Introducción

Tema 1

Índice

Profesores

Pablo Neira Ayuso

Carlos Falgueras García

Temario

- Variables, tipos y punteros
- Herramientas y espacio de trabajo
- Arrays y estructuras
- C modular
- Memoria dinámica
- Objetos
- Argumentos del main
- Listas encadenadas
- Entrada y salida
- Punteros a funciones
- Herramientas de depuración
- Interfaces gráficas con GTK
- Sockets

Material de clase

Wiki

http://1984.lsi.us.es/wiki-c

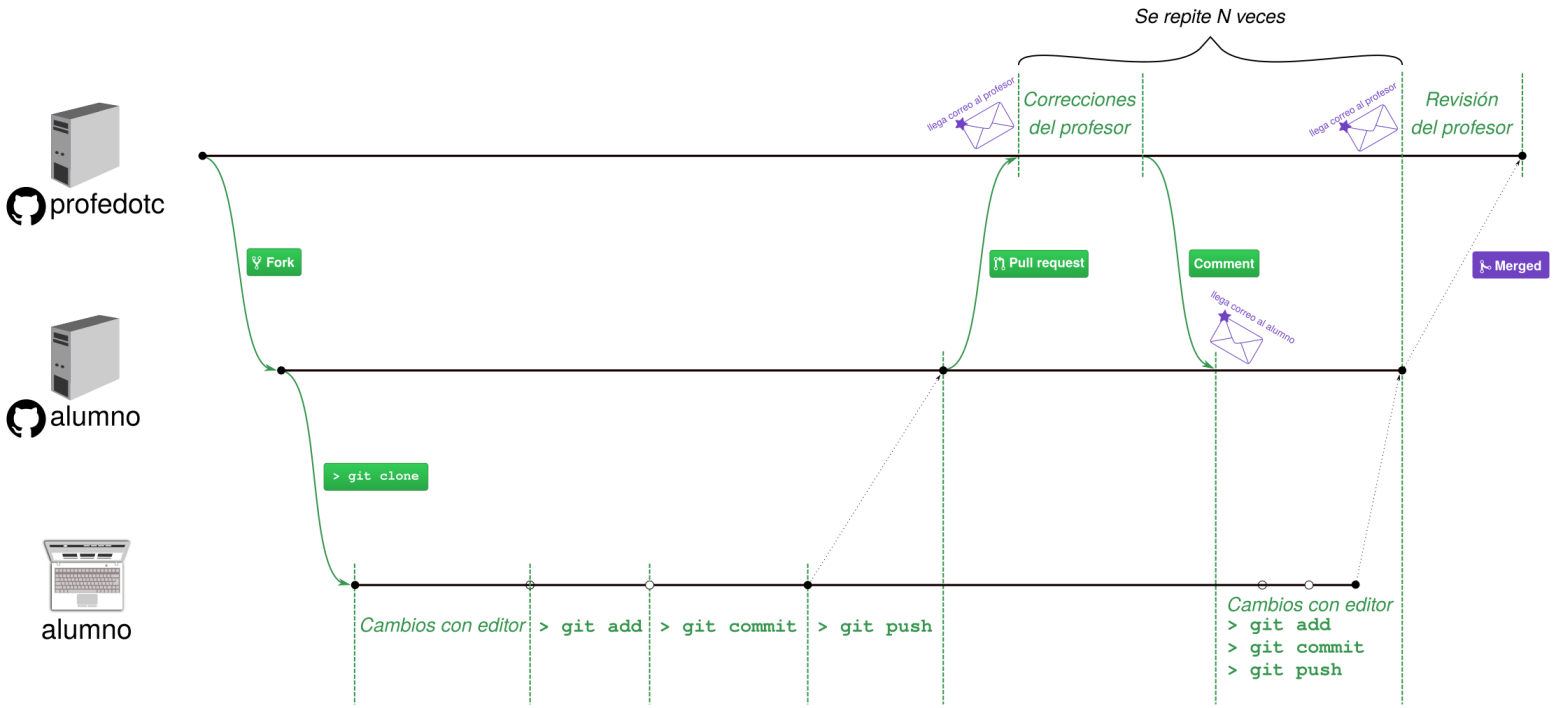
Lista de correo

https://listas.us.es/mailman/listinfo/programacion-c

Herramientas

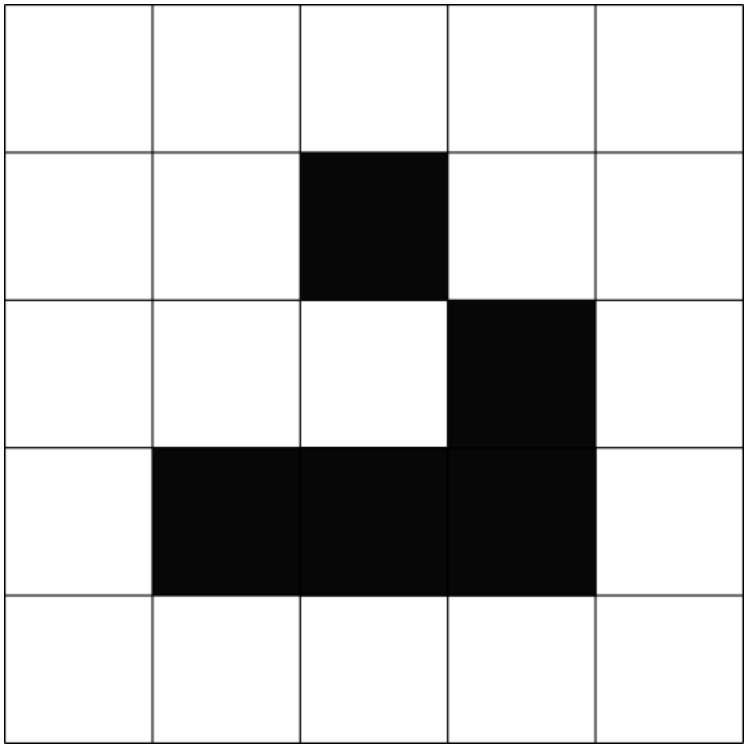
- Linux y terminal
- Editor de texto (*geany*)
- Compilador GCC
- *Make* para automatizar la compilación
- Repositorio de código (*git* y *GitHub*)

Flujo de trabajo



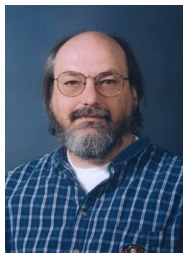
Evaluación

Juego de la vida de Conway



https://en.wikipedia.org/wiki/Conway's_Game_of_Life

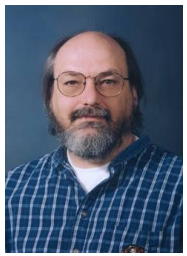
Inicios



- Ken Thompson
- Dennis Ritchie
- Brian Kernighan

- **Bell Labs** (de **AT&T**)
- Ensamblador y **B** insuficientes → diseñan **C**
- C fue desarrollado por **Dennis Ritchie** entre 1969 y **1973**
- **Unix** reescrito en C (1973)
- En 1973 *Brian Kernighan* y *Dennis Ritchie* publican **The C Programming Language** (**K&R**), que por muchos años sirvió como especificación informal del lenguaje.
- Posteriormente se añaden más funcionales a C y se estandariza.

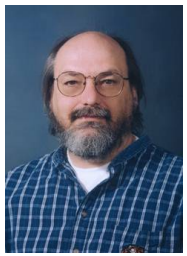
Inicios



- **Ken Thompson**
- **Dennis Ritchie**
- **Brian Kernighan**

- **Bell Labs** (de **AT&T**)
- Ensamblador y **B** insuficientes → diseñan **C**
- C fue desarrollado por **Dennis Ritchie** entre 1969 y **1973**
- **Unix** reescrito en C (1973)
- En 1973 *Brian Kernighan* y *Dennis Ritchie* publican **The C Programming Language** (**K&R**), que por muchos años sirvió como especificación informal del lenguaje.
- Posteriormente se añaden más funcionales a C y se estandariza.

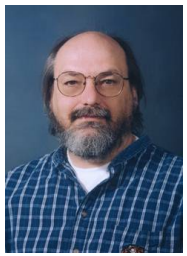
Inicios



- **Ken Thompson**
- **Dennis Ritchie**
- **Brian Kernighan**

- **Bell Labs** (de **AT&T**)
- Ensamblador y **B** insuficientes → diseñan **C**
- C fue desarrollado por **Dennis Ritchie** entre 1969 y **1973**
- **Unix** reescrito en C (1973)
- En 1973 *Brian Kernighan* y *Dennis Ritchie* publican **The C Programming Language** (**K&R**), que por muchos años sirvió como especificación informal del lenguaje.
- Posteriormente se añaden más funcionales a C y se estandariza.

