Tema 3: Asignación Dinámica de Memoria

Introducción a la programación II

Ana Isabel Sierra de las Heras
Marcos Novalbos
Tiago Manuel Louro Machado de Simas
Rodrigo Alonso Solaguren-Beascoa
Alfonso Castro Escudero

Índice

1 Introducción

- Los programas pueden crear variables globales o locales, cada una de las cuales tiene un ámbito diferente y se almacenan en diferentes áreas de la memoria. Las variables globales son declaradas en la cabecera del programa, antes de la declaración de funciones y antes del método main y son conocidas desde cualquier función, mientras que las variables locales son declaradas dentro de una función y sólo son conocidas dentro de ella.

- Las variables declaradas como globales se almacenan en posiciones fijas de memoria, en la zona conocida como **segmento de datos del programa**, y todas las funciones pueden utilizar estas variables.

- Las variables locales se almacenan en la **pila (stack)** y existen sólo mientras están activas las funciones en las que están declaradas.

- Es posible también crear variables `static` (similares a las globales) que se almacenan en posiciones fijas de memoria, pero solo están disponibles en el módulo o función en que se declaran.

1 Introducción

- Todas estas variables se definen cuando se compila el programa. Esto significa que el compilador reserva (define) espacio en memoria para almacenar valores de los tipos de datos declarados en el momento de la definición de las variables.

- Pero existen infinidad de aplicaciones en las cuales no se conoce la cantidad de memoria necesaria para almacenar los datos hasta que no se ejecuta el programa.

- Por ejemplo,
  - Si se desea almacenar una cadena de caracteres tecleada por el usuario, no se puede prever, a priori, el tamaño del array necesario, a menos que se reserve un array de gran dimensión y se malgaste memoria cuando no se utilice.

1 Introducción

- Además, la cantidad de memoria estática que usamos para variables, programas, etc... está limitada en tiempo de ejecución, siendo actualmente un tamaño máximo estándar de 10MB (depende de flags de compilación y SO).

- Es decir, no podemos crear arrays “infinitos”.

- Pero nuestros ordenadores tienen varios GB de RAM, ¿qué ocurre si se necesita más memoria?

  → **Hacer gestión explícita de memoria dinámica mediante peticiones de memoria al sistema operativo**.

1 Introducción

- La solución es pues recurrir a técnicas de asignación dinámica de memoria.
- En C se puede asignar memoria en el momento de la ejecución. La memoria que se gestiona dinámicamente se almacena en el **heap** (es la zona de memoria dinámica).
- El espacio de la variable se crea dinámicamente durante la ejecución del programa.
- El programa puede solicitar o liberar memoria dinámica en cualquier momento durante le ejecución.
- Las funciones **malloc(), realloc(), calloc() y free()** asignan y liberan memoria de manera dinámica durante la ejecución del programa.

Error típico de programación en C:
En C no se puede determinar el tamaño de un array en tiempo de ejecución

2 Función `malloc()`

- La forma más habitual en C para obtener bloques de memoria es mediante la llamada a la función **`malloc()`**
- La función `malloc()` reserva un bloque de memoria cuyo tamaño en bytes es el número que se le pasa como parámetro.
- `malloc()` devuelve un puntero, que es la dirección del primer byte del bloque asignado de memoria.
- El puntero se utiliza para referenciar el bloque de memoria asignado.
- El puntero que devuelve es del tipo void* (*puntero a void = puntero a cualquier tipo*).
- El prototipo es:

```
void* malloc (size_t n);
```

2 Función `malloc()`

- La forma de invocar la función malloc() es:

```
puntero = malloc (tamaño en bytes);
```

- Generalmente se hará una conversión al tipo del puntero:

```
puntero = (tipo *) malloc (tamaño en bytes);
```

- El operador unitario sizeof se utiliza con mucha frecuencia en las funciones de asignación de memoria para calcular el número de bytes que se desea solicitar. Este operador se aplicará a un tipo de dato (o una variable), para obtener el número de bytes que ocupa.

