Práctica 3:
Manejo dinámico de memoria

Estructuras de Datos y Algoritmos I

Autor: José Mauricio Matamoros de Maria y Campos

1. Objetivo

El alumno conocerá las funciones en lenguage C que permiten el manejo dinámico de memoria (reserva y liberación) para el almacenamiento de información no previsible en tiempo de ejecución.

2. Material

Se asume que el alumno cuenta con un una computadora con arquitectura Intel x86 o compatible, sistema operativo Linux/Unix, y compilador `gcc` instalado.

3. Instrucciones

1. Lea detenidamente los antecedentes teóricos sobre el manejo dinámico de memoria en lenguaje C en la Sección 4.
2. Realice cada una de las actividades detalladas en la Sección 5 apoyándose en el código de ejemplo del Apéndice A y responda las preguntas de las mismas.
3. Finalmente, responda el cuestionario de la Sección 6.

4. Antecedentes

Antes de estudiar las funciones que le permiten a un desarrollador reservar memoria de manera dinámica, es necesario entender por qué esto es necesario más allá de las obviedades de este proceso.

4.1. Memoria, programas y procesos

Antes de que se ejecute un programa, el Sistema Operativo tendrá que cargarlo en memoria para que el procesador tenga acceso a las instrucciones del programa y pueda ejecutarlo. Esta operación, en apariencia trivial, requiere de un gran número de operaciones y cálculos bastante complejos.

En un programa las funciones y los datos del programa no sólo están mezclados, sino que los primeros hacen referencia a los segundos mediante valores numéricos relativos (en el nivel más bajo, un programa no es sino un vector de números). Ahora bien, dentro del archivo que forma el programa, todas las referencias relativas existentes tienen como base el cero, es decir, la primer localidad de memoria disponible en una computadora que, por lo general, corresponde al vector de RESET y es donde se coloca la rutina de inicialización y, por obvias razones, no es posible cargar al programa en ésta dirección. Es por esta razón que, para poder ejecutarse, un programa requerirá que el sistema operativo *traduzca* las referencias y relaciones existentes en el programa para que sean compatibles con la dirección de memoria donde éste se va a cargar.

Aunado a lo anterior, el sistema operativo tendrá que reconocer todas aquellas referencias que apuntan a recursos compartidos **fuera** del programa y reemplazarlas por las direcciones en memoria de las funciones que permiten que

los programas se comuniquen entre sí, incluído el propio sistema operativo. Todos los programas utilizan estos recursos compartidos entre los cuales se encuentran el disco duro y sus archivos, todos los periféricos, la pantalla, la consola, etcétera.

De todos los recursos compartidos, los más críticos son la memoria y el procesador. El sistema operativo, además, está encargado de administrar el uso de estos dos recursos a fin de permitir que todos los programas se ejecuten de la forma más eficiente y segura posible; es decir, el sistema operativo tendrá que evitar que los programas monopolicen el procesador o la memoria.

Lo anterior tiene dos consecuencias inmediatas. Primero, ningún programa puede cargarse directamente en la memoria principal, ni siquiera aún después de haber sido traducido. El sistema operativo tendrá que generar una estructura especial en memoria que le permita controlar la ejecución del programa a fin de activarlo y desactivarlo a conveniencia, pues sólo así podrá ejecutar varios programas de forma concurrente y dar la impresión de que estos se ejecutan al mismo tiempo. A esta estructura se le denomina **proceso**.

Segundo. Para poder funcionar correctamente, un programa tiene que mantenerse como una unidad dentro de la memoria (los procesos no pueden fragmentarse). Si bien el sistema operativo sabe cuánta memoria requieren las funciones y variables del programa durante carga (el programador lo indica de manera explícita), es imposible predecir cuánta memoria adicional requerirán las interacciones con otras fuentes de datos (el usuario, los archivos, la red, otros programas, etc.). Dicho de otra manera, el sistema operativo no tiene manera de saber *a priori* cuánta memoria va a requerir un programa. Para solucionar este problema, el sistema operativo reserva un conjunto de bloques de memoria dentro de cada proceso creado para que, cuando ejecute el programa, éste pueda hacer uso de memoria disponible adicional. Sin embargo, hay ocasiones en las que el programa necesita más memoria de la que tiene asignada un proceso, por ejemplo como cuando se carga un archivo muy grande. En estos casos, el sistema operativo tendrá que *mover*[1] el proceso a otro lugar en la memoria donde haya suficiente espacio para garantizar que el proceso pueda seguir ejecutándose.

4.2. Memoria dinámica

Cuando un programa requiere una cantidad de memoria no conocida *a priori* por el programador, se dicr que el programa hace una *reserva dinámica de memoria* o que es un programa que utiliza *memoria dinámica*. Este nombre deriva del hecho de que la memoria se reserva y libera de forma dinámica durante la ejecución del proceso asociado mediante llamadas al sistema operativo.