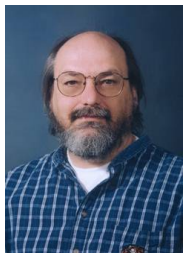
Inicios



- **Ken Thompson**
- **Dennis Ritchie**
- **Brian Kernighan**

- **Bell Labs** (de **AT&T**)
- Ensamblador y **B** insuficientes $\rightarrow$ diseñan **C**
- C fue desarrollado por **Dennis Ritchie** entre 1969 y **1973**
- **Unix** reescrito en C (1973)
- En 1973 *Brian Kernighan* y *Dennis Ritchie* publican **The C Programming Language** (**K&R**), que por muchos años sirvió como especificación informal del lenguaje.
- Posteriormente se añaden más funcionales a C y se estandariza.

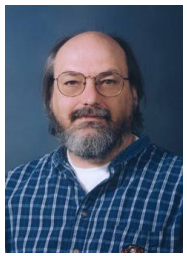
Inicios



- **Ken Thompson**
- **Dennis Ritchie**
- **Brian Kernighan**

- **Bell Labs** (de **AT&T**)
- Ensamblador y **B** insuficientes $\rightarrow$ diseñan **C**
- C fue desarrollado por **Dennis Ritchie** entre 1969 y **1973**
- **Unix** reescrito en C (1973)
- En 1973 *Brian Kernighan* y *Dennis Ritchie* publican **The C Programming Language** (**K&R**), que por muchos años sirvió como especificación informal del lenguaje.
- Posteriormente se añaden más funcionales a C y se estandariza.

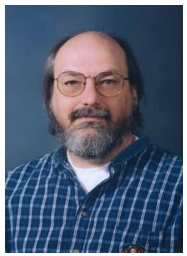
Inicios



- **Ken Thompson**
- **Dennis Ritchie**
- **Brian Kernighan**

- **Bell Labs** (de **AT&T**)
- Ensamblador y **B** insuficientes $\rightarrow$ diseñan **C**
- C fue desarrollado por **Dennis Ritchie** entre 1969 y **1973**
- **Unix** reescrito en C (1973)
- En 1973 *Brian Kernighan* y *Dennis Ritchie* publican **The C Programming Language** (**K&R**), que por muchos años sirvió como especificación informal del lenguaje.
- Posteriormente se añaden más funcionales a C y se estandariza.

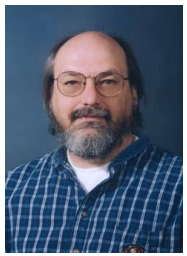
Inicios



- **Ken Thompson**
- **Dennis Ritchie**
- **Brian Kernighan**

- **Bell Labs** (de **AT&T**)
- Ensamblador y **B** insuficientes → diseñan **C**
- C fue desarrollado por **Dennis Ritchie** entre 1969 y **1973**
- **Unix** reescrito en C (1973)
- En 1973 *Brian Kernighan* y *Dennis Ritchie* publican **The C Programming Language** (**K&R**), que por muchos años sirvió como especificación informal del lenguaje.
- Posteriormente se añaden más funcionales a C y se estandariza.

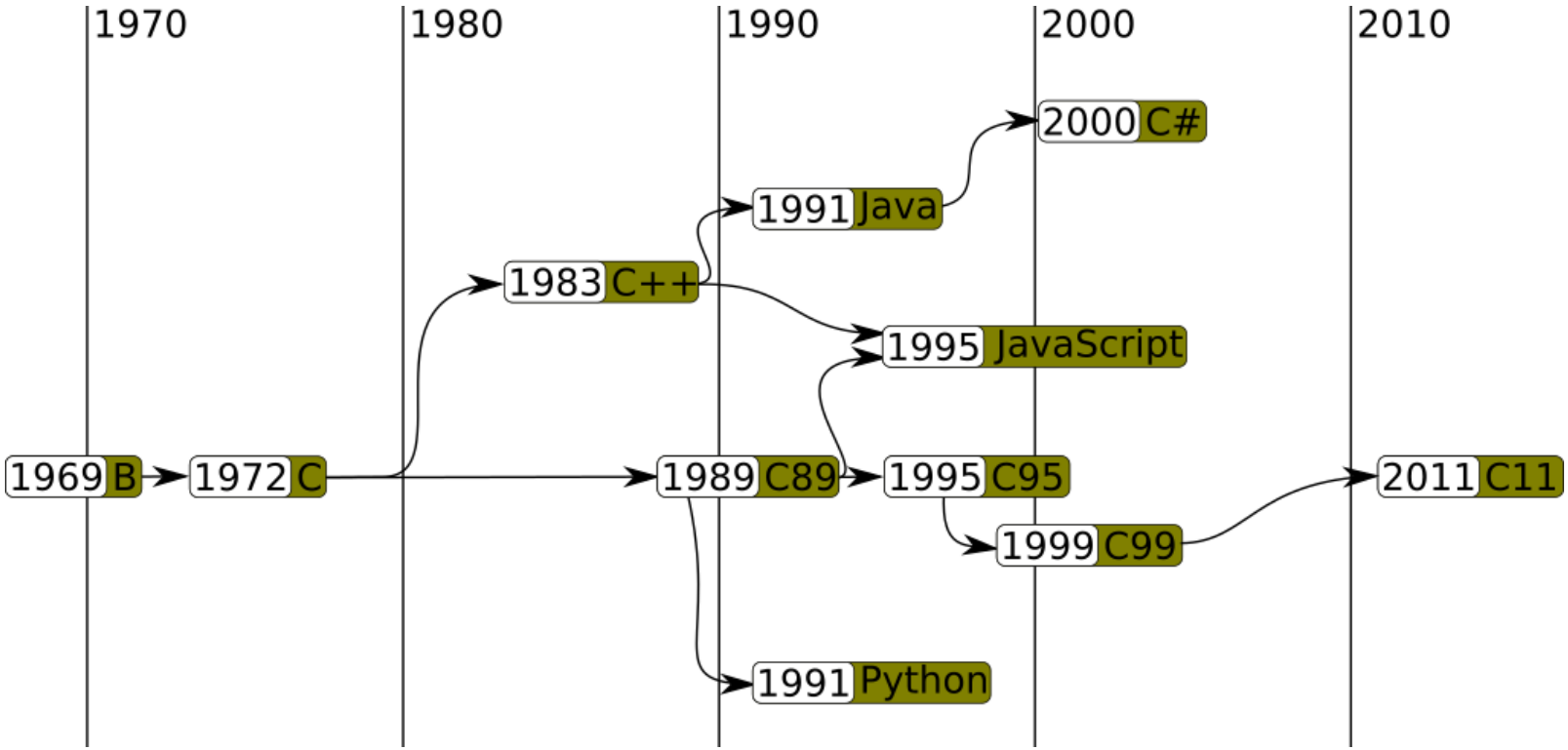
Inicios



- **Ken Thompson**
- **Dennis Ritchie**
- **Brian Kernighan**

- **Bell Labs** (de **AT&T**)
- Ensamblador y **B** insuficientes → diseñan **C**
- C fue desarrollado por **Dennis Ritchie** entre 1969 y **1973**
- **Unix** reescrito en C (1973)
- En 1973 *Brian Kernighan* y *Dennis Ritchie* publican **The C Programming Language** (**K&R**), que por muchos años sirvió como especificación informal del lenguaje.
- Posteriormente se añaden más funcionales a C y se estandariza.

