Memoria Dinámica

Organización del Computador II

David González Márquez → Daniel Kundro

→ Leandro Ezequiel Barrios

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

27 de Agosto de 2019

Temario de la clase

- Estructuras
  - Ejercicio 1
- Memoria Dinámica
  - Ejercicio 2
- Listas
  - Ejercicio 3
- Ejercicios para hoy
  - Ejercicio 1
  - Ejercicio 2
  - Ejercicio 3

Estructuras

Structs

- **¿Para qué?**
- **¿Cómo?**
- **¿Dónde se almacenan?**

Structs

- **¿Para qué?** Definir tipos de datos.
- **¿Cómo?**
- **¿Dónde se almacenan?**

Structs

- **¿Para qué?** Definir tipos de datos.
- **¿Cómo?** Componiendo otros tipos.
- **¿Dónde se almacenan?**

Structs

- **¿Para qué?** Definir tipos de datos.
- **¿Cómo?** Componiendo otros tipos.
- **¿Dónde se almacenan?** En memoria (como cualquier otro tipo de dato)

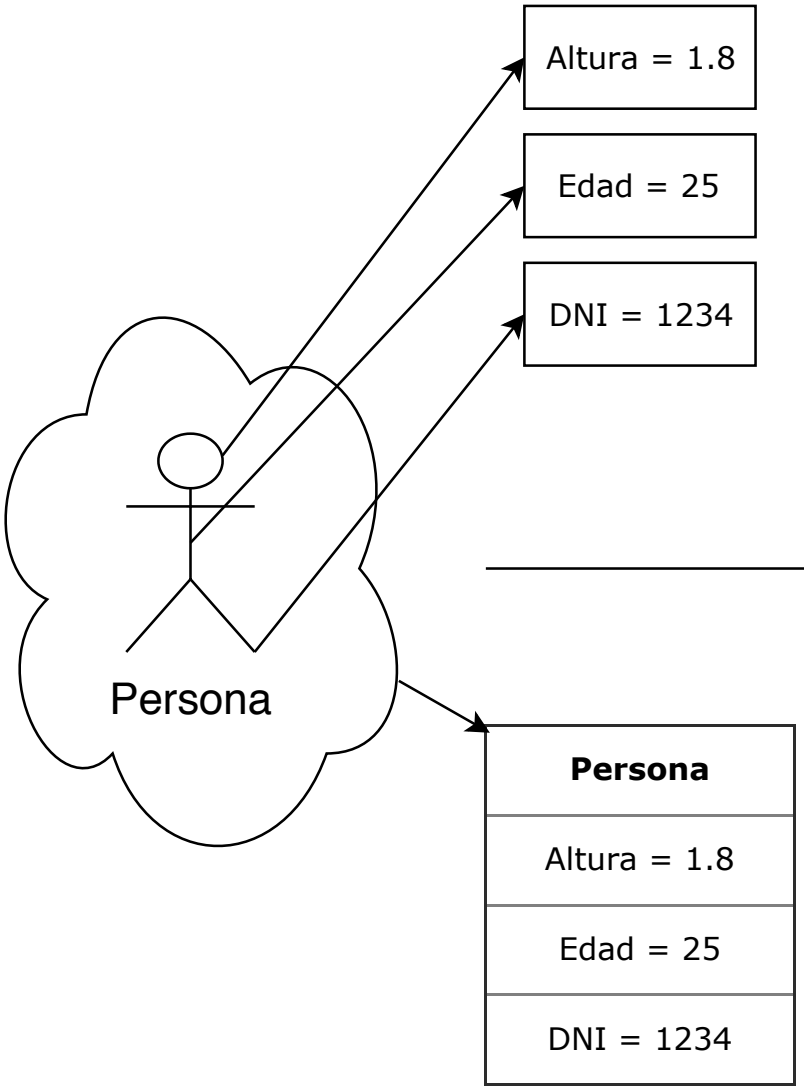
Structs

- **¿Para qué?** Definir tipos de datos.
- **¿Cómo?** Componiendo otros tipos.
- **¿Dónde se almacenan?** En memoria (como cualquier otro tipo de dato)

Structs

- **¿Para qué?** Definir tipos de datos.
- **¿Cómo?** Componiendo otros tipos.
- **¿Dónde se almacenan?** En memoria (como cualquier otro tipo de dato)
- **¿Cómo se almacenan?** ...

¿Por qué usar estructuras?




```
float altura_pedro = 1.7;
int edad_pedro = 25;
int dni_pedro = 1584;
float altura_carla = 1.8;
int edad_carla = 27;
int dni_carla = 1263;
float altura_ramiro = 1.5;
int edad_ramiro = 21;
int dni_ramiro = 2121;
float altura_julia = 1.6;
int edad_julia = 22;
int dni_julia = 2000;

printf("La edad de Julia es %d", edad_julia);
```

```
typedef struct persona_t {
        float altura;
        int edad;
        int dni;
} Persona;

Persona pedro = crear_persona(1.7, 25, 1584);
Persona carla = crear_persona(1.8, 27, 1263);
Persona ramiro = crear_persona(1.5, 21, 2121);
Persona julia = crear_persona(1.6, 22, 2000);

printf("La edad de Julia es %d", julia.edad);
```

¿Cómo se almacenan?

