Práctica de Arquitectura y Organización de Computadores

Introducción a C

Segundo Cuatrimestre 2024

Arquitectura y Organización de Computadores
DC - UBA

Introducción

¿Por qué C?

Características de C:

- C es de bajo nivel, ideal para *system programming*.

¿Por qué C?

Características de C:

- C es de bajo nivel, ideal para *system programming*.
- C es **pequeño** en comparación con otros lenguajes de programación.

¿Por qué C?

Características de C:

- C es de bajo nivel, ideal para *system programming*.
- C es **pequeño** en comparación con otros lenguajes de programación.
- C es permisivo. Asume que el programador sabe lo que está haciendo.

¿Por qué C?

Puntos fuertes:

- **Eficiente**. C fue pensado para reemplazar assembly.

¿Por qué C?

Puntos fuertes:

- **Eficiente**. C fue pensado para reemplazar assembly.
- Ampliamente utilizado en sistemas operativos, compiladores, intérpretes, etc.

¿Por qué C?

Puntos fuertes:

- **Eficiente**. C fue pensado para reemplazar assembly.
- Ampliamente utilizado en sistemas operativos, compiladores, intérpretes, etc.
- Ampliamente utilizado en sistemas embebidos o de bajos recursos.

¿Por qué C?

Puntos fuertes:

- **Eficiente**. C fue pensado para reemplazar assembly.
- Ampliamente utilizado en sistemas operativos, compiladores, intérpretes, etc.
- Ampliamente utilizado en sistemas embebidos o de bajos recursos.
- Permite manipulación directa de **memoria**.

¿Por qué C?

Puntos débiles:

- Los programas en C son más **propensos a errores**.

¿Por qué C?

Puntos débiles:

- Los programas en C son más **propensos a errores**.
- Los programas en C pueden ser más difíciles de entender

¿Por qué C?

Puntos débiles:

- Los programas en C son más **propensos a errores**.
- Los programas en C pueden ser más difíciles de entender
- Los programas en C pueden ser más difíciles de mantener.

Hola Orga!


```c
#include <stdio.h>              // Incluye la biblioteca stdio.h

int main() {                    // main es el punto de entrada
    printf("Hola Orga!\n");     // imprime en pantalla
    return 0;                   // devuelve un 0 al SO
}
```

Compilando y ejecutando

Compilando

```
$ gcc -c hola.c -o hola.o
$ gcc hola.o -o hola
```

Compilando y ejecutando

Compilando

```
$ gcc -c hola.c -o hola.o
$ gcc hola.o -o hola
```

O también se puede hacer:

```
$ gcc hola.c -o hola
```

Compilando y ejecutando

Compilando

```
$ gcc -c hola.c -o hola.o
$ gcc hola.o -o hola
```

O también se puede hacer:

```
$ gcc hola.c -o hola
```

Ejecutando

```
$ ./hola
Hello, World!
```

Tipos de datos:

- **Enteros:** char, short, int, long.
- **Reales:** float, double.
- **Caracteres:** char.
- **Booleanos:** bool.[1]

[1] C99: Hay que incluir `<stdbool.h>`

Enteros (tamaños típicos en 64 bits)

Tipo	Bytes	Rango
char	1	-128 a 127 -2^7 a $2^7 - 1$
unsigned char	1	0 a 255 0 a $2^8 - 1$
short	2	-32768 a 32767 -2^{15} a $2^{15} - 1$
unsigned short	2	0 a 65535 0 a $2^{16} - 1$
int	4	-2147483648 a 2147483647 -2^{31} a $2^{31} - 1$
unsigned	4	0 a 4294967295 0 a $2^{32} - 1$
long	8	$-9{,}2 \times 10^{18}$ a $9{,}2 \times 10^{18}$ -2^{63} a $2^{63} - 1$
unsigned long	8	0 a $1{,}84 \times 10^{19}$ 0 a $2^{64} - 1$

Enteros


```c
#include <stdio.h>

int main() {
    char c = 100;
    short s = -8712;
    int i = 123456;
    long l = 1234567890;

    printf("char(%lu):  %d \n", sizeof(c),c);
    printf("short(%lu): %d \n", sizeof(s),s);
    printf("int(%lu):   %d \n", sizeof(i),i);
    printf("long(%lu):  %ld \n", sizeof(l),l);

    return 0;
}
```

Enteros


```
char(1):  100
short(2): -8712
int(4):   123456
long(8):  1234567890
```

`sizeof()` devuelve el tamaño en bytes de un tipo de dato.

Enteros

Para no tener problemas con el tamaño de los enteros, se pueden usar los tipos de datos definidos en `stdint.h`:

- `int8_t`
- `int16_t`
- `int32_t`
- `int64_t`
- `uint8_t`
- `uint16_t`
- `uint32_t`
- `uint64_t`

Operadores

operador	tipo
++, --	incremento, decremento
+, -, *, /, %	aritméticos
<, <=, >, >=, ==, !=	comparación
&&, ||, !	lógicos
&, <<, >>, |, ~, ^	binarios
=, +=, -=, *=, /=, %=	asignación
?:	condicional

Operadores

Algunas cosas a tener en cuenta:

- `++i` incrementa `i` y luego devuelve el valor de `i`.