Influencias

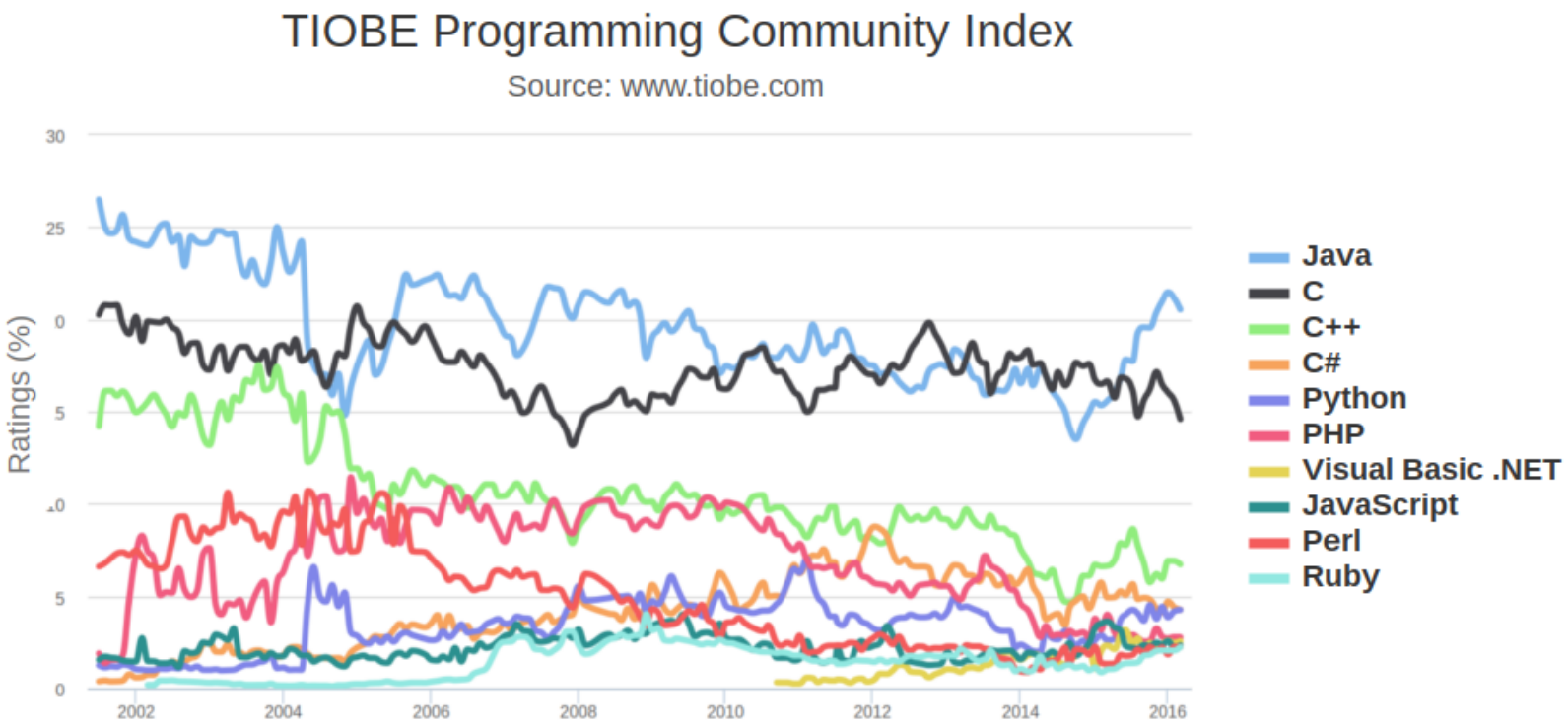


¿Por qué C?

- Simpleza
- Características de bajo nivel
- Madurez
- Eficiencia
- Portabilidad
- Numerosas bibliotecas y herramientas

¿Por qué C?

- Popularidad



¿Para qué C?

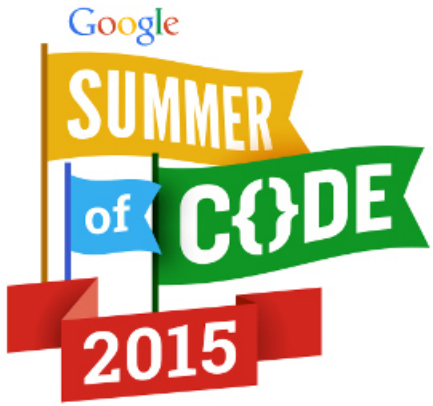
- **Ciencia:**
  - Simulaciones
  - Operaciones con grandes cantidades de tatos
- **Sistemas Empotrados:**
  - Sistemas Operativos en tiempo real
  - Electrodomésticos, ascensores, automovilismo ...
- **Robótica**
  - Drones
  - Robots humanoides
  - Coches autónomos
- **Medicina**
  - Prótesis robóticas
  - Equipamiento médico
- **Sistemas Operativos**

Proyectos en C

- Unix, GNU/Linux, kernel de MacOS y kernel de Windows
- Firefox y muchos otros exploradores (gumbo)
- Apache
- Gnome (GTK)
- Rover Curiosity (2.5 millones de lineas)

Google Summer Of Code

- Beca de Google para estudiantes
- Trabajas **3 meses** en un proyecto de **software libre**
- Experiencia
- Dinero: 5500$



https://summerofcode.withgoogle.com/

Outreachy

- Beca de Gnome para:
  - mujeres
  - grupos discriminados o con poca representación en el mundo tecnológico
  - **Que no hayan participado antes ni en Outreachy ni en GSOC**
- Trabajas **3 meses** en un proyecto de **software libre**
- Experiencia
- Dinero: 5500$

Entorno de trabajo

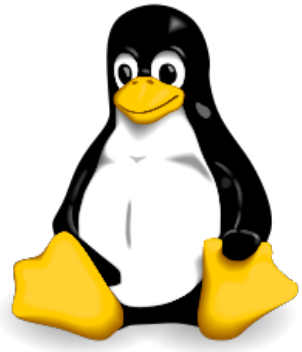
Tema 2

Índice

Linux

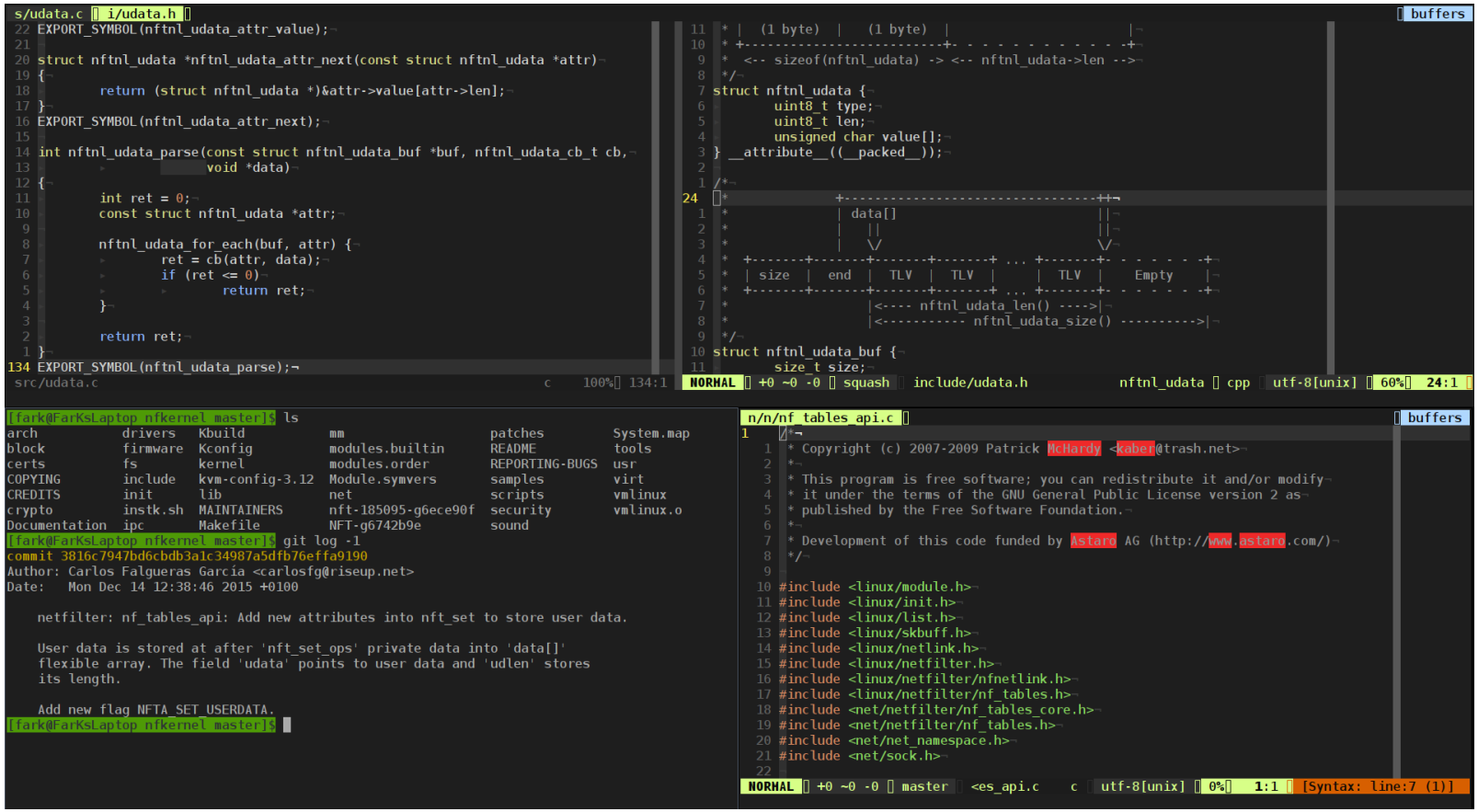
GNU/Linux es el sistema operativo que vamos a utilizar durante el curso.

