
UNRN
Universidad Nacional
de Río Negro
| unrionegro

Memoria

11

2023

UNRN

Universidad Nacional
de **Río Negro**

Los alcances de las variables (scoping)

Global

Las variables declaradas fuera de cualquier función tienen alcance global. Esto significa que son accesibles desde cualquier parte del programa.

¿Donde se declaran?

```
int variable_global = 10;

int main() {
  printf("La variable global es %d\n", variable_global);

  return 0;
}

void procedimiento(){
  printf("La variable_global existe acá %d\n", variable_local);
}
```


"We don't do that here"

Local

Las que venimos utilizando;

Las variables declaradas dentro de una función tienen alcance local. Esto significa que solo son accesibles desde dentro de la función en la que se declaran.

```
int main() {
  int variable_local = 20;
  printf("La variable local vale %d\n", variable_local);
  return 0;
}

void procedimiento(){
  printf("La variable_local no existe acá %d\n", variable_local);
}
```

Bloques { }

Las variables declaradas dentro de un bloque tienen alcance de bloque. Esto significa que solo son accesibles desde dentro del bloque en el que se declaran.

Las llaves pueden ir ‘sueltas’ :-)

```
int main()
{
  {
    int variable_bloque = 30;

    // La variable variable_bloque solo es accesible desde dentro del bloque
    printf("El valor de la variable de bloque es %d\n", variable_bloque);
  }

  // La variable variable_bloque no es accesible fuera del bloque
  printf("El valor de la variable de bloque es %d\n", variable_bloque);

  return 0;
}
```

Estructura (`struct`)

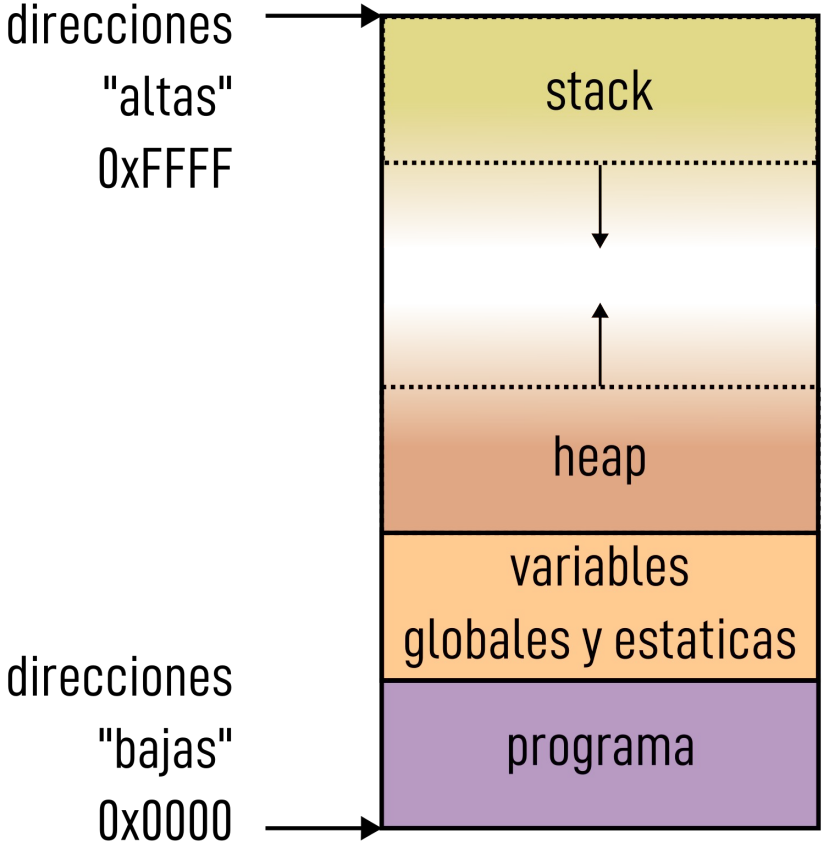
Los atributos declarados dentro de una estructura tienen alcance de estructura. Esto significa que son accesibles desde cualquier parte de la estructura, incluidas las funciones que están definidas dentro de la estructura.

```
struct persona {
  int edad;
};

void imprimir_edad(struct persona *p) {
    printf("La edad de la persona es %d\n", p->edad);
}
```

```
int main() {
  struct persona p1;
  struct persona p2;
  p1.edad = 40;
  p2.edad = 19;
  printf("La edad de la persona es %d\n", p.edad);
  imprimir_edad(&p);

  return 0;
}
```


¿Preguntas?