Operadores

Algunas cosas a tener en cuenta:

- `++i` incrementa `i` y luego devuelve el valor de `i`.
- `i++` devuelve el valor de `i` y luego incrementa `i`.

Operadores

Algunas cosas a tener en cuenta:

- `++i` incrementa `i` y luego devuelve el valor de `i`.
- `i++` devuelve el valor de `i` y luego incrementa `i`.
- `a = b = c = 0` asigna 0 a `a`, `b` y `c`.

Operadores

Algunas cosas a tener en cuenta:

- `++i` incrementa `i` y luego devuelve el valor de `i`.
- `i++` devuelve el valor de `i` y luego incrementa `i`.
- `a = b = c = 0` asigna 0 a `a`, `b` y `c`.
- `a = b = c` asigna el valor de `c` a `a` y `b`.

Operadores

Algunas cosas a tener en cuenta:

- `++i` incrementa `i` y luego devuelve el valor de `i`.
- `i++` devuelve el valor de `i` y luego incrementa `i`.
- `a = b = c = 0` asigna 0 a `a`, `b` y `c`.
- `a = b = c` asigna el valor de `c` a `a` y `b`.
- los operadores de comparación devuelven 1 si la condición es verdadera y 0 si es falsa.

Operadores

Algunas cosas a tener en cuenta:

- `++i` incrementa `i` y luego devuelve el valor de `i`.
- `i++` devuelve el valor de `i` y luego incrementa `i`.
- `a = b = c = 0` asigna 0 a `a`, `b` y `c`.
- `a = b = c` asigna el valor de `c` a `a` y `b`.
- los operadores de comparación devuelven 1 si la condición es verdadera y 0 si es falsa.
- un entero es considerado verdadero si es distinto de 0.

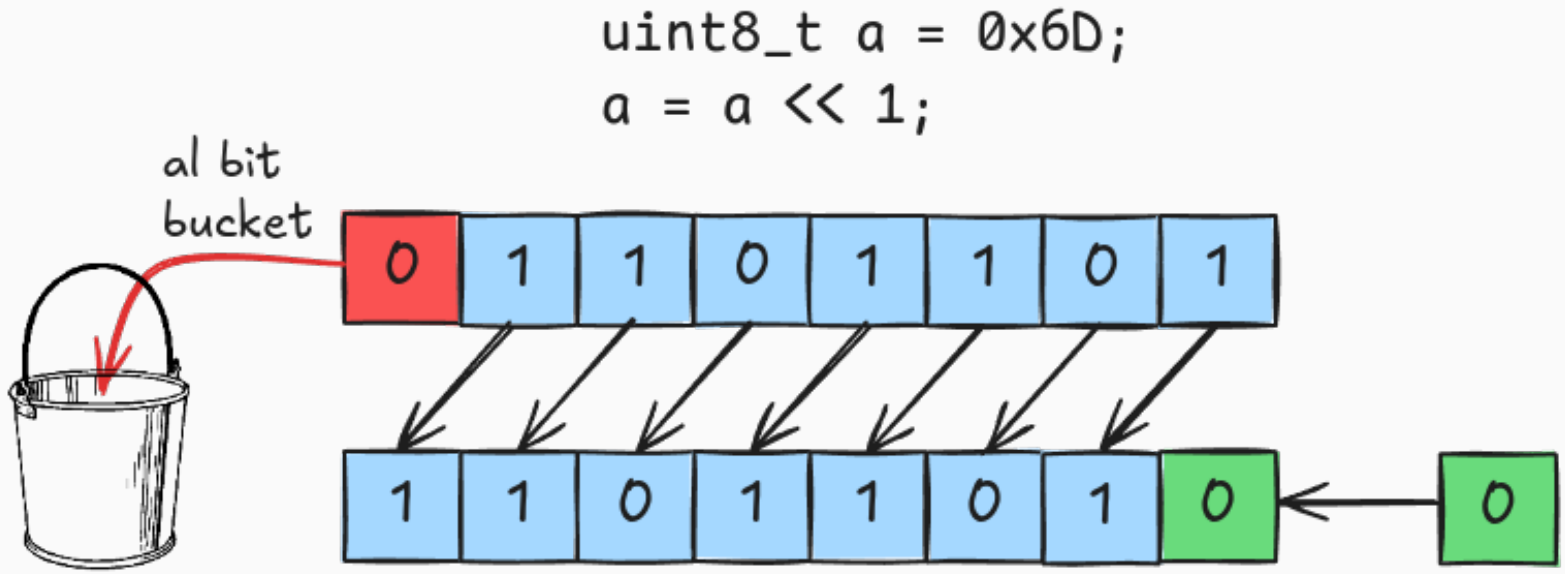
pre y post incremento

```c
#include <stdio.h>

int main() {
    int a = 0;

    a = 1;
    printf("a = %d\n",a);        // a = 1
    printf("a++ = %d\n",a++);    // a++ = 1
    printf("++a = %d\n", ++a);   // ++a = 3
    printf("a = %d\n",a);        // a = 3

    return 0;
}
```

