Clase 2. Arreglos y memoria estática.

Heap y stack

Heap, Stack, and Code

Cuando un programa se va a ejecutar la memoria se organiza en diferentes segmentos.

- Code o text segment. Es donde se carga el codigo o instrucciones de ejecución.

Heap, Stack, and Code

Cuando un programa se va a ejecutar la memoria se organiza en diferentes segmentos.

- Code o text segment. Es donde se carga el codigo o instrucciones de ejecución.
- Stack. Es una pila (LIFO), donde se cargan todas las variables de memoria estática. Es memoria de tamaño predefinido, usada de forma continua.

Heap, Stack, and Code

Cuando un programa se va a ejecutar la memoria se organiza en diferentes segmentos.

- Code o text segment. Es donde se carga el codigo o instrucciones de ejecución.
- Stack. Es una pila (LIFO), donde se cargan todas las variables de memoria estática. Es memoria de tamaño predefinido, usada de forma continua.
- Heap. Es memoria que puede estar fragmentada y que se requiere dinámicamente, el tamaño del heap, usualmente, está determinado por el tamaño de la RAM.

Adicionalmente pueden existir otras áreas:

Heap, Stack, and Code

Cuando un programa se va a ejecutar la memoria se organiza en diferentes segmentos.

- Code o text segment. Es donde se carga el codigo o instrucciones de ejecución.
- Stack. Es una pila (LIFO), donde se cargan todas las variables de memoria estática. Es memoria de tamaño predefinido, usada de forma continua.
- Heap. Es memoria que puede estar fragmentada y que se requiere dinámicamente, el tamaño del heap, usualmente, está determinado por el tamaño de la RAM.

Adicionalmente pueden existir otras áreas:

- Data o globals. El lugar donde se almancenan las variables globales y estáticas.

Heap, Stack, and Code

Cuando un programa se va a ejecutar la memoria se organiza en diferentes segmentos.

- Code o text segment. Es donde se carga el codigo o instrucciones de ejecución.
- Stack. Es una pila (LIFO), donde se cargan todas las variables de memoria estática. Es memoria de tamaño predefinido, usada de forma continua.
- Heap. Es memoria que puede estar fragmentada y que se requiere dinámicamente, el tamaño del heap, usualmente, está determinado por el tamaño de la RAM.

Adicionalmente pueden existir otras áreas:

- Data o globals. El lugar donde se almancenan las variables globales y estáticas.
- Memoria de solo lectura. Algunas definiciones o variables const, pueden ser almacenadas como de solo lectura, esto beneficia la velocidad de ejecución.

Stack

Las variables y arreglos de memoria estática se requieren en el stack.


```
1  /*Esto esta en el data segment*/
   char c[]={"Hola mundo"}
3  int funcion(float x){
   /*Memoria requerida al stack al llamar la funcion*/
5  float z,w;
   /*El codigo como este va al code segment*/
7  w=z+w;
   return 0;
9  }
   int main(){
11 /*La memoria de a esta en el stack*/
   int a;
13 /*La memoria de x esta en el stack*/
   float x[20];
15 /*La memoria de los argumentos va al stack*/
   funcion(x[0]);
17 }
```

Notas sobre eficiencia

- Normalmente usar la memoria del Stack es mas eficiente que la del Heap (mucha de la eficiencia depende del uso del cache). Pero esta memoria es mas accesible por ser una estructura LIFO.

Notas sobre eficiencia

- Normalmente usar la memoria del Stack es mas eficiente que la del Heap (mucha de la eficiencia depende del uso del cache). Pero esta memoria es mas accesible por ser una estructura LIFO.
- La memoria de solo lectura tambien puede eficientar la ejecución, ya que el compilador/sistema, saben que esa memoria esta memoria no cambiará en cierto segmento, y entonces no tienen que recargar su valor desde la RAM, e incluso se pueden cargar en registros de solo lectura (memoria de acceso super rápido en el mismo chip del procesador).

