# Memoria III

V

2024

UNRN

Universidad Nacional
de **Río Negro**

r20

# ¿Cómo les fue?

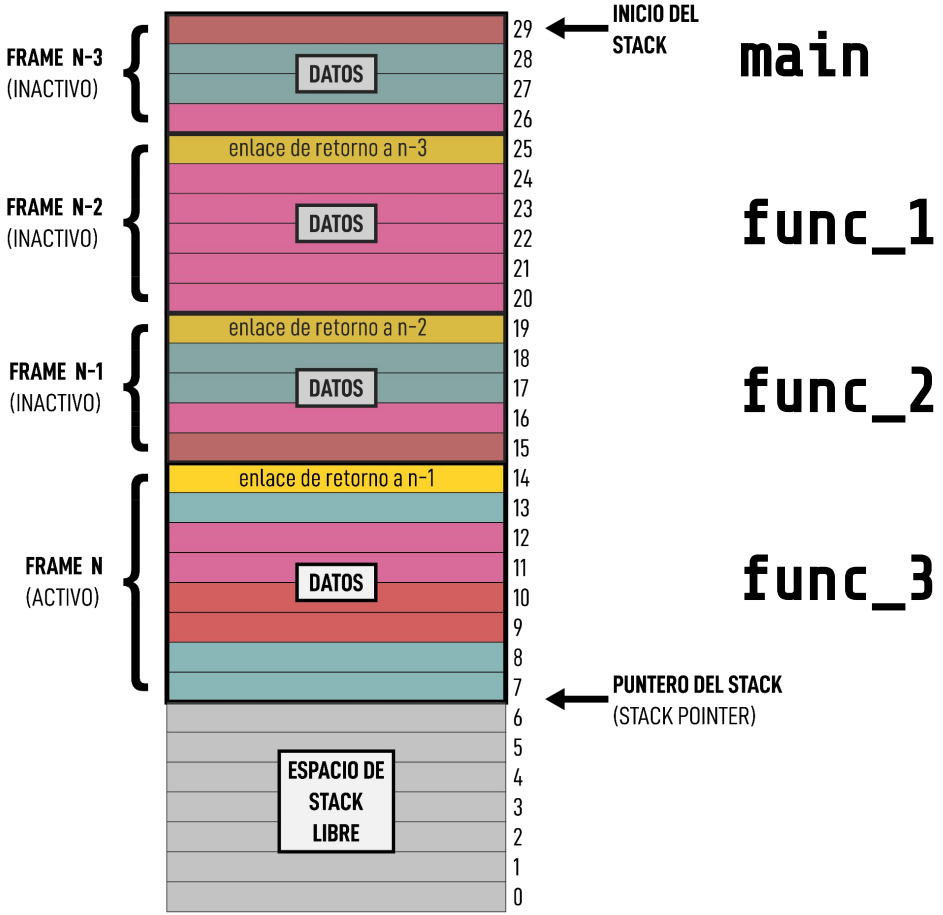

¿Preguntas?

# Memoria automática

# de Stack

# 1 alta eficiencia

No hay 'huecos'

