Memoria Dinámica

David Alejandro González Márquez

Clase disponible en: https://github.com/fokerman/computingSystemsCourse

Memoria

Memoria

Variable estática

Asignada en un espacio de memoria **reservado** que solo será utilizado para almacenar la variable en cuestión.

Ejemplo:

```
#include <stdio.h>
...
int numero = 10;
const char letra = '10'
...
```

Memoria

Variable estática

Asignada en un espacio de memoria **reservado** que solo será utilizado para almacenar la variable en cuestión.

Variable en la pila

Esta asignada dentro del espacio de **pila** del programa, puede existir solo en el contexto de ejecución de una función.

Ejemplo:

```
#include <stdio.h>
...
int numero = 10;
const char letra = '10'
...
int main() {
     ...
     int n = 10;
     int* puntero;
     ...
```

Memoria

Variable estática

Asignada en un espacio de memoria **reservado** que solo será utilizado para almacenar la variable en cuestión.

Variable en la pila

Esta asignada dentro del espacio de **pila** del programa, puede existir solo en el contexto de ejecución de una función.

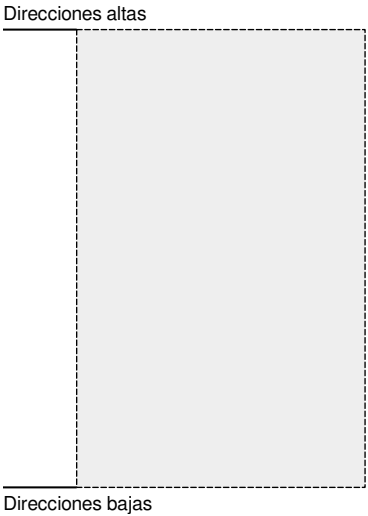
Variable dinámica

Esta asignada en un espacio de memoria **solicitado al sistema** operativo mediante una biblioteca de funciones, estas permiten solicitar y liberar memoria. (`malloc`)

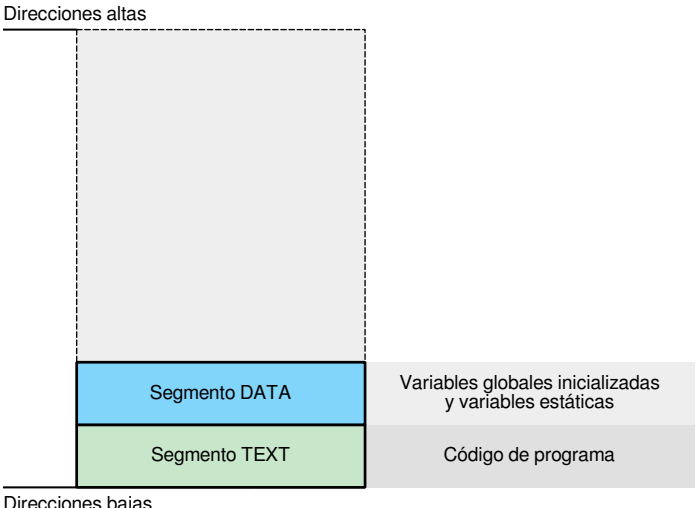
Ejemplo:

```
#include <stdio.h>
...
int numero = 10;
const char letra = '10'
...
int main() {
     ...
     int n = 10;
     int* puntero;
     ...
     ...
     ...
     int* puntero = malloc(10);
     ...
     free(puntero);
     ...
}
```