- Ofrece muchísimas facilidades al programador
- Software libre y gratuito
- Repositorios con infinidad de herramientas a nuestra disposición



Consola

CTRL + ALT + T



Consola

Comandos básicos:

`ls`:	Lista directorios
`cd <dir>`:	Cambia a directorio
`mkdir <dir>`:	Crea directorio
`touch <file>`:	Crea archivo vacío*
`rm <file>`:	Borra archivo
`rm -r <dir>`:	Borra directorio y lo que hay dentro

Instalando herramientas

Comandos para instalar:

`sudo <comando>`: Ejecuta un comando con permisos de administrador

`apt-get install <programa>`: Instala un programa del repositorio

Programas a instalar (`sudo apt-get install <programa>`):

- **gcc**: Compilador
- **make**: Automatización de tareas
- **git**: Gestor de versiones
- **geany**: Editor de texto gráfico

Hola Mundo

Abrir *geany* y guardar el siguiente archivo:

```c
#include <stdio.h>

int main()
{
  printf("Hola mundo!\n");

  return 0;
}
```

Compilación y ejecución:

- Hacer *cd* hasta el directorio dónde se encuentra *helloworld.c*
- `gcc <mi_prog.c>-o <mi_exe>`
- `./mi_exe`

Variables y tipos

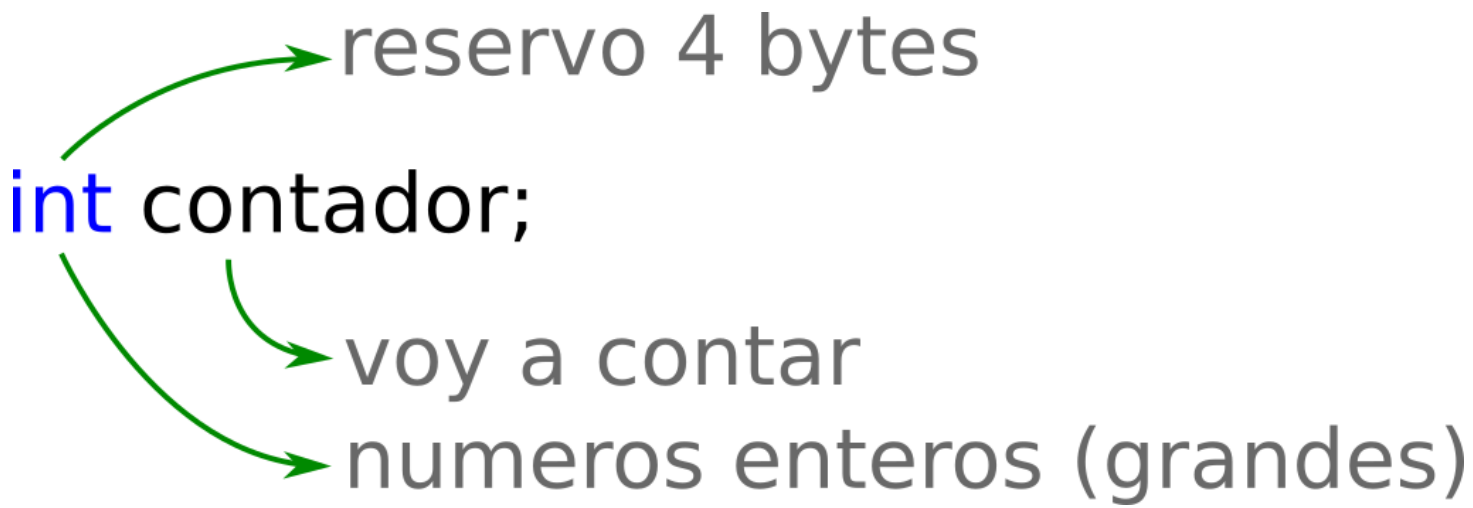
Tema 3

Índice

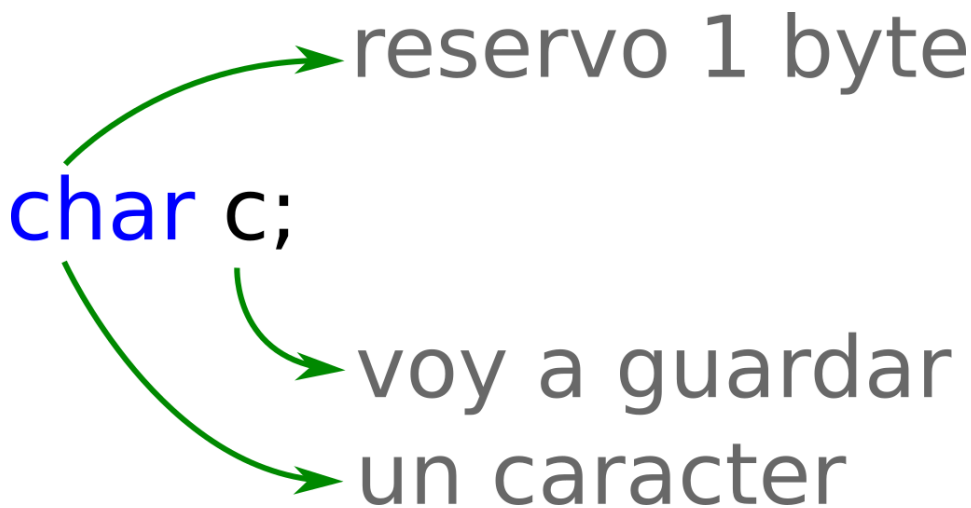
¿Qué es una variable?

- Variables como **zona de memoria** reservada de **tamaño específico**
- Los tipos:
  - Definen el tamaño
  - Dan una idea del uso que se le van a dar a los datos guardados

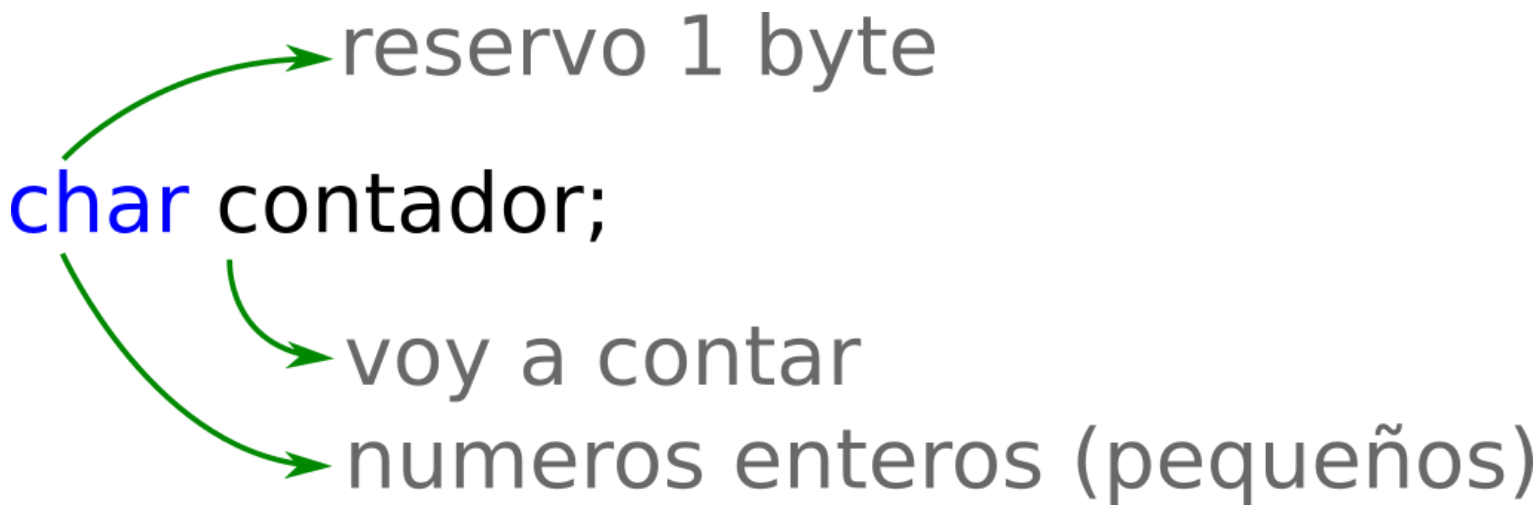
¿Qué es una variable?