Entero como boolean

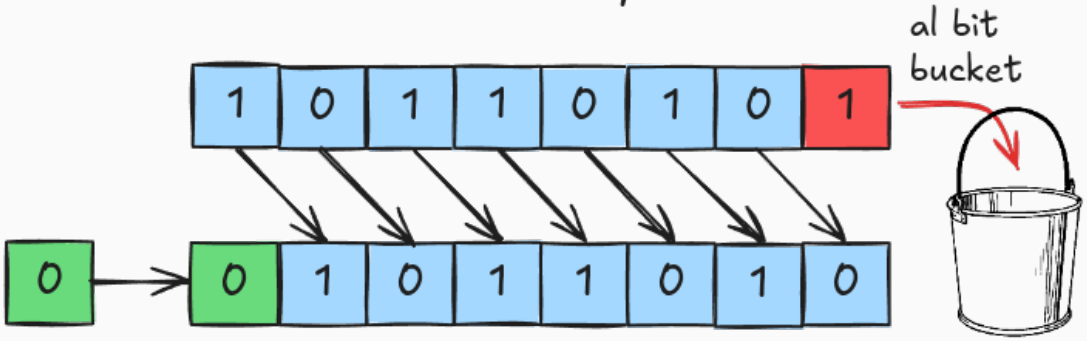
```c
#include <stdio.h>

int main() {
    int i = 10;

    while(i--){
        printf("i = %d\n",i); // imprime o no el 0?
    }
}
```

Shifting left

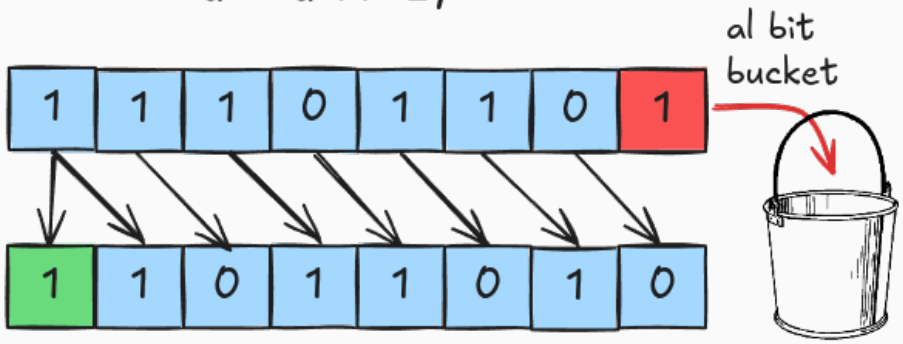


Shifting right

```
uint8_t a = 0xB5;
a = a >> 1;
```




```
int8_t a = 0xED;
a = a >> 1;
```



En general sucede esto. Pero en realidad es implementación dependiente.