Las regiones de memoria



Automática

Dinámica

Estática

Automática

Se asigna al inicio de cada función y se libera con su final. Esta región se utiliza para almacenar variables locales.

Estática

Se asigna al inicio del programa y se libera con su finalización. Esta región se utiliza para almacenar variables globales y constantes.

Dinámica

Se asigna y libera durante la ejecución del programa. Se utiliza para almacenar datos que no se conocen de antemano, como el tamaño de una matriz o la longitud de una cadena de caracteres.

Memoria Dinámica de heap

¿Por qué?

¿es necesaria?

Como se usa

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

El bloque pedido

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

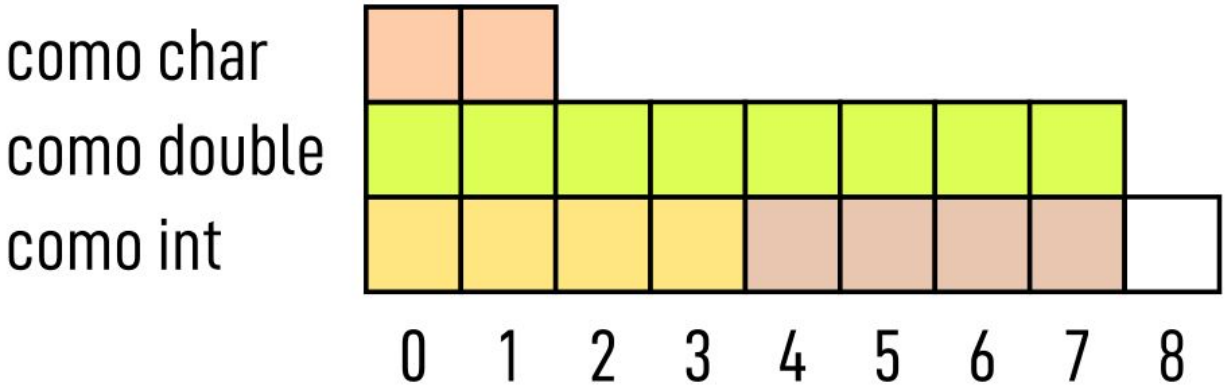
¿lo qué?

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

`void*`

Solo la ubicación en memoria sin noción de ‘ancho’

¿Ancho? (en memoria)



*no a escala

`void*` no contiene información sobre el ancho

No es válido un arreglo de `void*`

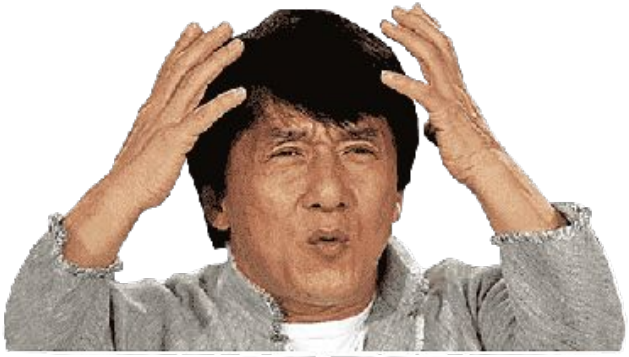
```
void arreglo[10];
```

Tampoco es válido

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

Pero entonces

¿¿Pero como se usa??



¿Cómo indicamos el ancho?