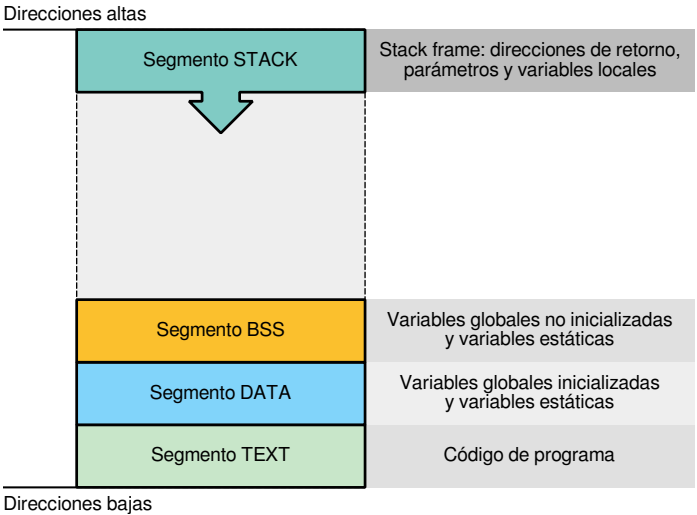
Memoria



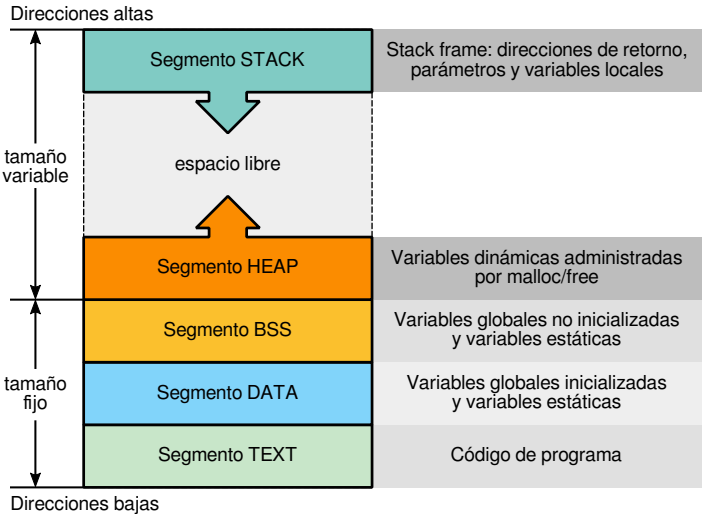
Memoria


Direcciones altas
Segmento DATA
Variables globales inicializadas y variables estáticas
Segmento TEXT
Código de programa
Direcciones bajas

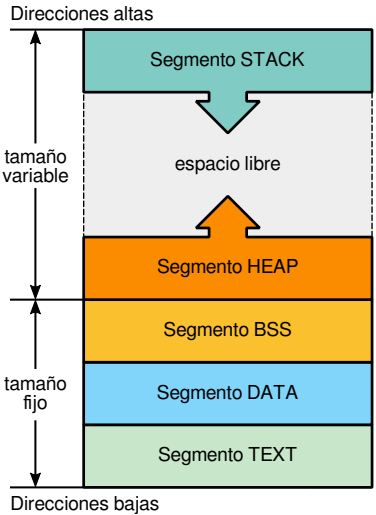
Memoria


Direcciones altas
Segmento STACK
Stack frame: direcciones de retorno, parámetros y variables locales
Segmento BSS
Variables globales no inicializadas y variables estáticas
Segmento DATA
Variables globales inicializadas y variables estáticas
Segmento TEXT
Código de programa
Direcciones bajas

Memoria

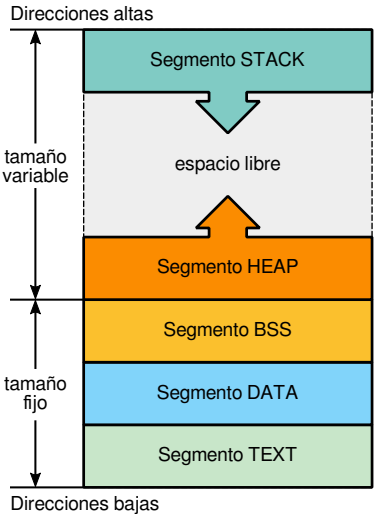

Direcciones altas
Segmento STACK
Stack frame: direcciones de retorno, parámetros y variables locales
tamaño variable
espacio libre
Segmento HEAP
Variables dinámicas administradas por malloc/free
Segmento BSS
Variables globales no inicializadas y variables estáticas
tamaño fijo
Segmento DATA
Variables globales inicializadas y variables estáticas
Segmento TEXT
Código de programa
Direcciones bajas

Memoria



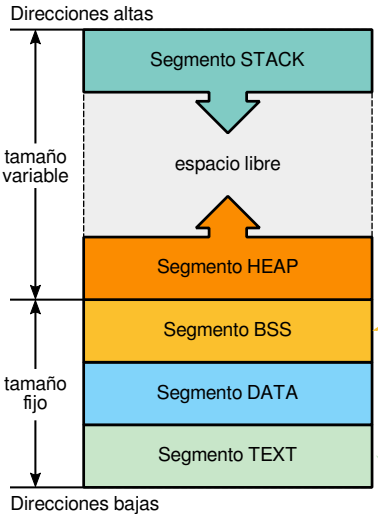

```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 35;

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

    int *data;
    int i;

    data = (int*)malloc(sizeof(int)*y);

    ...

    return 0;
}
```

Memoria



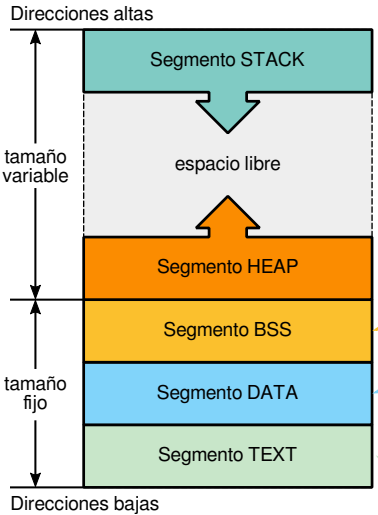

```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 35;

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

    int *data;
    int i;

    data = (int*)malloc(sizeof(int)*y);

    ...

    return 0;
}
```

Memoria



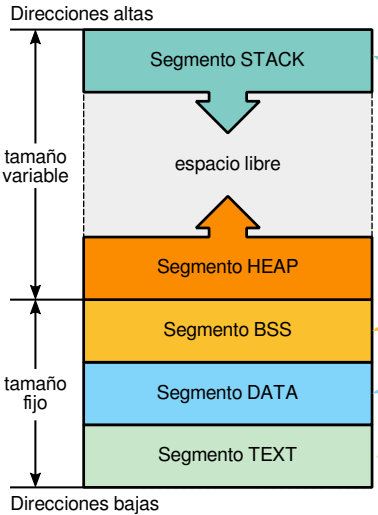

```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 35;

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

    int *data;
    int i;

    data = (int*)malloc(sizeof(int)*y);

    ...

    return 0;
}
```