Las funciones del sistema operativo que se encargan de la gestión de memoria son:

- **`void* malloc(size_t size)`** Reserva un bloque de memoria de tamaño `size` (en bytes) y devuelve un apuntador al primer byte del bloque. `malloc` no inicializa el contenido del bloque de memoria reservado, por lo que los valores almacenados en el bloque quedan indeterminados. `malloc` devolverá `NULL` (cero) si no se dispone de suficiente memoria libre para reservar el bloque.
- **`void* calloc(size_t size)`** Reserva un bloque de memoria de tamaño `size` (en bytes) y devuelve un apuntador al primer byte del bloque, inicializando en cero todos los bytes del bloque reservado. Al igual que `malloc`, `calloc` devolverá `NULL` (cero) si no se dispone de suficiente memoria libre para reservar el bloque.
- **`void* realloc(void* ptr, size_t size)`** Intenta extender el bloque de memoria reservado por `malloc` o `calloc` y apuntado por `ptr`; y devuelve un apuntador al primer byte del bloque. Al igual que `malloc` y `calloc`, `realloc` devolverá `NULL` (cero) si no se dispone de suficiente memoria libre para extender el bloque, en cuyo caso los datos existentes no se ven afectados.
- **`void free(void* ptr)`** Libera el bloque de memoria reservado por `malloc`, `calloc` o `realloc` y apuntado por `ptr`, dejando esta memoria disponible para su uso.

Todas estas funciones son parte de la librería estándar de C y estám contenidas en `stdlib.h`.

5. Desarrollo de la práctica

Lea cuidadosamente cada una de las actividades propuestas antes de realizar el programa o las modificaciones indicadas.

[1] Desde un punto de vista riguroso, el sistema operativo no mueve procesos sino que, tras suspender el proceso y buscar en la memoria un espacio lo suficientemente grande, crea un nuevo proceso donde se copian el programa y los datos de ejecución existentes, y finalmente destruye al proceso original. Si se diere el caso de que no hubiere suficiente memoria disponible, el sistema operativo devolverá un apuntador nulo que, de no ser controlado por el proceso que realiza la petición, causará la terminación del proceso.

5.1. Actividad 1: Primitivas de manejo de memoria

El programa del Apéndice A.3 reserva dos bloques de memoria a y b de tamaño N usando las primitivas funciones `malloc` y `calloc` respectivamente, para después redimensionar los bloques a un tamaño $2N$, imprimiendo el contenido de la memoria del bloque en cada vez.

Compile y ejecute el programa del Apéndice A.2 con $N = 10$ (N se pasa como argumento) y utilice la información para llenar la Tabla 1.

```
$ ./mem 10
```

Tabla 1: Direcciones de memoria y contenido de bloques reservados dinamicamente

Dirección	Contenido
a (`malloc`)	
b (`calloc`)	
a (`realloc`)	
b (`realloc`)	

5.2. Actividad 2: Máximos y mínimos

El programa del Apéndice A.3 imprime el máximo valor encontrado en arreglo unidimensional de tamaño fijo de números generados aleatoriamente con base en la semilla proporcionada.

[3 puntos] Modifique dicho programa para que imprima el valor **mínimo** encontrado en el arreglo. ¿Cuál es el mínimo reportado? _____

Ahora modifique el programa del Apéndice A.3 para que acepte como primer parámetro el tamaño del vector de números aleatorios a generar, y como segundo parámetro la semilla del generador de números aleatorios. Compile su programa y utilícelo para llenar la Tabla 2.

Tabla 2: Mínimos [2 puntos]

Parámetro 1	Parámetro 2	Máximo
10	0	
10	69	
10	80	
100	0	
100	69	
100	80	
1000	0	
1000	69	
1000	80	

5.3. Actividad 3: Producto punto

El programa del Apéndice A.4 imprime el el producto punto de dos arreglos unidimensionales de tamaño fijo compuestos por números generados aleatoriamente.

Modifique dicho programa para que imprima el producto punto $p = \bar{u} \cdot \bar{v}$ de dos vectores de números pseudo-aleatorios con diferente semilla. En esta ocasión, el programa tendrá que recibir tres parámetros:

i) el número de componentes en los vectores a generar (tamaño del arreglo).
ii) la semilla usada para generar las componentes del vector $\bar{u}$.
iii) la semilla usada para generar las componentes del vector $\bar{v}$.

Compile su programa y utilícelo para llenar la Tabla 3.

Tabla 3: Máximos [3 puntos]

Parámetro 1 ($\lvert\bar{u}\rvert = \lvert\bar{v}\rvert$)	**Parámetro 2**	**Parámetro 3**	$p = \bar{u} \cdot \bar{v}$
10	0	1	
10	69	80	
10	42	42	
100	0	1	
100	69	80	
100	42	42	
1000	0	1	
1000	69	80	
1000	42	42	

6. Cuestionario

Responda las siguientes preguntas considerando la información de la Tabla 1.