Arreglos

Un arreglo en C es una forma de almacenar datos del mismo tipo bajo un solo nombre de variable, accediendo a cada uno de los elementos por medio de un índice.
En general son listas ordenadas de datos de un mismo tipo.
Ej.

```
/*Arreglo de 6 caracteres*/
char c[6];
/*Arreglo de 11 caracteres de flotante de doble
   precision*/
double x[11];
/*Arreglo de 1000 enteros*/
int z[1000];
```

Arreglos

Los arreglos estáticos se almacenan en el *stack*, por lo tanto el tamaño máximo de los arreglos estáticos está limitado por el tamaño del stack, cuando se requiere mas memoria de la que se ha reservado para el stack es cuando sucede un **Stack Overflow**.

Arreglos

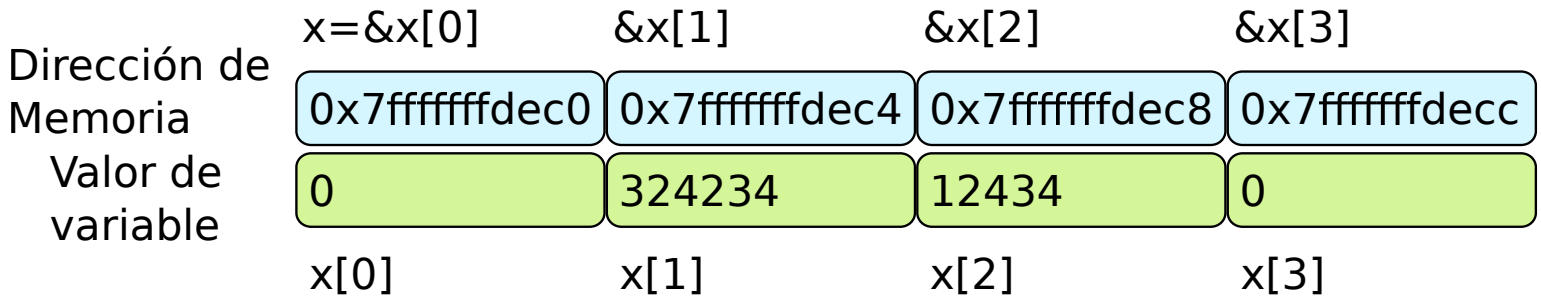
Los arreglos estáticos se almacenan en el *stack*, por lo tanto el tamaño máximo de los arreglos estáticos está limitado por el tamaño del stack, cuando se requiere mas memoria de la que se ha reservado para el stack es cuando sucede un **Stack Overflow**.

Existe una región denominada BSS (Block Started by Symbol), que puede guardar todas las variables no inicializadas en el alcance local. Sin embargo esta región (y todas las anteriores: stack, code,etc.), NO pertencen al estándar de C, sino son implementaciones del sistema operativo. Las variables en esta región se inicializan con 0, pero esto es dependiente del sistema operativo, NO es estándar.

Arreglos y direcciones de memoria

En la declaración siguiente, la memoria se requiere en el Stack de tamaño 5.

```
int main() {
  int x[4];
  return 0;
}
```



Note que:

- **x** sin ningún corchete, es la dirección de x[0].
- Las direcciones van aumentando de 4 en 4 (bytes).
- El operador & nos devuelve la dirección de cualquier posición.

Declaración de arreglos

Los arreglos estáticos (hasta C89) se declaran:
tipo nombre[const dimension];
Ejemplo:

```
int x[10];
```

También es valido:

```
int x[]={1,2,3,4,5,6,7,8,9,10};
```

Desde C99, se puede también:

```
int n;
scanf("%d",&n);
int x[n];
```

Cabe destacar que la última forma de declarar un arreglo requiere memoria en el Stack, entonces el tamaño máximo de este arreglo está limitado por el Stack.
En ambos casos, esta memoria solo es local a la función donde se declara la variable.

Arreglos en funciones


```
1 int funcion(int z
      []) {
    z[0]=z[1];
3 return 0;
  }
```

```
int funcion2(int []);
2 int funcion2(int z
      [15000]) {
    z[0]=z[3];
4 return 0;
  }
```

```
1 int main()
  {
3   int x[4];
    funcion(x);
5   return 0;
  }
```

Como se puede ver en la definición solo hay que poner el nombre, en la declaración no.

Es posible que acceder a posiciones mayores que las declaradas (sin que marque un segmentation fault) mientras esta memoria pertenezca al stack.

El apuntador de x se copia a z, pero las posiciones se acceden por el operador [].