- **Al invocar la función `malloc()` puede ocurrir que no haya memoria disponible, en este caso `malloc()` devuelve NULL.**

2 Función `malloc()`

- **La característica principal de un puntero void es que puede apuntar a la dirección de cualquier tipo de dato,** sin tener que hacer una conversión explícita del mismo. Veamos un ejemplo:

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

int main (void){
   void * ptrVoid;
   int a = 78, arr[20];
   float r = 283.91, * ptrFloat;
   char nombre[30] = "Ana Roncal";

   /* Las siguientes instrucciones son válidas, no requieren de una conversión explícita del dato asignado (cast)*/

   ptrVoid = &a;           // recibe la dirección de la variable entera a
   ptrVoid = arr;          // apunta al primer elemento del array de enteros arr
   ptrVoid = &arr[5];      // recibe la dirección del quinto elemento del array arr
   ptrVoid = &r;           // recibe la dirección de la variable de punto flotante r

   ptrFloat = malloc(sizeof(float));

   *ptrFloat = 456.78;
   ptrVoid = ptrFloat;      // apunta a la misma dirección a la que apunta el puntero ptrFloat

   ptrVoid = nombre;        // recibe la dirección de cadena nombre
}
```

3 Liberación de memoria Función `free()`

- Cuando se ha terminado de utilizar un bloque de memoria previamente asignado por `malloc()`, u otras funciones de asignación, se debe liberar el espacio de memoria y dejarlo disponible para otros usos.
- La función para liberar memoria es `free()`.
- **Por cada `malloc` escrito, se debe escribir un `free` en algún punto de nuestro código.**
- El bloque de memoria suprimido se devuelve al espacio de almacenamiento libre, de manera que habrá más memoria disponible para asignar bloques de memoria.
- El prototipo es:

```
void free (void *);
```

3 Función `malloc()+ free()`

- Veamos un ejemplo en el que se repasa el concepto de void * que devuelve el malloc:

```
#include <stdio.h>
                                                        //todo en un paso
void main (int argc, char* argv[]){

   //En memoria estática
   int valor;
   valor = 5;
   printf("valor = %d\n",valor);

   //En memoria dinamica
   //puntero a void vale para un puntero a cualquier tipo
   void *ptr = NULL;
   ptr = malloc (sizeof(int));

   //hasta que no pongamos el puntero de un tipo no vamos a
   // poder incluir un valor en el definicion de un puntero int
   int *prt_int = NULL;

   //conversión del puntero void al puntero int
   ptr_int = (int *) ptr;
   //Ahora ya se puede incluir un valor entero en la memoria dinamica
   *ptr_int = 5;
   printf ("ptr_int: %d \n", *ptr_int);

   free (ptr);
```

Ejemplo 1: uso de `malloc()` y `free()`

- Pedir al sistema operativo que nos devuelva memoria para almacenar 10 enteros e imprimir el contenido de la memoria reservada.

```
#include <stdio.h>
#include <stdlib.h>
#define NUM_ENTEROS 10

int b=15;
int *c = NULL; //Ahora nuestro array es un puntero sin inicializar

void main (int argc, char * argv[]){
  //Debemos inicializarlo antes, le pedimos al SO que nos de 40bytes (10 integers)
   c=(int*)malloc(40); //c = (int *)malloc (NUM_ENTEROS * sizeof(int));
   //comprobar si el malloc ha ido bien
   if (c ==NULL){
      printf ("No se ha podido reservar memoria\n" );
   }else{
      for(b=0;b< NUM_ENTEROS;b++){
         c[b]=b; //Inicializamos los valores de los enteros idem *(c+b)=b;
      }
      int* pc=c; //Declaramos otro puntero y lo hacemos apuntar a la reserva realizada
      for(b=0;b< NUM_ENTEROS;b++){ //Imprimimos de tres maneras diferentes
         printf("Los valores del array dinamico c[%d] = %d \n", b, c[b]);
         printf("El contenido apuntado por el puntero devuelto por el malloc: %d \n", *(c+b));
         printf("El contenido apuntado por el puntero que apunta a la direccion de memoria devuelta por el malloc: %d \n", *pc);
         pc++;
      }
   }
   //una vez usado, se libera
   free(c);
}
```

Ejercicio 1 – Gestión de memoria dinámica

- Crear un programa genérico, que pueda ser usado para cualquier grupo de U-tad, que calcule y pida los datos necesarios para:
  - La edad media de los alumnos del grupo
  - El máximo de las edades
  - El mínimo de las edades
  - Imprimir las edades de los alumnos.

Gestión de errores:

```
ptr_int = (int*)malloc(sizeof(int));
   if (ptr_int == NULL){
      printf ("Error: No hay espacio de memoria\n");
   }
   else{
      printf ("Se ha asignado la memoria ptr_int \n");
   }
```

4 Gestión de errores

- La gestión de errores en C se deja a cargo del programador, debido a cómo funciona el sistema:
  - Los errores de memoria se detectan a posteriori. Sólo conocemos/sabemos sus efectos, no lo que los causó.
  - Es posible que hubiéramos corrompido la memoria mucho antes. Es difícil de depurar, por lo que se recomienda ser lo más cuidadoso posible.

Consejo:
Por cada array reservado dinámicamente, llevad una variable “tamaño” que lo indique.

4 Gestión de errores

- La función malloc se encarga de pedir nuevos segmentos de memoria dinámica (heap) al sistema operativo.
- Pertenece al estándar de C, y cada implementación de la función realizará internamente la gestión de memoria necesaria para reducir las llamadas al SO:
  - Pedimos 10 integers (40 bytes) en memoria dinámica
  - malloc reservará bloques más grandes, y nos devolverá un puntero al inicio de ese bloque
  - Hemos de ser nosotros los que aseguremos que no nos salimos de esa memoria
  - El sistema operativo sólo se dará cuenta de los errores de memoria cuando nos hayamos salido de los bloques reservados por `malloc()`

Ejemplo 2: uso de `malloc()` y `free()`

Declarar dos listas de enteros a partir de su dirección de comienzo, el tamaño lo determinará el usuario al comienzo del programa. Inicializar una de ellas y copiar los valores en la otra. Mostrar los resultados.

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

int main (int argc,char * argv[]){

   int* c = NULL; //Ahora nuestro array es un puntero sin inicializar
   int* d =NULL; //Segundo array de memoria
   int a=15;

   //Debemos inicializarlo antes, le pedimos al SO que nos de 40bytes (10 integers)
   c=(int*)malloc(40);
   d=(int*)malloc(40);

   for(int b=0;b<a;b++){
               printf("%d \n", c[b]); //accesos inválidos en lectura (no muy importantes)
   }
   for(int b=0;b<a;b++){
               c[b]=d[b];//accesos inválidos en escritura. Se están corrompiendo zonas de memoria “desconocidas”
   }

   //una vez usado, se libera
   free(c);
   free(d);
   return 0;
}
```

5 Memoria dinámica: uso de sizeof()

- En temas anteriores vimos la importancia de usar los tipos de datos redefinidos con “typedef”, estamos creando un nuevo tipo de datos controlado por nosotros:
  - Sabemos su tamaño
  - Sabemos sus rangos válidos
  - Sabemos su estructura interna

- Para poder facilitar la reserva de memoria, tenemos la macro/palabra reservada “**sizeof**”. Esta función nos devuelve el tamaño en bytes de un tipo de datos definido anteriormente:
  - Tipos básicos integer, char, float...
  - Tipos estructurados /uniones
  - Tipos definidos con typedef

- NO USAR SIZEOF CON VARIABLES DE TIPO PUNTERO, no hay manera de saber el tamaño de memoria de una zona de memoria reservada con malloc (hay soluciones, pero no son estándar)

- SIZEOF TAMPOCO SIRVE PARA SABER LA LONGITUD DE ARRAYS/CADENAS
(Sólo funciona en arrays estáticos, y bajo circunstancias concretas)

6 Funciones `calloc()` y `realloc()`

- Los prototipos de cada una de las funciones de asignación dinámica de memoria son:

```
void *malloc(size_t size);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

- Ya vimos que `malloc()` devuleve un puntero con la dirección del bloque de memoria reservado dinámicamente, pero es importante tener en cuenta que `malloc()` no inicializa el contenido de este bloque de memoria.

- `malloc()` debe recibir como parámetro el tamaño del bloque que se desea reservar, es decir, nº elementos * tamaño de elemento.

6 Funciones `calloc()` y `realloc()`

- La función `calloc()` cumple la misma función de `malloc()` pero con alguna diferencia:
  - `calloc()` sí inicializa la memoria reservada poniendo el contenido a 0 de cada elemento del array dinámico.
  - En la definición de `calloc()` el primer parámetro es el número de elementos y a continuación el tamaño del elemento.