¿Qué es una variable?



¿Qué es una variable?



Tipos básicos

Tipos:

- char (`"%c"`)
- int (`"%i"`) ó (`"%d"`)
- float (`"%f"`)
- double (`"%f"`)
- bool

Modificadores:

- signed (`"%hh□"`)
- unsigned (`"%u"`)
- short (`"%h□"`)
- long (`"%l□"`)
- long long (`"%ll□"`)

Más info sobre formato de printf: `http://www.cplusplus.com/reference/cstdio/printf`

Tipos de tamaño fijo


```
#include <stdint.h>
```

```
[u]int_<size>_t
```

- int8_t
- int16_t
- int32_t
- int64_t

- uint8_t
- uint16_t
- uint32_t
- uint64_t

Arrays y tipos

Tema 4

Índice

Arrays

```
int array[5] = {1, 2, 3, 4, 5};
```

- Reserva de memoria **continua** de forma **estática**
- Usos:
  - Vector de elementos
  - Matrices (multidimensionales)
  - Cadenas de texto
  - Espacio de memoria (buffer)

Ejemplo

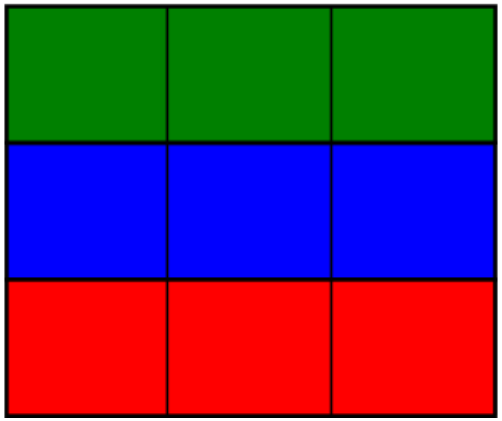

```c
#include <stdio.h>

int main()
{
  int i;
  int vector1[10];
  int vector2[] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};

  for (i = 0; i < 10; i++)
    vector1[i] = vector2[i];

  for (i = 0; i < 10; i++)
    printf("%d ", vector1[i]);

  return 0;
}
```

Cadenas


```c
#include <stdio.h>

int main()
{
  int i;
  char hola[] = {'h', 'o', 'l', 'a', '\0'};
  char mundo[] = "mundo";

  printf("%s %s\n", hola, mundo);

  for (i = 0; i < 4; i++)
    printf("%c ", hola[i]);

  for (i = 0; i < 5; i++)
    printf("%c ", mundo[i]);

  return 0;
}
```

Array multidimensional


```
int array[3][3] = {{11, 12, 13}, {21, 22, 23}, {31, 32, 33}};
```

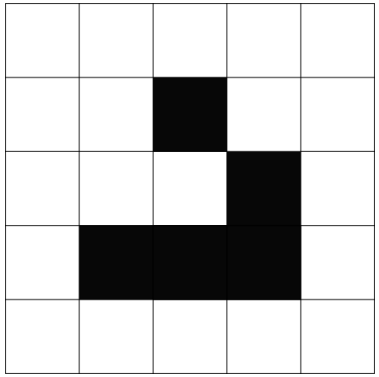


Juego de la vida

Tema 5

Índice

Introducción

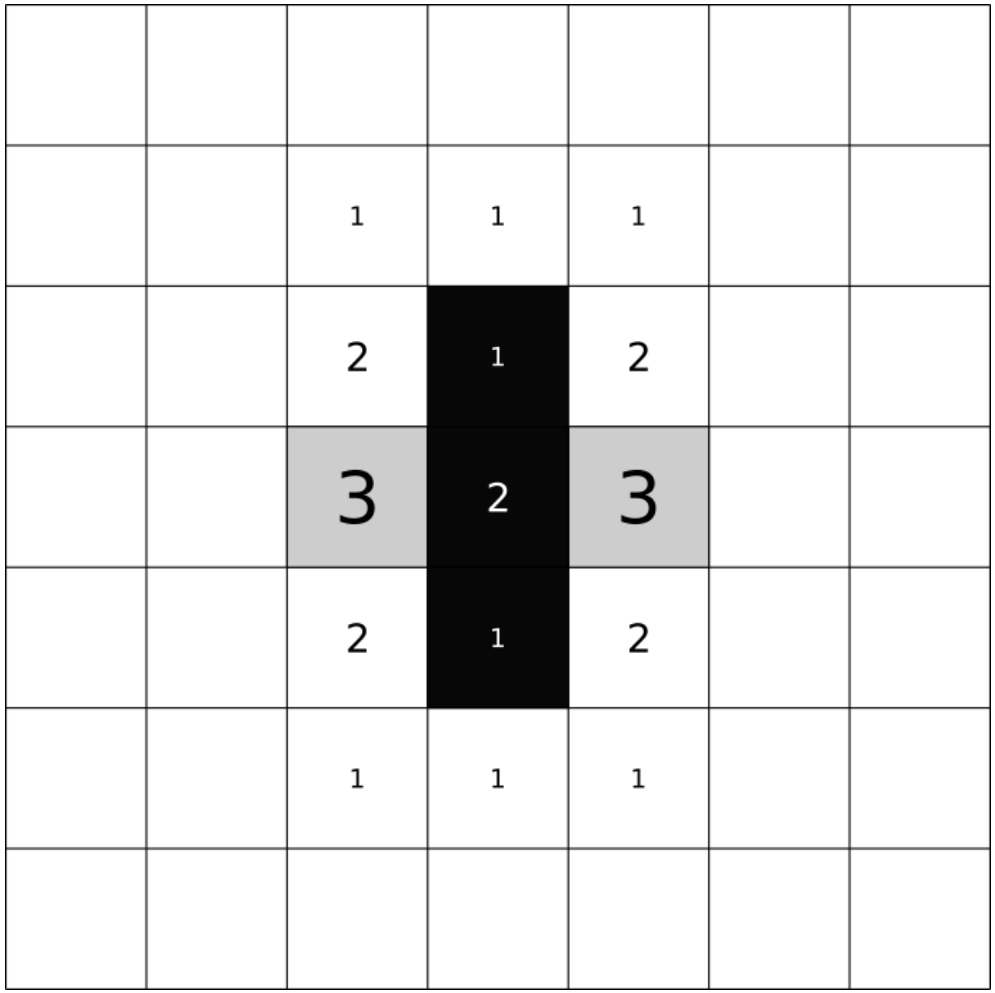


- juego de 0 jugadores
- Rejilla de células cuadradas como universo bidimensional ortogonal (infinito o no)
- Cada célula tiene dos estados (muerta o viva) e interactúa con sus 8 vecinas según unas reglas
- La regla más común es:
  - **Nacimiento**: Una célula muerta con exactamente 3 vecinas vivas estará viva en la siguiente iteración
  - **Supervivencia**: Una célula viva con 2 o 3 vecinas vivas seguirá viva en la siguiente iteración, de lo contrario morirá.