1. [2 puntos] ¿Cuál es la diferencia entre las funciones `malloc` y `calloc`?
2. [1 punto] ¿Para qué sirve la función `realloc`?
3. [1 punto] ¿Para qué sirve la función `free` y cuándo debe utilizarse?

A. Código de ejemplo

A.1. Archivo `src/Makefile`

src/Makefile

```
CC       = gcc
CFLAGS   = -Wall -O3 -std=c99 -pedantic
SILENT   = @
PROGRAMS = dot max mem
.PHONY: all clean $(PROGRAMS)

all: $(PROGRAMS)

$(PROGRAMS): %: %.c
  $(SILENT) $(CC) $(CFLAGS) -o $@ $<

clean:
  rm -f *.o
```

A.2. Archivo `src/mem.c`

src/mem.c

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

void print_mem_block(int* ptr, size_t size);

int main(int argc, char** argv){
  size_t block_size = 0;
  int *a, *b;
  if (argc > 1) block_size = (size_t)strtof(argv[1], NULL);
  if ((block_size < 1) || (block_size > 1024)) block_size = 10;

  a = (int*)malloc(block_size * sizeof(int));
  printf("Reservados %lu bytes en memoria con malloc\n", block_size);
  print_mem_block(a, block_size);

  b = (int*)calloc(block_size, sizeof(int));
  printf("Reservados %lu bytes en memoria con calloc\n", block_size);
  print_mem_block(b, block_size);

  a = (int*)realloc(a, block_size * sizeof(int));
  printf("Redimensionado (realloc) %p a %lu bytes\n", a, 2*block_size);
  print_mem_block(a, 2*block_size);

  b = (int*)realloc(b, block_size * sizeof(int));
  printf("Redimensionado (realloc) %p a %lu bytes\n", a, 2*block_size);
  print_mem_block(b, 2*block_size);

  free(a);
  free(b);
}

void print_mem_block(int* ptr, size_t size){
  printf(" %p={", ptr);
  for(size_t i = 0; i < size; ++i)
    printf(" %d", ptr[i]);
  printf(" }\n\n");

}
```

A.3. Archivo `src/max.c`

src/max.c

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

#define A_SIZE 1024
int a[A_SIZE];

void generate(int seed);
int main(int argc, char** argv);

void generate(int seed){
  srand(seed);
  for(int i = 0; i < A_SIZE; ++i) a[i] = 1 + rand()  % 69000;
}

int main(int argc, char** argv){
  int seed = 69;
  if(argc > 1) seed = strtof(argv[1], NULL);
  int max = a[0];
  int pmax = 0;
  // Populate the array with random numbers
  generate(seed);
  // Find max
  for(int i = 1; i < A_SIZE; ++i){
    if(a[i] > max){
      max = a[i];
      pmax = i;
    }
  }

  printf("Max: %d (%d)\n", max, pmax);
  return 0;
}
```

A.4. Archivo `src/dot.c`

src/dot.c

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

#define A_SIZE 10

int main(int argc, char** argv);
void populate(float* array, int seed);
float dot(float* u, float* v, size_t size);

int main(int argc, char** argv){
  float u[A_SIZE];
  float v[A_SIZE];

  int seedu = (argc > 1) ? strtof(argv[1], NULL) : 0;
  int seedv = (argc > 2) ? strtof(argv[2], NULL) : 0;

  // Populate the arrays with random numbers
  populate(u, seedu);
  populate(v, seedv);

  // Dot product
  printf("u dot v = %0.8f\n", dot(u, v, A_SIZE));

  return 0;
}

void populate(float* array, int seed){
  srand(seed);
  for(size_t i = 0; i < A_SIZE; ++i)
    array[i] = (float)rand() / RAND_MAX;
}

float dot(float* u, float* v, size_t size){
  float dp = 0;
  for (size_t i = 0; i < size; ++i)
    dp+= u[i]*v[i];
  return dp;
}
```

B. Reporte Escrito

El reporte de la práctica deberá ser entregada en un archivo en formato `PDF` siguiendo las siguientes especificaciones:

- La primera página del documento deberá ser la carátula oficial para prácticas de laboratorio disponible en `lcp02.fi-b.unam.mx/`
- El nombre del documento `PDF` deberá ser `nn-XXXX-L03.pdf`, donde:
  - `nn` es el número de lista del alumno a dos dígitos forzosos (ej. 01, 02, etc.).
  - `XXXX` corresponden a los cuatro primeros caracteres de su RFC o CURP.
- El reporte consiste en un documento de redacción propia donde el alumno explica de forma concisa y a detalle las actividades realizadas en la práctica, y reportando los resultados obtenidos.
- La longitud del reporte no deberá exceder las 3 páginas, sin contar la carátula.
- El reporte deberá seguir todos los lineamientos para documentos escritos establecidos al inicio del curso.
- Todas las referencias deberán estar debidamente citadas.

IMPORTANTE: No se aceptan archivos en otros formatos ni con nombres distintos a los especificados.