¿Cómo se almacenan? Ver siguientes diapos...

Desde el punto de vista de la memoria...

Los structs definen una forma de acceso a un área determinada de memoria.

Particularmente, a cada uno de sus componentes.

¿Qué necesitamos para aceder a un struct?

Estructuras

¿Qué necesitamos para aceder a un struct?

- El **tamaño** de cada uno de los tipos que lo componen

Estructuras

¿Qué necesitamos para aceder a un struct?

- El **tamaño** de cada uno de los tipos que lo componen
- El modo de **empaquetado y alineación** de esos componentes (packed vs unpacked)

Estructuras

¿Qué necesitamos para aceder a un struct?

- El **tamaño** de cada uno de los tipos que lo componen
- El modo de **empaquetado y alineación** de esos componentes (packed vs unpacked)

En base a esto, podremos calcular:

Estructuras

¿Qué necesitamos para aceder a un struct?

- El **tamaño** de cada uno de los tipos que lo componen
- El modo de **empaquetado y alineación** de esos componentes (packed vs unpacked)

En base a esto, podremos calcular:

- **offset** $\rightarrow$ La posición de cada componente *dentro del struct*.

Estructuras

¿Qué necesitamos para aceder a un struct?

- El **tamaño** de cada uno de los tipos que lo componen
- El modo de **empaquetado y alineación** de esos componentes (packed vs unpacked)

En base a esto, podremos calcular:

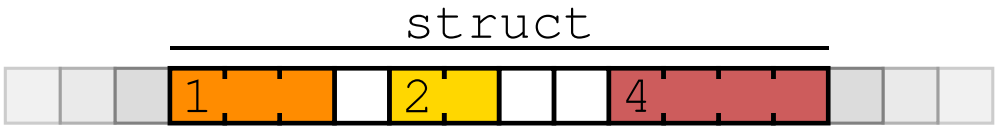
- **offset** $\rightarrow$ La posición de cada componente *dentro del struct*.
- **size** $\rightarrow$ El *tamaño total* del struct (importante para recorrer arreglos)

Estructuras

- Para conocer el tamaño de cada tipo de dato: **ver los archivos de la clase**
- Para conocer como se empaquetan y alinean los componentes: **ver la siguientes diapositiva**

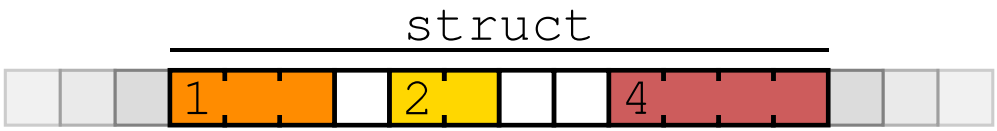
Alineación

- Alineación en los campos del struct:
  Cada campo esta alineado a su tamaño dentro del struct

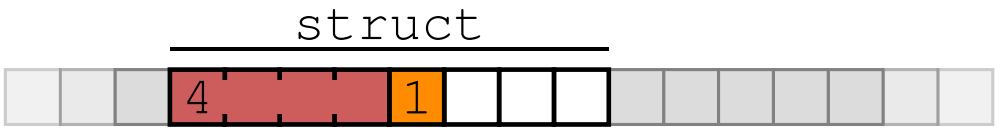


Alineación

- Alineación en los campos del struct:
  Cada campo esta alineado a su tamaño dentro del struct

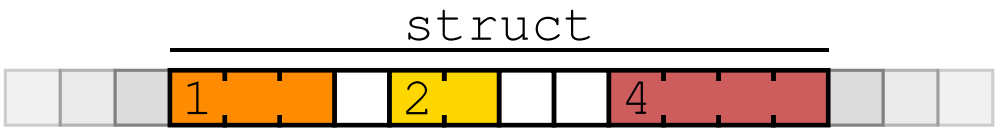


- Alineación del struct:
  Se alinea al tamaño del campo mas grande del struct

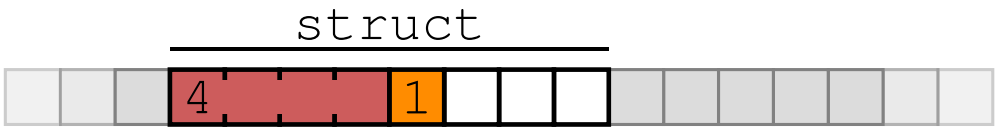


Alineación

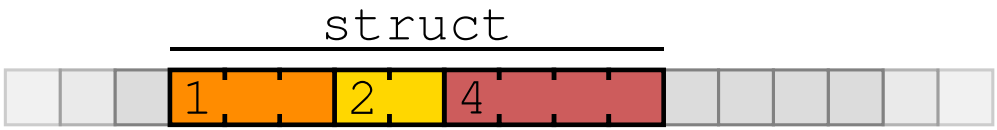
- Alineación en los campos del struct:
  Cada campo esta alineado a su tamaño dentro del struct