Memoria



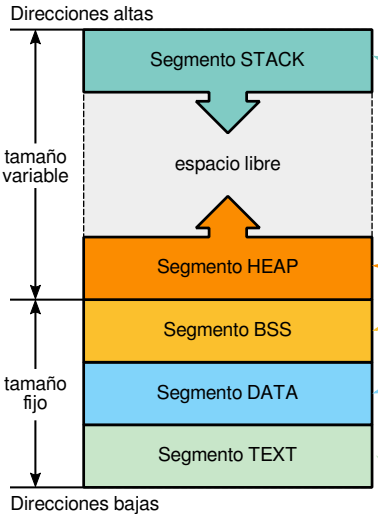

```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 35;

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

    int *data;
    int i;

    data = (int*)malloc(sizeof(int)*y);

    ...

    return 0;
}
```

Memoria




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

int x;
int y = 35;

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

    int *data;
    int i;

    data = (int*)malloc(sizeof(int)*y);

    ...

    return 0;
}
```

Memoria




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

int x;
int y = 35;

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

    int *data;
    int i;

    data = (int*)malloc(sizeof(int)*y);

    ...

    return 0;
}
```

Memoria Dinámica

Solicitar memoria

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

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

Memoria Dinámica

Solicitar memoria

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

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

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 posición.

Liberar memoria

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

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

"With a great power comes a great responsibility"

Memoria Dinámica - IMPORTANTE -

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

Memoria Dinámica - IMPORTANTE -

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

Caso contrario se PIERDE MEMORIA

Memoria Dinámica - IMPORTANTE -

Si se solicita memoria utilizando `malloc`, 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`

Memoria Dinámica - IMPORTANTE -

Si se solicita memoria utilizando `malloc`, 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

Uso:

```
$ valgrind --leak-check=full --show-leak-kinds=all -v ./holamundo
```

Instalación:

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

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
  (ejemplo: una variable cuyo valor es una dirección de memoria)

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
  (ejemplo: una variable cuyo valor es una dirección de memoria)

- Tiene un tipo y un nombre.

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
  (ejemplo: una variable cuyo valor es una dirección de memoria)
- Tiene un tipo y un nombre.
- Almacena una dirección de memoria.

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
  (ejemplo: una variable cuyo valor es una dirección de memoria)
- Tiene un tipo y un nombre.
- Almacena una dirección de memoria.
- Sirve para referenciar una posición de memoria.

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
  (ejemplo: una variable cuyo valor es una dirección de memoria)
- Tiene un tipo y un nombre.
- Almacena una dirección de memoria.
- Sirve para referenciar una posición de memoria.
- Operadores:

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
  (ejemplo: una variable cuyo valor es una dirección de memoria)
- Tiene un tipo y un nombre.
- Almacena una dirección de memoria.
- Sirve para referenciar una posición de memoria.
- Operadores:
  - `&` → Da como resultado la dirección de memoria de una variable.

Repaso de punteros

- Es una variable que referencia una **posición de la memoria**.
  (ejemplo: una variable cuyo valor es una dirección de memoria)
- Tiene un tipo y un nombre.
- Almacena una dirección de memoria.
- Sirve para referenciar una posición de memoria.
- Operadores:
  - `&` → Da como resultado la dirección de memoria de una variable.
  - `*` → Da como resultado el valor apuntado por un puntero.
    (Además de ser el indicador del tipo puntero)

Repaso de punteros

Ejemplos:

- `int *pepe`

Repaso de punteros

Ejemplos:

- `int *pepe`

  Declara un puntero de tipo entero con nombre *pepe*.

Repaso de punteros

Ejemplos:

- `int *pepe`

  Declara un puntero de tipo entero con nombre *pepe*.

- ```
  int x = 5
  pepe = &x
  ```
  ```

Repaso de punteros

Ejemplos:

- `int *pepe`

  Declara un puntero de tipo entero con nombre *pepe*.

- ```
  int x = 5
  pepe = &x
  ```

  Guarda en el puntero *pepe* la dirección de *x*. Se dice que *pepe* apunta a *x*.
  ```

Repaso de punteros

Ejemplos:

- `int *pepe`

  Declara un puntero de tipo entero con nombre *pepe*.

- ```
  int x = 5
  pepe = &x
  ```

  Guarda en el puntero *pepe* la dirección de *x*. Se dice que *pepe* apunta a *x*.

- `*pepe = 8`
  ```

Repaso de punteros

Ejemplos:

- `int *pepe`

  Declara un puntero de tipo entero con nombre *pepe*.

- ```
  int x = 5
  pepe = &x
  ```

  Guarda en el puntero *pepe* la dirección de *x*. Se dice que *pepe* apunta a *x*.

- `*pepe = 8`

  Guarda 8 en la posición apuntada por el puntero *pepe*.
  ```

Repaso de punteros

Ejemplos:

- `int *pepe`

  Declara un puntero de tipo entero con nombre *pepe*.

- ```
  int x = 5
  pepe = &x
  ```

  Guarda en el puntero *pepe* la dirección de *x*. Se dice que *pepe* apunta a *x*.