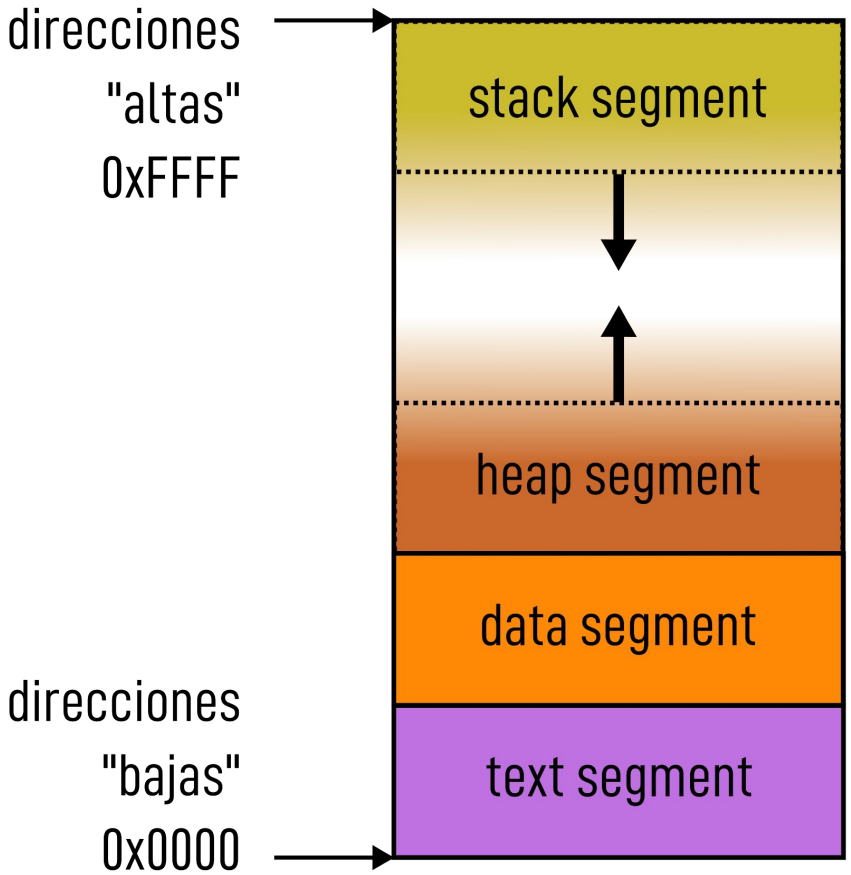


# 2

# La reutilización continua

En donde se alojan las variables

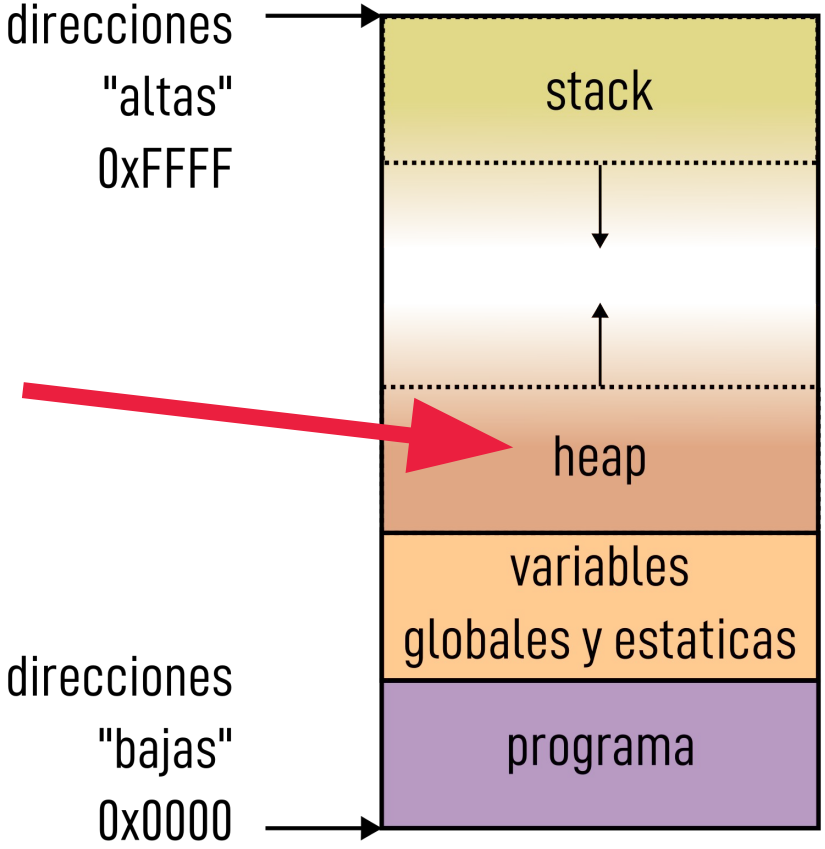
# El mapa de la memoria




¿Preguntas?

# sizeof

# Memoria Dinámica [de heap]


direcciones
"altas"
0xFFFF
stack
heap
variables
globales y estaticas
programa
direcciones
"bajas"
0x0000

# ¿Por qué?

# ventajas/ desventajas

# Como se usa

en `stdlib.h`

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

en `stdlib.h`


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

El bloque pedido

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

# ¿lo qué?

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

# `void` no era *sin retorno*

`void*`

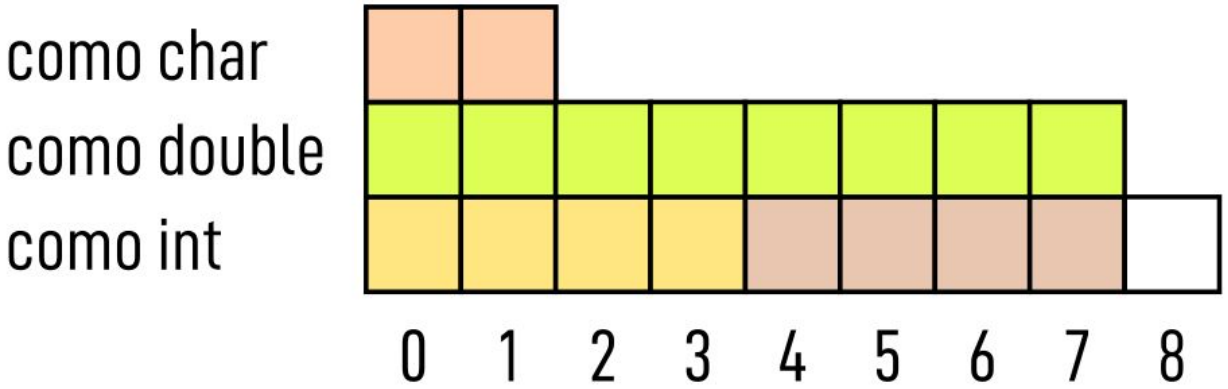
# ¡Un puntero!