- Alineación del struct:
  Se alinea al tamaño del campo mas grande del struct



- `__attribute__((packed))`:
  Indica que el struct va a estar empaquetado



Definir structs en C

struct

```
struct nombre_de_la_estructura {
        tipo_1      nombre_del_campo_1 ;
        ...
        tipo_n      nombre_del_campo_n ;
}
```

Definir structs en C

struct

```
struct nombre_de_la_estructura {
            tipo_1      nombre_del_campo_1 ;
            ...
            tipo_n      nombre_del_campo_n ;
}
```

Ejemplos:

```
struct p2D {
 int x;
 int y;
};
```

```
struct alumno {
 char* nombre;
 char comision;
 int dni;
};
```

Definir structs en C

struct

```
struct nombre_de_la_estructura {
            tipo_1      nombre_del_campo_1 ;
            ...
            tipo_n      nombre_del_campo_n ;
}
```

Ejemplos: → `SIZE`

```
struct p2D {
 int x;          → 4
 int y;          → 4
};
```

```
struct alumno {
 char* nombre;     → 8
 char comision;    → 1
 int dni;          → 4
};
```

Definir structs en C

struct

```
struct nombre_de_la_estructura {
            tipo_1      nombre_del_campo_1 ;
            ...
            tipo_n      nombre_del_campo_n ;
}
```

Ejemplos: → SIZE ⇒ OFFSET

```
struct p2D {                    → 4   ⇒ 0
 int x;                         → 4   ⇒ 4
 int y;                                ⇒ 8
};
```

```
struct alumno {
 char* nombre;                  → 8   ⇒ 0
 char comision;                 → 1   ⇒ 8
 int dni;                       → 4   ⇒ 12
};                                    ⇒ 16
```

Ejemplos

```
struct alumno {
 char* nombre;
 char comision;
 int dni;
};

struct alumno2 {
 char comision;
 char* nombre;
 int dni;
};

struct alumno3 {
 char* nombre;
 int dni;
 char comision;
} __attribute__((packed));
```

Ejemplos: → SIZE

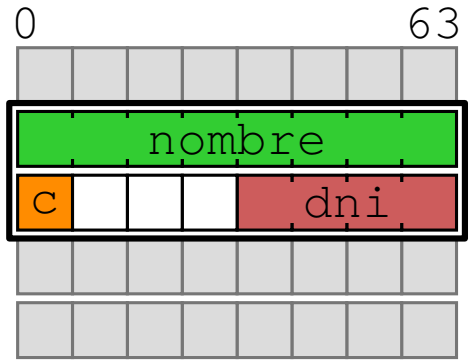
```
struct alumno {
 char* nombre;      → 8
 char comision;     → 1
 int dni;           → 4
};

struct alumno2 {
 char comision;     → 1
 char* nombre;      → 8
 int dni;           → 4
};

struct alumno3 {
 char* nombre;      → 8
 int dni;           → 4
 char comision;     → 1
} __attribute__((packed));
```

Ejemplos: $\rightarrow$ SIZE $\Rightarrow$ OFFSET

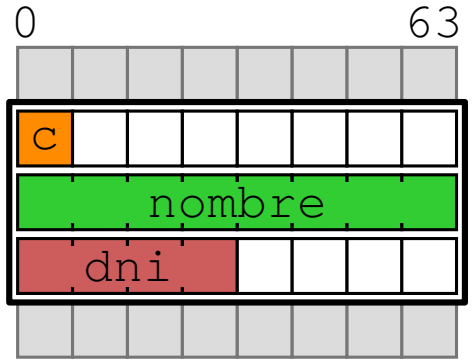
```
struct alumno {
 char* nombre;      → 8      ⇒ 0
 char comision;     → 1      ⇒ 8
 int dni;           → 4      ⇒ 12
};                           ⇒ 16

struct alumno2 {
 char comision;     → 1      ⇒ 0
 char* nombre;      → 8      ⇒ 8
 int dni;           → 4      ⇒ 16
};                           ⇒ 24

struct alumno3 {
 char* nombre;      → 8      ⇒ 0
 int dni;           → 4      ⇒ 8
 char comision;     → 1      ⇒ 12
} __attribute__((packed));   ⇒ 13
```

