**C Segundo semestre**

**(Memoria dinámica, Archivos, punteros dobles)**

Memoria dinámica

|  |
| --- |
|  |
|  |
|  |
|  |

Memoria global: variables mueren al terminar o cerrar el programa

Stack(memoria locales): son declaradas dentro de las funciones y mueren al terminar la función.

Memoria libre: puede agrandarse o achicarse para dar lugar al heap o stack

Heap: mucho mas espacio que en el stack (+98%). Viven las variables dinámicas. Estas variables viven hasta después de cerrar el programa por eso tenemos que destruirlas antes.

Hacer un strack overflow es quedarse sin memoria libre.

|  |  |
| --- | --- |
| sizeof(int) | 4 Bytes |
| sizeof(float) | 4 Bytes en binario (6bytes en txt) |
| sizeof(char) | 1 Bytes |
| sizeof(eEstructura) | La suma de todas sus variables.  Se guarda de a cuatro así que char reserva 4 pero solo ocupa 1 |
| typedef struct{  int id;  char nombre[20];  char sexo;  float altura;  int isEmpty;  }ePersona; | sizeof(ePersona)  **4+20+4+4+4:**  esta estructura ocupa 36 Bytes |

**MALLOC**

* Reserva un espacio en la memoria dinámica
* Devuelve la dirección del espacio
* Devuelve NULL si no encontró el espacio

|  |  |
| --- | --- |
| **ESTATICO** | **DINAMICO** |
| int numero;  numero=12;  scanf(“%d”, &numero);  printf(“%d”, numero); | int\* numero=(int\*)malloc(sizeof(int));  \*numero=12;  scanf(“%d”, numero);  printf(“%d”, \*numero); |
| int numeros[5];  numeros[1]= 12;  scanf(“%d”, &números[1]);  printf(“%d”, números[1]) | int\* numeros=(int\*) malloc(sizeof(int)\*5);  \*(numeros +1)=12;  scanf("%d", numeros +1);  printf("%d", \*(numeros +1)); |
| **Estructuras** | **typedef struct{**  **int id;**  **}ePersona;** |
| ePersona persona;  printf("%d", persona.id);  scanf("%d",&persona.id);  printf("%d", persona.id); | ePersona\* persona=(ePersona\*)malloc(sizeof(ePersona));  persona->id=12;  scanf("%d", &persona->id);  printf("%d", persona->id); |
| ePersona personas[5];  personas[1].id=12;  scanf("%d", &personas[1].id);  printf("%d", personas[1].id); | ePersona\*\* personas=(ePersona\*\*) malloc (sizeof(ePersona\*)\*5);  (\*personas + 1)->id=12;  scanf("%d", &(\*personas + 1)->id);  printf("%d", (\*personas + 1)->id); |
| **Notas adicionales a memoria dinámica:**   * Siempre poner NULL al crear una variable puntero: (**Int\* numero=NULL**) * Siempre asegurarse que haya espacio en la memoria digital. Ejemplo   **ePersona\*\* x=(ePersona\*\*) malloc (sizeof(ePersona\*)\*tam);**  **if(x==NULL) {**  **printf(“No hay espacio”);**  **exit(1);**  **}**   * Una vez concluido el programa liberar la memoria dinámica, ya que esta sigue en uso luego de cerrar el programa (a diferencia de la memoria estática). Ejemplo:   **free(numeros);**   * Sobre todo para los parámetros de las funciones, poner nombres distintivos ya que no usaremos **int num[]** , sino **int\* vecNum** | |
|  | |

* int motrarNumero(int\* x, int tam);
* int motrarNumero(ePersona\*\* x, int tam);

**REALLOC**

* devuelve la dirección de memoria del primero elemento de la memoria
* agranda o achica el array en dinámico. (se debe pasar el tamaño completo no solo en incrementog)
* puede que consiga espacio en otro lugar de la memoria, entonces mueve todo para ese espacio y libera el anterior espacio.
* Si no encuentra devuelve NULL.  **Usar un auxiliar en para no perder el archivo, validar y pasar y borrar el aux.**
* No hace falta usar aux para achicar.  **Pero realloc destruye los datos sobrantes.**
* **Nota: el aux=null luego de ser usado**

