

Estructuras de datos

Clases teóricas

por Pablo E. “Fidel” Martínez López

10. Punteros y arrays

C y la Memoria

Modelo de computación

- ❑ Modelo destructivo
 - ❑ También llamado *imperativo*
 - ❑ Se ejecutan comandos que producen efectos
 - ❑ Procedimientos para abstraer (secuencias de) comandos
 - ❑ Los procedimientos producen efectos sobre el estado
 - ❑ El programa va de un estado inicial a un estado final
 - ❑ ¿Qué compone el estado?
 - ❑ Tablero, en Gobstones
 - ❑ Memoria, en general

Modelo de computación

- ❏ ¿Qué es una memoria?
 - ❏ En bajo nivel
 - ❏ Una secuencia de celdas
 - ❏ Cada celda tiene cierta cantidad de bytes
 - ❏ Se debe interpretar el contenido en binario
 - ❏ En forma más abstracta
 - ❏ Un grupo de espacios de memoria
 - ❏ Las variables locales se organizan en *frames*

Modelo de computación

- ❑ Memoria estática
 - ❑ Se dice que es **estática** pues su comportamiento no depende de valores de ejecución
 - ❑ Es la memoria conformada por los *frames*
 - ❑ Cada función tiene su *frame* con sus variables locales
 - ❑ Al invocar un función se abre su *frame*
 - ❑ Al terminar el función se elimina su *frame*
 - ❑ Se comporta como una pila, por eso se la llama **Stack**
 - ❑ Y a los *frames*, se los llama **stack frames**

Modelo de computación

- ❑ Lenguaje C/C++
 - ❑ Las funciones tienen efectos permanentes
 - ❑ Los procedimientos son funciones que devuelven *void*
 - ❑ Las variables son locales (viven en el *stack frame* del procedimiento que las declara)
 - ❑ Los parámetros tienen también su espacio de memoria
 - ❑ El pasaje de parámetros es por copia
 - ❑ Los registros son varios espacios de memoria juntos, que se acceden a través de los nombres de campos

Motivación

Motivación

- ❏ ¿Se pueden hacer TADs en C/C++?
- ❏ Primera aproximación: memoria estática
- ❏ Interfaz (en **Persona.h**)

```
struct PersonaSt;  
typedef PersonaSt Persona;  
Persona nacer(string n);  
Persona cumplirAnios(Persona p);  
string nombre(Persona p);  
int edad(Persona p);  
void ShowPersona(Persona p);
```

Las zonas rojas NO son parte de la interfaz. ¿Por qué es necesario que estén en el .h?

Observar que un elemento de tipo **Persona** es un **struct** (en memoria estática)

Motivación

- ❏ ¿Se pueden hacer TADs en C/C++?
- ❏ Primera aproximación: memoria estática
- ❏ Implementación (en `Persona.cpp`)

```
struct PersonaSt {  
    string nombre;  
    int edad;  
};  
  
Persona nacer(string n) {  
    Persona p;  
    p.nombre = n; p.edad = 0;  
    return p;  
}
```

TODA la memoria estática de **p**
se copia en donde indique el
que llama a esta función

Motivación

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Implementación (parte 2, en `Persona.cpp`)

```
Persona cumplirAnios(Persona p) {  
    p.edad++;  
    return p;  
}  
  
string nombre(Persona p) {  
    return p.nombre;  
}  
  
int edad(Persona p) {  
    return p.edad;  
}
```

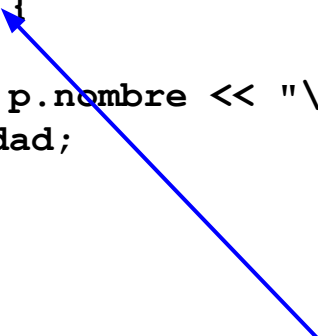
TODA la memoria estática de **p** se copia en donde indique el que llama a esta función

El parámetro **p** se copia TODA la memoria estática del argumento correspondiente

Motivación

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Implementación (parte 3, en `Persona.cpp`)

```
void ShowPersona(Persona p) {  
    cout << "Persona (";  
    cout << "nombre <- \"" << p.nombre << "\", "  
    cout << "edad <- " << p.edad;  
    cout << ")" << endl;  
}
```



El parámetro **p** se copia TODA
la memoria estática del
argumento correspondiente


Motivación

- ❏ ¿Se pueden hacer TADs en C/C++?
- ❏ Primera aproximación: memoria estática
- ❏ Uso de la interfaz

```
#include "Persona.h"


int main() {
    Persona carlos = nacer("Carlitos");
    carlos = cumplirAnios(carlos);
    ShowPersona(carlos);
}
```

Motivación



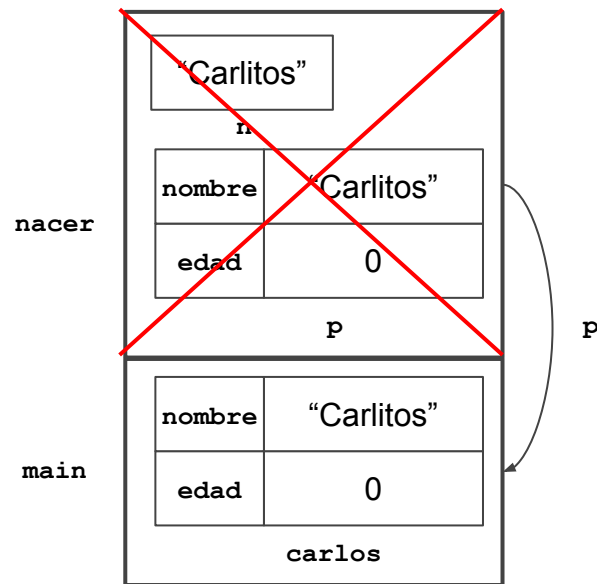
```
Persona nacer(string n) {  
    Persona p;  
    p.nombre = n; p.edad = 0;  
    return p;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz



```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    carlos = cumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se retorna p



Motivación



```
Persona cumplirAnios(Persona p) {  
    p.edad++;  
    return p;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"
```

```
int main() {
```

```
    Persona carlos = nacer("Carlitos");
```

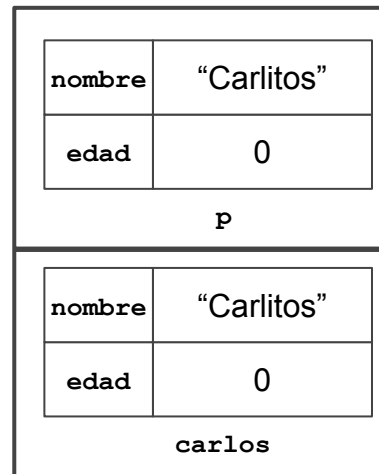
```
    carlos = cumplirAnios(carlos);
```

```
    ShowPersona(carlos);
```

```
}
```

Se invoca a `cumplirAnios`

`cumplirAnios`



Motivación



```
Persona cumplirAnios(Persona p) {  
    p.edad++;  
    return p;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"
```

```
int main() {
```

```
    Persona carlos = nacer("Carlitos");
```

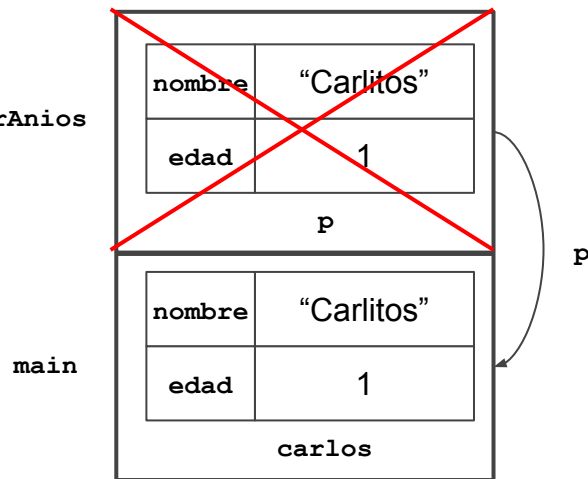
```
    carlos = cumplirAnios(carlos);
```

```
    ShowPersona(carlos);
```

```
}
```

Se retorna p

cumplirAnios



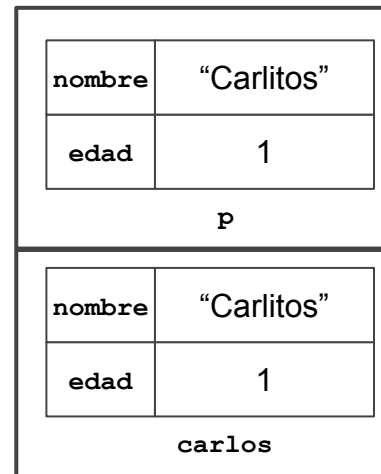
Motivación

```
void ShowPersona(Persona p) {  
    cout << "Persona (";  
    cout << "nombre <- \"" << p.nombre << "\", "  
    cout << "edad <- " << p.edad;  
    cout << ")" << endl;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    carlos = cumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se ejecuta ShowPersona



Motivación



```
Persona cumplirAños(Persona p) {  
    p.edad++;  
    return p;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Primera aproximación: memoria estática
- ❑ Posible problema: clonación involuntaria

```
#include "Persona.h"
```

```
int main() {
```

```
    Persona carlos = nacer("Carlitos");
```

```
    Persona carlosClon = cumplirAños(carlos);
```

```
    ShowPersona(carlos);
```

```
}
```

¡No es carlos el que cumple años!

cumplirAños

nombre	"Carlitos"
edad	1

p

main

nombre	"Carlitos"	nombre	"Carlitos"
edad	0	edad	1
carlos		carlosClon	

p

Motivación

- ❑ ¿Se pueden hacer TADs en C/C++?
 - ❑ Primera aproximación: memoria estática
 - ❑ Ventajas
 - ❑ El usuario puede olvidarse de la representación
 - ❑ Desventajas
 - ❑ Copia de ida y vuelta de parámetros y resultados
 - ❑ Posibilidad de “clonar” elementos (en forma no deseada)
- ❑ Si es imperativo, ¿no es mejor que **cumplirAnios** modifique a **carlos**, sin tener que reasignarlo?

Punteros y Memoria Dinámica

Memoria dinámica

- ❑ La memoria estática no es suficiente para TADs
 - ❑ Requiere copias innecesarias de información
 - ❑ Permite comportamientos no deseados
- ❑ Se hace necesario otro tipo de memoria
 - ❑ ¿Qué debería recibir **cumplirAños**?
 - ❑ ¿Es razonable copiar toda la memoria del **struct**?
 - ❑ Recordemos que un **struct** está en algún lugar de la memoria... ¿Y si se comparte la ubicación?
 - ❑ Se necesita un nuevo tipo de datos: **punteros**

Memoria dinámica

- ❑ Un **puntero** es una dirección de memoria
 - ❑ O sea, un *número* que indica dónde está cierto dato
 - ❑ El puntero se usa para acceder a esa memoria
 - ❑ También se puede copiar
 - ❑ Para declararlo y usarlo se usa un *****
 - ❑ Ej. de declaración: `int* punteroANum;`
 - ❑ Ej. de uso: `(*punteroANum)++;`
- ❑ ¿Cómo se crea un puntero?

Memoria dinámica

- ❑ ¿Es razonable compartir memoria estática?
 - ❑ Puede dar problemas
 - ❑ Conviene tener un lugar *aparte* donde esté la memoria que se va a compartir entre varios procedimientos
 - ❑ Esta nueva memoria se llama **memoria heap**
 - ❑ Su comportamiento es controlado por el programador
 - ❑ Por eso se la conoce como **memoria dinámica**
 - ❑ Se pide lugar en esta memoria con una operación **new**
 - ❑ Esta operación devuelve un puntero al espacio reservado

Memoria dinámica

■ Ejemplo

- Para indicar el tipo puntero se usa un asterisco (*)
- La operación **new** retorna un puntero a la memoria reservada en el espacio compartido

```
typedef int* Contador;  
  
Contador crearContador() {  
    int* punteroANum = new int;  
    *punteroANum = 0;  
    return punteroANum;  
}
```

- ¿Cómo funciona esto? Ver su uso desde quién lo llama...