Ejemplos: $\rightarrow$ SIZE $\Rightarrow$ OFFSET

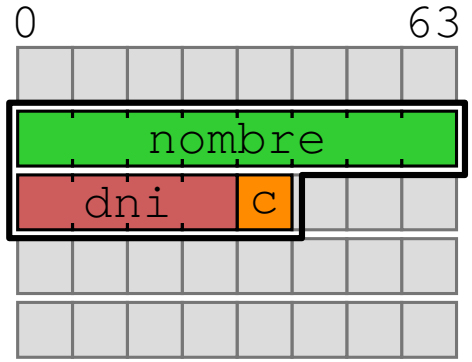
```
struct alumno {
 char* nombre;      → 8     ⇒ 0
 char comision;     → 1     ⇒ 8
 int dni;           → 4     ⇒ 12
};                          ⇒ 16
```




```
struct alumno2 {
 char comision;     → 1     ⇒ 0
 char* nombre;      → 8     ⇒ 8
 int dni;           → 4     ⇒ 16
};                          ⇒ 24
```




```
struct alumno3 {
 char* nombre;      → 8     ⇒ 0
 int dni;           → 4     ⇒ 8
 char comision;     → 1     ⇒ 12
} __attribute__((packed));  ⇒ 13
```



Uso

Definición:

```
struct alumno {
        char* nombre;
        char comision;
        int dni;
};
```

Uso

Definición:

```
struct alumno {
        char* nombre;
        char comision;
        int dni;
};
```

Uso en C:

```
struct alumno alu;
alu.nombre = ‘‘carlos’’;
alu.dni = alu.dni + 10;
alu.comision = ‘a’;
```

Uso en ASM:

```
%define off_nombre 0
%define off_comision 8
%define off_dni 12
mov rsi, ptr_struct
mov rbx, [rsi+off_nombre]
mov al, [rsi+off_comision]
mov edx, [rsi+off_dni]
```

Ejercicio 1

En el archivo de la clase tienen el ejercicio 1 con el siguiente struct:

```
struct alumno {
 short comision;
 char * nombre;
 int edad;
};
```

Implementar la función `mostrar_alumno(struct alumno * alumno)` que toma el `struct alumno` e imprime por pantalla sus valores.

Memoria

Memoria

Variable estática

Está asignada a un espacio de memoria reservado que sólo será utilizado para almacenar la variable en cuestión.

Memoria

Variable estática

Está asignada a un espacio de memoria reservado que sólo será utilizado para almacenar la variable en cuestión.

ej. ASM:

```
section .data:
numero: dd 10
section .rodata:
mensaje: db ‘‘hola pepe’’
section .bss
otro_numero: resd 1
```

ej. C:

```
int numero = 10;
const char* mensaje = ‘‘hola pepe’’;
int otro_numero;
```

Memoria

Variable estática

Está asignada a un espacio de memoria reservado que sólo será utilizado para almacenar la variable en cuestión.

Variable en la pila

Está asignada dentro del espacio de pila del programa, puede existir sólo en el contexto de ejecución de una función.

Memoria

Variable estática

Está asignada a un espacio de memoria reservado que sólo será utilizado para almacenar la variable en cuestión.

Variable en la pila

Está asignada dentro del espacio de pila del programa, puede existir sólo en el contexto de ejecución de una función.

ej. ASM: `sub rsp, 8` (ahora `rsp` apunta a nuestra variable `numero`)

ej. C:

```
void f(){
int* numero;
}
```

Memoria

Variable estática

Está asignada a un espacio de memoria reservado que sólo será utilizado para almacenar la variable en cuestión.