```
void *calloc(size_t nmemb, size_t size);
```

  - `calloc()` computacionalmente es algo más cara.

6 Funciones `calloc()` y `realloc()`

- La función `realloc()` sirve para redimensionar un bloque de memoria asignado anteriormente de forma dinámica.

- Su prototipo y sus parámetros son:

```
void *realloc(void *ptr, size_t size);
```

  - **ptr**: el puntero a redimensionar
  - **size_t**: el nuevo tamaño (nº elementos * tamaño de elemento)

  Devuelve un puntero a la dirección de memoria con el nuevo tamaño.

- Esta función guarda el contenido que se tiene almacenado en el bloque de memoria a redimensionar.

6 Funciones `calloc()` y `realloc()`

- Si se solicita "agrandar" el tamaño y al redimensionar no existe espacio consecutivo para el nuevo bloque `realloc()` cambia de lugar del bloque y devuelve un puntero a la nueva dirección.
- En el caso de se haga más pequeño elimina el contenido sobrante.
- Si se pasa cero (0) como tamaño del bloque a redimensionar, entonces se libera el bloque de memoria al que está apuntando el puntero primer argumento, y la función devuelve NULL.
- Si el primer argumento tiene el valor de NULL, la función reserva tanta memoria como la indicada por el segundo argumento, como malloc ( ).
- Por esto siempre se debe guardar el puntero obtenido con esta función.

Ejemplo 3: uso de `malloc(),calloc() y realloc()`

- Crear un array de int dinámicamente de 100 posiciones:

```
int *v, *p;
v = (int *) malloc(100*sizeof(int));
v = (int *) calloc(100,sizeof(int));
```

- Redimensionar v para que tenga 150 posiciones.

```
p= (int *) realloc(v,150*sizeof(int));
```

Ejercicio 2 – `malloc(),realloc ()`

- Crear un programa que recoja el nombre de un alumno, mediante el uso de reserva de memoria dinámica. Genera para ello una función con el siguiente prototipo

```
char *leeLineaDinamica ();
```

- Presentarlo después por pantalla

Ejercicio 3 – `malloc(),realloc ()`

- Crear un programa que recoja el nombre de los alumnos de una clase de Utad, mediante el uso de reserva de memoria dinámica. Utiliza para ello la función creada en el anterior programa con el siguiente prototipo