Operadores

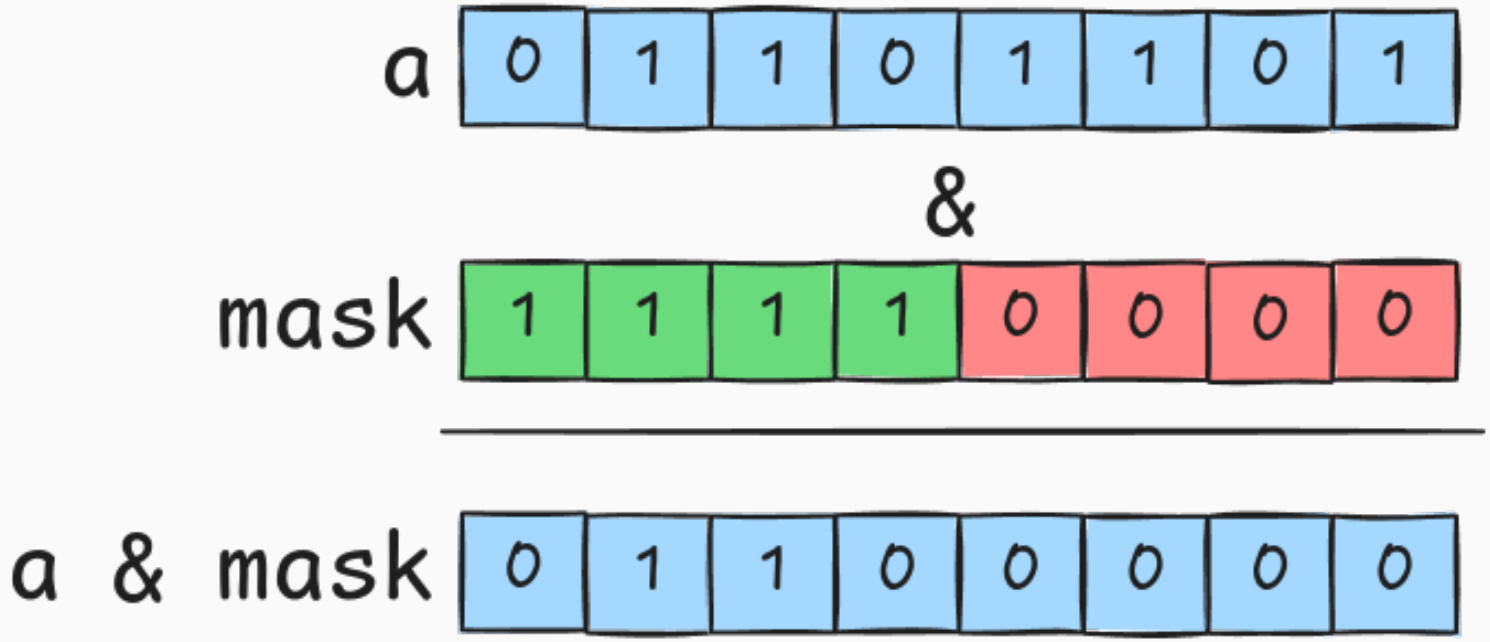
$a << b$, $a >> b$

- b no puede ser mayor que el tamaño del tipo de dato de a.
- b no puede ser negativo.

```c
#include <stdio.h>

int main() {
    uint32_t a = 50;

    a = a >> 32; // comportamiento indefinido
    a = a << -1; // comportamiento indefinido
}
```

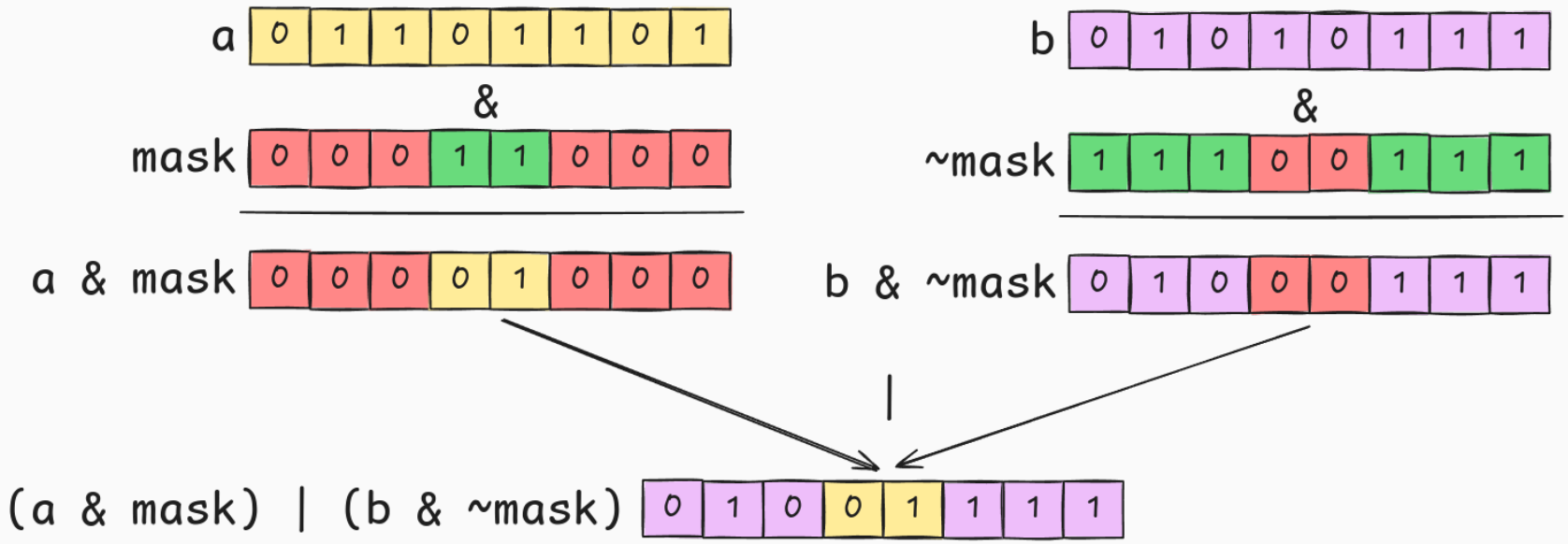
Máscaras

```
uint8_t a = 0x6D;
uint8_t mask = 0xF0;
a &= mask;
```



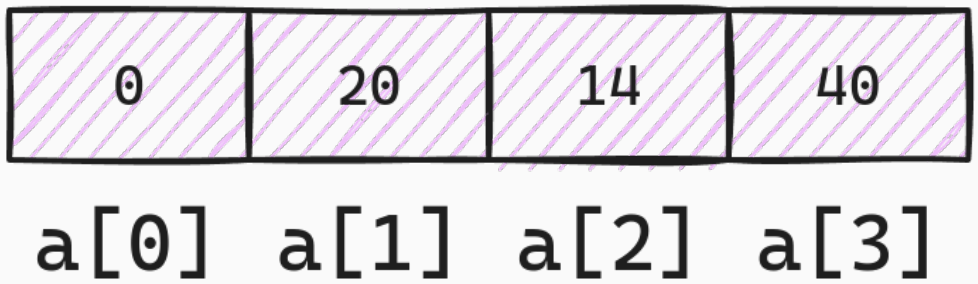
Máscaras

```
uint8_t a = 0x6D;
uint8_t b = 0x57;
uint8_t mask = 0x18;
uint8_t mix_ab = (a & mask) | (b & ~mask);
```