Arreglos en funciones


```
int funcion(int *z)
{
    z[0]=z[1];
return 0;
}
```

```
1 int main()
{
3   int x[4];
    funcion(x);
5   return 0;
}
```

La dirección de memoria de z se copia a x. ¿Cual es la diferencia entre:

int z[]; y int *z; ?
Es que z[], es una dirección de memoria estática, no se puede asignar un valor diferente a z (es rvalue).
Mientras que int *z, z es una variable (lvalue) al que se le puede asignar cualquier valor posible de las direcciones de memoria.

Arreglos en funciones

Ejemplo. Supongamos que declaramos:
int x[12];

Arreglos en funciones

Ejemplo. Supongamos que declaramos:
int x[12];

y que la dirección de memoria en x es 0x7fffffffdec0. Esta es la dirección de memoria de donde comienza el vector, es decir la dirección de memoria de &x[0].

Arreglos en funciones

Ejemplo. Supongamos que declaramos:
int x[12];

y que la dirección de memoria en x es 0x7fffffffdec0. Esta es la dirección de memoria de donde comienza el vector, es decir la dirección de memoria de &x[0].

Cuando se asigna x[0]=5;
El operador [], indica que:
x[0], es el contenido de la dirección a la que apunta x, 0 posiciones adelante de donde empieza.
x[1], es el contenido, que tiene la dirección que está una posición adelante de x, como x es entero, una posición son 4 bytes, entonces: 0x7fffffffdec0 + 4= 0x7fffffffdec4=&x[1],

Avance del un arreglo en un for

El tipo del arreglo (o apuntador) determina el avance en las posiciones.

```
int main(){
 float z[]={1,2,3,4,5,6,7,8,9,10};
 float *v=z;
 for (int i=0; i<10; i++){
  printf("&z[i]=%p  v=%p  z[i]=%f  *v=%f\n",&z[i],v,z[i],*v);
  v=v+1;
 }
return 0;
}
```

Note que el apuntador v si puede ser modificado (pero z no), y que avanzar uno en este apuntador es recorrer 4 bytes (el tamaño de un float).

Avance del un arreglo en un for

Los apuntadores son del mismo tamaño, no importa si apuntan a char, float, etc. Así que un apuntador de un tipo puede ser almacenado en uno de otro tipo, sin embargo el tipo determina el avance.

```
1  int main()
   {
3    float z[]={1,2,3,4,5,6,7,8,9,10};
     char *y;
5    for (int i=0; i <10; i++)
     {
7          printf("%p %c %f %f\n",&y[i],y[i],y[i],z[i]);
     }
9    return 0;
   }
```

Arreglos multidimensionales estáticos


```
float z[2][3]={1,2,3,4,5,6};
float x[2][3][2]={1,2,3,4,5,6,7,8,9,10,11,12};
float *v=z; //v[n] es igual a z[i][j], n=dim2*i+j
v=x; //v[n] es igual a x[i][j][k], n=dim2*dim3*i+dim3
  *j+k
```

Un arreglo multidimensional estático (solo los estáticos) en realidad es un arreglo unidimensional, solo se indiza de esta forma por facilidad, pero en realidad es un arreglo unidimensional.

Arreglos multidimensionales estáticos


```
float x[2][3][2]={1,2,3,4,5,6,7,8,9,10,11,12};
```

x es la direccion donde inicia la memoria.
x[1] es la direccion donde inicia la matriz 1 (la segunda matriz).
x[1][1] es la dirección donde inicia el segundo renglń de la segunda matríz.
x[1][1][0] es el contenido de la dirección anterior.