```
char *leeLineaDinamica ();
```

- Presentarlos después por pantalla
- Completad el programa para que recoja el nombre de los alumnos de todos los grupos de U-tad

Ejercicio 4 `malloc(),realloc ()`

- Crear un tipo estructura “libro” con los siguientes miembros:
  - NumPag de tipo entero
  - Autor que va a ser una cadena un tamaño indeterminado
  - Título que va a ser una cadena un tamaño indeterminado

- Crear una biblioteca preparando un menú con las siguientes opciones:
  - Introducción de un nuevo ejemplar
  - Finalizar la introducción de ejemplares

- Cuando el usuario haya acabado de introducir libros, se debe escribir un listado con el nombre y el número de páginas de cada libro.

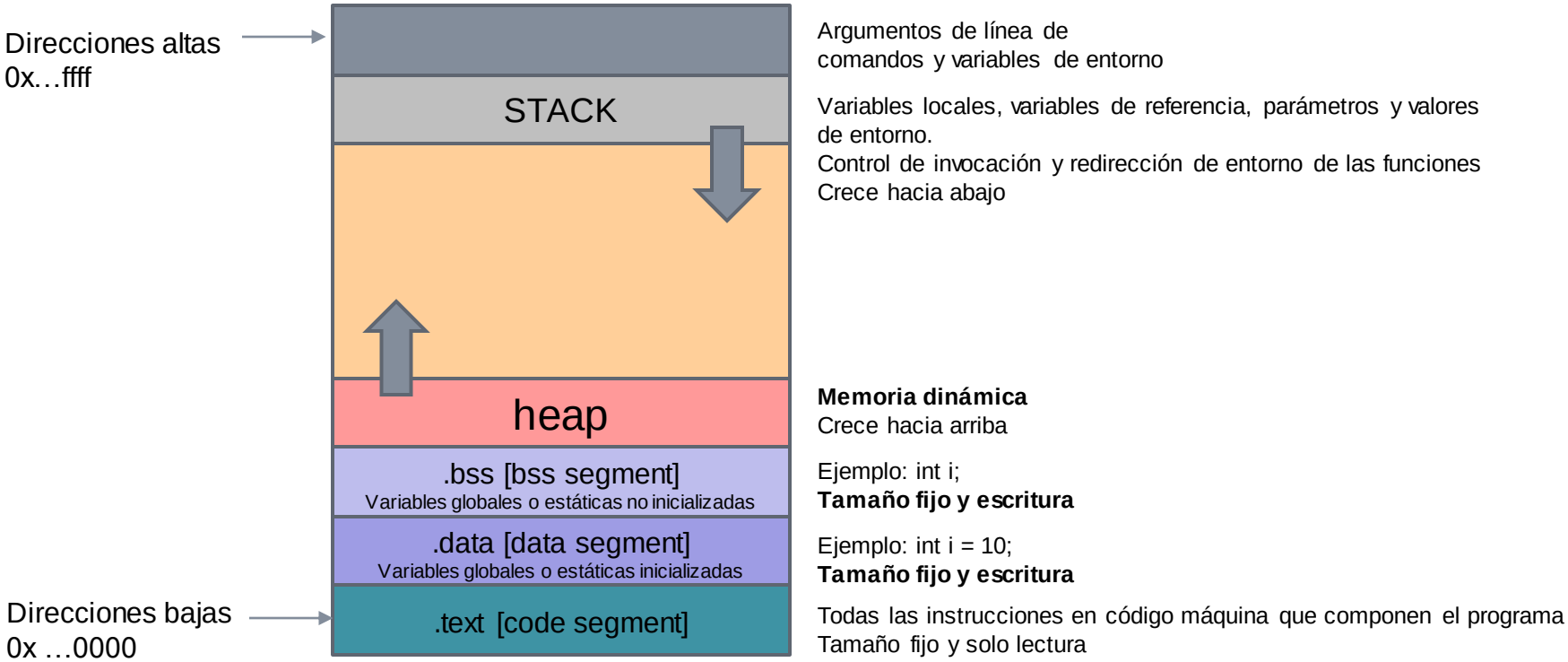
- Incluir en el programa la funcionalidad de préstamo.

Ejercicio 5 `malloc(),realloc ()`

- Solicitar al usuario que introduzca por teclado un conjunto de números enteros.
- Estos deben irse almacenando mientras que sean menores que el primero introducido.
- En el momento que esta condición no se cumpla no se deben pedir más números al usuario y deben escribirse todos los que haya introducido hasta ese momento.

7 Secciones de memoria de un programa

*La forma en como se divide la memoria de un programa en ejecución, se conoce como **segmentación de memoria de un programa**. En la siguiente figura podemos ver cuales son sus segmentos, que objetivos tienen y que se almacena en cada uno de ellos.*



7 Secciones de memoria de un programa

- ***.text :*** Es de tamaño fijo y de solo lectura. En esta parte se almacenan todas y cada una de las instrucciones en **código máquina** que componen el programa que se está ejecutando. Como curiosidad añadir que en caso de existir varias instancias en ejecución del mismo programa o binario el sistema operativo actúa de forma inteligente manteniendo una sola copia en memoria del código y permitiendo que los procesos puedan compartirla para ahorrar recursos.

- ***.data [data segment]*:** Aquí se almacenan las variables globales inicializadas del programa. De tamaño fijo y permite la escritura, se puede escribir en él.

- ***.bss [bss segment]:*** Aquí se almacenan las variables globales sin inicializar. De tamaño fijo y permite la escritura, se puede escribir en él.

7 Secciones de memoria de un programa

- ***Heap [heap segment]:*** Segmento de memoria reservado para la **memoria dinámica** del programa. El tamaño de este segmento no está predefinido, va variando. Crece hacia arriba, en el mismo sentido que las direcciones de memoria. Para reservar memoria utilizamos, por ejemplo, las conocidas funciones de asignación malloc(), calloc() o realloc() del lenguaje C. Para liberar memoria utilizaremos la función free().

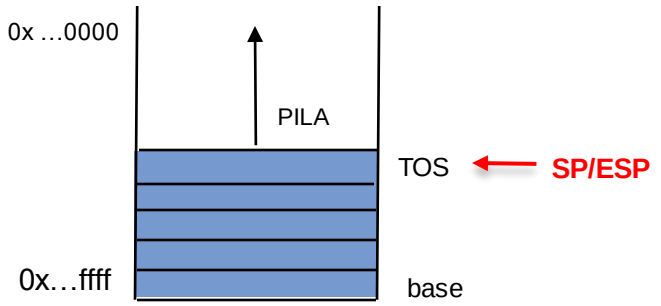
- ***STACK [stack segment]:*** La pila o stack. Aquí se guardan los argumentos pasados al programa, las variables del entorno donde éste es ejecutado (el comando env permite su visualización), argumentos pasados a las funciones, las variables locales y además, es donde se almacena el registro IP (Instruction Pointer) cuando se invoca una función (dirección o punto de retorno). El tamaño de este segmento es fijo y se establece durante la compilación del programa. Crece hacia abajo, en sentido inverso que las direcciones de memoria. El tamaño predeterminado de la pila suele ser de unos pocos megabytes, pero puede variar según el sistema operativo y el compilador