Arrays


```
1  #define N 4
2
3  uint32_t a[N];
4  a[0] = 0;
5  a[1] = 20;
6  a[2] = 14;
7  a[3] = 40;
```



Arrays


```
1  #define N 100
2
3  uint32_t a[N];
4  int i = 0;
5  while(i < N){
6      a[i] = i;
7      i++;
8  }
```

Arrays


```
1  #define N 100
2
3  uint32_t a[N];
4  int i = 0;
5  while(i < N){
6      a[i] = i;
7      i++;
8  }
```

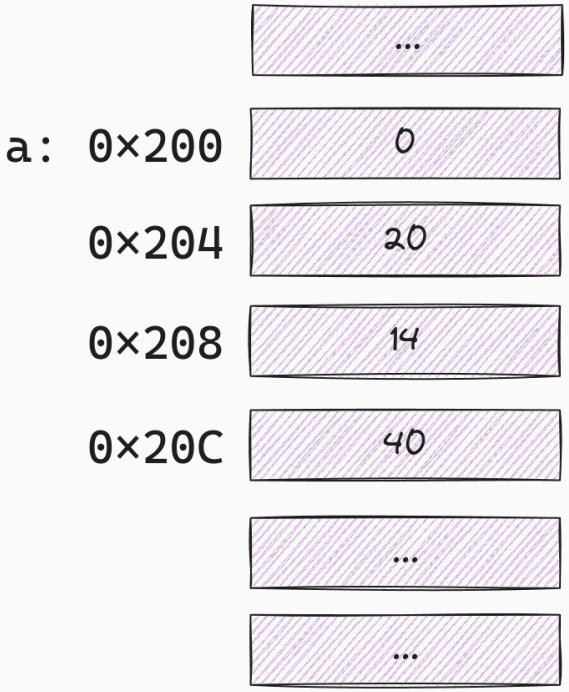
```
1  #define N 100
2
3  uint32_t a[N];
4  for (int i = 0; i < N; i++){
5      a[i] = i;
6  }
7
```

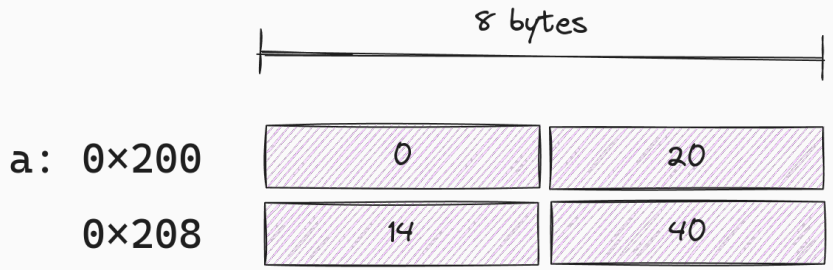
Definiciones alternativas

```
1  #define N 4
2  uint32_t a[N] = {0, 20, 14, 40};
3  uint32_t b[] = {0, 20, 14, 40};
4  uint32_t c[N] = {0}; // c es {0, 0, 0, 0}
5  uint32_t d[] = {[1] = 20, [2] = 14, [3] = 40};
6  uint32_t e[N];
7
8  for (int i = 0; i < N; i++){
9      e[i] = 0;
10 }
```