`void*`

`void*`

**a solo la ubicación en memoria sin noción de ‘ancho’**

# ¿Ancho? (en memoria)






*no a escala

# `void*` contiene solo la dirección

**Por lo que**

# No es válido un arreglo de void*

```
void arreglo[10];
```

# Tampoco es válido

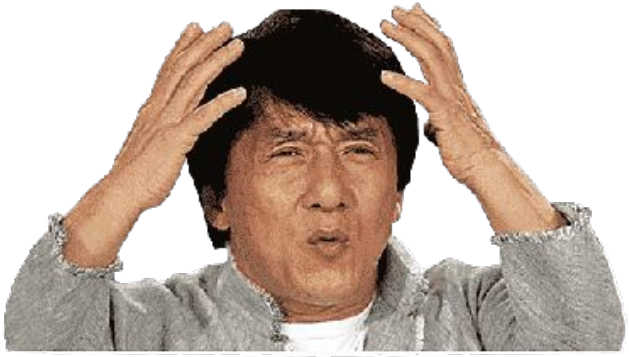
```
void* arreglo = ...;
arreglo[n] = 10
a = arreglo[n]
```

```
          char*
void* ≠ int*
          double*
```


¿Preguntas?

# Pero entonces

# ¿¿Pero como se usa??



# ¿Cómo indicamos el ancho?

# Con ejemplo simple

```
int numero = 10;
void* ptr = &numero;
```

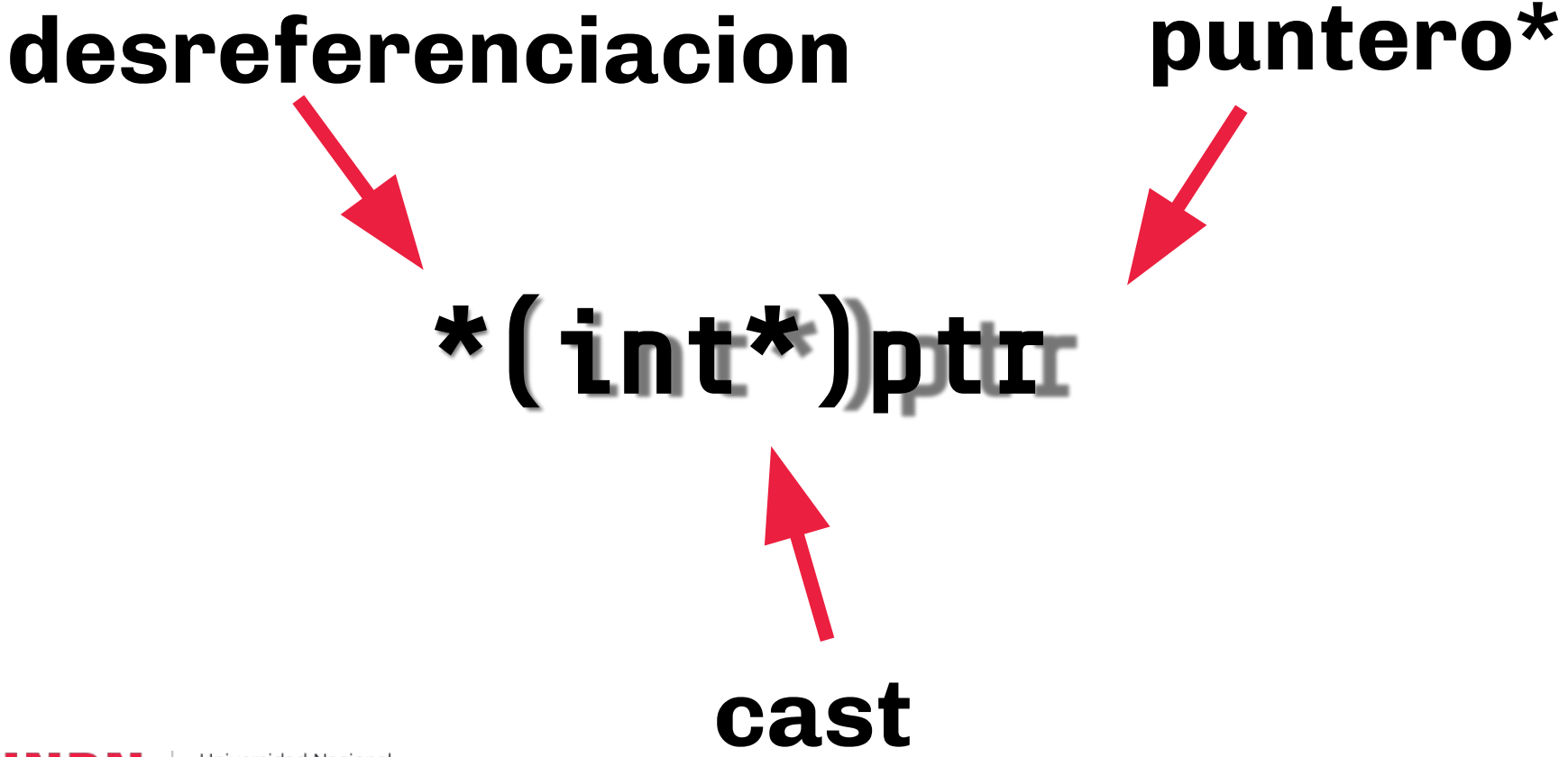
# ¿podemos desreferenciar `ptr`?

```
int numero = 10;
void* ptr = &numero;
```

```
int numero = 10;
void* ptr = &numero;

printf("Es %d\n",  *ptr );
```