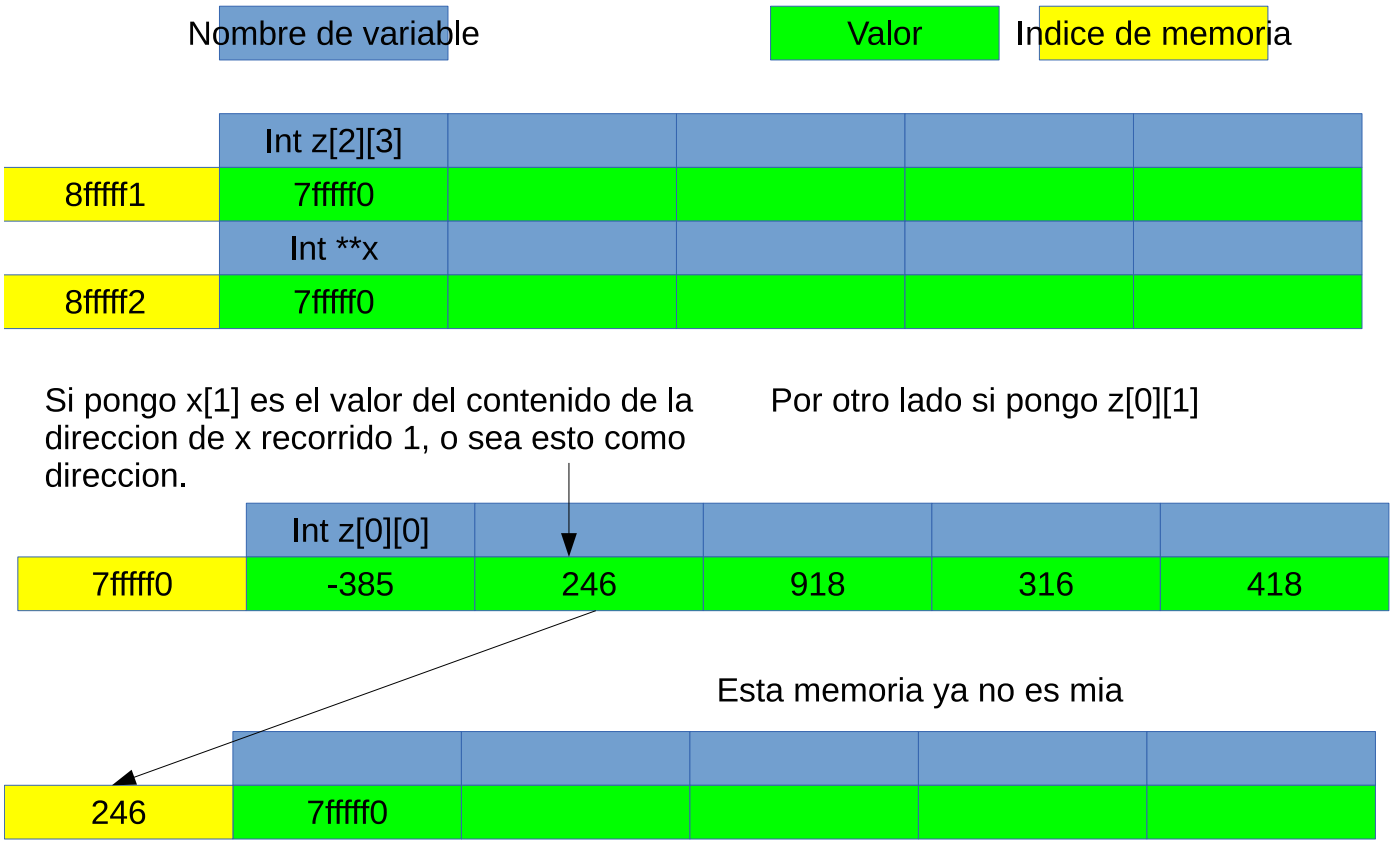
Arreglos multidimensionales estáticos

Ya que los arreglos estáticos multidimensionales, en realidad son unidimensionales. El avance en los indices (cuantas posiciones recorrer por renglón), lo determina la dimensión al momento de

```
1  int fun(float x[4][4]){
       printf("%f\n",x[1][0]);
3  }
   int main(){
5      float z[4][4];
       float x[2][3];
7      for (int i=0; i<4;i++)
           for (int j=0; j<4; j++)
9              z[i][j]=i*4+j+1;

11     for (int i=0; i<2;i++)
           for (int j=0; j<3; j++)
13             x[i][j]=i*4+j+1;
   return 0;
15 }
```

Ejemplo



Tarea 2

prog2.1 Escriba funciones para números float, int y double que reciban un vector del tipo indicado y un entero con el tamaño del vector y calcule: la suma, el promedio, la desviación estándar, varianza, el máximo, el mínimo, la norma L2. Si ya tienen una función (por ejemplo la suma), la pueden usar para otra de las funciones (por ejemplo varianza). Utilice las funciones en un programa de prueba.