En memoria

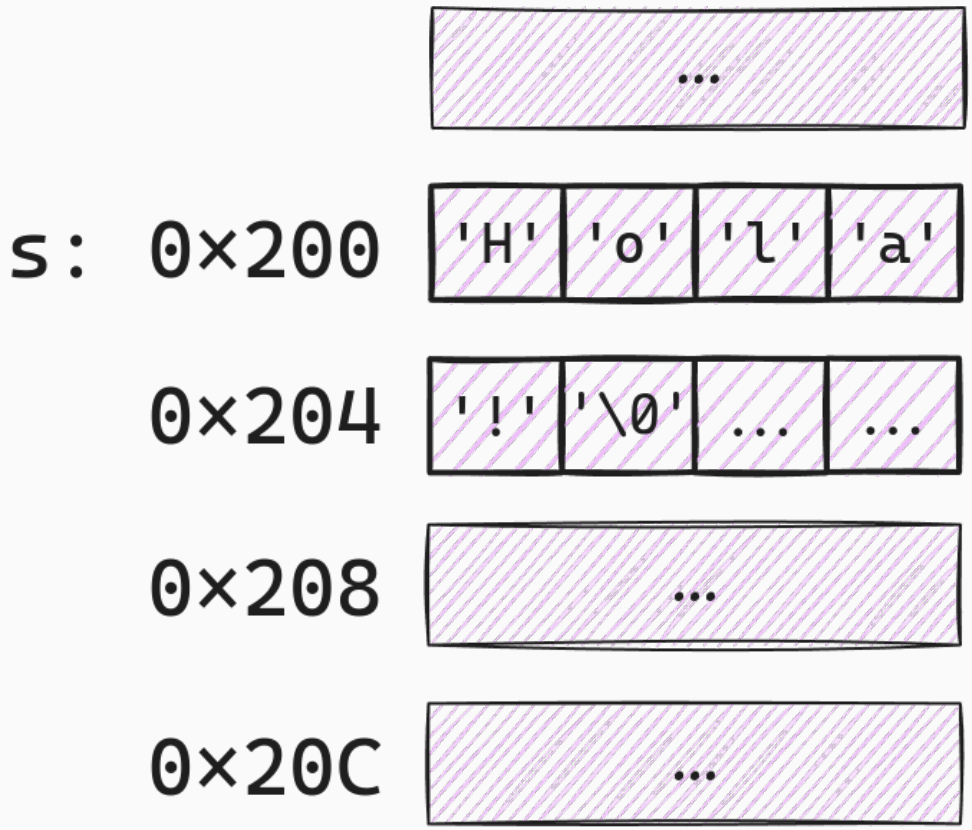
```
uint32_t a[] = {0,20,14,40};
```





Strings


```
1  char s[] = "Hola!";     // string literal
2  char u[] = {'H', 'o', 'l', 'a', '!', '\0'}; //char literals
```



```c
int main(){
    char s[] = "Hola!";  // s es el nombre del array
    char *u =  "string"; // u es un puntero a un char

    printf("s = %s\n", s);
    printf("u = %s\n", u);
    s[0] = 'h'; // s = "hola!"
    u[0] = 'S'; // ERROR
}
```

```
$ gcc -Wall -Wextra -pedantic strings.c   -o strings
$ ./strings
s = Hola!
u = string
[1]    65377 segmentation fault (core dumped)  ./strings
```

```c
#include <stdio.h>

int suma(int a, int b){ // definición de la función
    return a + b;
}

int main(){
    int a = 10, b = 20;

    int c = suma(a,b);
    printf("suma = %d\n", c);
}
```

Funciones

```c
#include <stdio.h>

int suma(int a, int b); // declaración de la función

int main(){
    int a = 10, b = 20;

    printf("suma = %d\n", suma(a,b));
}

int suma(int a, int b){ // definición de la función
    return a + b;
}
```

Scope y Duración

Scope y Duración son 2 conceptos completamente distintos:

- **Scope**: se refiere al lugar en el código donde un identificador puede ser accedido. Los scopes pueden ser:
  - **block scope**
  - **file scope**

Scope y Duración

Scope y Duración son 2 conceptos completamente distintos:

- **Scope**: se refiere al lugar en el código donde un identificador puede ser accedido. Los scopes pueden ser:
  - **block scope**
  - **file scope**
- **Duración**: es el tiempo durante el cual el objeto en cuestión existe. Puede ser:
  - **estático**
  - **dinámico**
  - **automático**

Scope y Duración


```c
#include <stdio.h>
#define FELIZ 0
#define TRISTE 1

void ser_feliz(int estado);
void print_estado(int estado);

int main(){
    int estado = TRISTE;  // automatic duration. Block scope
    ser_feliz(estado);
    print_estado(estado); // qué imprime?
}

void ser_feliz(int estado){
    estado = FELIZ;
}

void print_estado(int estado){
    printf("Estoy %s\n", estado == FELIZ ? "feliz" : "triste");
}
```

Scope y Duración


```c
#include <stdio.h>
#define FELIZ 0
#define TRISTE 1
int estado = TRISTE; // static duration. File scope

void ser_feliz();
void print_estado();

int main(){
    print_estado();
    ser_feliz();
    print_estado(); // qué imprime?
}

void ser_feliz(){
    estado = FELIZ;
}

void print_estado(){
    printf("Estoy %s\n", estado == FELIZ ? "feliz" : "triste");
}
```

Funciones


```c
#include <stdio.h>
#define FELIZ 0
#define TRISTE 1
int estado = TRISTE; // static duration. File scope

void alcoholizar();
void print_estado();

int main(){
    print_estado();
    alcoholizar();
    print_estado();
    alcoholizar();alcoholizar();alcoholizar();
    print_estado();  // que imprime?
}
void alcoholizar(){
    static int cantidad = 0; // static duration. block scope
    cantidad++;
    if(cantidad < 3){
        estado = FELIZ;
    }else{
        estado = TRISTE;
    }
}
void print_estado(){
    printf("Estoy %s\n", estado == FELIZ ? "feliz" : "triste");
}
```

Veamos la diferencia entre **declaración** y **definición**

- **Declaración**: le dice al compilador que cierto símbolo existe y permite referenciarlo.

Veamos la diferencia entre **declaración** y **definición**

- **Declaración**: le dice al compilador que cierto símbolo existe y permite referenciarlo.
- **Definición**: Implementa el símbolo, es decir, reserva memoria para el mismo.

Funciones


```
return-type function-name (parameters){
        declarations
        statements
}
```

- Las funciones deben ser declaradas antes de ser usadas.

Funciones


```
return-type function-name (parameters){
        declarations
        statements
}
```

- Las funciones deben ser declaradas antes de ser usadas.
- Puede haber muchas declaraciones de una función, pero solo una definición.

Funciones


```
return-type function-name (parameters){
        declarations
        statements
}
```

- Las funciones deben ser declaradas antes de ser usadas.
- Puede haber muchas declaraciones de una función, pero solo una definición.
- Las funciones pueden ser definidas en otro archivo

Funciones


```
return-type function-name (parameters){
        declarations
        statements
}
```

- Las funciones deben ser declaradas antes de ser usadas.
- Puede haber muchas declaraciones de una función, pero solo una definición.
- Las funciones pueden ser definidas en otro archivo
- Los parámetros de una función son pasados por valor. **Siempre**.