Con ejemplo simple

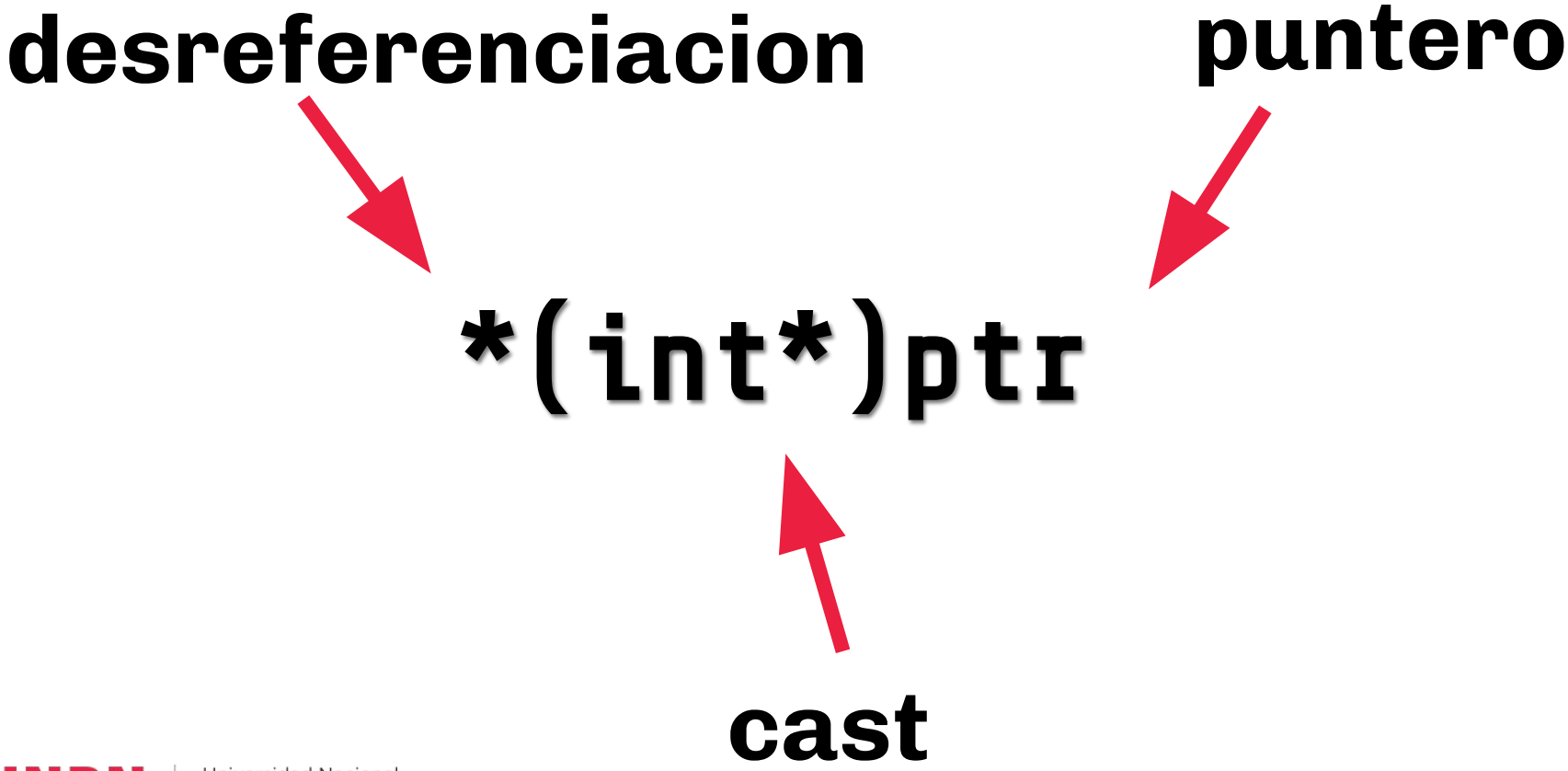
```
int numero = 10;
void* ptr = &numero;
```

¿podemos desreferenciar `ptr`?

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

```c
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);
```

O todo junto

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

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


¿Preguntas?