prog2.2 Escriba funciones para datos double y float que reciban 2 vectores y calculen: la suma de los vectores, el producto punto, la proyeccion de uno sobre el otro, y normalice (que la norma L2 valga 1) un vector de entrada y escriba su valor normalizado en el otro, la multiplicación de un vector por un escalar (y almacenar en otro).

prog2.3 Escriba funciones para double y float que realicen el producto de dos vectores, considerando que uno es de Nx1 y otro de 1xM, y se obtenga una matriz de NxM.

prog2.4 Escriba un programa que realice un Stack Overflow, reportar el sistema operativo, el procesador y RAM del sistema donde se ejecutó.

prog2.5 Imprimir direcciones de memoria de los arreglos estáticos antes y depués de enviarlo a una función. Verificar que pasa sí: declaro un arreglo de un tipo, pero lo envio a un arreglo de un tipo diferente, por ejemplo; mi funcion recibe double y envío un float o entero. Mostrar cuanto avanza el apuntador en un for en los casos anteriores. Mostrar que el avance depende del tipo de dato y del número de renglones o columnas, etc. que se definieron.

prog2.6 Realice funciones (considerando arreglos estáticos) que: multipliquen dos vectores de Mx1 y 1xN, para obtener un matriz de MxN. sumen dos matrices, resten dos matrices, multipliquen dos matrices, multipliquen una matriz por un vector (almacenar en otro vector), multipliquen una matriz por un escalar (y almacenar en otra).

Lectura y escritura de datos a un vector

La lectura y escritura es similar a una variable si, solo hay que utilizar el índice;

```
1 float x[5], y[4][5], z[2][3][3];
  printf("%f %f %f",x[3],y[2][3],z[1][1][0]);
3 scanf("%f %f %f",&x[3],&y[2][3],&z[1][1][0]);
```

Solo existe un formato para leer y escribir un vector completo, que es %c. Ejemplo:

```
1 char c[]={"Hola"}; //Vector tamanio 5
  printf("%s",c);
3 scanf("%s",c);
```

Para asegurar que no haya un acceso mal de memoria o corrupción, se debe de requerir suficiente memoria en c.

Lectura y escritura de archivos

Lectura y escritura de archivos

- Primero se requiere una libreria que permita leer y escribir de y hacia archivo, que es la misma que la de entrada y salida estándar: stdio.h.
- Se abre el archivo y se almacena la dirección en un apuntador.
- Se escribe o lee del archivo usando el apuntador.
- Se cierra el archivo.

Ejemplo, escritura:

```
1 FILE *out;
  //Abrir archivo
3 out=fopen("nombre.txt","w");
  //Escribir en el archivo
5 fprintf(out,"Hola\n");
  //Cerrar archivo
7 fclose(out);
```

Modos de apertura

Modos de apertura.
Los de lectura necesitan que el archivo exista para abrirlo para lectura.
Los de escritura sobreescriben el archivo.
Los de anexar abren el archivo al final de su contenido actual para escritura.

```
1  //Abrir archivo para lectura
   FILE *escritura=fopen("nombre.txt","w");
3  FILE *anexar=fopen("nombre.txt","a");
   FILE *lectura=fopen("nombre.txt","r");
5  FILE *escrituralectura=fopen("nombre.txt","w+");
   FILE *lecturaescritura=fopen("nombre.txt","r+");
7  FILE *lecturaescrituraanexar=fopen("nombre.txt","a+");
   fclose(escritura);
9  fclose(anexar);
   fclose(lectura);
11 ...
```


Si un archivo no puede abrirse la función regresa NULL

Tarea 2

En la tarea 2, lea y escriba todos los datos desde y hacia archivo de texto.

Referencias

Arrays:
http://www.tutorialspoint.com/cprogramming/c_arrays.htm
Heap, code, stack.
http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch10s04.htm
Lectura y escritura de archivos
http://forum.codecall.net/topic/51524-reading-and-writing-files-in-

Temas futuros no cubiertos en está clase:

- Arreglos con memoria dinámica.
- Optimización utilizando el cache.
- Paso de argumentos a funciones con memoria dinámica.
- Arreglos de tipos de datos declarados.
- Algoritmos de ordenamiento y otras operaciones con vectores.
- Archivos binarios.