Funciones


```
return-type function-name (parameters){
        declarations
        statements
}
```

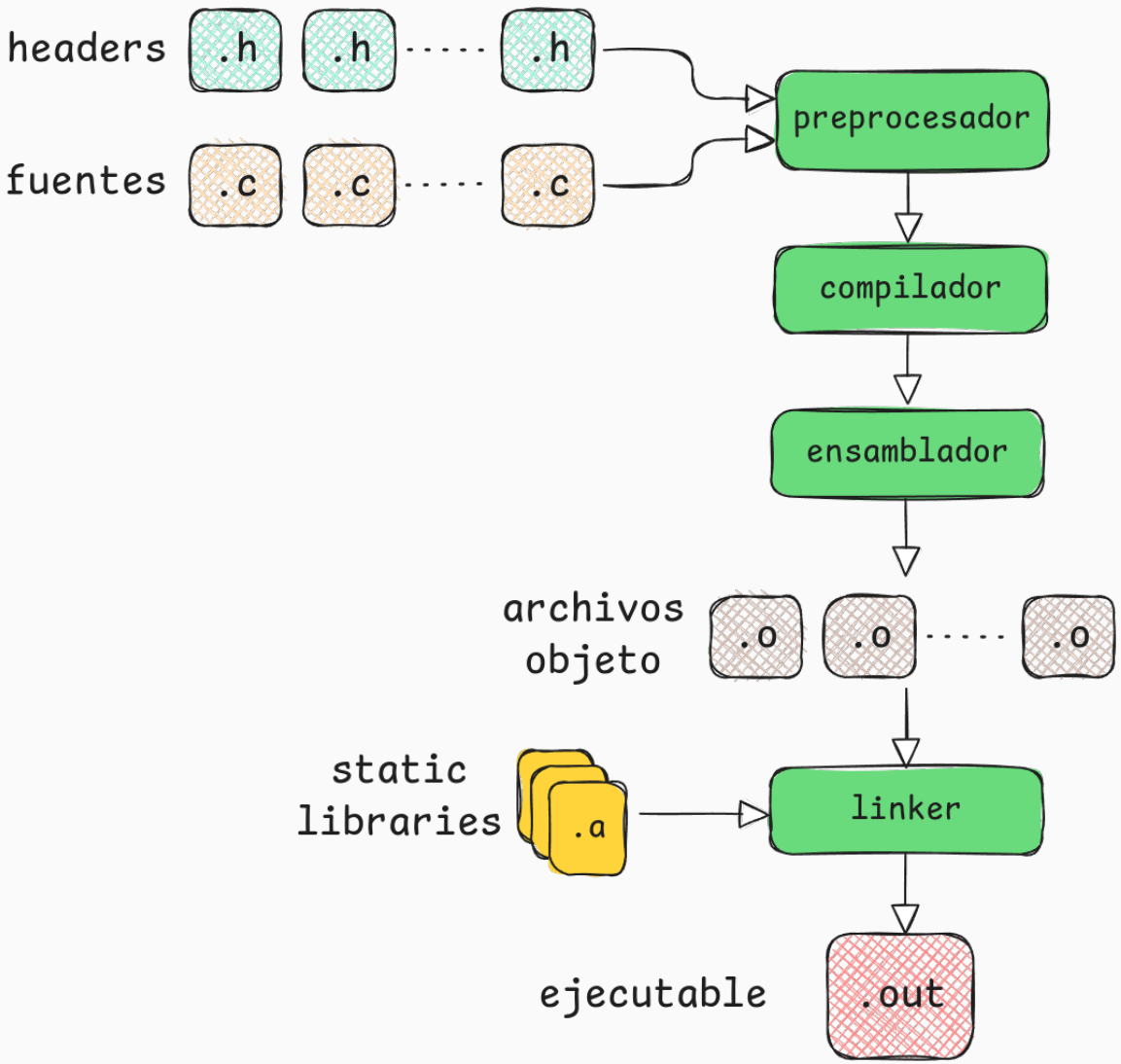
- Las funciones deben ser declaradas antes de ser usadas.
- Puede haber muchas declaraciones de una función, pero solo una definición.
- Las funciones pueden ser definidas en otro archivo
- Los parámetros de una función son pasados por valor. **Siempre**.
- Las funciones pueden devolver un valor. Si no devuelven nada, se usa `void`.

Funciones


```
return-type function-name (parameters){
        declarations
        statements
}
```

- Las funciones deben ser declaradas antes de ser usadas.
- Puede haber muchas declaraciones de una función, pero solo una definición.
- Las funciones pueden ser definidas en otro archivo
- Los parámetros de una función son pasados por valor. **Siempre**.
- Las funciones pueden devolver un valor. Si no devuelven nada, se usa `void`.
- No se pueden devolver arrays.