casts


desreferenciacion
puntero*
*(int*)ptr
cast

# Con una variable intermedia

```
int numero = 10;
void* ptr = &numero;
int* int_ptr = (int*)ptr;

printf("Es %d\n",  *int_ptr );
```

¡Ahora sí!

# O todo junto

```
int numero = 10;
void* ptr = &numero;

printf("Es %d\n",  *(int*)ptr );
```


¿Preguntas?

**void** solo es nada

**pero**

**void*** es cualquier cosa


¿Preguntas?

# Efectos en las funciones

1

2

# Si una función trabaja con `void*`, no se espera modificar el contenido

# Volviendo al `malloc`

# ¿Cómo se usa?

# un ejemplo de uso (para un valor individual)

```
char *cadena;
cadena = (char*) malloc(sizeof(char));
```

cast

el tamaño individual

# un ejemplo de uso (para un arreglo)

que podemos multiplicar

```
char *cadena;
cadena = (char*) malloc(sizeof(char) * cantidad);
```

cast

el tamaño individual

# Después, a usarlo como un arreglo más

# { Lo que sale del `malloc` no está garantizado en `0` }


¿Preguntas?

# ¿Puede fallar?


¡Sí!

`malloc` puede no tener memoria para dar

# En tal caso, el puntero apunta a NULL

# Completando el ejemplo

```
char *cadena;
cadena = (char *) malloc(sizeof(char) * cantidad);
if (cadena == NULL)
{
  abort(); //Oops, nos quedamos sin memoria!
}
//magic happens, le damos uso
```

# Combinado en una sola expresión.

```
char *cadena;
if ((cadena = (char *) malloc(sizeof(char) * cantidad)){
    //OK
} else {
    // Falló
}
```


¿Preguntas?