Volviendo al `malloc`

Cómo usarlo para un arreglo

un ejemplo de uso (para un valor individual)

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

cast

el tamaño individual

un ejemplo de uso

que podemos multiplicar

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

cast

el tamaño individual

Después, a usarlo como un arreglo más


¿Preguntas?

¿Puede fallar?

¡Si!

`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
```


¿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

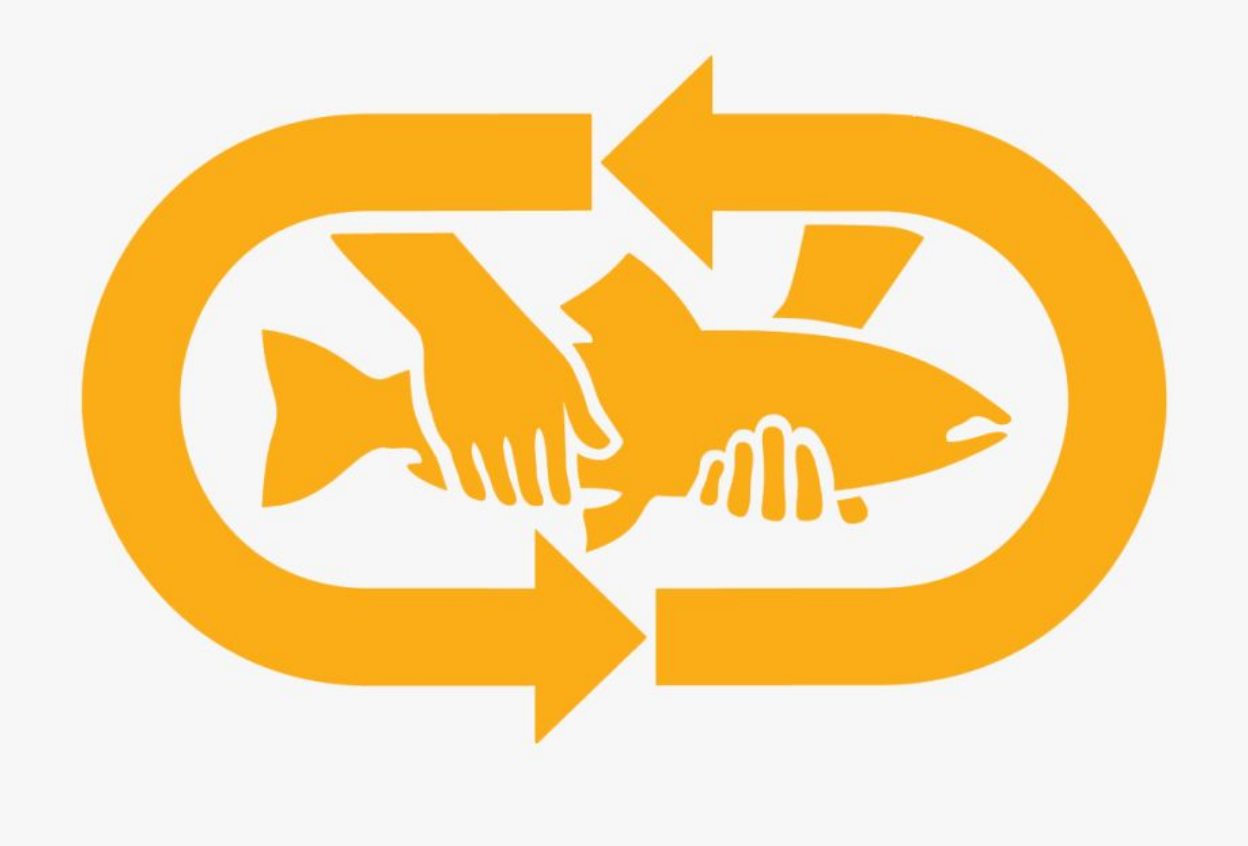
Liberar la memoria