Memoria dinámica

```
typedef int* Contador;  
Contador crearContador() {  
    int* punteroANum = new int;  
    *punteroANum = 0;  
    return punteroANum;  
}
```

■ Ejemplo

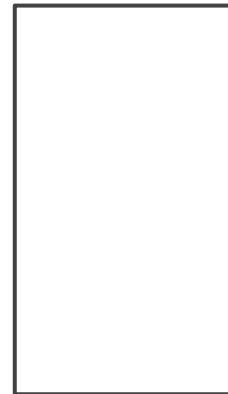
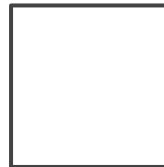
- Para indicar el tipo puntero se usa un asterisco (*)
- La operación **new** retorna un puntero a la memoria reservada en el espacio compartido



```
int main() {  
    Contador n = crearContador();  
    cout << *n;  
}
```


Se comienza la ejecución

main



Memoria
Heap


Memoria dinámica



```
typedef int* Contador;  
Contador crearContador() {  
    int* punteroANum = new int;  
    *punteroANum = 0;  
    return punteroANum;  
}
```

■ Ejemplo

- Para indicar el tipo puntero se usa un asterisco (*)
- La operación **new** retorna un puntero a la memoria reservada en el espacio compartido



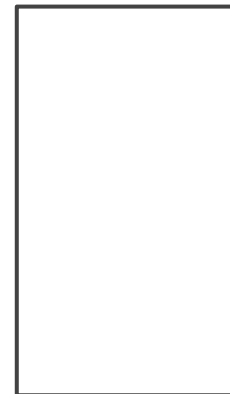
```
int main() {  
    Contador n = crearContador();  
    cout << *n;  
}
```

Se crea memoria para **n** y se llama a la función

crearContador

main

n



Memoria
Heap

Memoria dinámica



```
typedef int* Contador;  
Contador crearContador() {  
    int* punteroANum = new int;  
    *punteroANum = 0;  
    return punteroANum;  
}
```

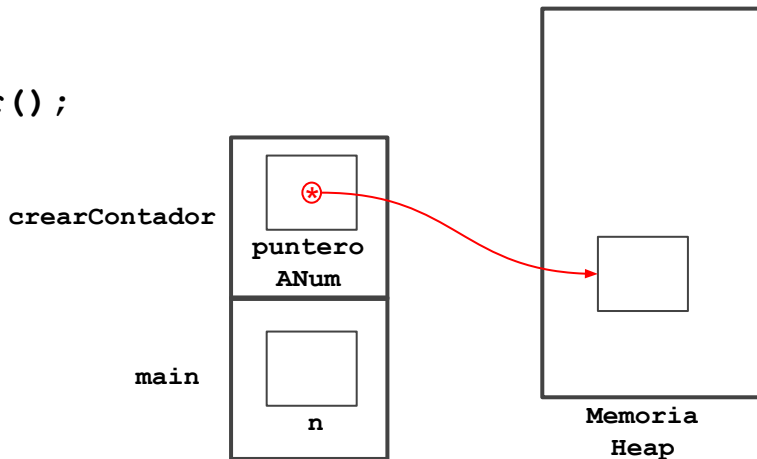
Ejemplo

- Para indicar el tipo puntero se usa un asterisco (*)
- La operación **new** retorna un puntero a la memoria reservada en el espacio compartido



```
int main() {  
    Contador n = crearContador();  
    cout << *n;  
}
```

Se crea memoria para el puntero y se pide memoria en la Heap



Memoria dinámica

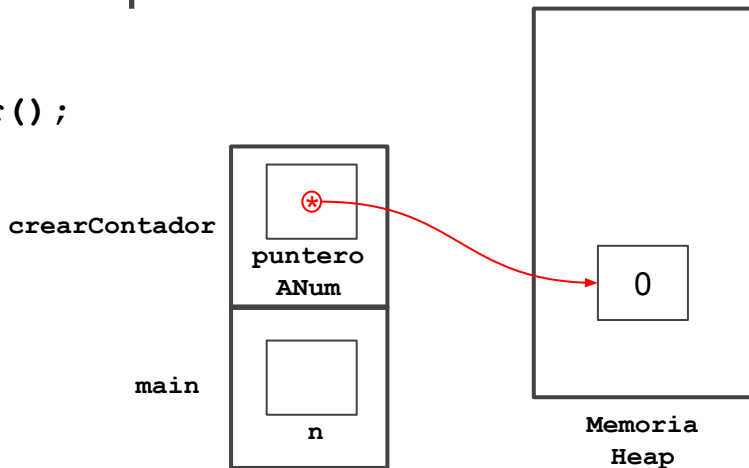
```
typedef int* Contador;  
Contador crearContador() {  
    int* punteroANum = new int;  
    *punteroANum = 0;  
    return punteroANum;  
}
```

Ejemplo

- Para indicar el tipo puntero se usa un asterisco (*)
- La operación **new** retorna un puntero a la memoria reservada en el espacio compartido

```
int main() {  
    Contador n = crearContador();  
    cout << *n;  
}
```

Se accede a lo apuntado por el puntero y se inicializa



Memoria dinámica

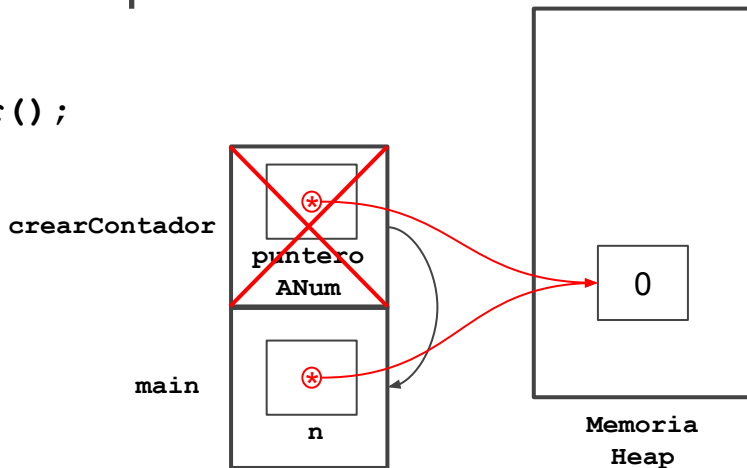
```
typedef int* Contador;  
Contador crearContador() {  
    int* punteroANum = new int;  
    *punteroANum = 0;  
    return punteroANum;  
}
```

Ejemplo

- Para indicar el tipo puntero se usa un asterisco (*)
- La operación **new** retorna un puntero a la memoria reservada en el espacio compartido

```
int main() {  
    Contador n = crearContador();  
    cout << *n;  
}
```

Se retorna SOLAMENTE el puntero




Memoria dinámica

```
typedef int* Contador;  
Contador crearContador() {  
    int* punteroANum = new int;  
    *punteroANum = 0;  
    return punteroANum;  
}
```

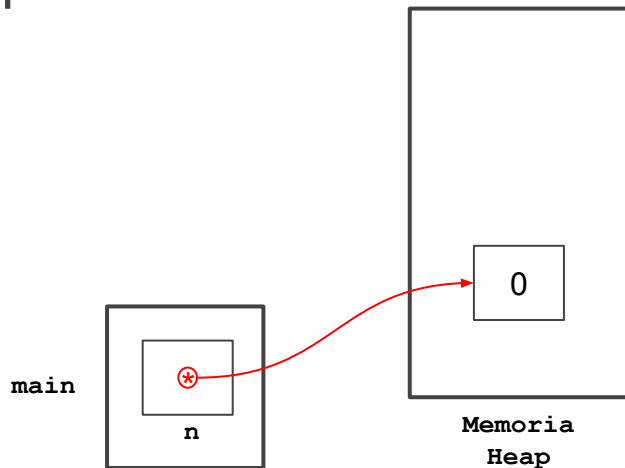
Ejemplo

- Para indicar el tipo puntero se usa un asterisco (*)
- La operación **new** retorna un puntero a la memoria reservada en el espacio compartido



```
int main() {  
    Contador n = crearContador();  
    cout << *n;  
}
```

Se procede con el resto del programa



Memoria dinámica

■ Ejemplo: completemos el **Contador**

- Interfaz (en el archivo **.h**)
 - Crear e incrementar contador, y retornar su valor
 - Observar que el puntero queda solamente del lado de la representación

```
typedef int* Contador;  
Contador crearContador();  
void Incrementar(Contador c);  
int leerContador(Contador c);
```

Memoria dinámica

```
typedef int* Contador;  
Contador crearContador();  
void Incrementar(Contador c);  
int leerContador(Contador c);
```

■ Ejemplo: completemos el **Contador**

■ Implementación (en el archivo **.cpp**)

```
Contador crearContador() {  
    int* c = new int;  
    *c = 0;  
    return c;  
}  
  
void Incrementar(Contador c) {  
    (*c)++;  
}  
  
int leerContador(Contador c) {  
    return *c;  
}
```

Observar que un **Contador**
ES un puntero (a memoria
dinámica), y se debe
recorrer para usarlo

TADs en Memoria Dinámica

TADs y Memoria Dinámica

```
struct PersonaSt;  
typedef PersonaSt Persona;  
Persona nacer(string n);  
Persona cumplirAnios(Persona p);  
string nombre(Persona p);  
int edad(Persona p);  
void ShowPersona(Persona p);
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Interfaz (en **Persona.h**)

```
struct PersonaSt;  
typedef PersonaSt* Persona;  
Persona nacer(string n);  
void CumplirAnios(Persona p);  
string nombre(Persona p);  
int edad(Persona p);  
void ShowPersona(Persona p);
```

Observar que un elemento de tipo **Persona** ahora es un PUNTERO a un **struct** (en memoria dinámica)

Observar que ahora esta operación es un procedimiento (MODIFICA la memoria común)

TADs y Memoria Dinámica

```
typedef PersonaSt Persona;  
Persona nacer(string n) {  
    Persona p;  
    p.nombre = n; p.edad = 0;  
    return p;  
}
```

```
typedef PersonaSt* Persona;
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Implementación (en `Persona.cpp`)

```
struct PersonaSt {  
    string nombre;  
    int edad;  
};  
  
Persona nacer(string n) {  
    PersonaSt* p = new PersonaSt;  
    (*p).nombre = n; (*p).edad = 0;  
    return p;  
}
```

SOLAMENTE se copia el puntero;
la memoria dinámica es accedida
siguiendo el mismo

TADs y Memoria Dinámica

❑ ¿Se pueden hacer TADs en C/C++?

❑ Segunda aproximación: memoria dinámica

❑ Implementación (parte 2, en `Persona.cpp`)

```
Persona cumplirAnios(Persona p) {  
    p.edad++;  
    return p;  
}  
  
string nombre(Persona p) {  
    return p.nombre;  
}  
  
int edad(Persona p) {  
    return p.edad;  
}
```

```
void CumplirAnios(Persona p) {  
    (*p).edad++;  
}  
  
string nombre(Persona p) {  
    return ((*p).nombre);  
}  
  
int edad(Persona p) {  
    return ((*p).edad);  
}
```

¡No hace falta retornar nada,
porque se modifica la memoria
compartida (en la Heap)!