- `*pepe = 8`

  Guarda 8 en la posición apuntada por el puntero *pepe*.

- ```
  int y
  y = *pepe
  ```

  Guarda en *y* el valor apuntado por *pepe*.

Repaso de punteros

Ejemplos:

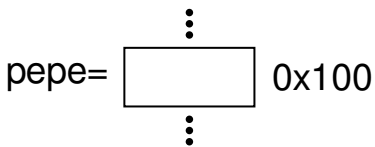
1. `int *pepe`

2. ```
   int x = 5
   pepe = &x
   ```

3. `*pepe = 8`

4. ```
   int y
   y = *pepe
   ```

1.



Repaso de punteros

Ejemplos:

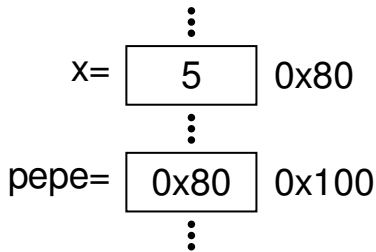
1. `int *pepe`

2. ```
   int x = 5
   pepe = &x
   ```

3. `*pepe = 8`

4. ```
   int y
   y = *pepe
   ```

2.



Repaso de punteros

Ejemplos:

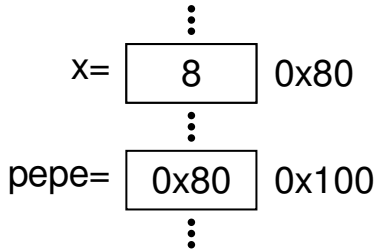
1. `int *pepe`

2. ```
   int x = 5
   pepe = &x
   ```

3. `*pepe = 8`

4. ```
   int y
   y = *pepe
   ```

3.



Repaso de punteros

Ejemplos:

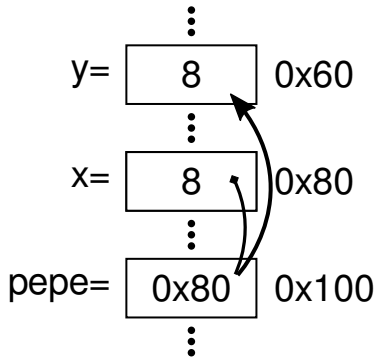
1. `int *pepe`

2. ```
   int x = 5
   pepe = &x
   ```

3. `*pepe = 8`

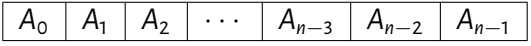
4. ```
   int y
   y = *pepe
   ```

4.



Arreglos

Un arreglo o vector es una secuencia ordenada de elementos consecutivos en memoria de un tamaño fijo.



Arreglos

Declaremos un vector *v* en C:

Arreglos

Declaremos un vector *v* en C:

```
int v[5];
```

¿Cómo está guardado en memoria?

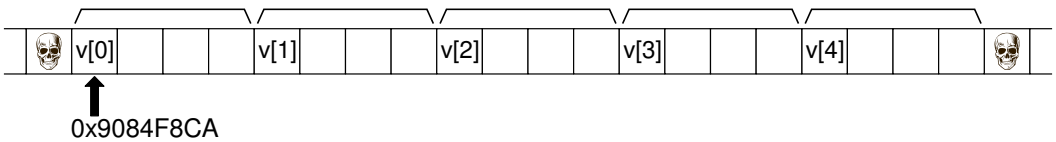
Arreglos

Declaremos un vector *v* en C:

```
int v[5];
```

¿Cómo está guardado en memoria?

Como 5 enteros (*doublewords* / 4 bytes) **consecutivos:**



Arreglos

Si rememoramos el ejemplo del hola mundo en ASM:

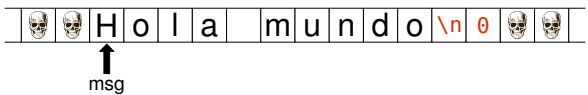
```
section .rodata:
msg: DB 'Hola mundo', 10, 0
largo: EQU $-msg-1
```

Arreglos

Si rememoramos el ejemplo del hola mundo en ASM:

```
section .rodata:
msg: DB 'Hola mundo', 10, 0
largo: EQU $-msg-1
```

msg es una etiqueta que, vista como un puntero,
es un vector de caracteres almacenados de la siguiente manera:

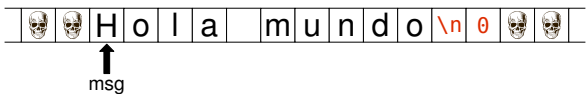


Arreglos

Si rememoramos el ejemplo del hola mundo en ASM:

```
section .rodata:
msg: DB 'Hola mundo', 10, 0
largo: EQU $-msg-1
```

msg es una etiqueta que, vista como un puntero,
es un vector de caracteres almacenados de la siguiente manera:



msg es un char*, es como si en C hiciéramos:

```
char msg[11] = "Hola mundo\n";
```