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

acepta el puntero ‘casteado’

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

Catch and release!



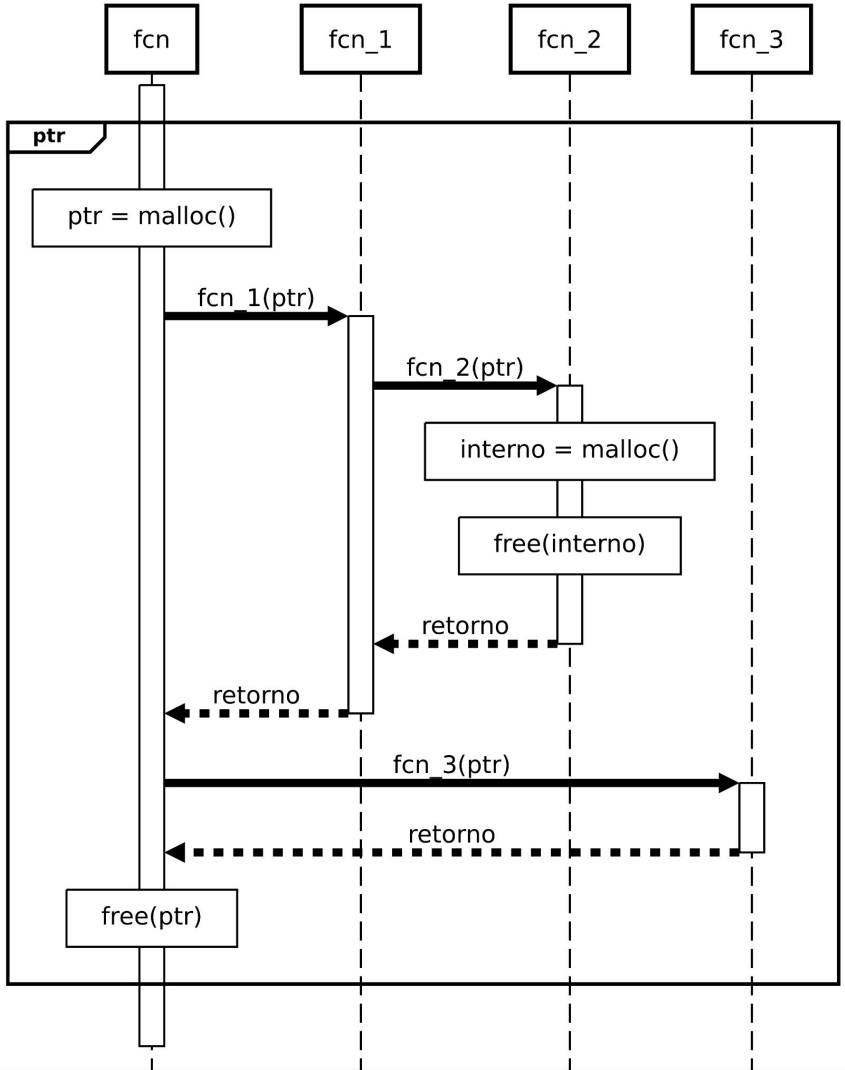
Completando el ejemplo

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


¿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)

Ser ordenados ayuda a liberar todo lo pedido


¿Preguntas?

También, para mantener todo en orden

¡👁️!

Ojo 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

Con un gran poder viene una gran responsabilidad