TADs y Memoria Dinámica

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Implementación (parte 3, en `Persona.cpp`)

```
void ShowPersona(Persona p) {  
    cout << "Persona (";  
    cout << "nombre <- \"" << (*p).nombre << "\", ";  
    cout << "edad <- " << (*p).edad;  
    cout << ")" << endl;  
}
```

Memoria dinámica

- ❑ Un **puntero** es una dirección de memoria
 - ❑ O sea, un *número* que indica dónde está cierto dato
 - ❑ El puntero se usa para acceder a esa memoria
 - ❑ El acceso a registros mediante punteros:
 - ❑ Usa el mecanismo **(*p) . campo** muy seguido
 - ❑ Por eso, se puede abreviar con **->**
 - ❑ En lugar de **(*p) . campo**, **p->campo**
 - ❑ Se recorre el puntero Y se accede al campo
 - ❑ Solamente sirve para punteros a registros

TADs y Memoria Dinámica

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Uso de la interfaz

```
#include "Persona.h"
```

```
➡ int main() {  
    Persona carlos = nacer("Carlitos");  
    CumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se comienza el programa

main

Memoria Heap

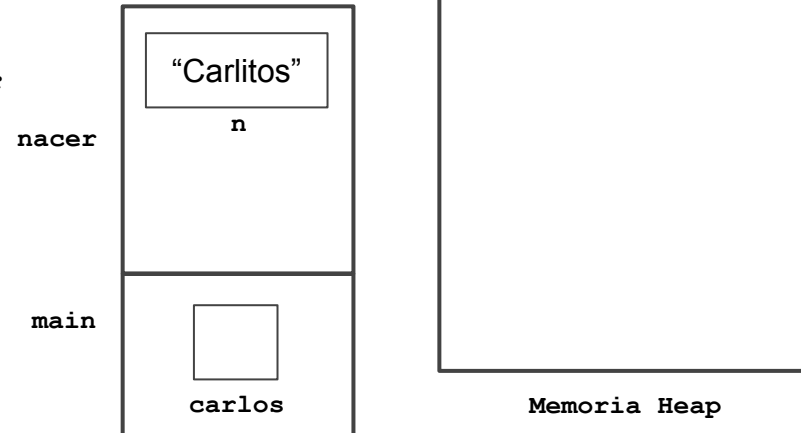
TADs y Memoria Dinámica

```
Persona nacer(string n) {  
    PersonaSt* p = new PersonaSt;  
    p->nombre = n; p->edad = 0;  
    return p;  
}
```

- ¿Se pueden hacer TADs en C/C++?
- Segunda aproximación: memoria dinámica
- Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    CumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se crea memoria para **carlos** y se invoca a **nacer**



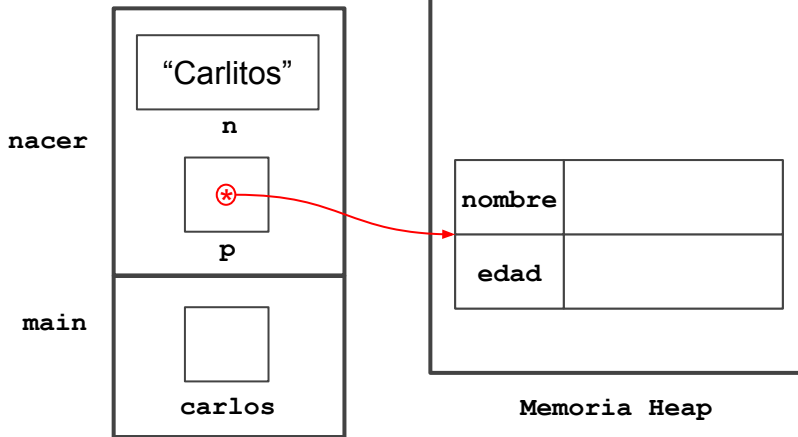
TADs y Memoria Dinámica

```
Persona nacer(string n) {  
    PersonaSt* p = new PersonaSt;  
    p->nombre = n; p->edad = 0;  
    return p;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    CumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se crea `p` y se pide memoria para la persona en la Heap



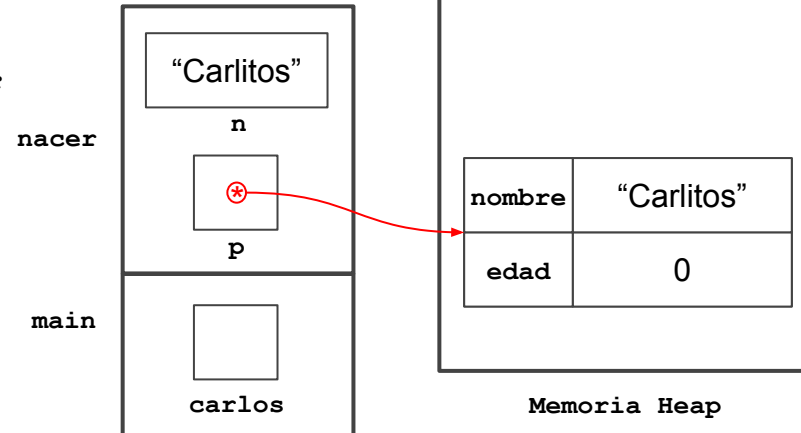
TADs y Memoria Dinámica

```
Persona nacer(string n) {  
    PersonaSt* p = new PersonaSt;  
    p->nombre = n; p->edad = 0;  
    return p;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    CumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se accede a `p` y se asignan sus campos



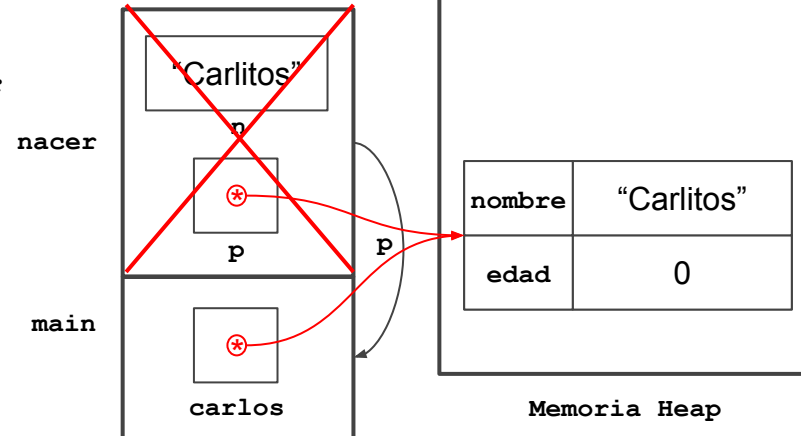
TADs y Memoria Dinámica

```
Persona nacer(string n) {  
    PersonaSt* p = new PersonaSt;  
    p->nombre = n; p->edad = 0;  
    return p;  
}
```


- ¿Se pueden hacer TADs en C/C++?
- Segunda aproximación: memoria dinámica
- Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    CumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se retorna p



TADs y Memoria Dinámica



```
void cumplirAnios(Persona p) {  
    p->edad++;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Uso de la interfaz

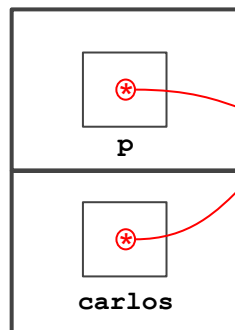
```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    CumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```



Se invoca a **CumplirAnios**

CumplirAnios

main



nombre	"Carlitos"
edad	0

Memoria Heap

TADs y Memoria Dinámica

```
void cumplirAnios(Persona p) {  
    p->edad++;  
}
```

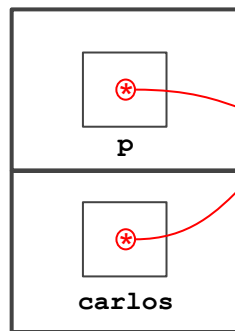
- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    CumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se accede a **p** y se modifican su campos

CumplirAnios

main



nombre	"Carlitos"
edad	1

Memoria Heap

TADs y Memoria Dinámica

```
void cumplirAnios(Persona p) {  
    p->edad++;  
}
```

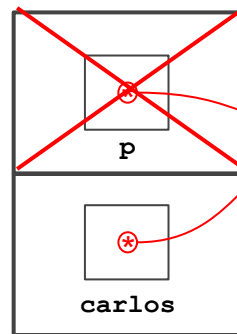
- ¿Se pueden hacer TADs en C/C++?
- Segunda aproximación: memoria dinámica
- Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    CumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se completa CumplirAnios

CumplirAnios

main



nombre	"Carlitos"
edad	1

Memoria Heap

TADs y Memoria Dinámica

```
void ShowPersona(Persona p) {  
    cout << "Persona (";  
    cout << "nombre <- \"" << p->nombre << "\", "  
    cout << "edad <- " << p->edad;  
    cout << ")" << endl;  
}
```

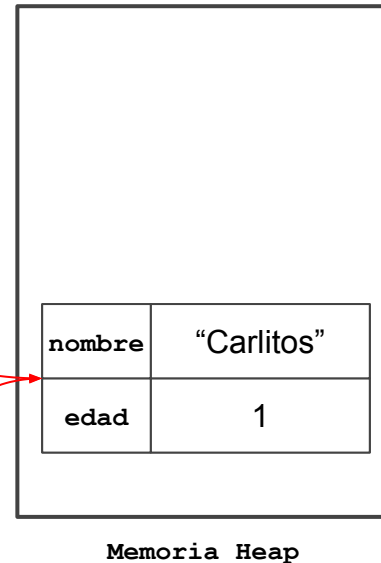
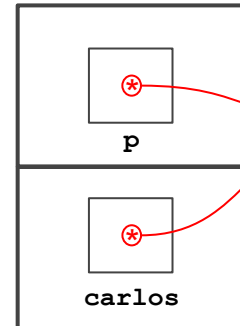
- ¿Se pueden hacer TADs en C/C++?
- Segunda aproximación: memoria dinámica
- Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    CumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se invoca a ShowPersona

ShowPersona

main



TADs y Memoria Dinámica

```
void ShowPersona(Persona p) {  
    cout << "Persona (";  
    cout << "nombre <- \"" << p->nombre << "\", "  
    cout << "edad <- " << p->edad;  
    cout << ")" << endl;  
}
```

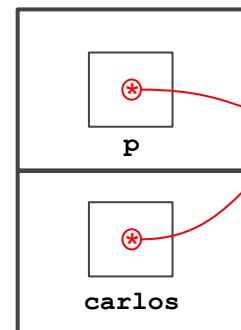
- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    CumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

¡Observar que los campos se acceden desde la memoria común!

ShowPersona

main



nombre	"Carlitos"
edad	1

Memoria Heap

TADs y Memoria Dinámica

```
void ShowPersona(Persona p) {  
    cout << "Persona (";  
    cout << "nombre <- \"" << p->nombre << "\", "  
    cout << "edad <- " << p->edad;  
    cout << ")" << endl;  
}
```

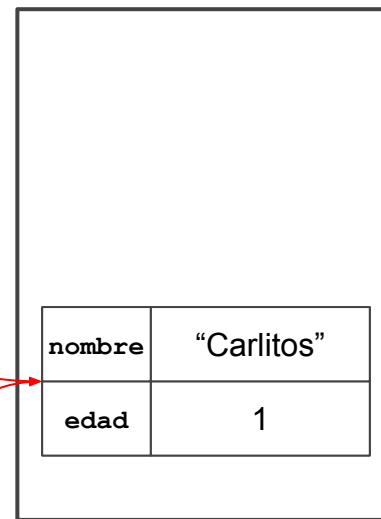
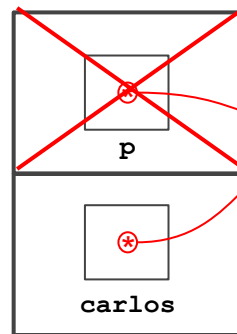
- ¿Se pueden hacer TADs en C/C++?
- Segunda aproximación: memoria dinámica
- Uso de la interfaz

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    CumplirAnios(carlos);  
    ShowPersona(carlos);  
}
```

Se completa ShowPersona

ShowPersona

main



Memoria Heap

TADs y Memoria Dinámica

- ❑ ¿Se pueden hacer TADs en C/C++?
 - ❑ Segunda aproximación: memoria dinámica
 - ❑ Ventajas
 - ❑ NO se copian los datos de ida y vuelta (solo punteros)
 - ❑ NO hay posibilidad de “clonar” elementos (en forma no deseada)
 - ❑ Desventajas
 - ❑ Pueden producirse pérdidas de memoria por accidente (*memory leaks*)

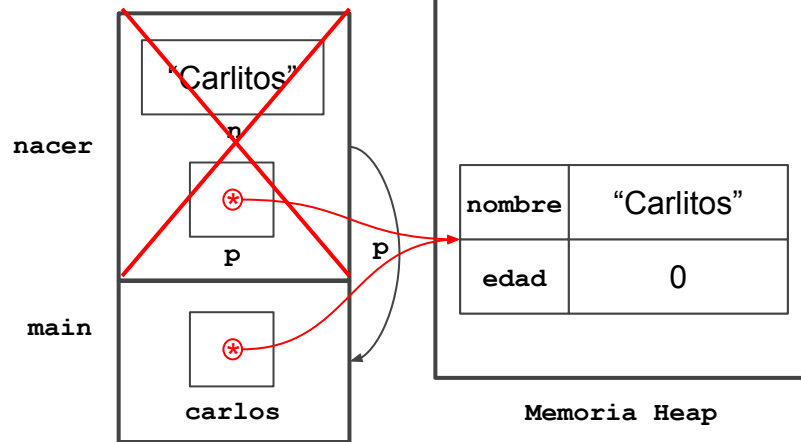
TADs y Memoria Dinámica

```
Persona nacer(string n) {  
    PersonaSt* p = new PersonaSt;  
    p->nombre = n; p->edad = 0;  
    return p;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Posible problema: **memory leak**

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    carlos = nacer("Juan");  
    ...  
}
```