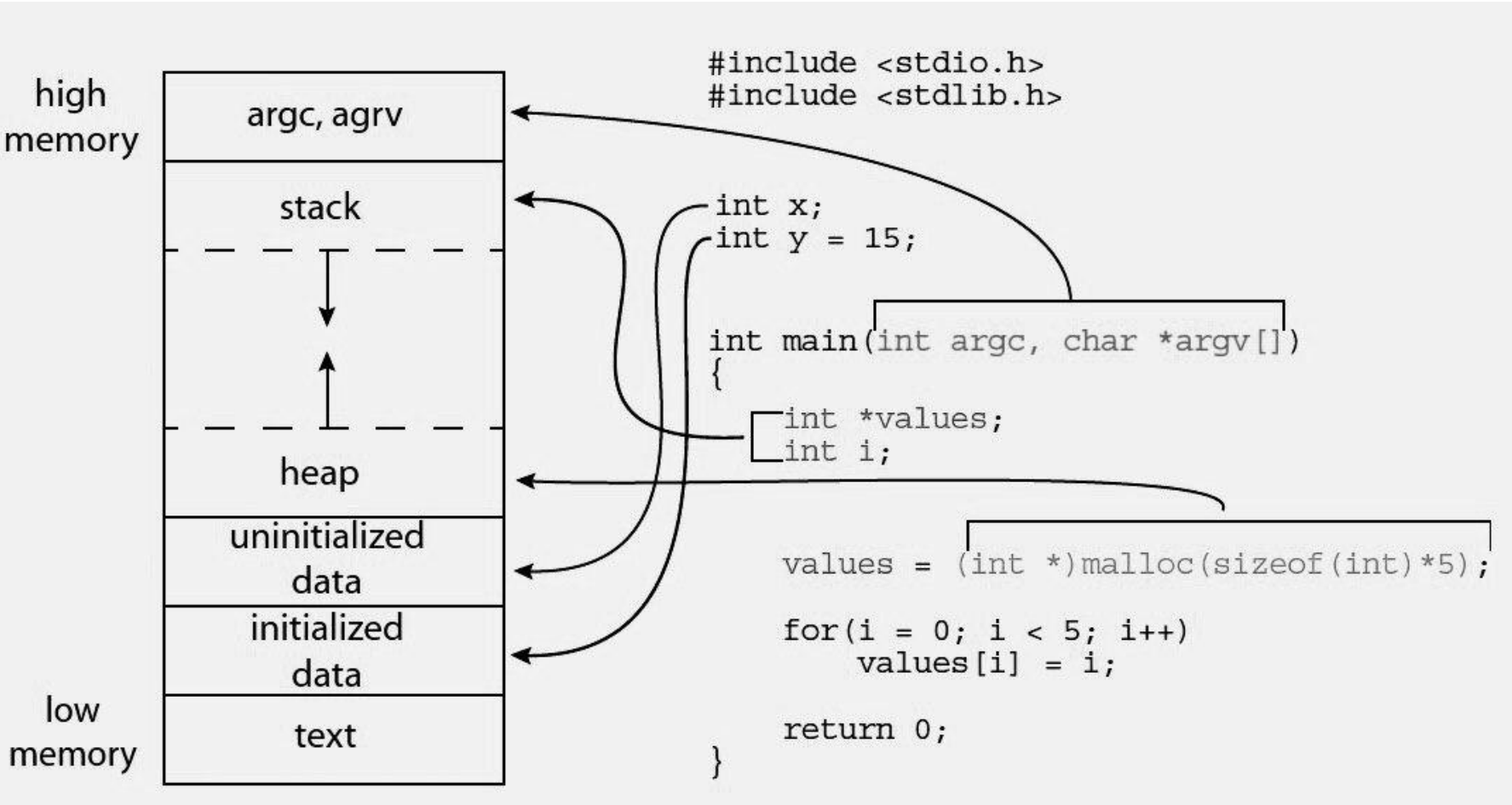
Variable en la pila

Está asignada dentro del espacio de pila del programa, puede existir sólo en el contexto de ejecución de una función.

Variable dinámica

Está asignada a un espacio de memoria solicitado al sistema operativo mediante una biblioteca de funciones que permiten solicitar y liberar memoria. (`malloc`)

Memoria


high memory
argc, agrv
stack
heap
uninitialized data
initialized data
text
low memory
#include <stdio.h>
#include <stdlib.h>
int x;
int y = 15;
int main(int argc, char *argv[])
{
int *values;
int i;
values = (int *)malloc(sizeof(int)*5);
for(i = 0; i < 5; i++)
values[i] = i;
return 0;
}

Memoria Dinámica

Solicitar memoria

```
void *malloc(size_t size)
```

Asigna `size` bytes de memoria y nos devuelve su dirección.

Liberar memoria

```
void free(void *pointer)
```

Libera la memoria en `pointer`, previamente solicitada por `malloc`.

Memoria Dinámica

Solicitar memoria

```
void *malloc(size_t size)
```

Asigna `size` bytes de memoria y nos devuelve su dirección.

Liberar memoria

```
void free(void *pointer)
```

Libera la memoria en `pointer`, previamente solicitada por `malloc`.

"With great power comes great responsibility"


UN GRAN PODER CONLLEVA
UNA GRAN RESPONSABILIDAD

Memoria Dinámica

Solicitar memoria desde ASM

```
mov rdi, 24   ; solicitamos 24 Bytes de memoria
call malloc   ; llamamos a malloc que devuelve en rax
              ; el puntero a la memoria solicitada
```

Liberar memoria desde ASM

```
mov rdi, rax  ; rdi contiene el puntero a la memoria
              ; solicitada a malloc previamente
call free     ; llamamos a free
```

Memoria Dinámica

Solicitar memoria desde ASM

```
mov rdi, 24   ; solicitamos 24 Bytes de memoria
call malloc   ; llamamos a malloc que devuelve en rax
              ; el puntero a la memoria solicitada
```

Liberar memoria desde ASM

```
mov rdi, rax  ; rdi contiene el puntero a la memoria
              ; solicitada a malloc previamente
call free     ; llamamos a free
```

"With great power comes great responsibility"

(Sí, también en ASM)

IMPORTANTE

Si se solicita memoria utilizando `malloc` entonces se DEBE liberar utilizando `free`. Toda memoria que se solicite DEBE ser liberada durante la ejecución del programa.

IMPORTANTE

Si se solicita memoria utilizando `malloc` entonces se DEBE liberar utilizando `free`. Toda memoria que se solicite DEBE ser liberada durante la ejecución del programa.