# Consideraciones de uso

Sí sabemos el tamaño de las cosas antes de compilar

**Mejor un arreglo común**

# Y no es muy útil para valores individuales

# Ya que es MUY* recomendable

*Léase SIEMPRE

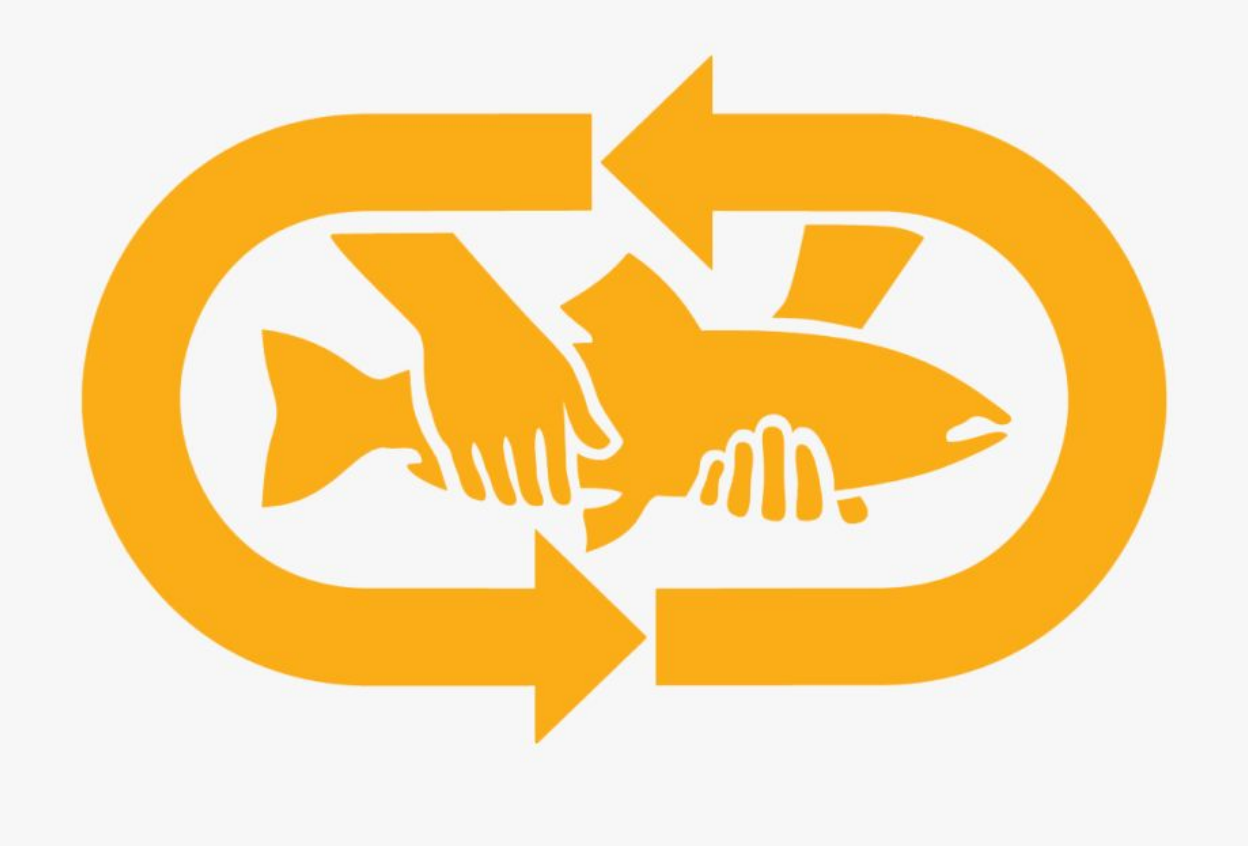
# Liberar la memoria

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

# acepta el puntero ‘casteado’

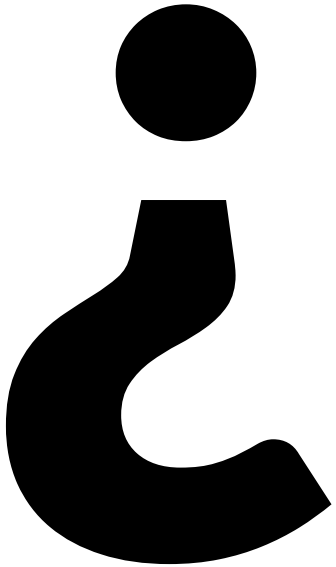
```
void free(void* ptr);
```

# Catch and release!



# Completando el ejemplo

```
char* cadena;
cadena = (char*) malloc(sizeof(char) * cantidad);
if (cadena == NULL)
{
  abort();//Oops, nos quedamos sin memoria!
}
//magic happens, le damos uso
free(cadena);
```



# Que pasa si...

```
void free(NULL);
```


Kaboom?