Se crea la primera persona que nace



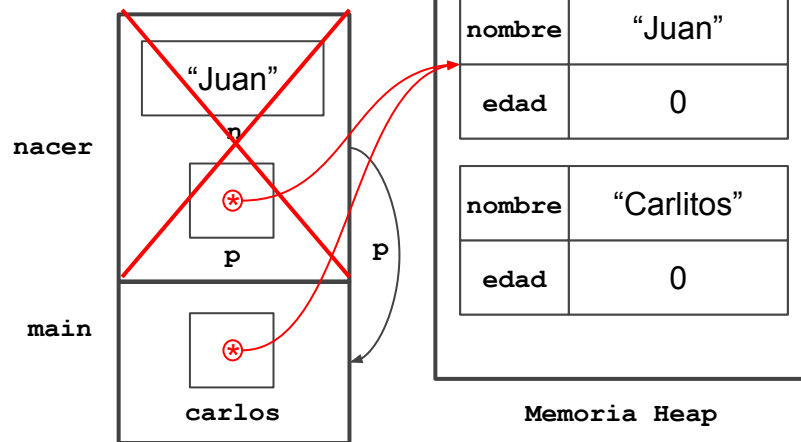
TADs y Memoria Dinámica

```
Persona nacer(string n) {  
    PersonaSt* p = new PersonaSt;  
    p->nombre = n; p->edad = 0;  
    return p;  
}
```

- ¿Se pueden hacer TADs en C/C++?
- Segunda aproximación: memoria dinámica
- Posible problema: **memory leak**

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    carlos = nacer("Juan");  
    ...  
}
```

Se sobrescribe el puntero
con la nueva persona
¿Qué pasó con la anterior?



TADs y Memoria Dinámica

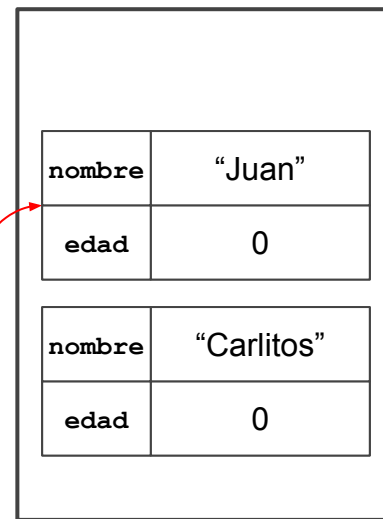
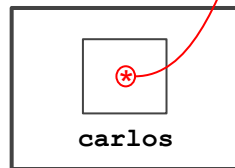
- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Posible problema: **memory leak**

```
#include "Persona.h"

int main() {
    Persona carlos = nacer("Carlitos");
    carlos = nacer("Juan");
    ...
}
```

La memoria de "Carlitos" es inaccesible

main



Memoria Heap

Problemas con la Memoria Dinámica

- ❑ Posible problema: *memory leak*
 - ❑ Memoria solicitada que no puede ser accedida
 - ❑ ¿Cómo se soluciona?
 - ❑ Es necesaria una operación para decir que una memoria solicitada no es más necesaria
 - ❑ **delete**: Libera la reserva sobre una memoria en la heap
 - ❑ Luego del **delete** NO puede usarse el puntero (hasta que sea asignado nuevamente)

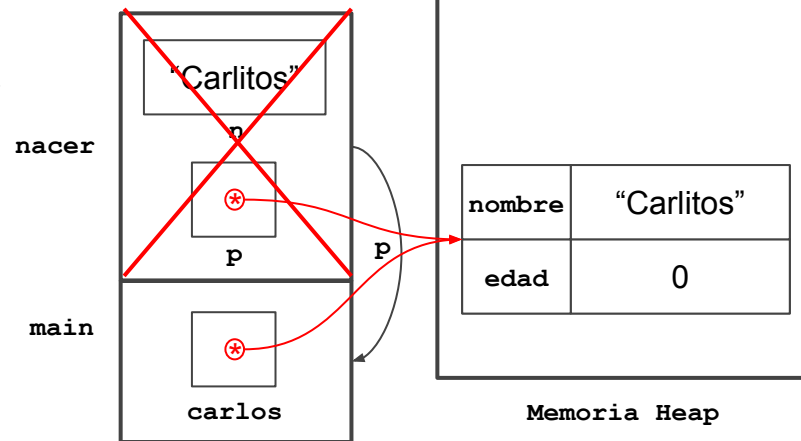
TADs y Memoria Dinámica

```
void Morir(Persona p) {  
    delete p;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Posible problema: **memory leak**

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    Morir(carlos);  
    carlos = nacer("Juan");  
    ...  
}
```

Se crea la primera persona que nace



TADs y Memoria Dinámica

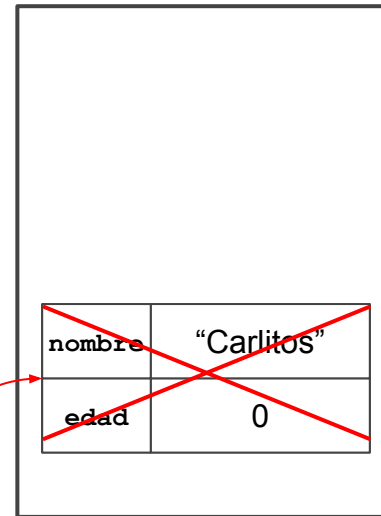
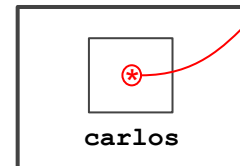
```
void Morir(Persona p) {  
    delete p;  
}
```

- ¿Se pueden hacer TADs en C/C++?
- Segunda aproximación: memoria dinámica
- Posible problema: **memory leak**

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    Morir(carlos);  
    carlos = nacer("Juan");  
    ...  
}
```

Se libera la memoria del
puntero...

main



Memoria Heap

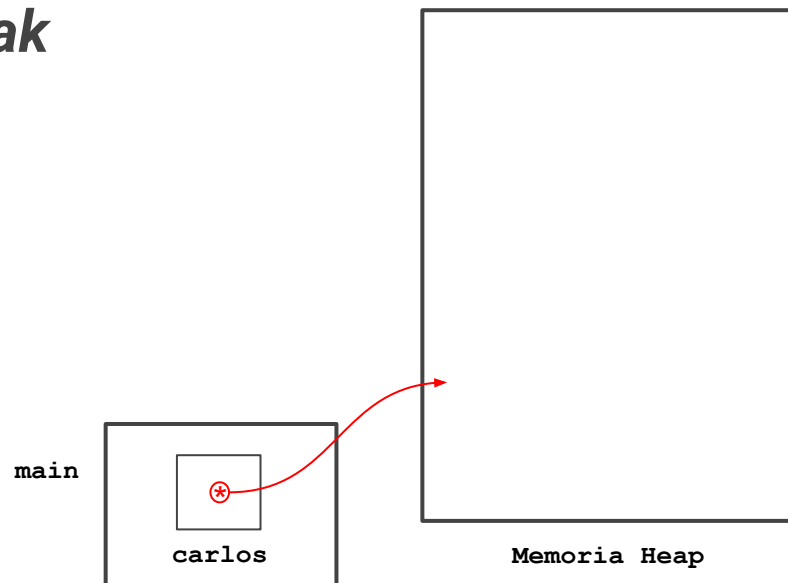
TADs y Memoria Dinámica

```
void Morir(Persona p) {  
    delete p;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Posible problema: **memory leak**

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    Morir(carlos);  
    carlos = nacer("Juan");  
    ...  
}
```

... por lo que NO debe
accederse sin reasignar



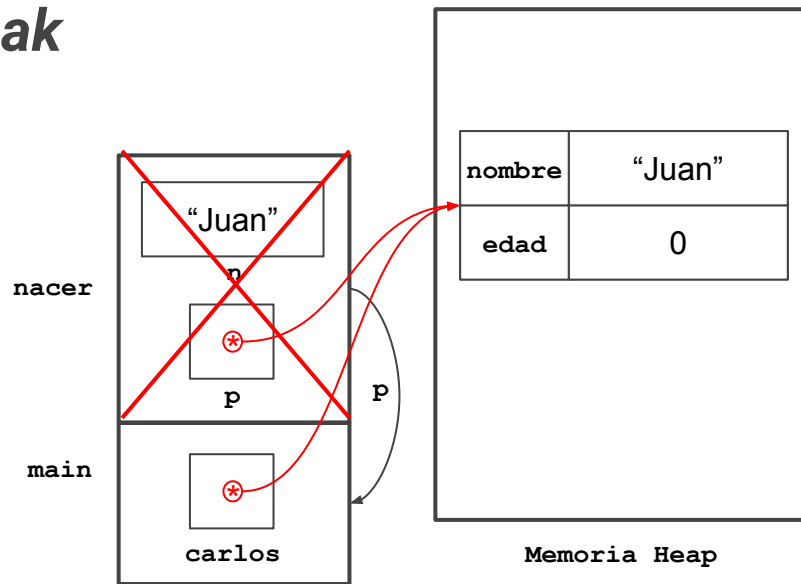
TADs y Memoria Dinámica

```
Persona nacer(string n) {  
    PersonaSt* p = new PersonaSt;  
    p->nombre = n; p->edad = 0;  
    return p;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Posible problema: **memory leak**

```
#include "Persona.h"  
  
int main() {  
    Persona carlos = nacer("Carlitos");  
    Morir(carlos);  
    carlos = nacer("Juan");  
    ...  
}
```

Se sobrescribe el puntero con nuevo valor y NO hay *memory leak*



TADs y Memoria Dinámica

```
Persona nacer(string n) {  
    PersonaSt* p = new PersonaSt;  
    p->nombre = n; p->edad = 0;  
    return p;  
}
```

- ❑ ¿Se pueden hacer TADs en C/C++?
- ❑ Segunda aproximación: memoria dinámica
- ❑ Posible problema: **memory leak**

```
#include "Persona.h"  
int main() {  
    Persona carlos = nacer("Carlitos");  
    Morir(carlos);  
    carlos = nacer("Juan");  
    ...  
}
```

NO hay *memory leak*

main

carlos

nombre	"Juan"
edad	0

Memoria Heap

Ejemplos más complejos

Ejemplos más complejos

- ❏ Personas con mascotas
 - ❏ Agregar mascotas
 - ❏ Agregar que las personas tengan una mascota

Ejemplos más complejos

Personas con mascotas

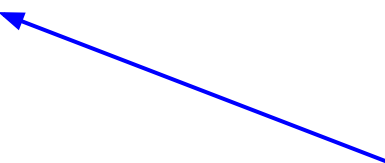
Interfaz (en Mascota.h)

```
struct MascotaSt;  
typedef MascotaSt* Mascota;  
Mascota mascotaNula();  
Mascota nacerMascota(string n, string hab);  
void MorirMascota(Mascota m); // PRECOND: la mascota no es nula  
string nombreMascota(Mascota m); // ídem  
string habilidad(Mascota m); // ídem  
void ShowMascota(Mascota m); // ídem
```

Ejemplos más complejos

- Personas con mascotas
 - Implementación (en `Mascota.cpp`)

```
...  
Mascota mascotaNula() {  
    return NULL;  
}  
...
```



NULL es un puntero que no apunta a ninguna memoria

Ejemplos más complejos

Personas con mascotas

Interfaz (en `Persona.h`)

```
struct PersonaSt;  
typedef PersonaSt* Persona;  
  
Persona nacer(string n);  
void Morir(Persona p);  
void CumplirAnios(Persona p);  
string nombre(Persona p);  
int edad(Persona p);  
Mascota mascota(Persona p);  
void AdoptarMascota(Persona p, Mascota m);  
void LiberarMascota(Persona p);  
void ShowPersona(Persona p);
```

Ejemplos más complejos

- ❏ Personas con mascotas
- ❏ Implementación (en `Persona.cpp`, parte 1)

```
struct PersonaSt {  
    string nombre;  
    int edad;  
    Mascota mascota;  
};  
  
Persona nacer(string n) {  
    PersonaSt* p = new PersonaSt;  
    p->nombre = n; p->edad = 0;  
    p->mascota = mascotaNula();  
    return p;  
}
```

Se precisa para abstraer el hecho
de que es un puntero

Ejemplos más complejos

- Personas con mascotas
- Implementación (en `Persona.cpp`, parte 1)