Caso contrario se PIERDE MEMORIA

IMPORTANTE

Si se solicita memoria utilizando `malloc` entonces se DEBE liberar utilizando `free`. Toda memoria que se solicite DEBE ser liberada durante la ejecución del programa.

Caso contrario se PIERDE MEMORIA

Para detectar problemas en el uso de la memoria se puede utilizar:

`Valgrind`

valgrind –leak-check=full –show-leak-kinds=all -v ./ejecutable

- Ubuntu/Debian: sudo apt-get install valgrind
- Otros Linux/Mac OS: http://valgrind.org/downloads/current.html
- Windows: usen Linux

Ejercicio 2 (memoria)

En el archivo de la clase tienen el ejercicio 2. El mismo está compuesto por una serie de funciones en C y ASM que sirven para controlar un reactor nuclear, y que lamentablemente contienen errores. Se debe buscar la forma de reparar este programa, pero sin alterar su funcionamiento.

La consigna es:

A. Compilar el ejercicio (tip: ver Makefile).

B. Correr el binario y leer el output.

C. Correr el ejercicio con valgrind (tip: ver correr_valgrind.sh).

D. Identificar, y arreglar todos los errores encontrados en los puntos anteriores.

Listas

Listas

Estructuras:

```
struct lista {
 nodo *primero;
};
```

```
struct nodo {
 int dato;
 nodo *prox;
};
```

Listas

Estructuras: $\rightarrow$ SIZE $\Rightarrow$ OFFSET

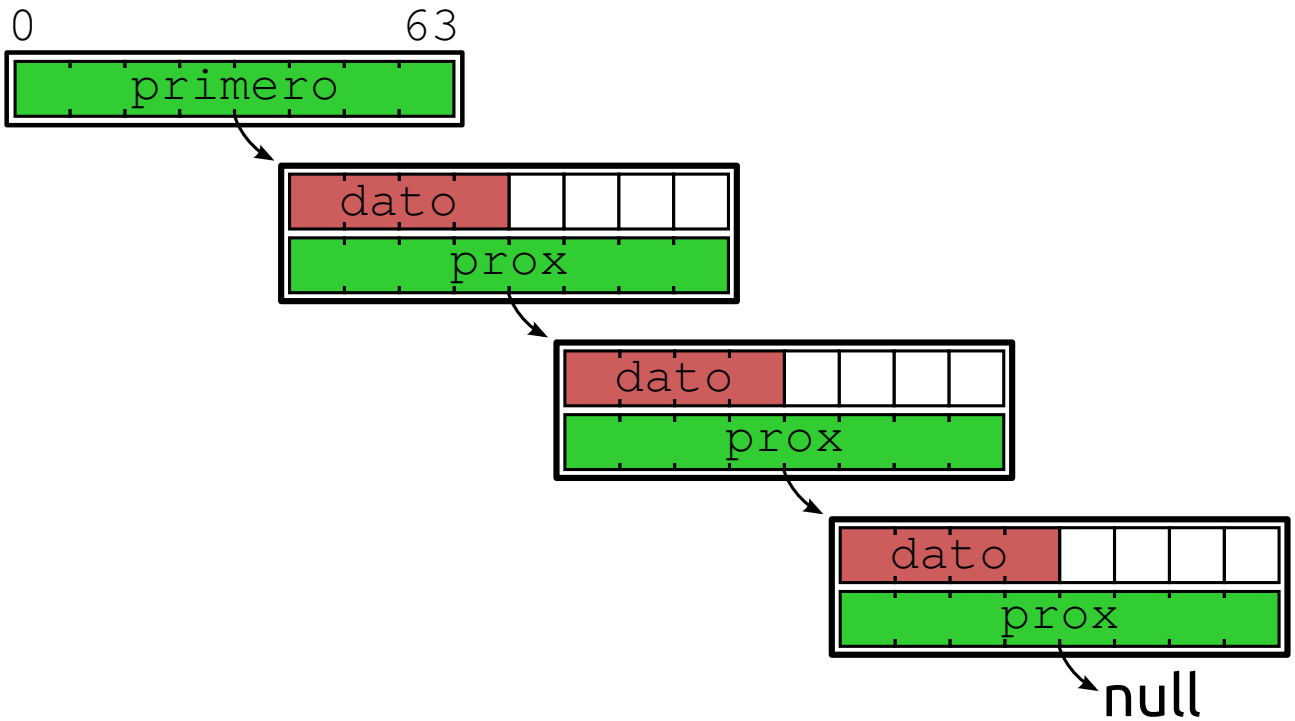
```
struct lista {                  → 8   ⇒ 0
 nodo *primero;
};                                    ⇒ 8
```

```
struct nodo {
int dato;          → 4   ⇒ 0
nodo *prox;        → 8   ⇒ 8
};                       ⇒ 16
```