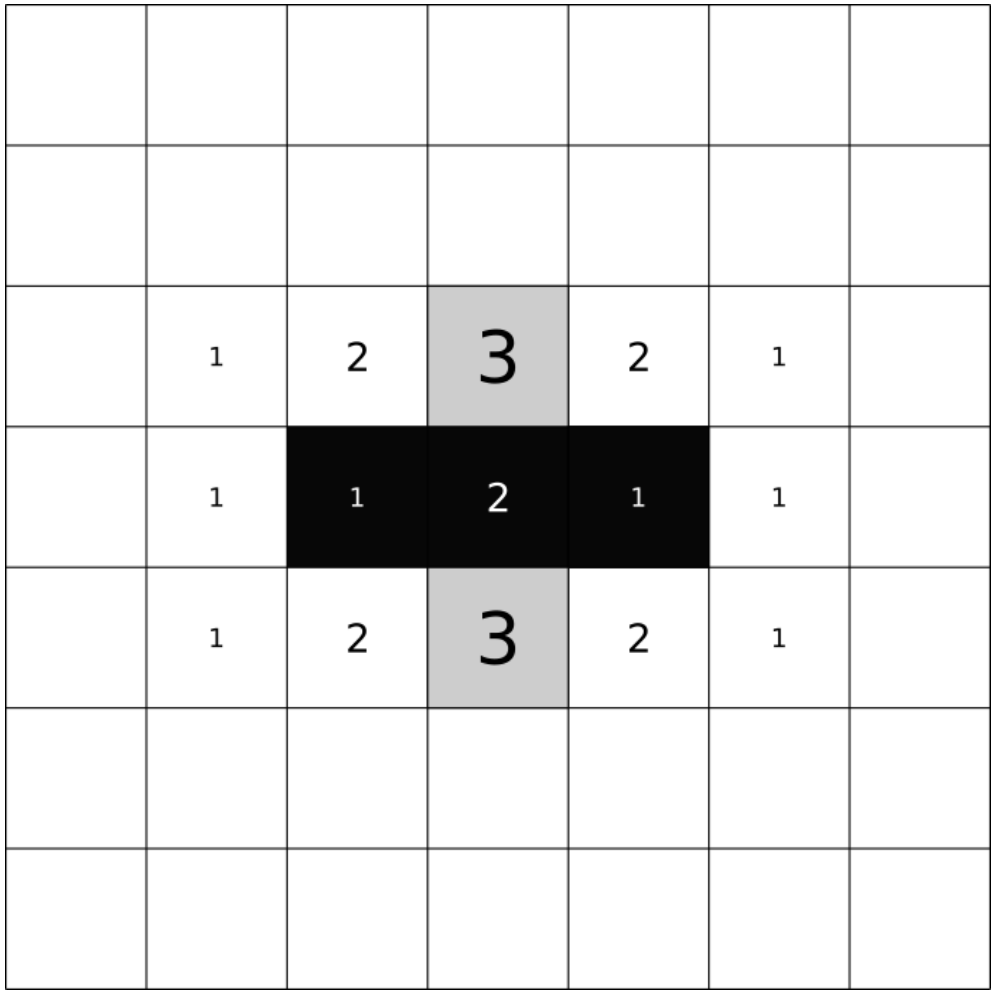
Ejemplo

	1	2	3	2	1	
	1	1	2	1	1	
	1	2	3	2	1	

Ejemplo



Ejemplo



Enlaces de interés

- Más información: es.wikipedia.org/wiki/Juego_de_la_vida
- Simulador (muy bueno): golly.sourceforge.net/
- Simulador web: pmav.eu/stuff/javascript-game-of-life-v3.1.1/

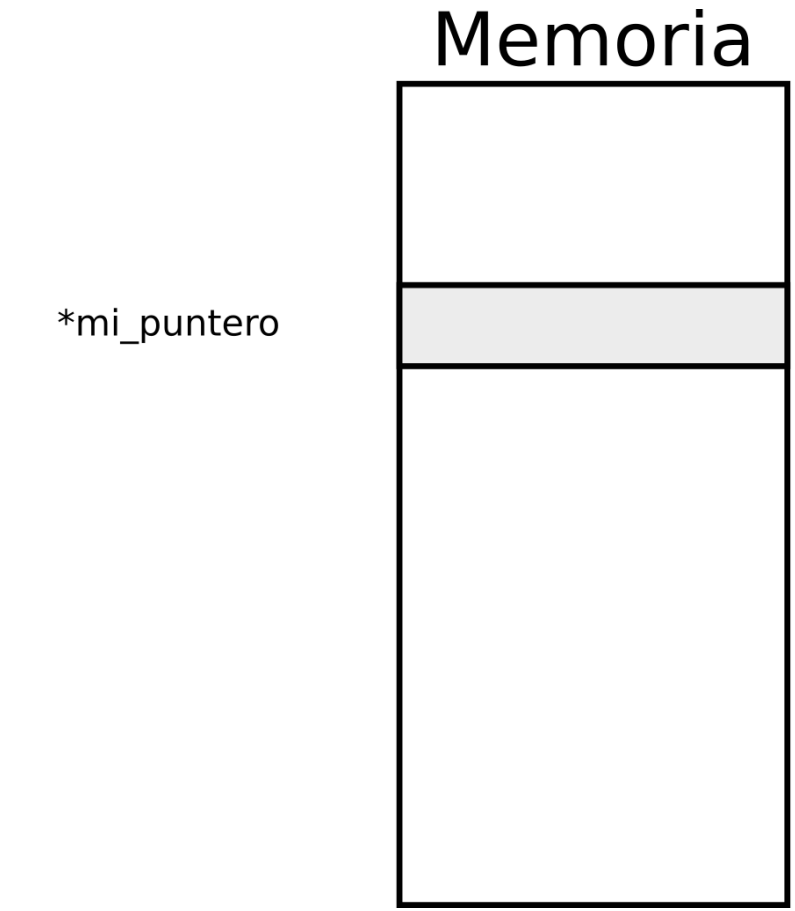
Punteros

Tema 6

Índice

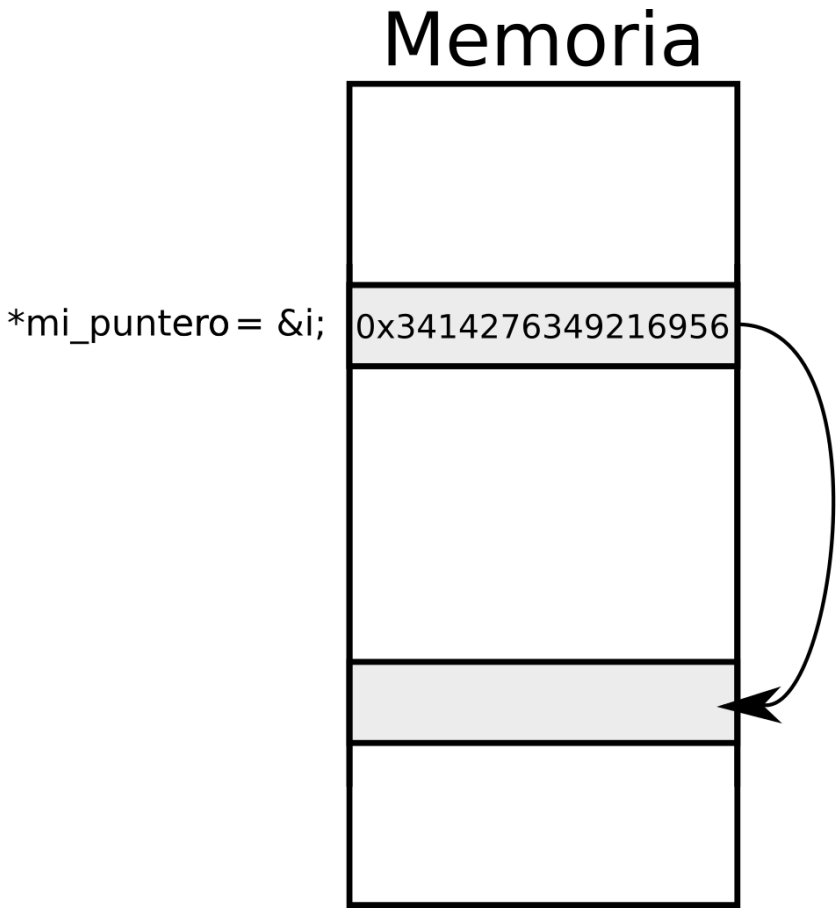
¿Qué es un puntero?