```
void AdoptarMascota(Persona p, Mascota m) {  
    p->mascota = m;  
}  
  
void LiberarMascota(Persona p) {  
    p->mascota = mascotaNula();  
}  
  
void MorirPersona(Persona p) {  
    delete p;  
}
```

¡No se puede usar **delete** sobre la mascota, porque no se sabe si es compartida!

Ejemplos más complejo

Personas con mascotas

Uso

```
int main() {  
    Persona constanza = nacer("Constanza");  
    Persona fabiana = nacer("Fabiana");  
    Mascota gatucha = nacerMascota("Gatucha", "ronronear");  
    AdoptarMascota(constanza, gatucha);  
    AdoptarMascota(fabiana, gatucha);  
    LiberarMascota(constanza);  
    ...  
}
```

```
Persona nacer(string n);  
void Morir(Persona p);  
void CumplirAños(Persona p);  
string nombre(Persona p);  
int edad(Persona p);  
void AdoptarMascota(Persona p; Mascota m);  
void LiberarMascota(Persona p);  
void ShowPersona(Persona p);
```

Se comparte la mascota
entre dos personas

Si esta operación hiciese **delete** de la mascota,
fabiana tendría un puntero a la nada

Ejemplos más complejos

- ❑ Liberar memoria es complicado
 - ❑ Requiere razonar si el puntero es o no compartido
 - ❑ Solamente se libera si no hay más que lo compartan
- ❑ Existen soluciones para facilitar esto
 - ❑ Punteros “inteligentes”
(saben cuántas veces se compartieron)
 - ❑ Manejo automático de memoria (***garbage collection***)
(en la mayoría de los lenguajes tradicionales)

Arrays

Arrays

- ❑ ¿Cómo manejar muchos datos del mismo tipo?
 - ❑ Posible solución: muchas celdas de memoria contiguas
 - ❑ Nuevo tipo de datos: **arrays** (arreglos)
- ❑ Sintaxis
 - ❑ Se utilizan corchetes para
 - ❑ Indicar que deben reservarse muchas celdas
 - ❑ Acceder a una celda particular
 - ❑ También se puede usar un puntero
 - ❑ Se debe indicar la cantidad a reservar

Arrays

■ Ejemplo 1

■ Inicialización y acceso, estático

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}

int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```

Arrays

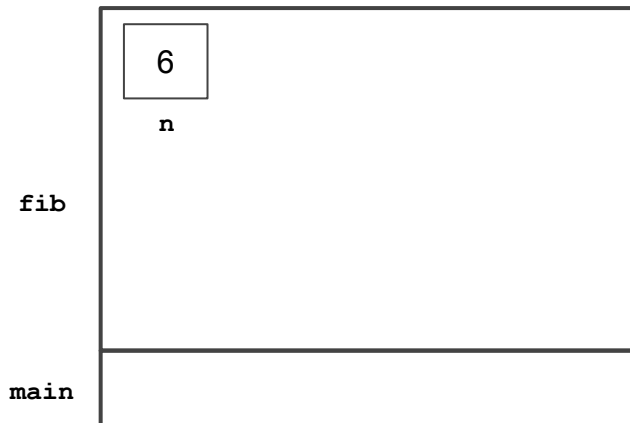
■ Ejemplo 1

■ Inicialización y acceso, estático

Se inicia el programa y se llama a la función `fib`

```
→ int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
→ int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```



Arrays

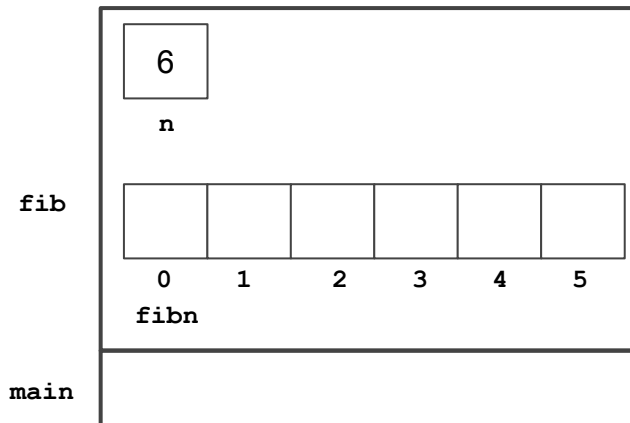
■ Ejemplo 1

■ Inicialización y acceso, estático

Se reservan n celdas para números enteros (un array)