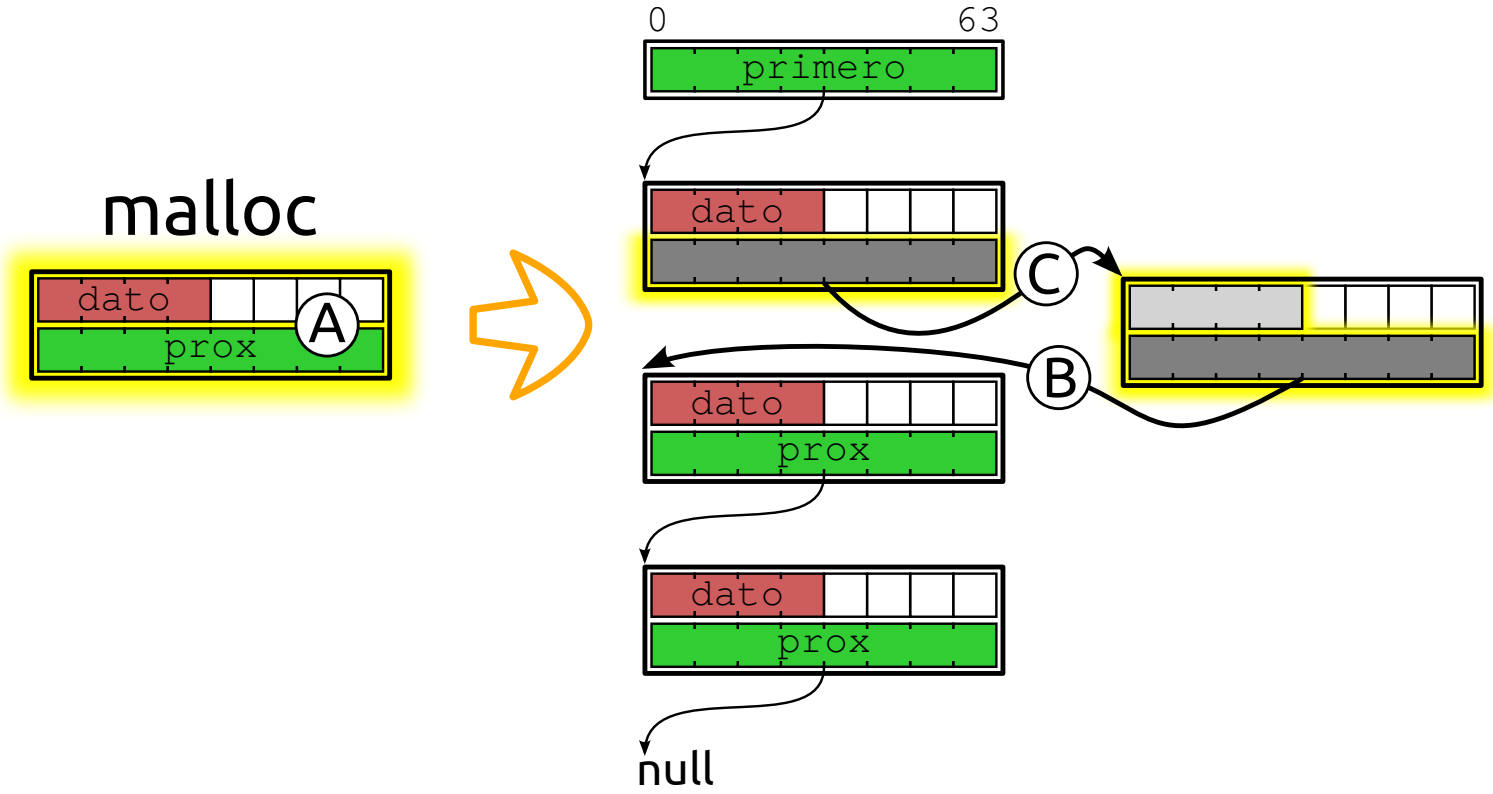
Listas

Estructuras: → SIZE ⇒ OFFSET

```
struct lista {                    struct nodo {
 nodo *primero;   → 8   ⇒ 0 int dato;      → 4   ⇒ 0
};                              ⇒ 8 nodo *prox;   → 8   ⇒ 8
                                  };                    ⇒ 16
```