Arreglos

Dado:

```
int v[5];
```

Arreglos

Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

Arreglos

Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```

Arreglos

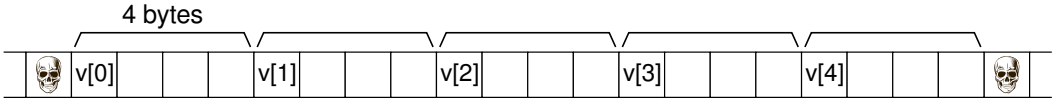
Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```



Arreglos

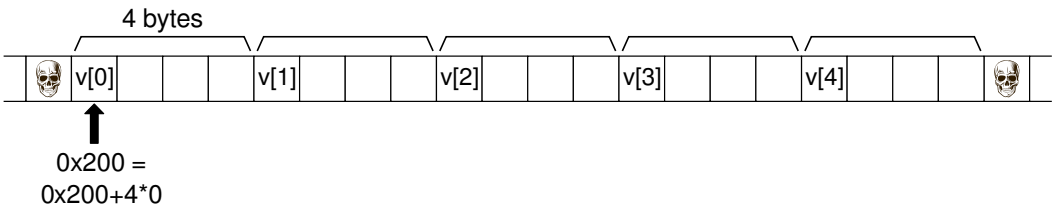
Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```



Arreglos

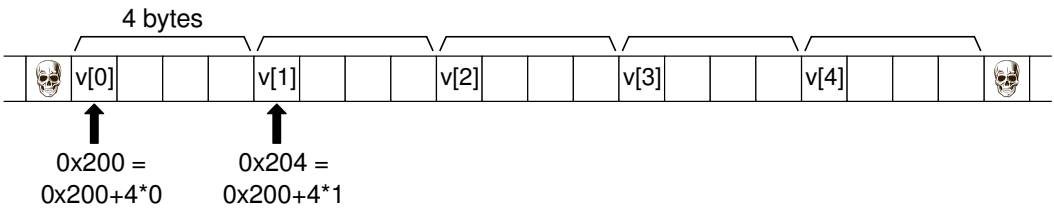
Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```



Arreglos

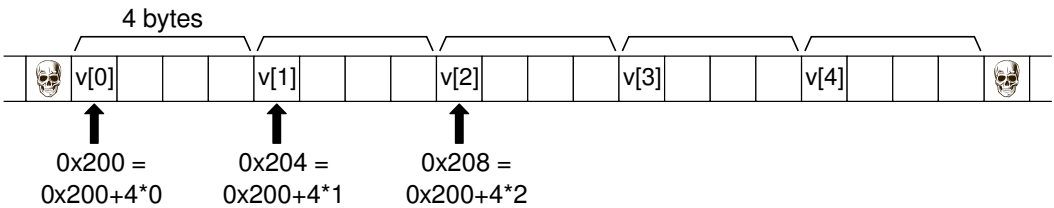
Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```



Arreglos

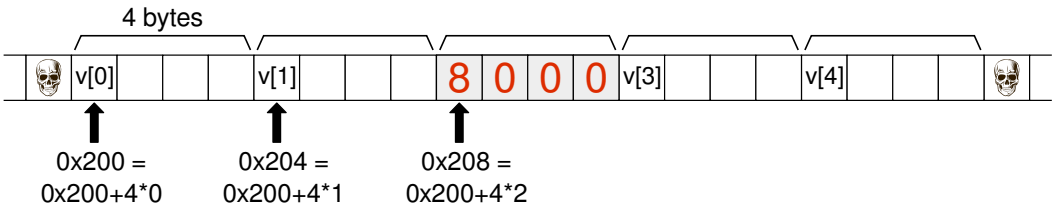
Dado:

```
int v[5];
```

Si suponemos que el primer elemento se encuentra almacenado en la dirección 0x200 de memoria

¿cómo se realiza la siguiente asignación?

```
v[2] = 8;
```



Arreglos y Punteros

Si tenemos:

```
int v[5];
```

v es un puntero al primer elemento del vector.

Arreglos y Punteros

Si tenemos:

```
int v[5];
```

v es un puntero al primer elemento del vector.

Luego, vale en C:

```
int *p_v = v; ⟵ tomo el puntero al vector
```

ó

```
int *p_v = &v[0]; ⟵ tomo la dirección del primer elemento
```

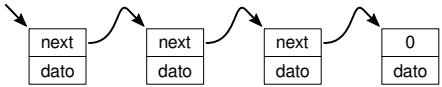
Listas

Es una estructura de datos que consiste en nodos **enlazados** mediante punteros.

Cada nodo contiene datos y un **puntero al siguiente** nodo.

Los datos pueden ser tipos básicos o punteros a **cualquier tipo** de datos.

Una lista termina cuando su último nodo apunta a **null**.



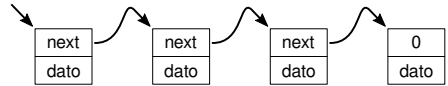
Listas

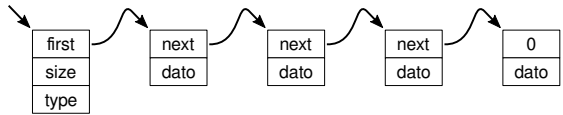
Es una estructura de datos que consiste en nodos **enlazados** mediante punteros.