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```



Arrays

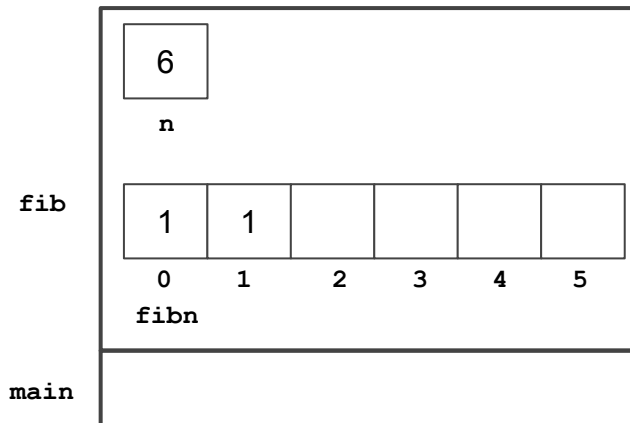
Ejemplo 1

Inicialización y acceso, estático

Se inicializan las primeras 2 posiciones

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```



Arrays

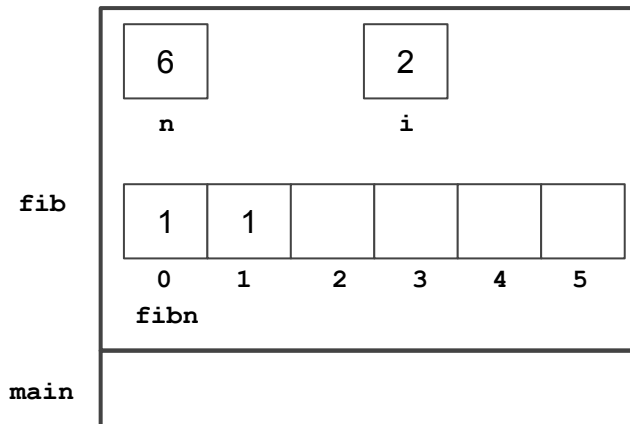
■ Ejemplo 1

■ Inicialización y acceso, estático

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```

Se comienza la repetición



Arrays

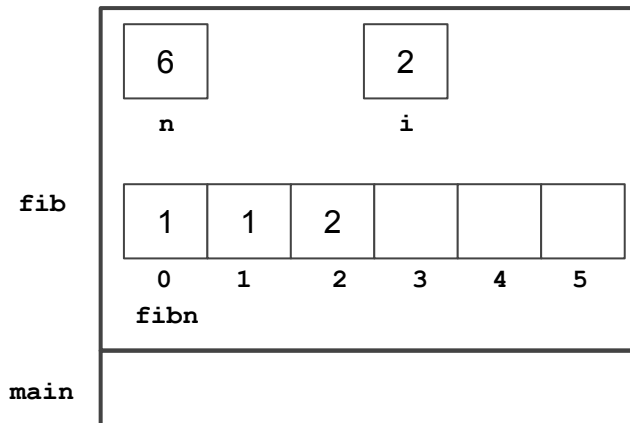
Ejemplo 1

Inicialización y acceso, estático

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```

Primera repetición



Arrays

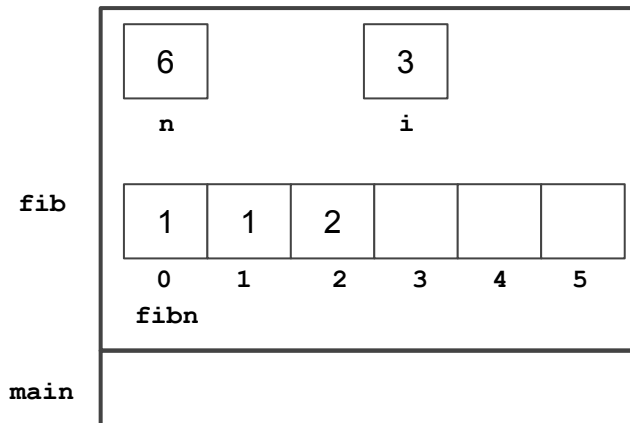
■ Ejemplo 1

■ Inicialización y acceso, estático

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```

Se incrementa y analiza el final



Arrays

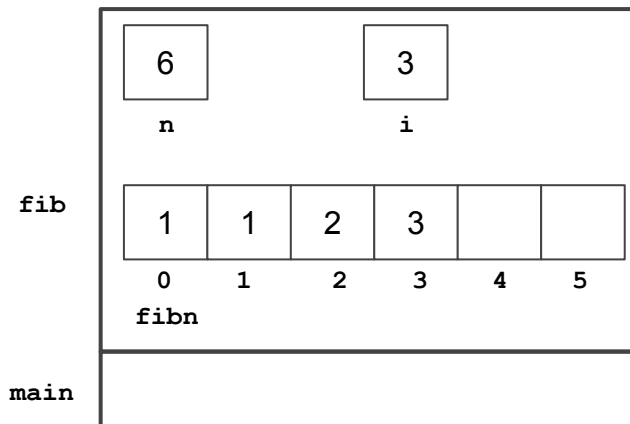
■ Ejemplo 1

■ Inicialización y acceso, estático

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```

Segunda repetición



Arrays

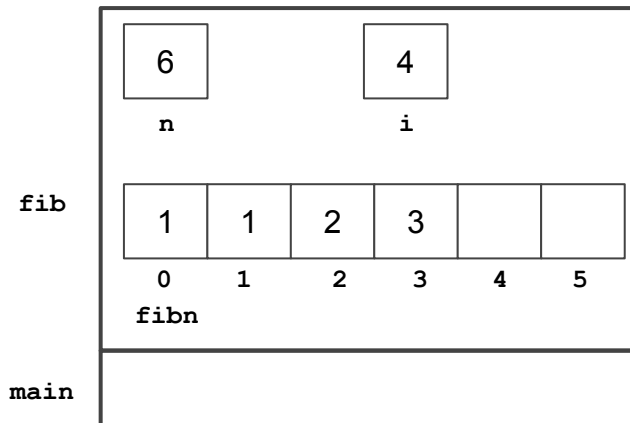
■ Ejemplo 1

■ Inicialización y acceso, estático

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```

Se incrementa y analiza el final



Arrays

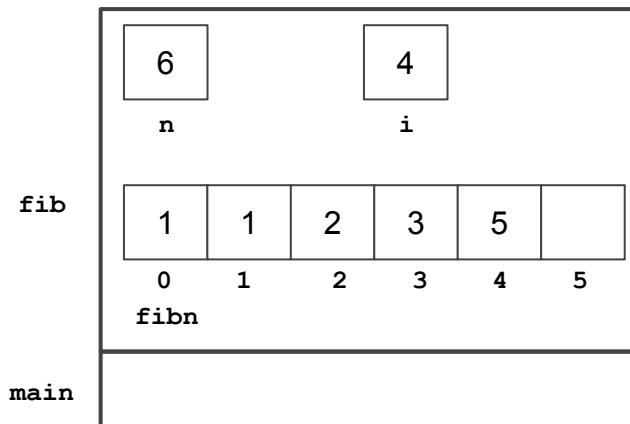
■ Ejemplo 1

■ Inicialización y acceso, estático

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```

Tercera repetición



Arrays

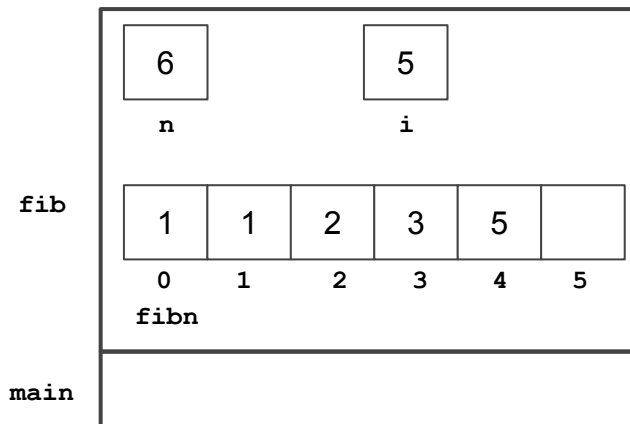
■ Ejemplo 1

■ Inicialización y acceso, estático

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```

Se incrementa y analiza el final



Arrays

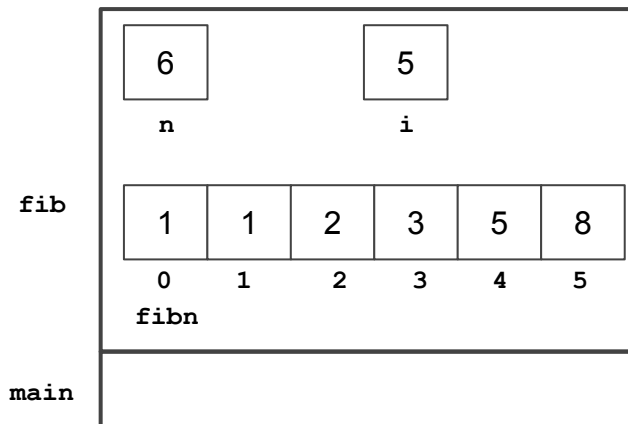
Ejemplo 1

Inicialización y acceso, estático

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```

Cuarta repetición



Arrays

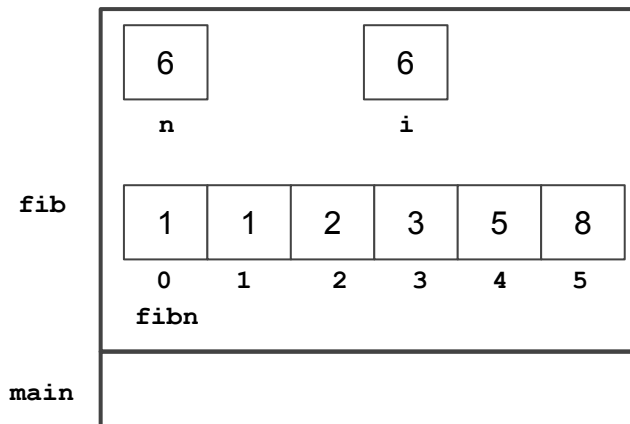
■ Ejemplo 1

■ Inicialización y acceso, estático

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```

Se incrementa y analiza el final



Arrays

■ Ejemplo 1

■ Inicialización y acceso, estático

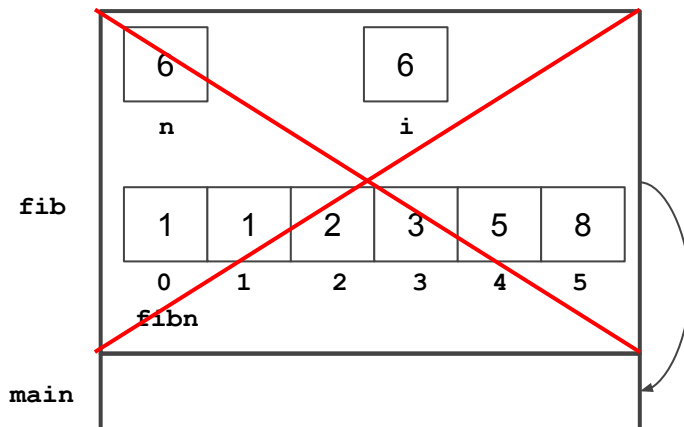
```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```



```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```



Se completa la repetición y
se retorna el valor buscado



Arrays

■ Ejemplo 1

■ Inicialización y acceso, estático

```
int fib(int n) { // PRECOND: n>0
    int fibn[n];
    fibn[0] = 1; fibn[1] = 1;
    for (int i=2; i<n; i++) {
        fibn[i] = fibn[i-2] + fibn[i-1];
    }
    return(fibn[n-1]);
}
```

```
int main() {
    cout << "fib(6) = " << fib(6) << endl;
}
```

Se completa la repetición y se retorna el valor buscado

main

Arrays

- ❑ Al utilizar memoria estática
 - ❑ El array deja de existir al completar la función
 - ❑ No se puede retornar un array
- ❑ Para retornarlo se requiere usar memoria dinámica

Arrays

■ Ejemplo 2

■ Inicialización y acceso, dinámico

```
int* countdown(int n, int j) { // PRECOND: 0<=j<n
    int* ns = new int[n-j+1];
    for (int i=0; i<=n-j; i++) {
        ns[i] = n-i;
    }
    return ns;
}

int main() {
    int* ns = countdown(10,6);
    for (int t=0; t<5; t++) {
        cout << ns[t] << " ";
    }
}
```

Arrays

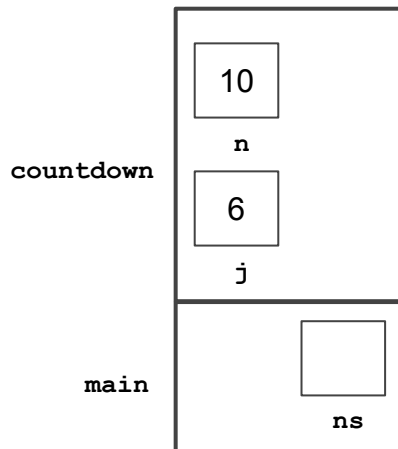
Ejemplo 2

Inicialización y acceso, dinámico

Se comienza el programa y se llama a la función

```
int* countdown(int n, int j) { // PRECOND: 0<=j<n
    int* ns = new int[n-j+1];
    for (int i=0; i<=n-j; i++) {
        ns[i] = n-i;
    }
    return ns;
}
```

```
int main() {
    int* ns = countdown(10,6);
    for (int t=0; t<5; t++) {
        cout << ns[t] << " ";
    }
}
```



Memoria Heap

Arrays

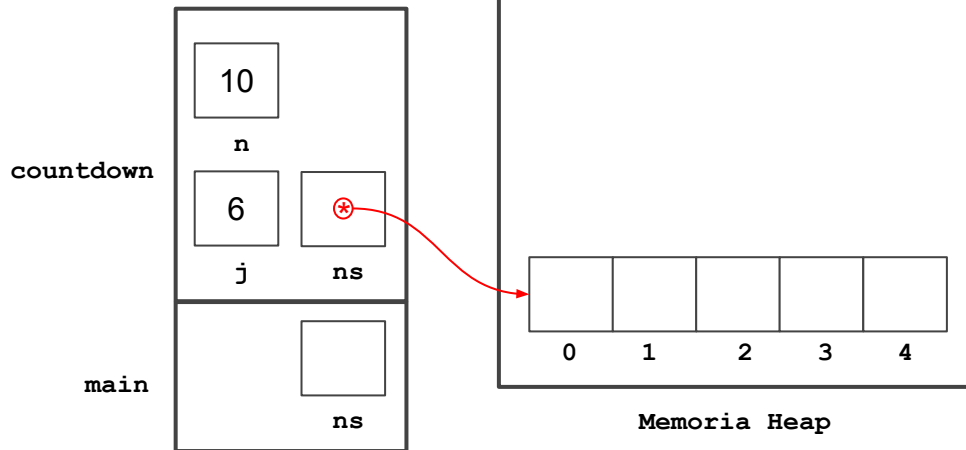
■ Ejemplo 2

■ Inicialización y acceso, dinámico

Se reserva memoria para el array (en la Heap)

```
int* countdown(int n, int j) { // PRECOND: 0<=j<n
    int* ns = new int[n-j+1];
    for (int i=0; i<=n-j; i++) {
        ns[i] = n-i;
    }
    return ns;
}
```

```
int main() {
    int* ns = countdown(10,6);
    for (int t=0; t<5; t++) {
        cout << ns[t] << " ";
    }
}
```



Arrays

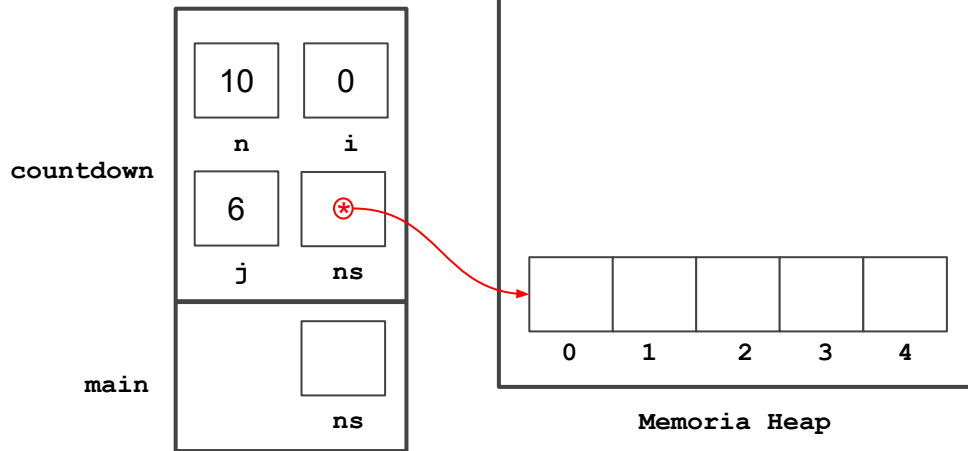
Ejemplo 2

Inicialización y acceso, dinámico

Se comienza la repetición

```
int* countdown(int n, int j) { // PRECOND: 0<=j<n
    int* ns = new int[n-j+1];
    for (int i=0; i<=n-j; i++) {
        ns[i] = n-i;
    }
    return ns;
}
```

```
int main() {
    int* ns = countdown(10,6);
    for (int t=0; t<5; t++) {
        cout << ns[t] << " ";
    }
}
```



Arrays

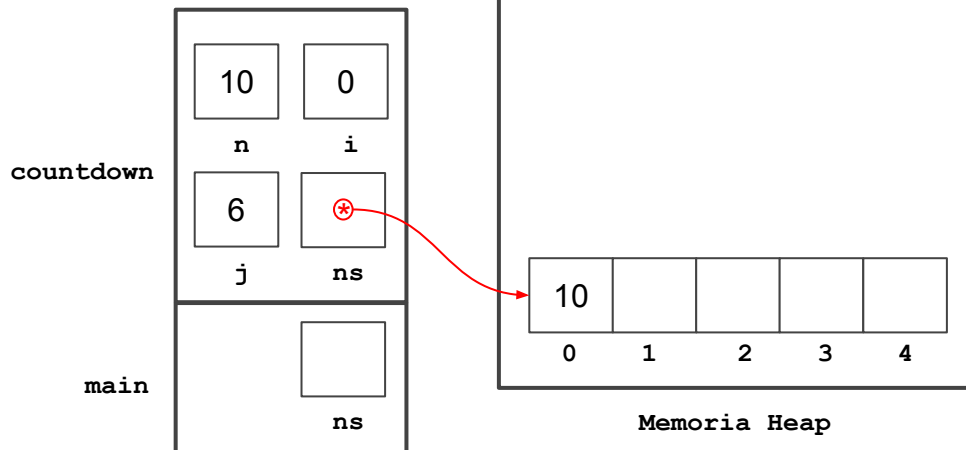
Ejemplo 2

Inicialización y acceso, dinámico

Primera repetición

```
int* countdown(int n, int j) { // PRECOND: 0<=j<n
    int* ns = new int[n-j+1];
    for (int i=0; i<=n-j; i++) {
        ns[i] = n-i;
    }
    return ns;
}
```

```
int main() {
    int* ns = countdown(10,6);
    for (int t=0; t<5; t++) {
        cout << ns[t] << " ";
    }
}
```



Arrays

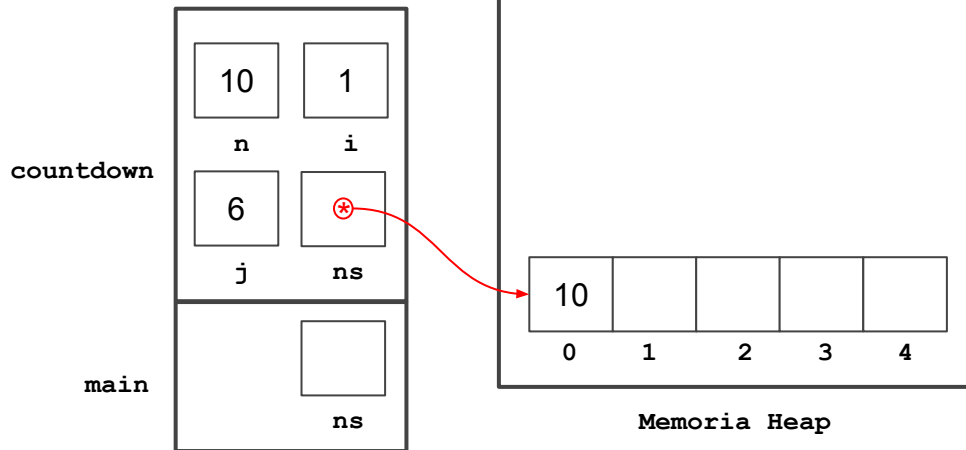
Ejemplo 2

Inicialización y acceso, dinámico

Se incrementa y analiza el final

```
int* countdown(int n, int j) { // PRECOND: 0<=j<n
    int* ns = new int[n-j+1];
    for (int i=0; i<=n-j; i++) {
        ns[i] = n-i;
    }
    return ns;
}
```

```
int main() {
    int* ns = countdown(10,6);
    for (int t=0; t<5; t++) {
        cout << ns[t] << " ";
    }
}
```