**Int\* aux=realloc(vecDim, sizeof(int)\*10);**

**CALLOC**

Hace lo mismo que malloc pero sin basura, ósea deja los valores inicializados en 0 o /0

Int\* nums= (int\*)calloc(2,sizeof(int)); **ahora es array**

**ARCHIVOS**

**TIPOS (**al final ambos tipos de guardado son en binario**)**

* Texto (ASCII)
* Binario
  + Int: complemento A2
  + Float: IEEE754
  + Char: Ascii
  + Char[]: Asciiz (mas el ‘\0’)

**Nota: al final de cada archivo vamos tener un eof al igual que \0**

|  |  |
| --- | --- |
| **r** | **Leer** |
| **w** | **Escribir** |
| **rb** | **Leer de binario** |
| **wb** | **Escribir en binario** |
| **a** | **Append (escribir al final del archivo)** |

**Preparar para escribir**

**FILE\* f= fopen(“nombreArchivo.bin”, “wb”);**

**FILE\* f= fopen(“nombreArchivo.txt”, “w”);**

**…**

**…**

**fclose(f);**

* Si no se puede cargar devuelve NULL
* Si no existe, lo crea(solo en w )
* Si ya existe va a pisar todo lo guardado anteriormente.
* En r necesita validar que no sea NULL
* Fclose crear automáticamente el feof del archivo.

**Escribir y leer en txt**

|  |  |
| --- | --- |
| Escribir | |
| fprintf(f,”%d”, num);  fprintf(f,”%s”, varString); | Usa el txt como el buffer de etrada |
| Leer | |
| fscanf(f,”%d”,& num);  fscanf(f,”%s %s”,var1,var2);  fscanf(f,”%[^,],%[^,],%d”,var1,var2, varInt); | Guardamos un entero  fscanf lee hasta encontrase con un espacio o enter  Caracter especial: lee hasta toparse con el carácter especial, en este caso “,” |
| **Nota importante devoluciones:** | **Fprintf: la cantidad de caracteres que escribió.**  **Sprintf: la cantidad de variables que guardo exitosamente** |
| While(!feof(f)){  fgetc(f);  } | Feof: indica si se llegó el final del archivo  Va carácter a carácter |
| While(!feof(f)){  fgets(auxStr,10,f);  }  While(!feof(f)&& cont<Max\_Lineas){  fgets(auxStr,10,f);  } | Lee de a bloques en este caso 10 caracteres o hasta que llegue al salto de linea  Nota: usar un contador para evitar que repita la cadena final por el salto del feof fantasma |

**Escribir y leer en binario**

**Escribir**

|  |  |
| --- | --- |
| fwrite(&x,sizeof(int),1,file)  fwrite(nombre,sizeof(char),TAM,file);  fwrite(alumno,sizeof(eAlumno),1,file);  fwrite(alumnos,sizeof(eAlumno),TAM,file); | Guardar un entero  guardar una cadena  guardar una estructura  Guardar un array de estructuras |
| * Dirección de memoria de lo que queremos guardar * Tamaño del tipo de variable * **Cantidad de estas variables vamos a guardar, OSEA MULTIPLICA, en caso de una cadena poner el TAM de la cadena(no usar strlen)** * puntero a FILE | |

NOTA TANTO fwrite y fread devuelven la cantidad de valores que cargaron, en este ejemplo solo 1 ya que es una sola variable, en caso de ser un array devolvería la cantidad de index del array cargo

LINKEDLIST

|  |  |
| --- | --- |
| struct Node  {  void\* pElement;  struct Node\* pNextNode;  }typedef Node;  struct LinkedList  {  Node\* pFirstNode;  int size;  }typedef LinkedList;  #endif | Aca se pone struct por que aun no se ha definido  Aca ya esta definido , por eso mas a bajo no usamos struct |

Este tipo de declaración de funciones son usadas solamente dentro del archivo .h ()

**static** Node\* getNode(LinkedList\* this, int nodeIndex)