Tio Ben dixit


¿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/

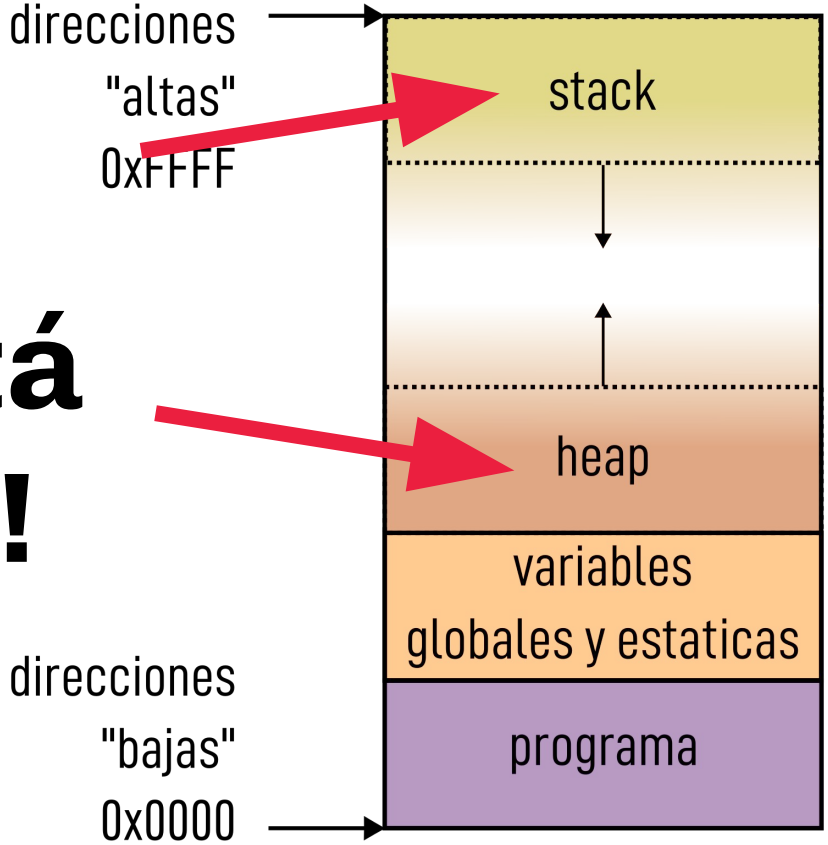
```
void funcion(int cantidad)
{
  int arreglo[cantidad];
  … //resto de la funcion
}
```

¿Qué pasa si no hay más memoria?


QUE SIGA LA
FIESTA

Pero lo peor es que no nos enteramos

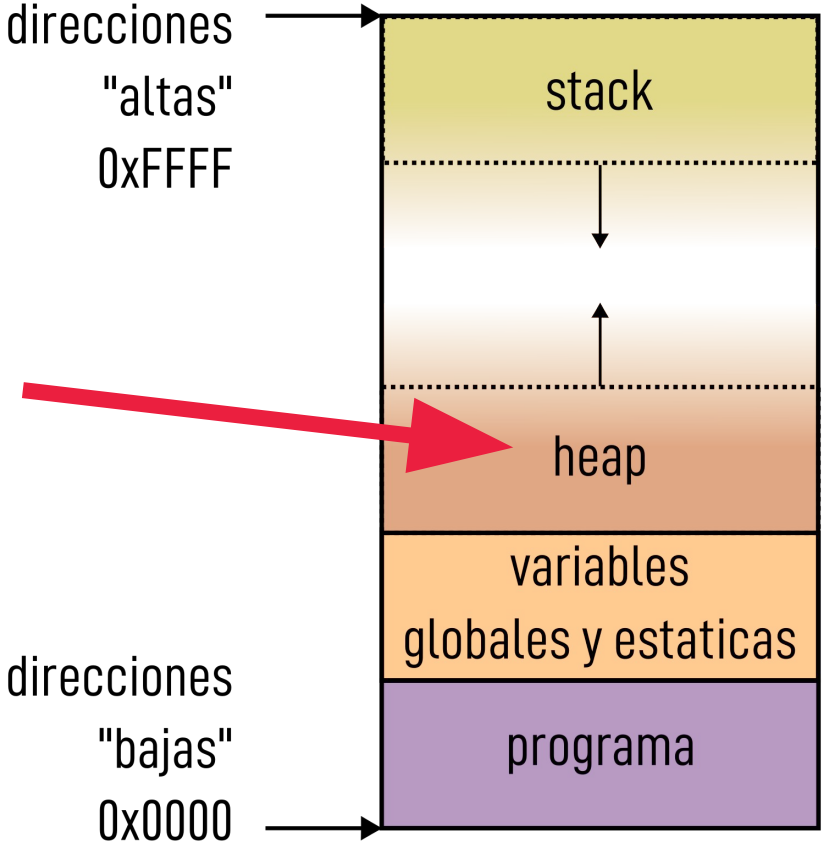
¡Que esto está superpuesto!



provocando un stack overflow

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


¿Preguntas?


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

Estructuras y punteros II