Arrays

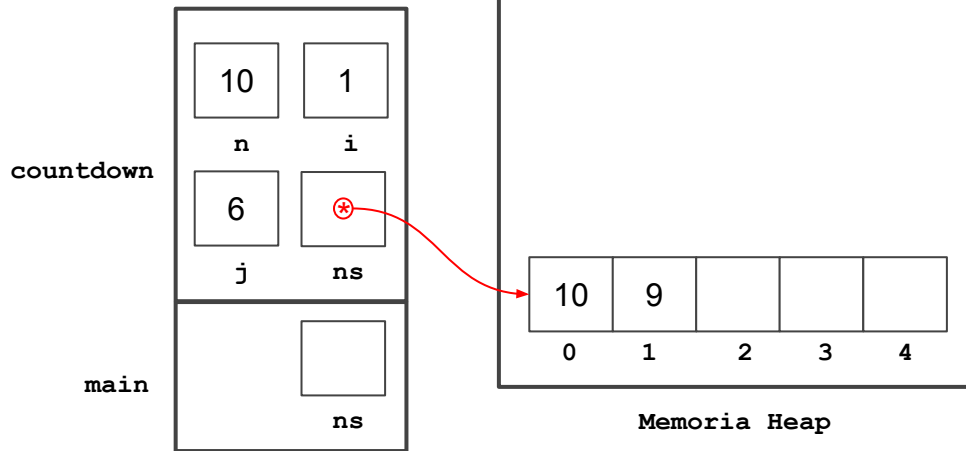
Ejemplo 2

Inicialización y acceso, dinámico

Segunda repetición

```
int* countdown(int n, int j) { // PRECOND: 0<=j<n
    int* ns = new int[n-j+1];
    for (int i=0; i<=n-j; i++) {
        ns[i] = n-i;
    }
    return ns;
}

int main() {
    int* ns = countdown(10,6);
    for (int t=0; t<5; t++) {
        cout << ns[t] << " ";
    }
}
```



Arrays

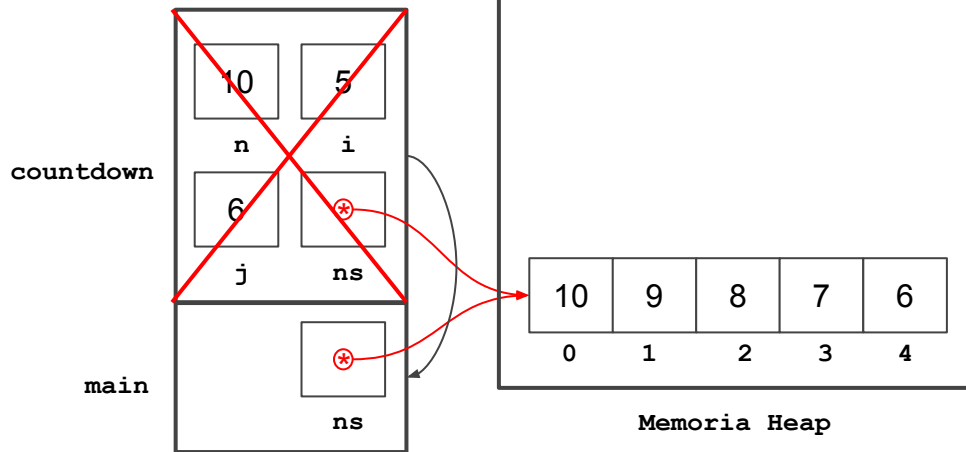
Ejemplo 2

Inicialización y acceso, dinámico

Se completa la repetición y se retorna el valor buscado

```
int* countdown(int n, int j) { // PRECOND: 0<=j<n
    int* ns = new int[n-j+1];
    for (int i=0; i<=n-j; i++) {
        ns[i] = n-i;
    }
    return ns;
}
```

```
int main() {
    int* ns = countdown(10,6);
    for (int t=0; t<5; t++) {
        cout << ns[t] << " ";
    }
}
```



Arrays

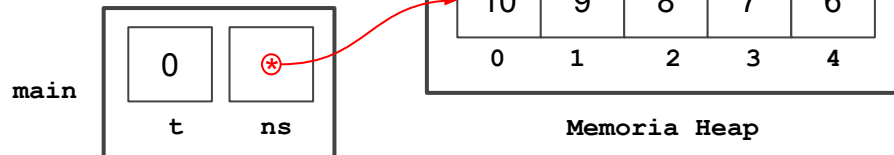
■ Ejemplo 2

■ Inicialización y acceso, dinámico

Se continúa con el programa

```
int* countdown(int n, int j) { // PRECOND: 0<=j<n
    int* ns = new int[n-j+1];
    for (int i=0; i<=n-j; i++) {
        ns[i] = n-i;
    }
    return ns;
}

int main() {
    int* ns = countdown(10,6);
    for (int t=0; t<5; t++) {
        cout << ns[t] << " ";
    }
}
```



Arrays

❏ Ventajas

- ❏ El acceso a cualquiera de los datos es $O(1)$
(sabiendo la posición)

❏ Desventajas

- ❏ Debe conocerse de antemano la cantidad máxima
- ❏ Si se llena, no se puede usar más
(esto puede aliviarse con algo de trabajo)
- ❏ Pueden quedar “agujeros”
(hay que prestar atención o mover todos los elementos)

Arrays en TADs

Ejemplo 3: personas con múltiples mascotas

```
struct PersonaSt;  
typedef PersonaSt* Persona;  
  
Persona nacer(string n);  
void Morir(Persona p);  
void CumplirAnios(Persona p);  
string nombre(Persona p);  
int edad(Persona p);  
Mascota mascotaNro(Persona p, int i);  
void AdoptarNuevaMascota(Persona p, Mascota m);  
void LiberarMascota(Persona p, string nombre);  
void ShowPersona(Persona p);
```


Arrays en TADs

Ejemplo 3: personas con múltiples mascotas

```
int main() {  
    Persona constanza = nacer("Constanza");  
    AdoptarNuevaMascota(constanza  
                        ,nacerMascota("Gatucha", "ronronear"));  
    AdoptarNuevaMascota(constanza  
                        ,nacerMascota("Dinamita", "recuperar palitos"));  
    AdoptarNuevaMascota(constanza  
                        ,nacerMascota("Nemo", "aleta feliz"));  
    ShowPersona(constanza);  
}
```

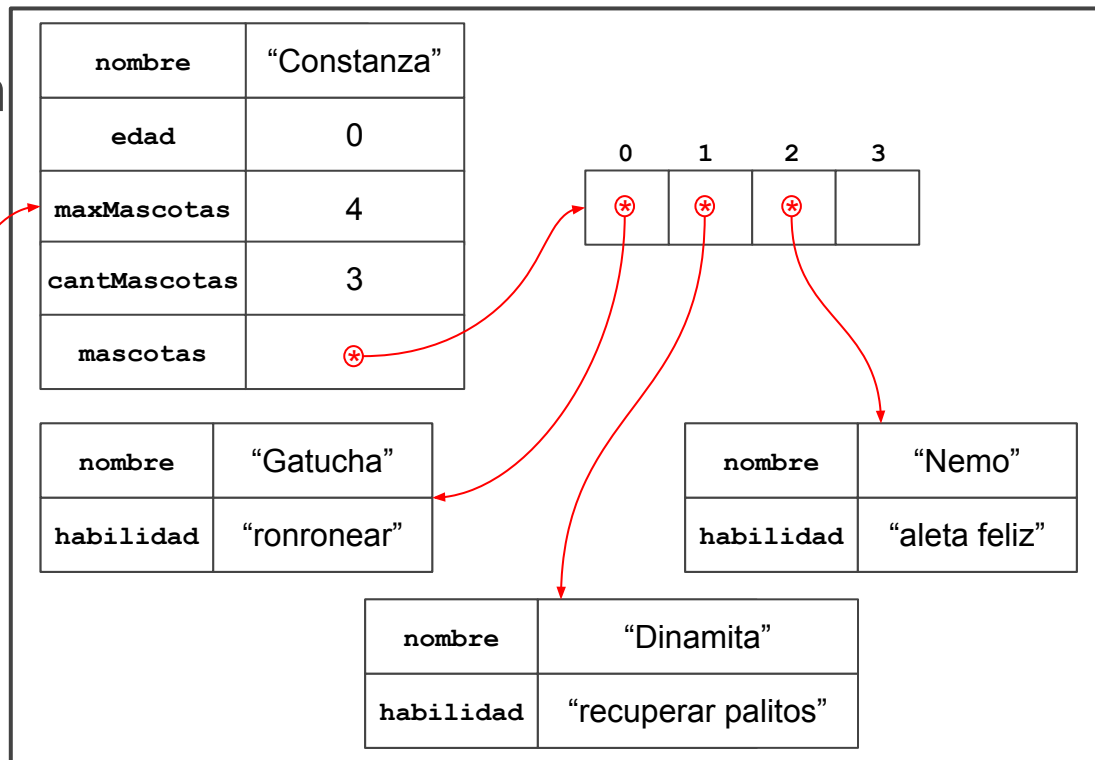
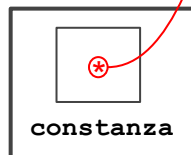
Arrays en TADs

```
int main() {  
    Persona constanza = nacer("Constanza");  
    AdoptarNuevaMascota(constanza, nacerMascota("Gatucha", "ronronear"));  
    AdoptarNuevaMascota(constanza, nacerMascota("Dinamita", "recuperar palitos"));  
    AdoptarNuevaMascota(constanza, nacerMascota("Nemo", "aleta feliz"));  
    ShowPersona(constanza);  
}
```

■ Ejemplo 3: person

■ Así debería
quedar la
memoria

main



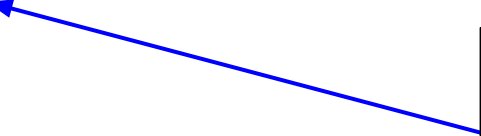
Memoria Heap

Arrays en TADs

❏ Ejemplo 3: personas con múltiples mascotas

```
struct PersonaSt { // INV.REP.: cantMascotas<=maxMascotas
    string nombre;
    int edad;
    int maxMascotas;
    int cantMascotas;
    Mascota* mascotas;
};

Persona nacer(string n) {
    PersonaSt* p = new PersonaSt;
    p->nombre = n; p->edad = 0;
    p->maxMascotas = 2; p->cantMascotas = 0;
    p->mascotas = new Mascota[p->maxMascotas];
    return p;
}
```



El máximo indica el tamaño del array y la cantidad indica cuántas hay realmente

Arrays en TADs

■ Ejemplo 3: personas con múltiples mascotas

```
void DuplicarEspacioDeMascotas(Persona p) {  
    Mascota* temp = new Mascota[p->maxMascotas*2];  
    for(int i=0; i<p->maxMascotas; i++)  
        { temp[i] = p->mascotas[i]; }  
    delete p->mascotas;  
    p->maxMascotas = p->maxMascotas*2;  
    p->mascotas = temp;  
}
```

Se duplica el espacio y se copia el contenido anterior en el nuevo

```
void AdoptarNuevaMascota(Persona p, Mascota m) {  
    if (p->cantMascotas == p->maxMascotas) {  
        DuplicarEspacioDeMascotas(p);  
    }  
    p->mascotas[p->cantMascotas++] = m;  
}
```

Si esto sucede, entonces no hay más espacio para guardar mascotas

Arrays en TADs



Ejemplo 3: personas con múltiples mascotas

```
void Morir(Persona p) {  
    delete p->mascotas;  
    delete p;  
}  
  
Mascota mascotaNro(Persona p, int i) {  
    // PRECOND: 0 < i <= p->cantMascotas  
    if (i > 0 && i <= p->cantMascotas) {  
        return p->mascotas[i-1];  
    }  
    return mascotaNula();  
}
```