Cada nodo contiene datos y un **puntero al siguiente** nodo.

Los datos pueden ser tipos básicos o punteros a **cualquier tipo** de datos.

Una lista termina cuando su último nodo apunta a **null**.





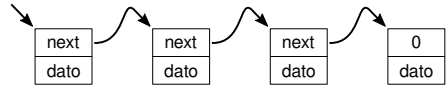
Listas

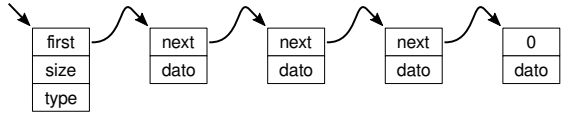
Es una estructura de datos que consiste en nodos **enlazados** mediante punteros.

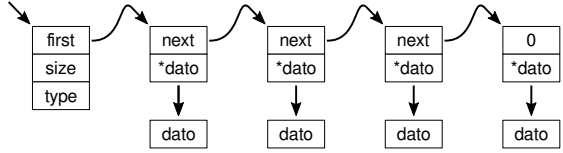
Cada nodo contiene datos y un **puntero al siguiente** nodo.

Los datos pueden ser tipos básicos o punteros a **cualquier tipo** de datos.

Una lista termina cuando su último nodo apunta a **null**.







Listas

Estructuras:

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

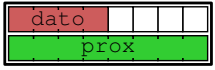
```
struct nodo {
 int dato;
 nodo *prox;
};
```

Listas

Estructuras:

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

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

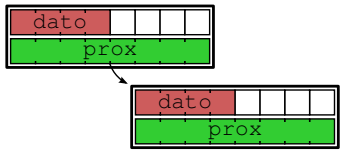


Listas

Estructuras:

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

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

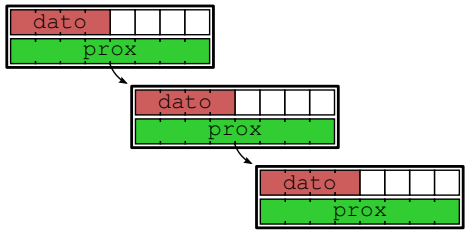


Listas

Estructuras:

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

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

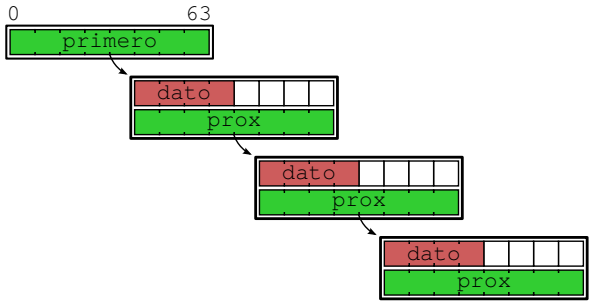


Listas

Estructuras:

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

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

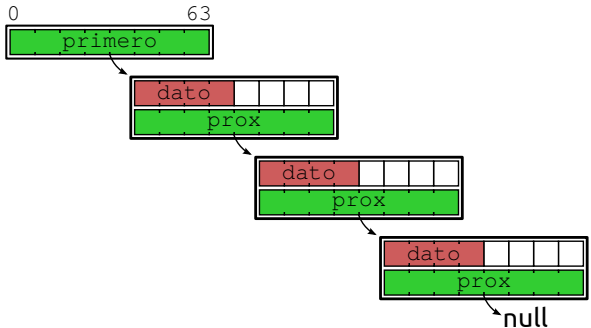


Listas

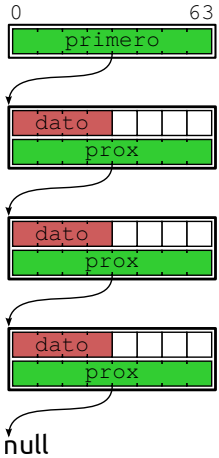
Estructuras:

