t size);}

Permite reservar un bloque de memoria de manera dinámica y devuelve un apuntador tipo void. Además, dicha memoria está inicializada con 0

* Realloc. {void \*realloc (void \*ptr, size\_t size);}

Permite redimensionar el espacio asignado previamente de forma dinámica. Si la reasignación no se pudo realizar, la función devuelve un apuntador a null, dejando intacto el apuntador que se pasa como parámetro

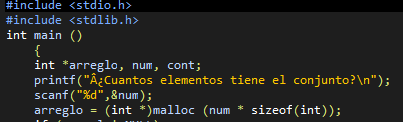
* Free {void free(void \*ptr);}

El almacenamiento en tiempo de ejecución se debe liberar de manera explícita, es decir, desde la aplicación se debe enviar la orden para hacerlo. Free permite realizar dicha acción.

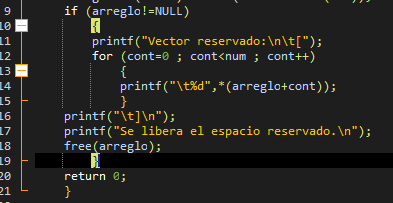
# Desarrollo:

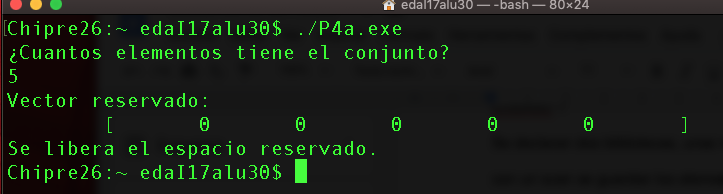
### Código (malloc)

Se declaran dos bibliotecas, stdio.h y stdlib.h. En el main se declara una variable apuntador y 2 enteros de nombre “num” y “cont”. Se imprimen las indicaciones al usuario y con un scan se guardan los elementos del conjunto en la variable num. Se le asigna al arreglo entero el tipo del dato que necesita (pues se castea de void a int) seguido de malloc y en los parámetros num y lo multiplica por sizeof (que regresa el tamaño del entero).

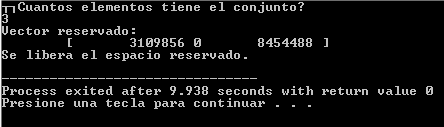


Con un condicional if que define que mientras no sea null (Si no se puede reservar la memoria de manera dinámica, la función malloc regresa NULL, por lo que solo implica que se pueda reservar memoria) con ayuda de un ciclo for se recorre con apuntadores la posición a la que apunta y se muestra lo que contiene esa localidad de memoria. Al terminar el ciclo for, con la función free se libera memoria que se reservó de manera dinámica.

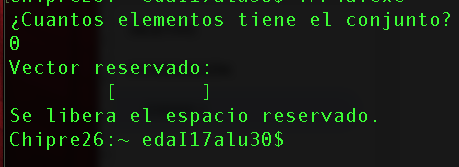




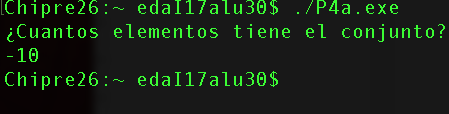
Pueden surgir datos basura si la memoria no está limpia.



Si se ingresa el número 0, no se reservará memoria.



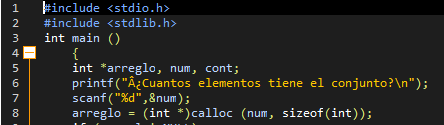
No se pueden reservar números negativos de memoria, ya que físicamente es imposible que un equipo tenga memoria negativa, por lo que el programa no hará nada.



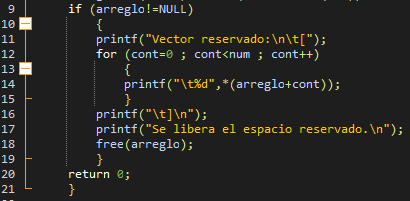
### Código (calloc)

Se declaran dos bibliotecas, stdio.h y stdlib.h. En el main se declara una variable apuntador y 2 enteros de nombre “num” y “cont”. Se imprimen las indicaciones al usuario y con un scan se guardan los elementos del conjunto en la variable num. Se le asigna al arreglo entero el tipo del dato que necesita (pues se castea de void a int) seguido de calloc y en los parámetros num y sizeof (que regresa el tamaño del entero).