- Son **variables normales** y corrientes
- Pensadas para guardar una **dirección de memoria**
- El tipo del puntero hace referencia al tipo de dato **al que apunta**



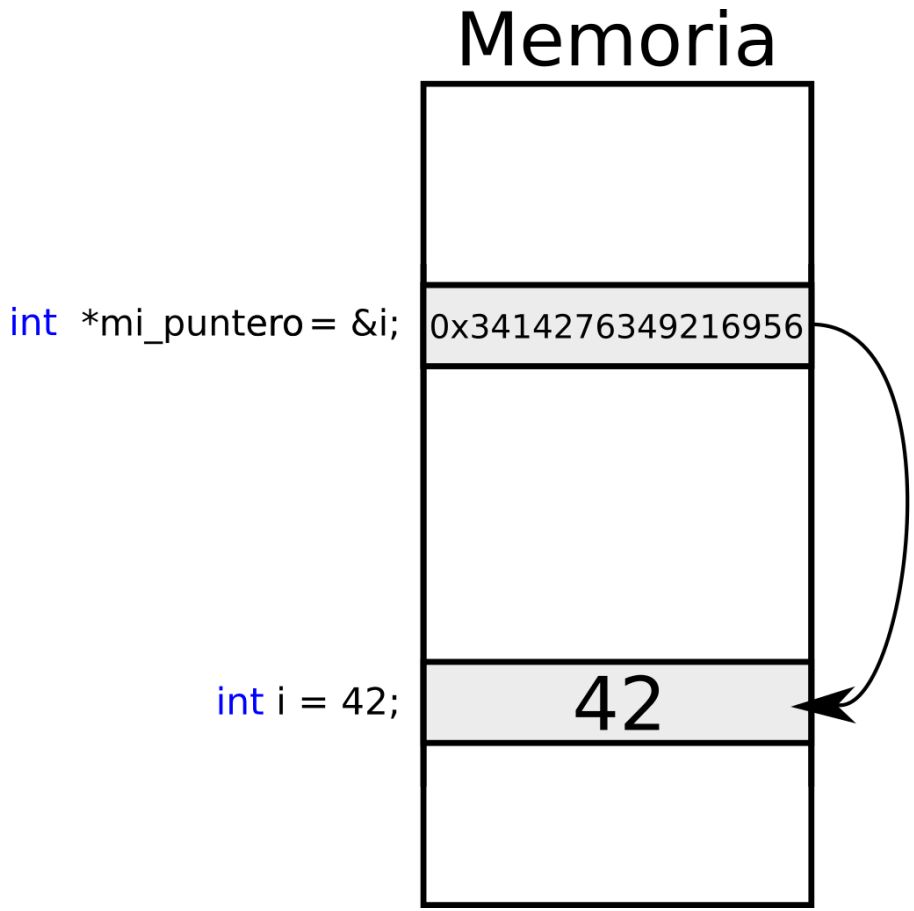
¿Qué es un puntero?

- Son **variables normales** y corrientes
- Pensadas para guardar una **dirección de memoria**
- El tipo del puntero hace referencia al tipo de dato **al que apunta**

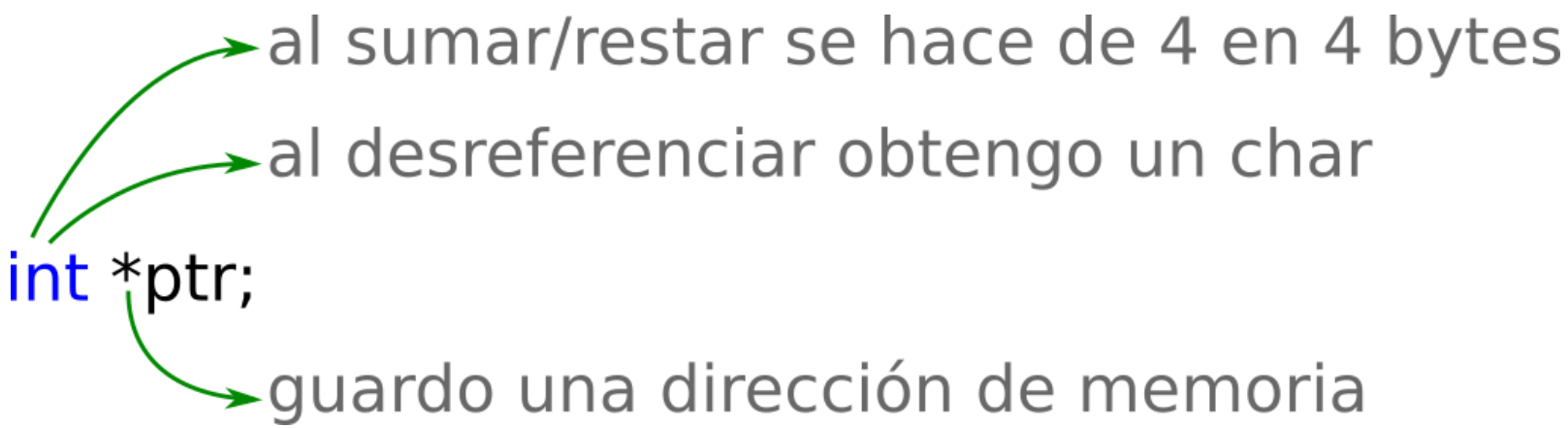


¿Qué es un puntero?

- Son **variables normales** y corrientes
- Pensadas para guardar una **dirección de memoria**
- El tipo del puntero hace referencia al tipo de dato **al que apunta**



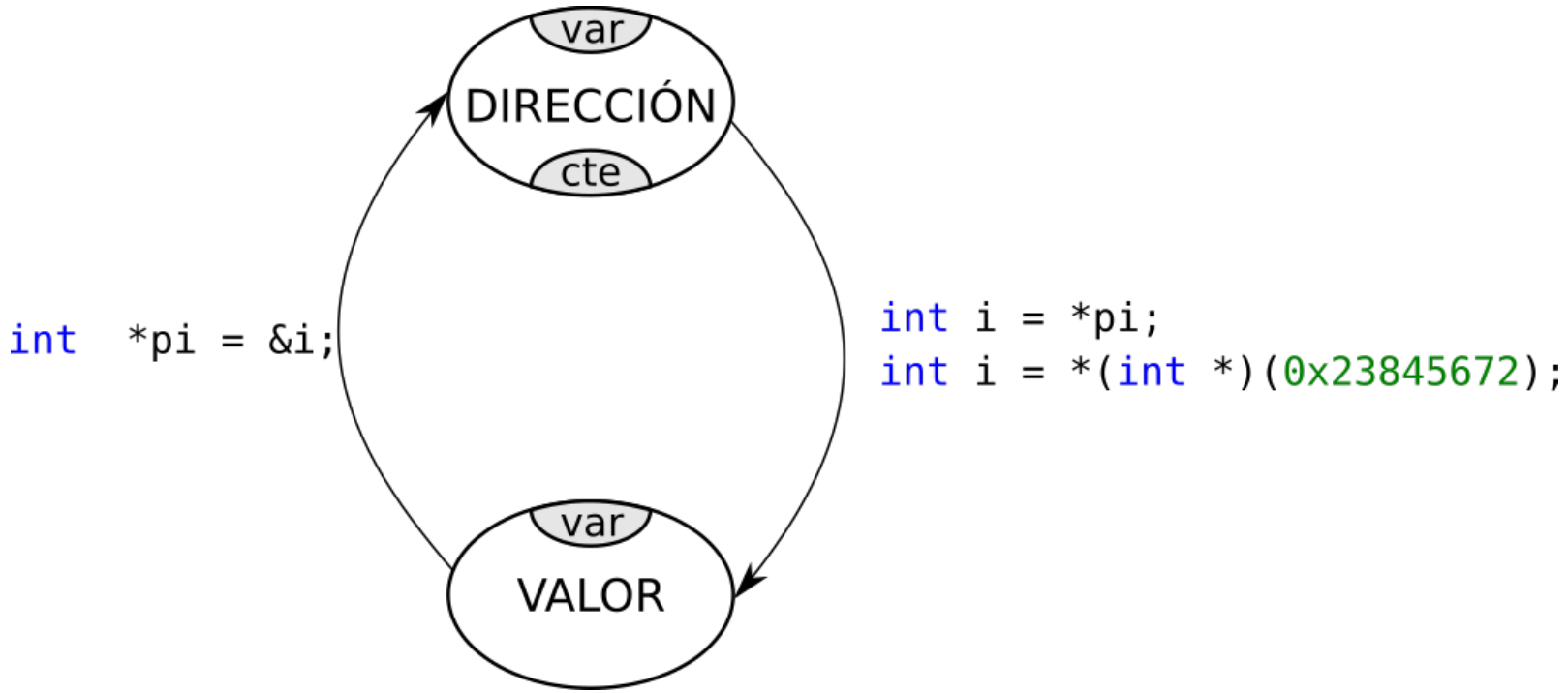
¿Qué es un puntero?