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

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

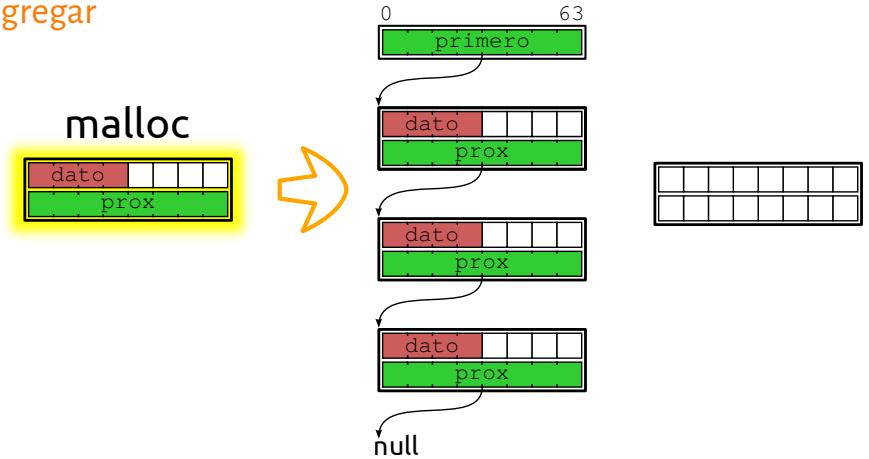


Listas - Agregar



- 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 - Agregar

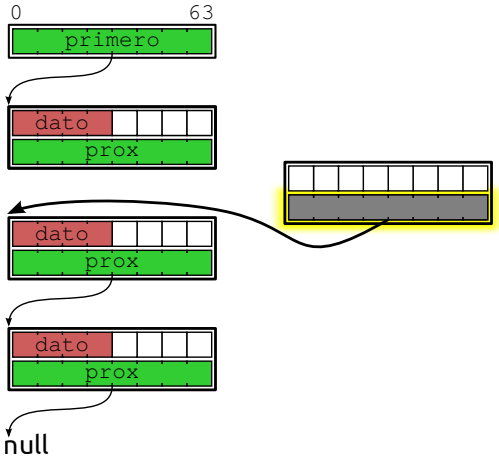


A. Crear el nuevo nodo usando `malloc` y asignar su contenido

B. Conectar el nuevo nodo a su siguiente en la lista

C. Conectar el puntero anterior en la lista al nuevo nodo

Listas - Agregar

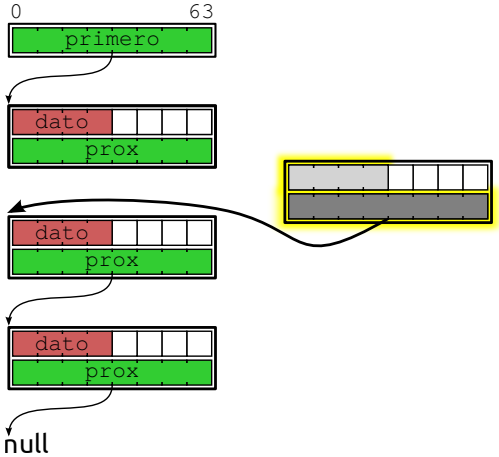


A. Crear el nuevo nodo usando `malloc` y asignar su contenido

B. Conectar el nuevo nodo a su siguiente en la lista

C. Conectar el puntero anterior en la lista al nuevo nodo

Listas - Agregar

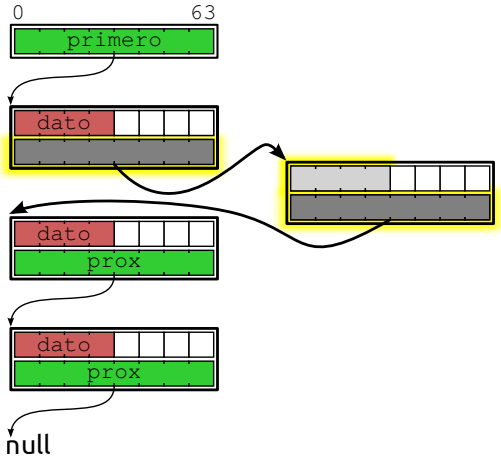


A. Crear el nuevo nodo usando `malloc` y asignar su contenido

B. Conectar el nuevo nodo a su siguiente en la lista

C. Conectar el puntero anterior en la lista al nuevo nodo

Listas - Agregar

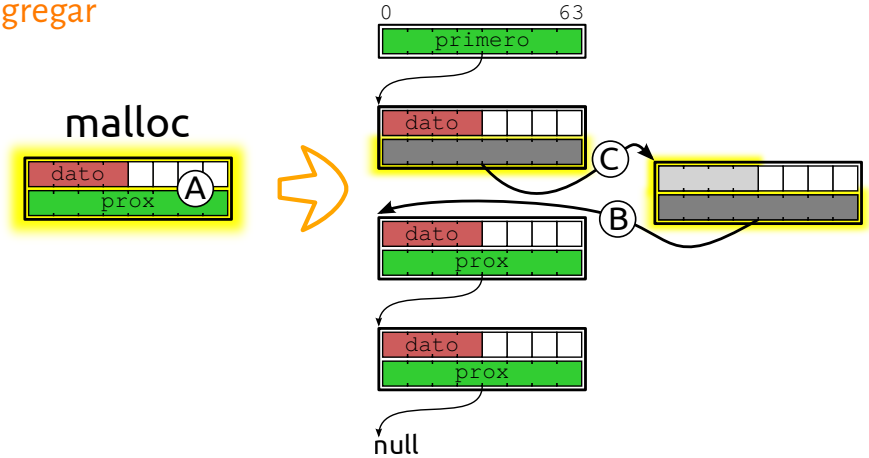


A. Crear el nuevo nodo usando `malloc` y asignar su contenido

B. Conectar el nuevo nodo a su siguiente en la lista

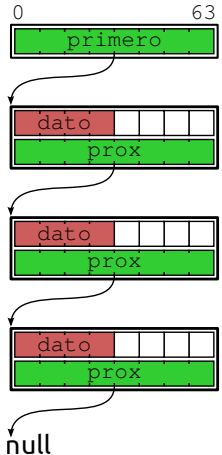
C. Conectar el puntero anterior en la lista al nuevo nodo