# Nope, no pasa nada

# Nope, *no pasa nada*

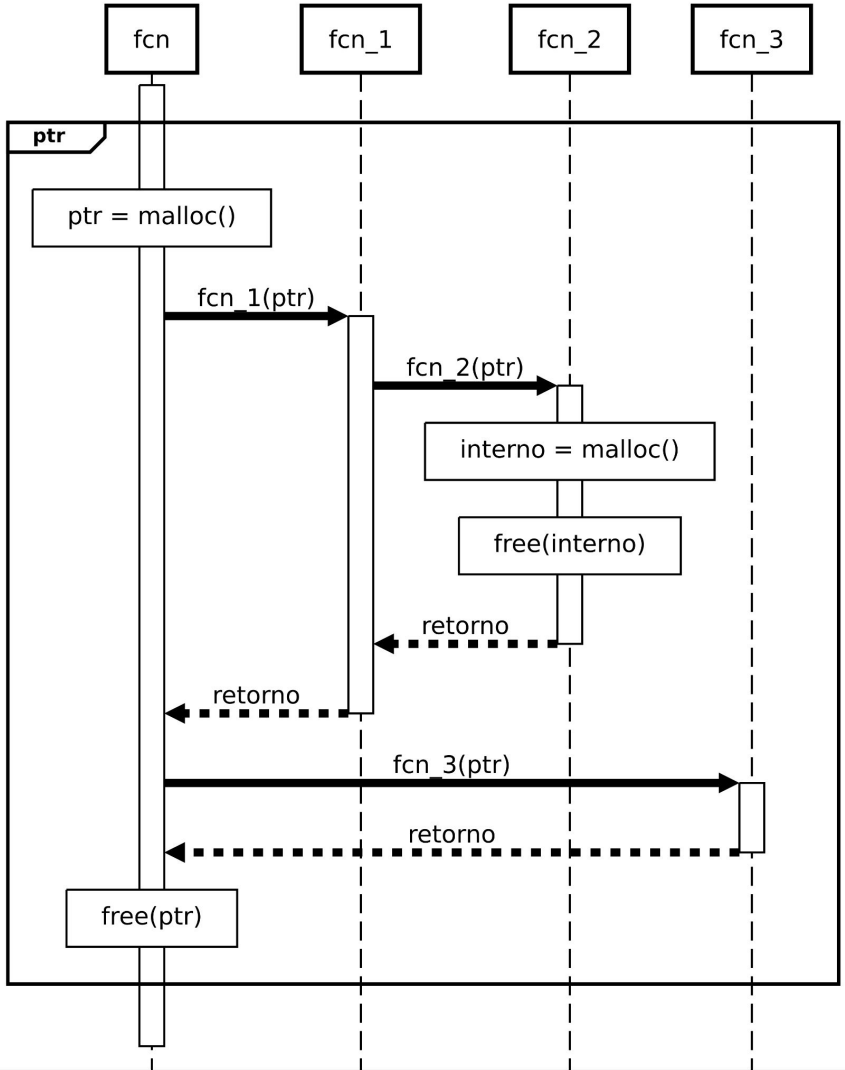
pero esto puede ser un problema

# Si estábamos esperando liberar memoria y *la perdimos* por el camino


¿Preguntas?

# Para hacer esta tarea mas facil

# Pidan y liberen memoria al mismo *nivel**


fcn
fcn_1
fcn_2
fcn_3
ptr
ptr = malloc()
fcn_1(ptr)
fcn_2(ptr)
interno = malloc()
free(interno)
retorno
retorno
fcn_3(ptr)
retorno
free(ptr)

# Esto significa que:

1

8

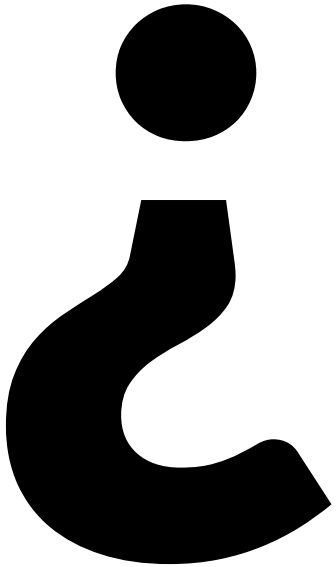
# La memoria se pide en un lugar[1] diferente a donde se usa

19

# La memoria se libera en el mismo lugar[1] donde se pide

# Ser ordenados ayuda a liberar todo lo pedido

# Pero además



# ¿Es posible distinguir memoria dinámica de automática?

UNRN | Universidad Nacional de Río Negro

# Si liberan algo no pedido dentro de una función...

**Tal vez** intenten liberar algo que no se puede liberar

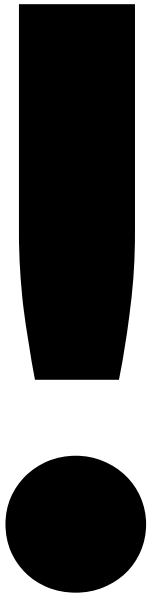
# ¿En qué segmento de la memoria esta cadena?

```
void funcion(char* cadena)
{
    free(cadena);
}
```

# ¿Que pasa acá?

```
int main()
{
    char automatica[] = “hola mundo”;
    funcion(automatica);
}
```

# Comportamiento no definido



# `free` solo puede liberar lo que `malloc` dá

# esto potencialmente rompe la información que usa `malloc`



https://vulncat.fortify.com/en/detail?id=desc.controlflow.cpp.code_correctness_memory_free_on_stack_variable


¿Preguntas?

# También, para mantener todo en orden

¡  !

# con pedir memoria en un lazo

# Que después hay que liberarla

# No es que esté prohibido, pero el potencial para meter la pata es grande


¿Preguntas?

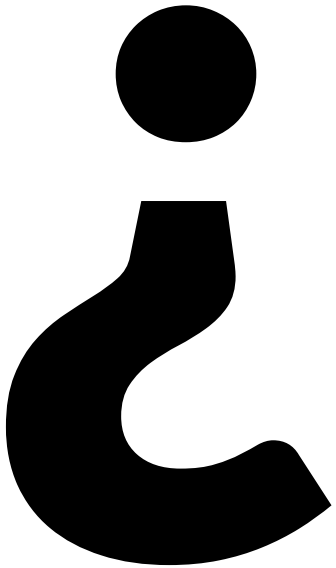
# Aparte del catch and release

# Después se usa como un arreglo

Tio Ben dixit

# Con un gran poder viene una gran responsabilidad


¿Preguntas?



# Pero se dice que es dinámica

# calloc

Para crear un arreglo dinámico y en cero.