7 Secciones de memoria de un programa

- ***STACK [stack segment]:***
  - La estructura del stack o pila es una lista o estructura ordenada en la cual, el almacenamiento y acceso a los datos se corresponde con un esquema **LIFO** (Last in, First out, último en entrar, primero en salir).
  - Para el manejo de datos, cuenta con dos operaciones: push/pop que permiten manipular la pila o stack, de manera que los elementos
    - se almacenan/apilan con **push**
    - se retiran/desapilan con **pop**.
  - La pila se utiliza para almacenar: valores de registros de manera temporal, variables locales, parámetros de funciones y direcciones de retorno.
  - En cada momento sólo se tiene acceso a la parte superior de la pila, es decir, al último objeto apilado (denominado TOS, **Top of Stack** en inglés).

Puntero de pila:

- El **stack pointer, SP** (o extended stack pointer, ESP) es un puntero especial que apunta al "tope de la pila", es decir al último elemento almacenado en ella.
- Cuando se almacena un nuevo valor en la pila con push, el valor del puntero se actualiza para siempre apuntar al tope de la pila.
- Cuando se desapila un dato con pop, el dato o información apuntada por el SP se almacena en el registro indicado en la instrucción, y el valor del SP se actualiza con el nuevo tope de pila.

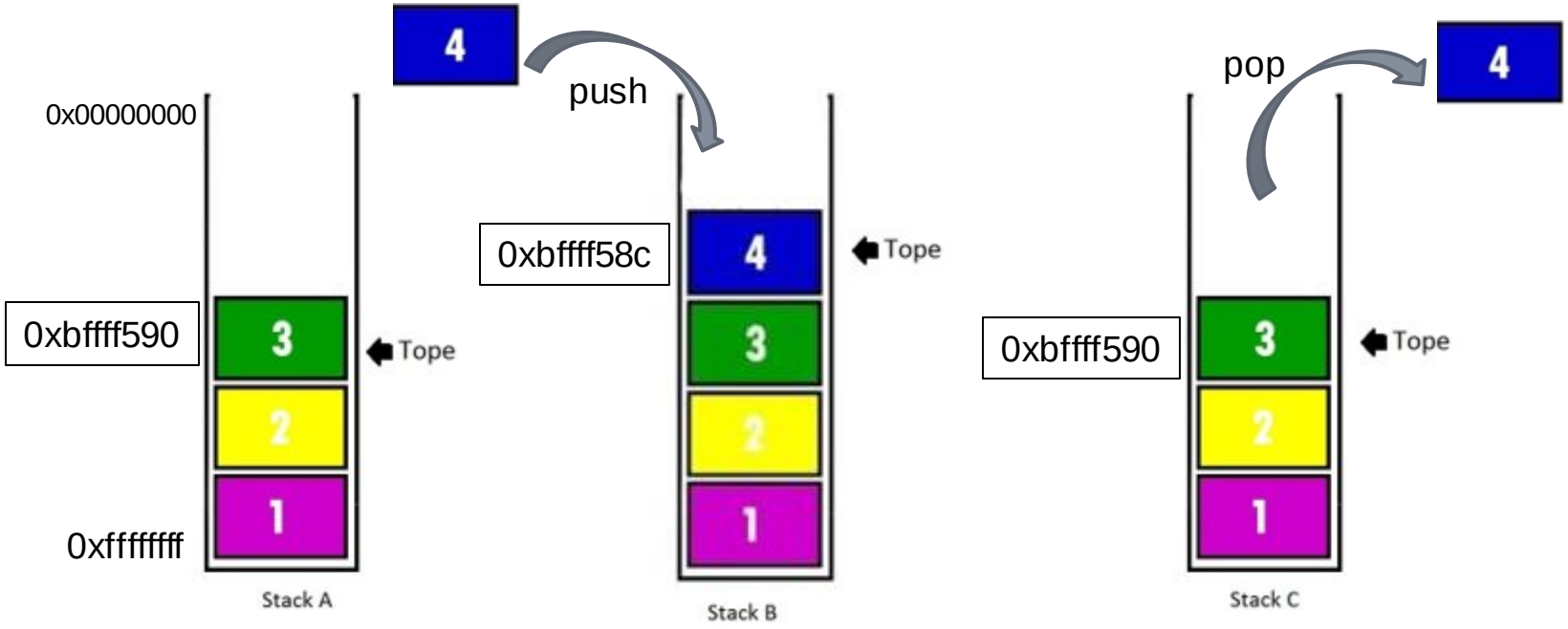


7 Secciones de memoria de un programa

- ***STACK [stack segment]:***
  - La pila crece desde direcciones numéricas mayores (que son usadas primero) hacia las direcciones de memoria menores. Es decir, crece desde 0xf...fff hacia 0x0...000.
  - Como la pila crece desde su base -desde la dirección más alta- hacia direcciones menores de memoria, al apilar un nuevo elemento se debe decrementar el puntero de la pila y al desapilar un elemento se debe incrementar el puntero de la pila. (Por eso con push de un int se resta SP= SP - 4 y con pop de un int se suma SP = SP + 4)
  - Por tanto, y en contra de lo que nos parece intuitivo, al almacenar un nuevo valor en la pila con push, la pila crece hacia las direcciones numéricas menores.
  - Por ejemplo:
    - Al añadir un nuevo elemento al stack, el SP de la pila pasa de la dirección 0xbffff590 a estar en 0xbffff58c.
    - Al sacar un elemento el proceso es el inverso, el SP crece desde 0xbffff58c hasta 0xbffff590, es decir “decrece” hacia direcciones mayores.

7 Secciones de memoria de un programa.

Ejemplo de operaciones push y pop sobre el stack



Ejemplo 4: Direcciones de memoria

- Programa con el que se visualizan las ubicación relativa en memoria de: Variables globales, argumentos del pasados al programa, variables locales y memoria dinámica

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

int c;
int d;
void main(int argc, char** argv){
    int a;
    int b;
    int *ptr=(int*)malloc(sizeof(int)*100000);
    printf("global 1   :%p \n", &c);
    printf("global 2   :%p \n", &d);
    printf("cabecera 1 :%p \n", &argc);
    printf("cabecera 2 :%p \n", &argv);
    printf("local 1    :%p \n", &a);
    printf("local 2    :%p \n", &b);
    printf("malloc     :%p \n",ptr);
}
```