Listas - Agregar



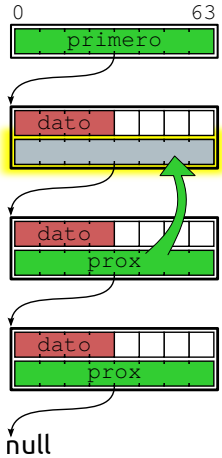
- A Crear el nuevo nodo usando `malloc` y asignar su contenido
- B Conectar el nuevo nodo a su siguiente en la lista
- C Conectar el puntero anterior en la lista al nuevo nodo

Listas - Borrar



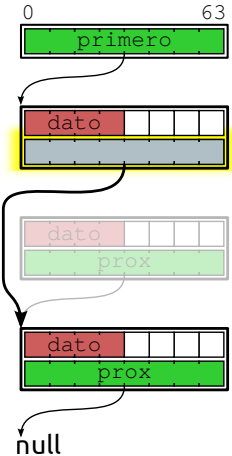
- A Leer el valor del puntero al siguiente nodo
- B Conectar el nodo anterior al siguiente del nodo a borrar
- C Borrar el nodo usando `free`

Listas - Borrar



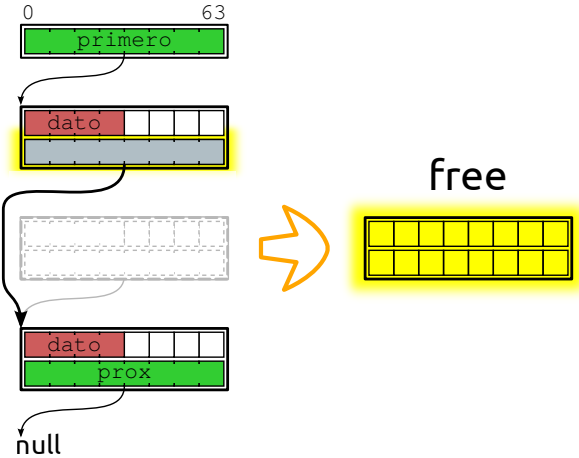
- A Leer el valor del puntero al siguiente nodo
- B Conectar el nodo anterior al siguiente del nodo a borrar
- C Borrar el nodo usando `free`

Listas - Borrar



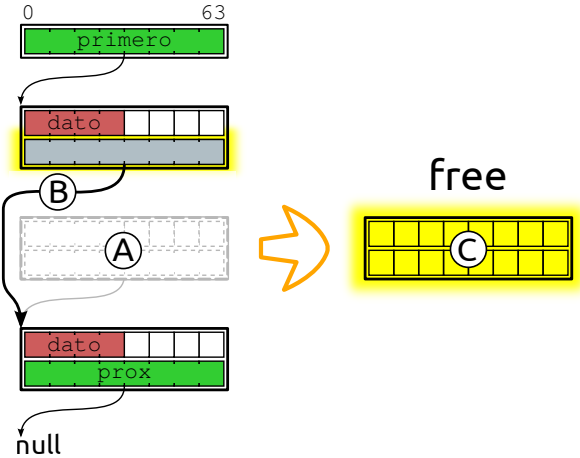
- A Leer el valor del puntero al siguiente nodo
- B Conectar el nodo anterior al siguiente del nodo a borrar
- C Borrar el nodo usando `free`

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`

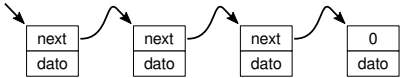
Listas - Borrar



- 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`

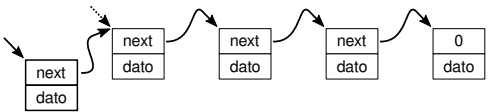
Ejercicio: Agregar primero en una lista

```
struct node {
    int data;
    struct node *next;
};
```




```
int main() {
    struct node *n = NULL;
    n = addFirst(n, 10);
    n = addFirst(n, 20);
    n = addFirst(n, 30);
    return 0;
}
```

```
struct node* addFirst(struct node *p, int data) {
   struct node *newNode = (struct node*) malloc(sizeof(struct node));
    newNode->data = data;
    newNode->next = p;
    return newNode;
}
```



Ejercicio: Borrar último nodo de una lista

```
struct node* removeLast(struct node *p) {
    struct node *prev = 0;
    struct node *current = p;
    // Un solo nodo
    if(current->next == 0) {
        free(current);
        return 0;
    }
    prev = current;
    current = current->next;
    // busco el ultimo
    while(current->next != 0) {
        prev = current;
        current = current->next;
    }
    free(current);
    prev->next = 0;
    return p;
}
```

```
struct node {
    int data;
    struct node *next;
};
```

Bibliografía

- Brian Kernighan y Dennis Ritchie. “El lenguaje de programación C”, 2da Edición, 1991.

- Mike Banahan, Declan Brady and Mark Doran, “The C Book”, 2da Edición, 1991.
  Disponible online: `https://publications.gbdirect.co.uk/c_book/`

¡Gracias!

Recuerden leer los comentarios adjuntos
en cada clase por aclaraciones.