# Crea un arreglo de cantidad de tamaño

```
void* calloc( size_t cantidad, size_t tamaño );
```

# Un arreglo dinámico de 4 elementos

```
int* arreglo = (int*) calloc(4, sizeof(int));
```

# realloc

Para cambiar el tamaño de un bloque pedido

# El nuevo tamaño no debe ser cero

```
void* realloc( void* bloque, size_t new_size );
```

# Pasamos de un arreglo de 4 a uno de 10

```
int* arreglo = (int*) calloc(4, sizeof(int));
arreglo = realloc(arreglo, sizeof(int) * 10);
```

# memcpy

Para copiar bloques de memoria

# Retorna dest

```
void* memcpy( void *destino, void* origen, size_t cuenta );
```

# Malloc y copia

```
const int patron[] = {1, 2, 3, 4, 5, 6, 7, 8};
const int largo_patron = sizeof(patron) / sizeof(int);
int *memoria = NULL;

if ((memoria = (int*)malloc(largo_patron * sizeof(int)))){
    memoria = memcpy(memoria, patron, sizeof(patron));
} else {
    exit(2);
}
```


¿Preguntas?

# Mención especial a

# Arreglos de largo variable ALV (VLA)

# Que es un ALV y porqué está prohibido su uso



https://nullprogram.com/blog/2019/10/27/

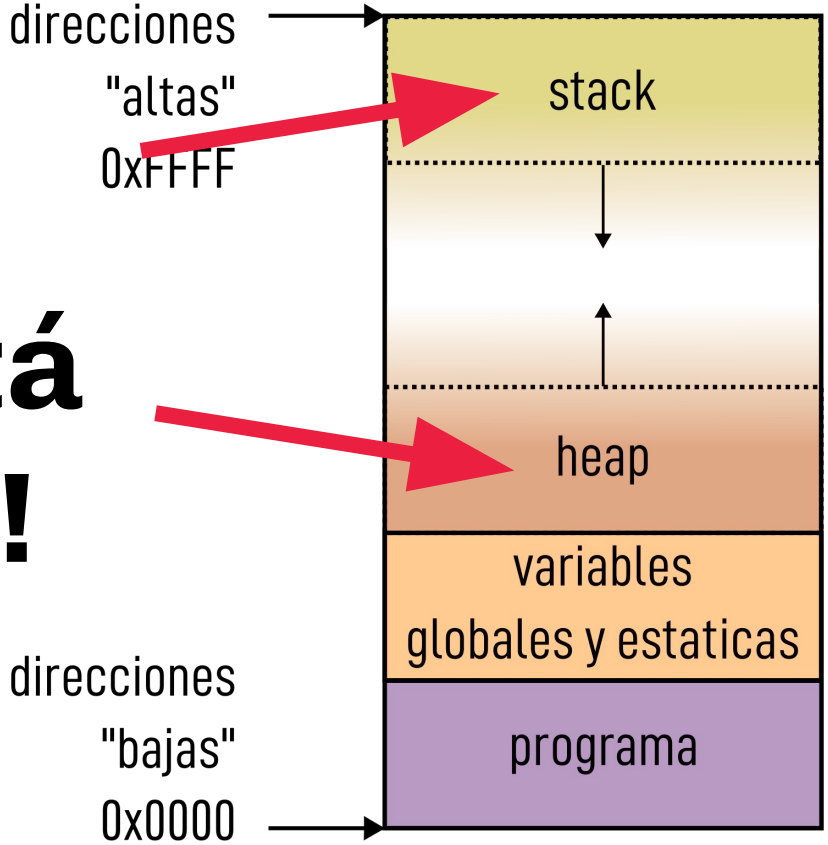
```
void funcion(int cantidad)
{
   int arreglo[cantidad];
   … //resto de la funcion
}
```

# ¿Qué pasa si no hay más memoria?


QUE SIGA LA
FIESTA

# Pero lo peor es que no nos enteramos

# ¡Que esto está superpuesto!



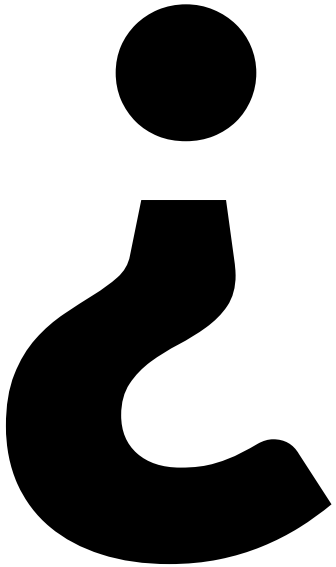
**provocando un stack overflow**

**aparte de que *técnicamente* resulta en un programa más lento**


¿Preguntas?

# Uso de memoria dinámica en funciones


¿Preguntas?