Compilación



Header files


```c
/* funca.h */
#ifndef FUNCA_H
#define FUNCA_H

void a();

#endif // FUNCA_H
```

```c
/* funca.c */
#include "funca.h"
#include <stdio.h>

void a(){
    printf("Hola, soy A!\n");
}
```

Header files


```c
/* funca.h */
#ifndef FUNCA_H
#define FUNCA_H

void a();

#endif // FUNCA_H
```

```c
/* main.c */
#include "funca.h"

int main() {
    a();
}
```

```c
/* funca.c */
#include "funca.h"
#include <stdio.h>

void a(){
    printf("Hola, soy A!\n");
}
```

```
$ gcc -c funca.c -o funca.o
$ gcc -c main.c -o main.o
$ gcc funca.o main.o -o bin
$ ./bin
Hola, yo soy A!
```

Header files. Otro ejemplo


```c
/* funca.h */
#ifndef FUNCA_H
#define FUNCA_H

void a();

#endif // FUNCA_H
```

```c
/* funca.c */
#include "funca.h"
#include <stdio.h>

void a(){
    printf("Hola, soy A!\n");
}
```

Header files. Otro ejemplo


```c
/* funca.h */
#ifndef FUNCA_H
#define FUNCA_H

void a();

#endif // FUNCA_H
```

```c
/* funcb.h */
#ifndef FUNCB_H
#define FUNCB_H

void b();

#endif // FUNCB_H
```

```c
/* funca.c */
#include "funca.h"
#include <stdio.h>

void a(){
    printf("Hola, soy A!\n");
}
```

```c
/* funcb.c */
#include "funcb.h"
#include <stdio.h>

void b(){
    printf("Hola, soy B!\n");
}
```

Header files. Otro ejemplo (cont.)


```c
/* main.c */
#include "funca.h"
#include "funcb.h"
int main() {
    a();
    b();
}
```

```
$ gcc -Wall -c funca.c -o funca.o
$ gcc -Wall -c funcb.c -o funcb.o
$ gcc -Wall -c main.c -o main.o
$ gcc -Wall funca.o funcb.o main.o -o binario
$ ./binario
Hola, soy A!
Hola, soy B!
```

Y con las variables? Cómo se hace?

```h
// file1.h
#ifndef FILE1_H
#define FILE1_H

int count;
void print_count();

#endif // FILE1_H
```

```c
// file1.c
#include <stdio.h>

void print_count() {
    printf("Count: %d\n", count);
}
```

Variables globales


```c
// main.c
#include <stdio.h>
#include "file1.h"

void increment_count() {
    count++;
}

int main() {
    count = 10;
    print_count();
    increment_count();
    printf("Count en main: %d\n", count);
    return 0;
}
```

Variables globales


```
$ gcc -Wall -c file1.c -o file1.o
$ gcc -Wall -c main.c -o main.o
$ gcc -Wall file1.o main.o -o binario
/usr/bin/ld: main.o:(.bss+0x0): multiple definition
of `count'; file1.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
```

```h
// file1.h
#ifndef FILE1_H
#define FILE1_H

extern int count; //declaración
void print_count();

#endif // FILE1_H
```

```c
// file1.c
#include "file1.h"
#include <stdio.h>

void print_count() {
    printf("Count: %d\n", count);
}
```

```c
// main.c
#include <stdio.h>
#include "file1.h"

int count; // definición

void increment_count() {
    count++;
}

int main() {
    count = 10;
    print_count();
    increment_count();
    printf("Count in main: %d\n", count);
    return 0;
}
```

Variables globales


```
$ gcc -Wall -c file1.c -o file1.o
$ gcc -Wall -c main.c -o main.o
$ gcc -Wall file1.o main.o -o binario
$ ./binario
Count: 10
Count in main: 11
```

Makefile


```
1  CC:=gcc
2  CFLAGS:= -Wall -Wextra -pedantic -std=c11 -O0 -ggdb
3
4  main: main.o
5      $(CC) $(CFLAGS) $^ -o $@
6
7  main.o: main.c
8      $(CC) $(CFLAGS) -c $< -o $@
```

```
$ make
gcc -Wall -Wextra -pedantic -std=c11 -O0 -ggdb -c main.c -o main.o
gcc -Wall -Wextra -pedantic -std=c11 -O0 -ggdb main.o -o main
$ make
make: 'main' is up to date.
```

Makefile


```
1  CC:=gcc
2  CFLAGS:= -Wall -Wextra -pedantic -std=c11 -O0 -ggdb
3
4  TARGET := binario
5  .PHONY: all clean rebuild
6
7  all: $(TARGET)
8
9  $(TARGET): main.o funca.o funcb.o
10     $(CC) $(CFLAGS) $^ -o $@
11
12 funca.o: funca.c funca.h
13     $(CC) $(CFLAGS) -c $< -o $@
14
15 funcb.o: funcb.c funcb.h
16     $(CC) $(CFLAGS) -c $< -o $@
17
18 main.o: main.c funca.h funcb.h
19     $(CC) $(CFLAGS) -c $< -o $@
20
21 clean:
22     rm -rf *.o
23     rm -rf $(TARGET)
24
25 rebuild: clean all
```

Próxima clase

Próxima clase

Próxima clase

Próxima clase

¿Consultas?