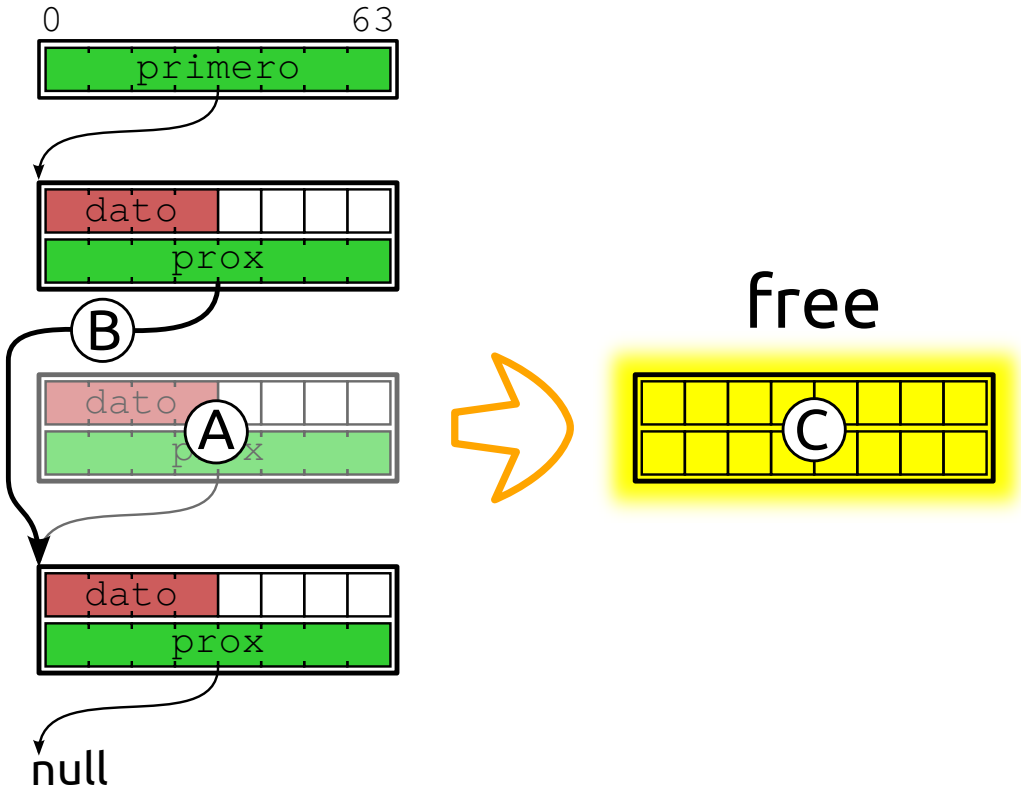


Listas - Agregar



- A Crear el nuevo nodo usando `malloc` y asignar su contenido
- B Conectar el nuevo nodo a su siguiente en la lista
- C Conectar el puntero anterior en la lista al nuevo nodo

Listas - Borrar



- A Leer el valor del puntero al siguiente nodo
- B Conectar el nodo anterior al siguiente del nodo a borrar
- C Borrar el nodo usando `free`

Ejercicios para hoy

Ejercicio 1 (structs)

En el archivo de la clase tienen el ejercicio 1 con el siguiente struct:

```
struct alumno {
 short comision;
 char * nombre;
 int edad;
};
```

Implementar la función `mostrar_alumno(struct alumno * alumno)` que toma el `struct alumno` e imprime por pantalla sus valores.

Ejercicio 2 (memoria)

En el archivo de la clase tienen el ejercicio 2. El mismo está compuesto por una serie de funciones en C y ASM que sirven para controlar un reactor nuclear, y que lamentablemente contienen errores. Se debe buscar la forma de reparar este programa, pero sin alterar su funcionamiento.

La consigna es:

A. Compilar el ejercicio (tip: ver Makefile).

B. Correr el binario y leer el output.

C. Correr el ejercicio con valgrind (tip: ver correr_valgrind.sh).

D. Identificar, y arreglar todos los errores encontrados en los puntos anteriores.

Ejercicio 3 (listas)

Estructuras: → SIZE ⇒ OFFSET

```
struct lista {                  → 8   ⇒ 0
 nodo *primero;                       ⇒ 8
};
```

```
struct nodo {
 int dato;          → 4   ⇒ 0
 nodo *prox;        → 8   ⇒ 8
};                         ⇒ 16
```

1. Escribir en ASM las siguientes funciones:
   - `void agregarPrimero(lista* unaLista, int unInt);`
     Toma una lista y agrega un nuevo nodo en la primera posición. Su dato debe ser el valor de `unInt` pasado por parámetro.
   - `void borrarUltimo(lista *unaLista);`
     Toma una lista cualquiera y de existir, borra el ultimo nodo de la lista.

Ejercicio 3 bis (listas)

Estructuras: → SIZE ⇒ OFFSET

```
struct lista {                    → 8   ⇒ 0
 nodo *primero;                         ⇒ 8
};
```

```
struct nodo {
 int dato;          → 4   ⇒ 0
 nodo *prox;        → 8   ⇒ 8
};                         ⇒ 16
```

1. Escribir en ASM las siguientes funciones:
   - `void borrarPrimero(lista *unaLista);`
     Toma una lista cualquiera y de existir, borra el primer nodo de la lista.
   - `void agregarUltimo(lista* unaLista, int unInt);`
     Toma una lista y agrega un nuevo nodo en la ultima posición. Su dato debe ser el valor de `unInt` pasado por parámetro.

¡GRACIAS!