- La salida del programa con las direcciones de memoria de cada variable:

```
global 1   :0x5597508e5048
global 2   :0x5597508e5044
cabecera 1 :0x7ffc2b9f651c
cabecera 2 :0x7ffc2b9f6510
local 1    :0x7ffc2b9f6528
local 2    :0x7ffc2b9f652c
malloc     :0x7f79fbf30010
```

Ejemplo 5: Direcciones del stack y del heap

- Programa con el que se visualizan la forma en la que el stack y el heap crece.

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

void func(int valor){
  unsigned int c;
  unsigned int *ptr;

  ptr = malloc(1);
  printf("Stack: %p | Heap: %p\n", &c, ptr);
    if(valor <= 0) return;
  func(valor - 1);
}

int main(){
  func(10);
  return 0;
}
```

- La salida del programa con las direcciones de memoria del stack o del heap:

```
Stack: 0xffffcbe4 | Heap: 0x6000003a0
Stack: 0xffffcba4 | Heap: 0x600048470
Stack: 0xffffcb64 | Heap: 0x600048490
Stack: 0xffffcb24 | Heap: 0x6000484b0
Stack: 0xffffcae4 | Heap: 0x6000484d0
Stack: 0xffffcaa4 | Heap: 0x6000484f0
Stack: 0xffffca64 | Heap: 0x600048510
Stack: 0xffffca24 | Heap: 0x600048530
Stack: 0xffffc9e4 | Heap: 0x600048550
Stack: 0xffffc9a4 | Heap: 0x600048570
Stack: 0xffffc964 | Heap: 0x600048590
```

Ejemplo 6: Segmentación de memoria

- Programa con el que se muestra como el compilador va generando la imagen completa de memoria, para ello se utiliza la herramienta `objdump`.

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

int b=15;
int c[10]={0};
int main (int argc,char * argv[]){
   int a=10;
   for(b=0;b<a;b++){
       printf("%d \n", c[b]);
   }
   return 0;
}

home$> gcc -g -c ejem1.c -o ejem1.o
home$> objdump.exe -s -j .data ejem1.o
home$> objdump.exe -d -S ejem1.o
```