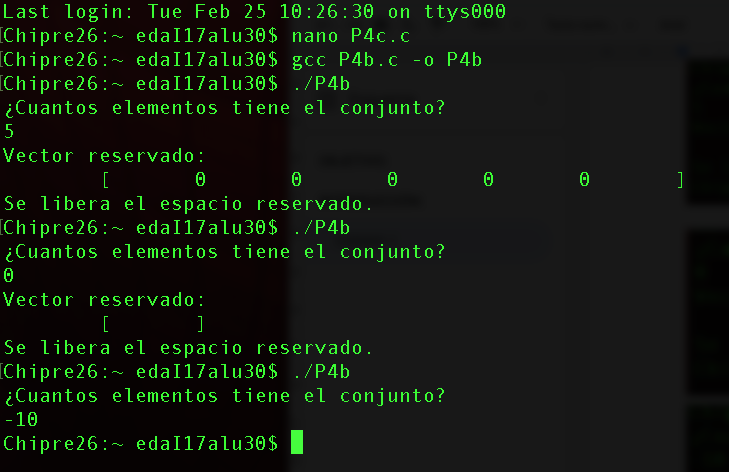
Primero recibe el tamaño y como segundo parámetro el tamaño, a diferencia de malloc que únicamente multiplica el espacio



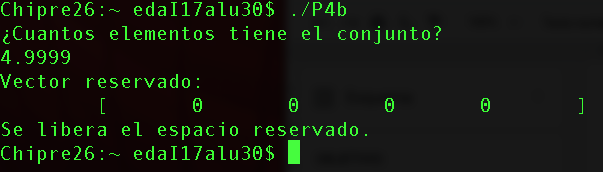
Con un condicional if que define que mientras no sea null (proceso igual que el de la función malloc, por lo que solo implica que se pueda reservar memoria) con ayuda de un ciclo for se recorre con apuntadores la posición a la que apunta y se muestra lo que contiene esa localidad de memoria. Al terminar el ciclo for, con la función free se libera memoria que se reservó de manera dinámica.



De igual forma que en la función malloc, el 0 no reservará memoria y un número negativo no puede ser procesado, pero siempre tendrá la memoria reservada con ceros, lo que la diferencia de malloc.

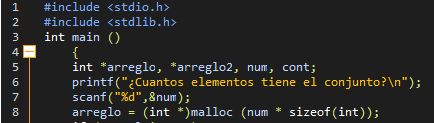


Si se prueba un decimal, se redondea hacia abajo, ya que no se consideran los dígitos después del punto por la naturaleza del tipo de dato int.

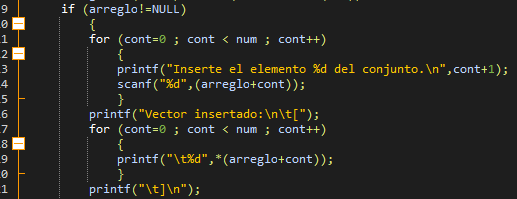


### Código (realloc)

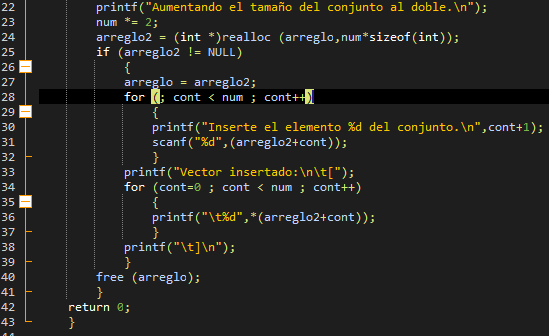
Se declaran dos bibliotecas, stdio.h y stdlib.h. En el main se declaran dos variables apuntador, una para cada arreglo, y 2 enteros de nombre “num” y “cont”. Se imprimen las indicaciones al usuario y con un scan se guardan los elementos del conjunto en la variable num. Se le asigna al arreglo entero el tipo del dato que necesita (pues se castea de void a int) seguido de malloc.