# Cómo se documenta

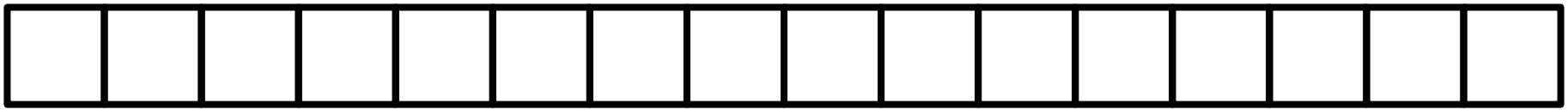

¿Preguntas?

# problemas en lo dinámico

# fragmentación

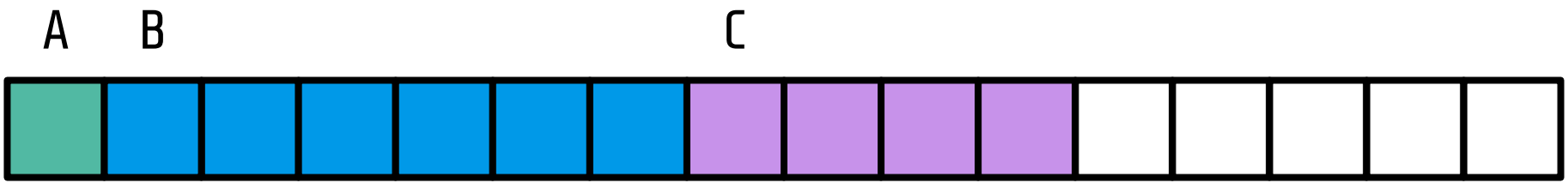
_

16 bloques libres



**Sea esta toda la memoria disponible para `malloc`**

5 bloques libres

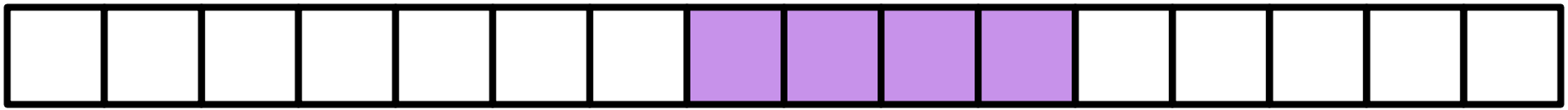



**Hacemos tres pedidos de memoria (A, B y C)**

_

12 bloques libres

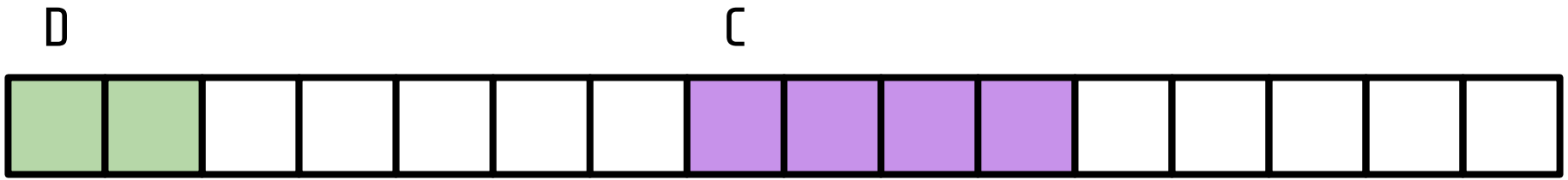
C



## Liberamos A y B

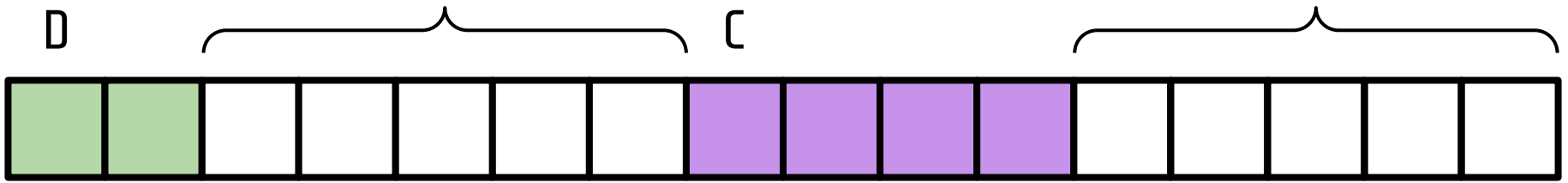
_

10 bloques libres




**Pedimos un bloque D**

10 bloques libres




**Pero solo se puede alojar un tamaño de 5**

# Esto es un problema en programas que deben funcionar mucho tiempo


¿Preguntas?

# uso luego de free

# Después de liberar un bloque de memoria, el puntero *sigue ahí*

# Es lo mismo que retornar un puntero a una variable automática...

# TP5

## Memoria dinámica

# Hasta la próxima

unrn.edu.ar

UNRN Universidad Nacional de Río Negro

| unrionegro