No hay que olvidar eliminar el espacio del array (NO de las mascotas)

Recordar que el array arranca en 0

Arrays en TADs

Ejemplo 3: personas con múltiples mascotas

```
void ShowPersona(Persona p) {  
    cout << "Persona[" << p << "]" (";  
    cout << "nombre <- \"\" << p->nombre << "\", ";  
    cout << "edad <- \" << p->edad << \",\";  
    cout << "mascotas <- [ ";  
    for (int i=0; i < p->cantMascotas; i++) {  
        if (i>0) { cout << ", "; }  
        ShowMascota(p->mascotas[i]);  
    }  
    cout << " ]\"";  
}
```



La primera mascota no
lleva el separador (,)

Arrays en TADs

Se busca la posición
donde está esa mascota

■ Ejemplo 3: personas con múltiples mascotas

```
void LiberarMascota(Persona p, string nombre) {  
    int i = 0;  
    while (i < p->cantMascotas  
        && nombreMascota(p->mascotas[i]) != nombre) {  
        i++;  
    }  
    if (i < p->cantMascotas) {  
        p->cantMascotas--;  
    }  
    while (i < p->cantMascotas) {  
        p->mascotas[i] = p->mascotas[i+1];  
        i++;  
    }  
}
```

Todos los que siguen se mueven
un lugar a la izquierda (para
evitar "agujeros")

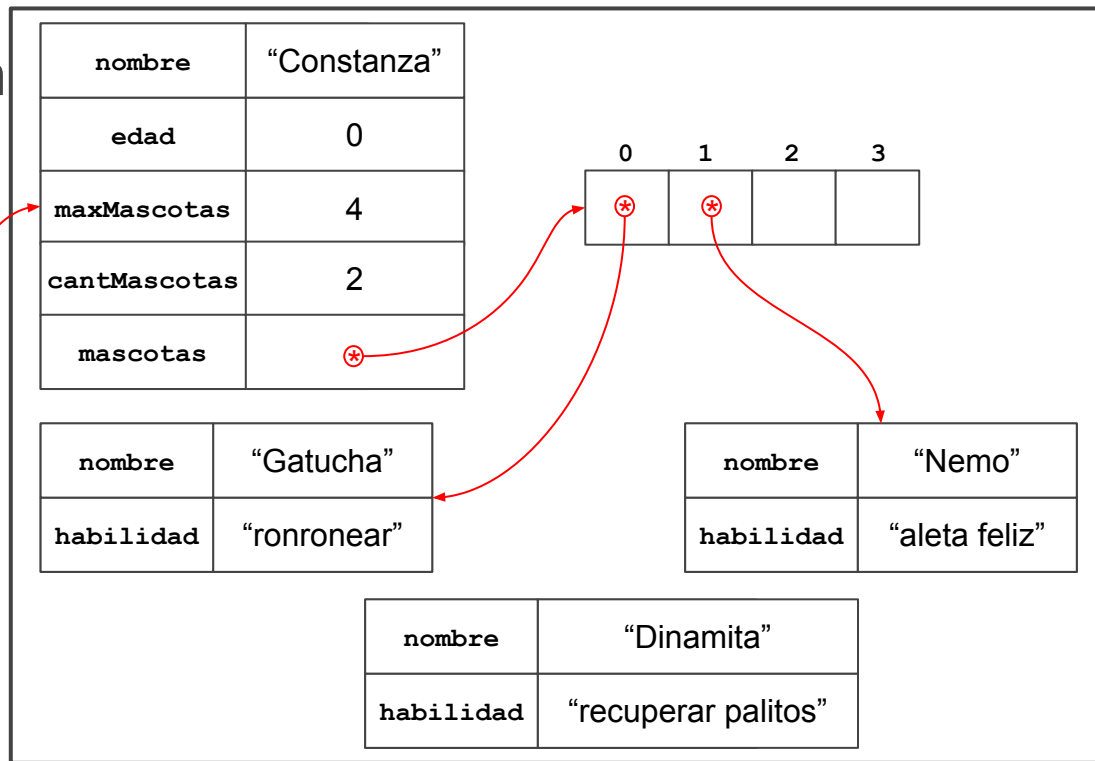
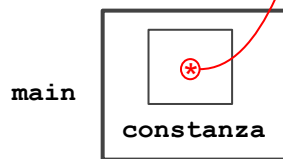
// i+1 es a lo sumo
cantMascotas-1 pero
cantMascotas fue decrementada

Arrays en TADs

```
int main() {  
    Persona constanza = nacer("Constanza");  
    AdoptarNuevaMascota(constanza, nacerMascota("Gatucha", "ronronear"));  
    AdoptarNuevaMascota(constanza, nacerMascota("Dinamita", "recuperar palitos"));  
    AdoptarNuevaMascota(constanza, nacerMascota("Nemo", "aleta feliz"));  
    LiberarMascota(constanza, "Dinamita");  
    ShowPersona(constanza);  
}
```

■ Ejemplo 3: person

■ Así debería
quedar la
memoria



Memoria Heap

Arrays

- ❑ El manejo para solucionar el problema del tamaño máximo se puede generalizar
- ❑ Estructura de datos: ***ArrayList***
 - ❑ Reserva una cantidad fija; si hace falta más, se duplica
 - ❑ Los elementos del array viejo se copian al nuevo
 - ❑ Se libera el array viejo

Arrays de más dimensiones

- ❑ Los arrays pueden tener más de una dimensión
- ❑ En el caso de 2 dimensiones, se llaman ***matrices***
- ❑ Para 3 o más dimensiones, se suelen llamar ***tensores***
- ❑ Ejemplo: el tablero de Gobstones
- ❑ Pero C/C++ no maneja arrays de más dimensiones
 - ❑ Opción 1: array de 1 dimensión y hacer cuentas
 - ❑ Opción 2: array de punteros a arrays

Arrays de más dimensiones

■ Ejemplo 4: el tablero de Gobstones

```
typedef int Color;  
#define Azul 0  
#define Negro 1  
#define Rojo 2  
#define Verde 3
```

El **#define** es para hacer alias
que reemplacen expresiones

```
typedef int Direccion;  
#define Norte 0  
#define Este 1  
#define Sur 2  
#define Oeste 3
```

Arrays de más dimensiones

Ejemplo 4: el tablero de Gobstones

```
struct CeldaSt;  
typedef CeldaSt* Celda;  
  
Celda celdaVacía();  
void BorrarCelda(Celda c);  
void PonerEnCelda(Celda c, Color color);  
void SacarEnCelda(Celda c, Color color);  
int  nroBolitasEnCelda(Celda c, Color color);  
bool hayBolitasEnCelda(Celda c, Color color);  
void ShowCelda(Celda c);
```

Arrays de más dimensiones

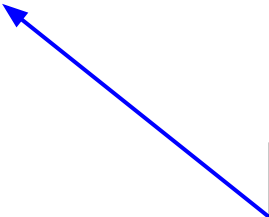
Ejemplo 4: el tablero de Gobstones

```
struct TableroSt;  
typedef TableroSt* Tablero;  
  
Tablero tableroNuevo(int columnas, int filas);  
void      EliminarTablero(Tablero t);  
void      Poner(Tablero t, Color color);  
void      Sacar(Tablero t, Color color);  
void      Mover(Tablero t, Direccion dir);  
int       nroBolitas(Tablero t, Color color);  
bool      hayBolitas(Tablero t, Color color);  
bool      puedeMover(Tablero t, Direccion dir);  
void      ShowTablero();
```

Arrays de más dimensiones

■ Ejemplo 4: el tablero de Gobstones

```
struct TableroSt {  
    int maxCol;    int maxRow; // Tamaño del tablero  
    int curCol;    int curRow; // Pos. del cabezal  
    Celda* celdas;  
};  
  
int index(Tablero t, int col, int row) {  
    return(row + col * t->maxRow);  
}
```



Esto calcula la posición lineal de la celda en la posición (**col**, **row**)

Arrays de más dimensiones

■ Ejemplo 4: el tablero de Gobstones

```
Tablero tableroNuevo(int columnas, int filas) {  
    TableroSt* t = new TableroSt;  
    t->maxCol = columnas; t->maxRow = filas;  
    t->curCol = 0;          t->curRow = 0;  
    t->celdas = new Celda[t->maxCol * t->maxRow];  
    for(int c=0; c<t->maxCol; c++) {  
        for(int f=0; f<t->maxRow; f++) {  
            t->celdas[index(t,c,f)] = celdaVacía();  
        }  
    }  
    return t;  
}
```

Se crea una matriz
como un array con
todas las celdas en fila

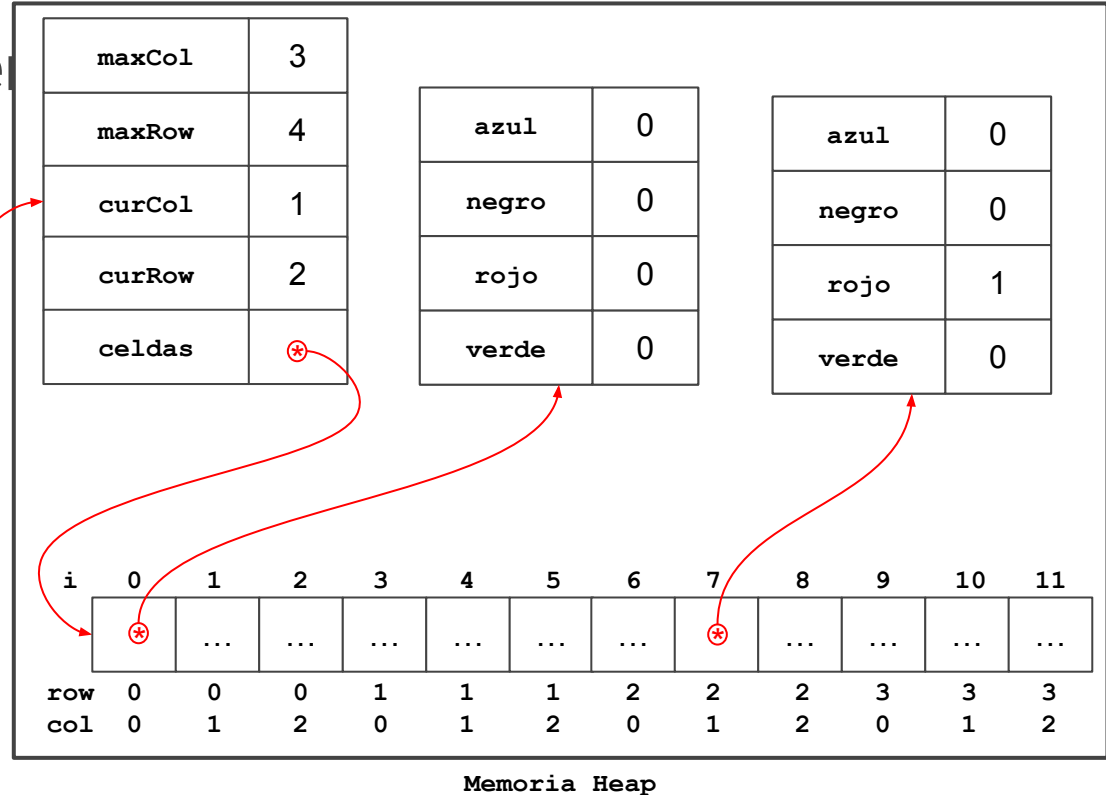
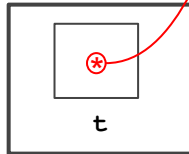
Se accede a una posición de la
matriz calculando la posición lineal

Arrays de más dimensiones

Ejemplo 4: el tablero

Así debería quedar la memoria

main



Arrays de más dimensiones

- Ejemplo 4: el tablero de Gobstones
 - Esta implementación tiene demasiados punteros
 - Otra mejor no usaría el tipo Celda, sino 4 arrays de ints
 - Cada array sería las bolitas de un color
 - El cálculo de índices lineales es el mismo

Resumen

Resumen

- ❏ Memoria dinámica: controlada por el programador
 - ❏ Operaciones: **new**, **delete**
 - ❏ Tipo de datos: punteros (*****, **->**)
 - ❏ Problema: *memory leaks*
- ❏ Arrays: celdas contiguas
 - ❏ En memoria estática o dinámica
 - ❏ Debe recordarse el tamaño
 - ❏ Problema: acceder fuera los límites
 - ❏ Problema: acceder a celdas no inicializadas