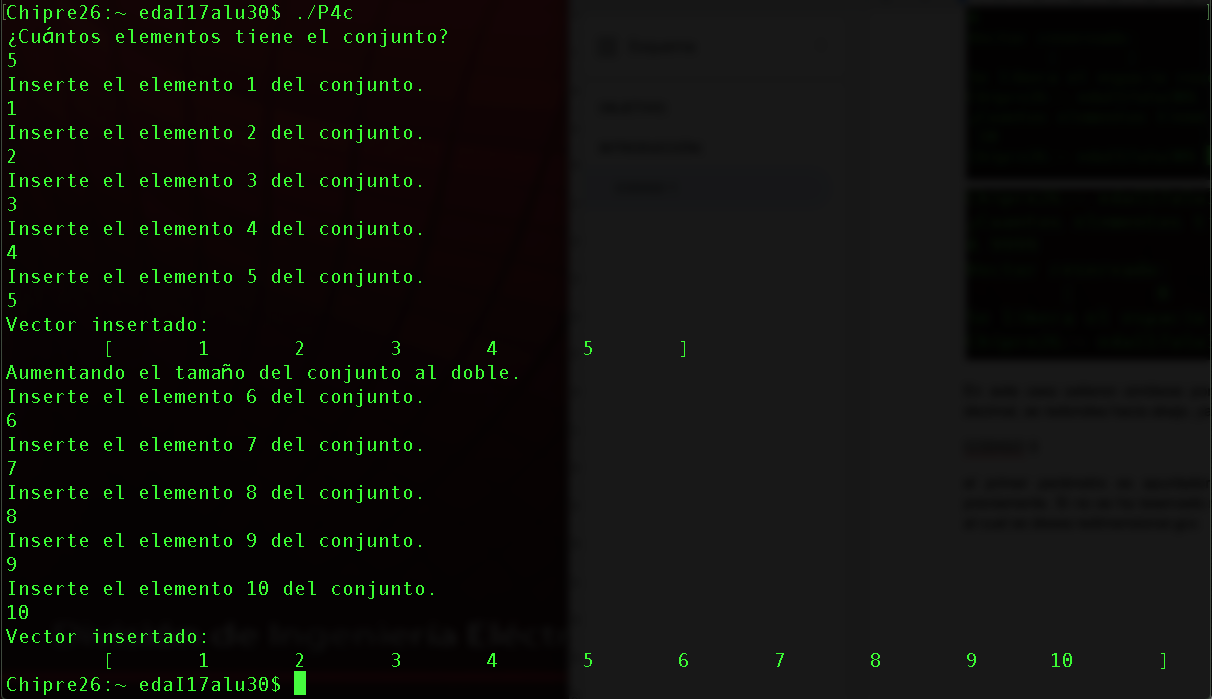


Con un condicional if que define que mientras no sea null (que implica que se pueda reservar memoria) con ayuda de un ciclo for se recorre con apuntadores la posición a la que apunta y se guarda en esa localidad de memoria un valor dado por el usuario. Al terminar muestra en forma de arreglo lo ingresado por el usuario.

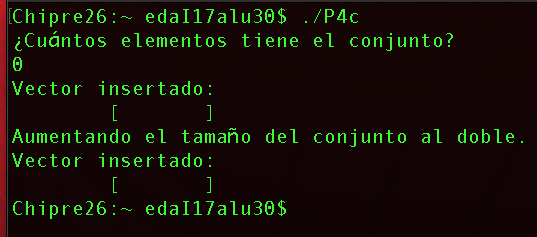


El programa duplicará el tamaño del arreglo al doble, por lo que cambia el valor de num a 2. Se le asigna al arreglo2 el tipo del dato que necesita (pues se castea de void a int) seguido de realloc. El primer parámetro es apuntador al tamaño de la memoria que se ha reservado previamente. Si no se ha reservado no puede usarse; el segundo parámetro es el tamaño al cual se desea redimensionar, que será dado por el tamaño del usuario multiplicado por el dos previamente asignado. Con un condicional if que define que mientras no sea NULL, y se realiza el mismo proceso que anteriormente se realizó para guardar los siguientes elementos del conjunto. Al terminar muestra el arreglo completo ingresado por el usuario. Al terminar el ciclo for, con la función free se libera memoria que se reservó de manera dinámica.

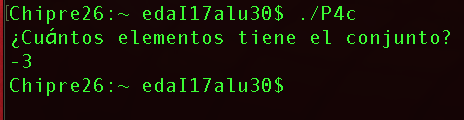




De igual manera que en los casos anteriores, el doble de 0 sigue siendo cero, por lo que no guardará dato alguno, pero sí puede funcionar el programa.



Al seleccionar un número negativo, el programa no reservará memoria puesto que es imposible.



## Códigos en Github:

<https://github.com/RodrigoPG-W/Practica4_EDA1#practica4_eda1>

# Conclusión:

A pesar de la simplicidad de almacenar las variables de nuestros programas en el stack de la memoria, es necesario hacer uso de la memoria dinámica y sus diferentes funciones cuando no se conoce el número de datos que se van a manejar para así optimizar el uso de recursos del equipo, ya que reservar recursos de más sólo es una solución ineficinte, que muchas veces puede incluso hacer que no se puedan ejecutar nuestros programas en equipos de bajos recursos computacionales.

Por otro lado, esto puede ser un arma de dos filos, ya que el almacenamiento dinámico puede afectar el rendimiento del programa debido a que se realizan arduas tareas en tiempo de ejecución: buscar un bloque de memoria libre y almacenar el tamaño de la memoria asignada, cosa que no todos los procesadores pueden hacer con tanta facilidad, y mal implementado hará imposible correr el programa. Por ello, es importante saber que función es la adecuada para cada situación, las cuales hacen muy amplio el catálogo de soluciones que puede brindar el lenguaje C.

# Referencias:

* Brian W. Kernighan, Dennis M. Ritchie. (1991). “*El lenguaje de programación C*”. Segunda edición, USA, Pearson Educación.
* Gottfried, BS. (1997). “*Programación en C*” (2ª ed.). Madrid, España: Mc Graw Hill.