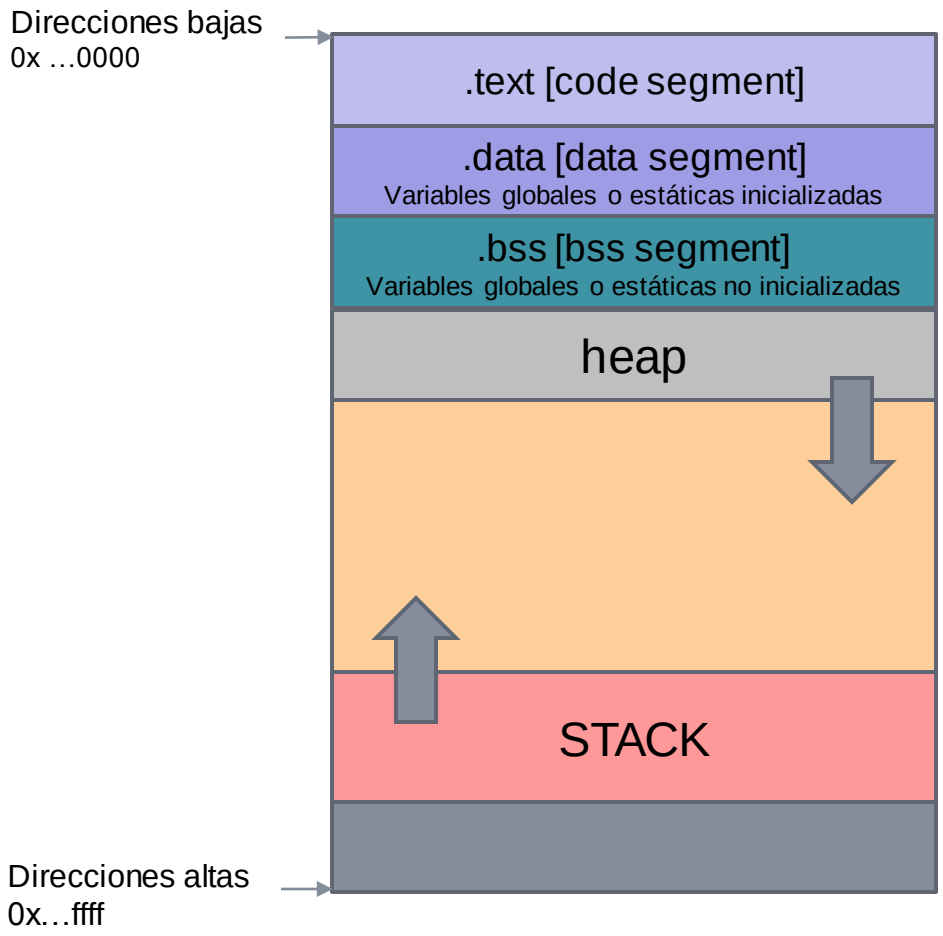
Ejercicio 6 `malloc(),realloc (), pila`

- Implementar un programa que gestione una pila de números, implementado las siguientes funciones:
  - calcularTamanio: devuelve el tamaño de la pila
  - apilar, (push): agregar un elemento
  - desapilar, (pop): quitar el último elemento; es decir, el elemento superior
  - leerUltimo: leer el elemento superior de la pila
  - vacia: indica si la pila está vacía
  - imprimirPila: recorrer la pila e imprimir sus valores
- Implementar una estructura "nodo". Se enlazarán varios nodos para almacenar los datos de la pila, y se recorrerán esos nodos para mostrarla.

```
// Un nodo tiene un dato, el cual es el número almacenado. Y otro nodo al que
apunta
struct nodo {
    int numero;
    struct nodo *siguiente;
};
```

Asignatura/Tema

2 Secciones de memoria de un programa



Todas las instrucciones en código máquina que componen el programa
Tamaño fijo y escritura

Ejemplo: int i = 10;
Tamaño fijo y escritura

Ejemplo: int i;
Tamaño fijo y escritura

Memoria dinámica
Crece hacia arriba (en el mismo sentido que las direcciones)

Variables locales, variables de referencia, parámetros y valores de entorno.
Control de invocación y retorno de las funciones
Crece hacia abajo (en sentido inverso a las direcciones)

Argumentos de línea de comandos y variables de entorno