```
typedef struct nodo{
    int valor;
    struct nodo* next;
}nodo_t;
```

```
#define MAX 10
nodo_t repositorio[MAX];
```

```
typedef struct{
    short estado[MAX];
    nodo_t nodos[MAX];
}repositorio_t;
```

Punteros II

genéricamente hablando

void solo es nada

pero

void* es cualquier cosa

char*

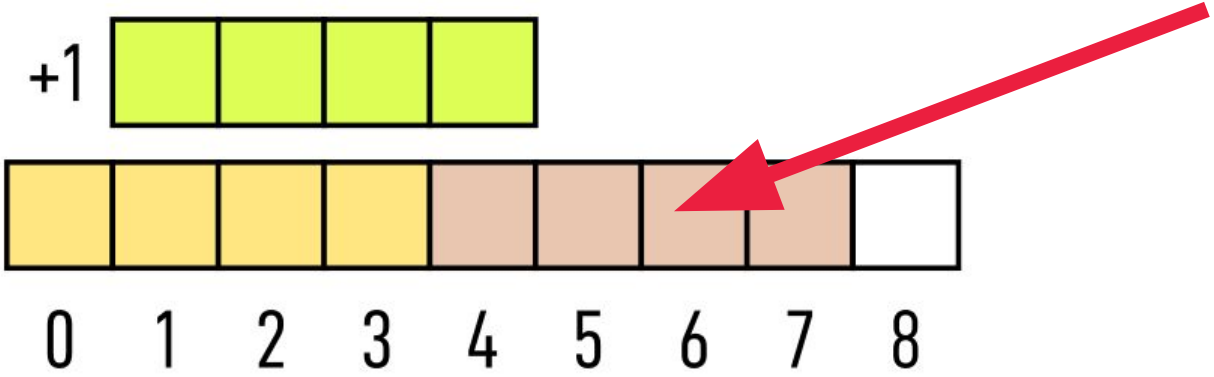
void* ≠ int*

double*

void* no tiene información del “ancho”

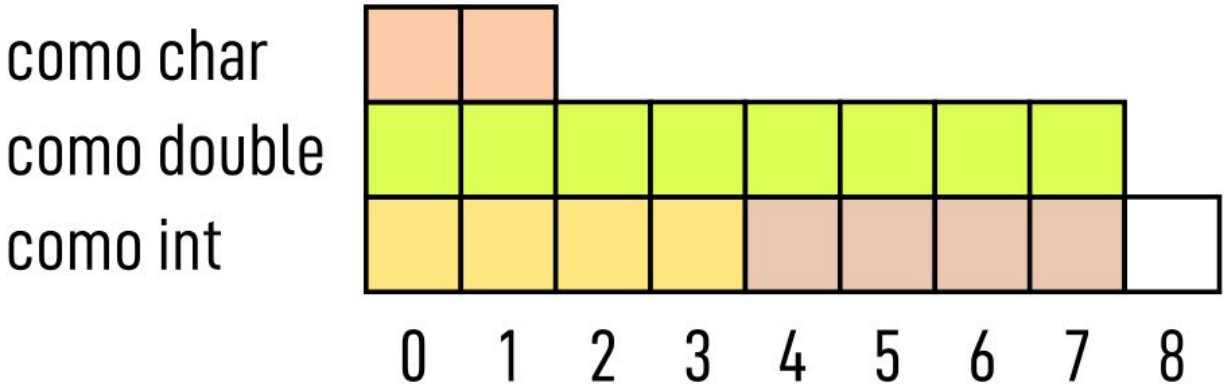
1. No es posible hacer aritmética de punteros

Corre peligro la alineación



2. No hay que des-referenciar

Desreferenciando



Afortunadamente

Son las conversiones de tipo

Con la conversión podemos des referenciar y hacer aritmética


¿Preguntas?

unrn.edu.ar

UNRN Universidad Nacional de Río Negro

| unrionegro