Ejemplo


```c
#include <stdio.h>

int main()
{
  int i = 42;
  int *pi;

  pi = &i;
  printf("dir = %p\n",  pi);
  printf("val = %d\n", *pi);

  *pi = 24;
  printf("val = %d\n", i);

  return 0;
}
```

Sintaxis



Arrays y punteros

- **Arrays**:
  - Son prácticamente punteros constantes (no se puede modificar la dirección a la que apunta)
  - Apuntan al primer elemento del array
  - Mediante el tipo y el índice se obtiene la dirección del elemento deseado
- **Punteros**:
  - Soportan las operaciones de suma y resta de enteros
  - Al sumar un entero y un puntero estamos sumando a la dirección de memoria ese entero por el tamaño del tipo del puntero
  - Se pueden indexar como un array

Arrays y punteros

- **Arrays**:
  - Son prácticamente punteros constantes (no se puede modificar la dirección a la que apunta)
  - Apuntan al primer elemento del array
  - Mediante el tipo y el índice se obtiene la dirección del elemento deseado
- **Punteros**:
  - Soportan las operaciones de suma y resta de enteros
  - Al sumar un entero y un puntero estamos sumando a la dirección de memoria ese entero por el tamaño del tipo del puntero
  - Se pueden indexar como un array

Arrays y punteros

- **Arrays**:
  - Son prácticamente punteros constantes (no se puede modificar la dirección a la que apunta)
  - Apuntan al primer elemento del array
  - Mediante el tipo y el índice se obtiene la dirección del elemento deseado
- **Punteros**:
  - Soportan las operaciones de suma y resta de enteros
  - Al sumar un entero y un puntero estamos sumando a la dirección de memoria ese entero por el tamaño del tipo del puntero
  - Se pueden indexar como un array

Arrays y punteros

- **Arrays**:
  - Son prácticamente punteros constantes (no se puede modificar la dirección a la que apunta)
  - Apuntan al primer elemento del array
  - Mediante el tipo y el índice se obtiene la dirección del elemento deseado
- **Punteros**:
  - Soportan las operaciones de suma y resta de enteros
  - Al sumar un entero y un puntero estamos sumando a la dirección de memoria ese entero por el tamaño del tipo del puntero
  - Se pueden indexar como un array

Arrays y punteros

- **Arrays**:
  - Son prácticamente punteros constantes (no se puede modificar la dirección a la que apunta)
  - Apuntan al primer elemento del array
  - Mediante el tipo y el índice se obtiene la dirección del elemento deseado
- **Punteros**:
  - Soportan las operaciones de suma y resta de enteros
  - Al sumar un entero y un puntero estamos sumando a la dirección de memoria ese entero por el tamaño del tipo del puntero
  - Se pueden indexar como un array

Ejemplo

Ejemplo:

```
1  int array[3] = {1, 2, 3};
2  int *p = array;
3  int i;
4
5  /* Todas las direcciones iguales */
6  printf("%p\n%p\n%p\n", array, p, &array[0]);
7
8  p[2] = 22;
9  p += 1;
10 *p = 33;
11 *(p - 1) = 11;
12
13 /* Que imprimira? */
14 for (i = 0; i < 3; i++)
15   printf("%d\n", array[i]);
```

Formas de recorrer un array


```
1  int main() {
2    int i;
3    p = array;
4
5    // forma 1: Contador e incremento de puntero
6    for (i = 0; i < 5; i++)
7      printf("%d\t", *p++);
8
9    // forma 2: Incremento de puntero y comparacion de
         direcciones
10   for (p = array; p <= &array[4]; p++)
11     printf("%d\t", *p);
12
13   // forma 3: Contador y puntero como array
14   for (i = 0, p = array; i < 5; i++)
15     printf("%d\t", p[i]);
16
17   return 0;
18 }
```

Jugando con Punteros


```c
#include <stdio.h>

int main()
{
  int magic = 0x00796177;
  printf("\nmagic = %0X\n", magic);
  printf("magic = \"%s\"\n", (char *)(&magic));

  return 0;
}
```

Funciones

Tema 7

Índice

Funciones

En C las funciones:

- Retornan un solo valor o nada (*void*)
- De cero a N parámetros
- Cada parámetro es de un tipo específico
- Todos los parámetros se pasan **por copia**
- Tienen una **declaración** y una **definición**
- Una función ha de estar declarada antes de ser llamada
- Una función no tiene por que estar definida a la hora de ser llamada

```c
#include <stdio.h>

/* Declaracion */
int f(int a, int b);

int main()
{
  /* Llamada */
  printf("%d\n", f(2, 3));

  return 0;
}

/* Definicion */
int f(int a, int b)
{
  return a + b;
}
```

Paso por copia

- **Siempre se copia** el parámetro (variable o constante) que se le pasa a la función al llamarla
- Dentro de la función se trabaja con la copia
- Las variables originales no se ven afectadas

```
1  #include <stdio.h>
2
3  void f(int a)
4  {
5     a = 33;
6  }
7
8  int main()
9  {
10    int a = 3;
11    f(a);
12    printf("%d\n", a);
13
14    return 0;
15 }
```

Paso por referencia

- Para poder modificar las variables originales dentro de una función, esta ha de trabajar con la **referencia** a la variable, no con la original
- Esto se consigue con **punteros**

```
1  #include <stdio.h>
2
3  void f(int *a)
4  {
5     *a = 33;
6  }
7
8  int main()
9  {
10    int a = 3;
11    f(&a);
12    printf("%d\n", a);
13
14    return 0;
15 }
```

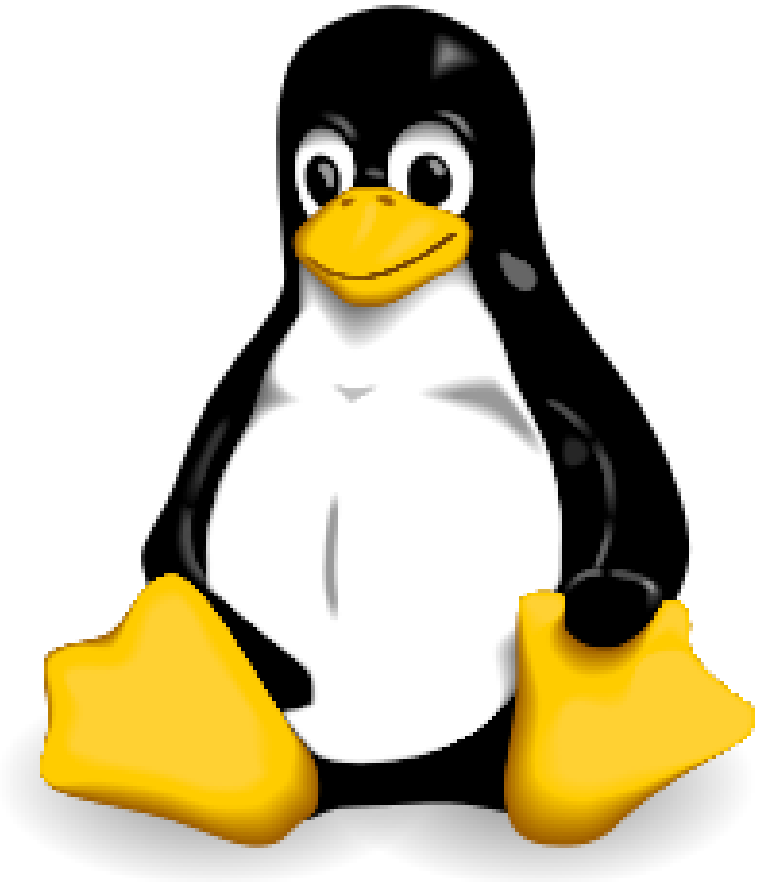
GIT

Tema 8

Índice

Git

git

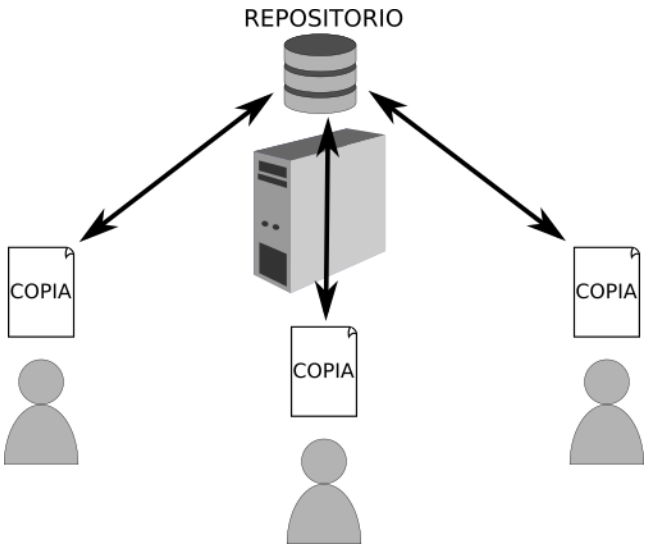


Características

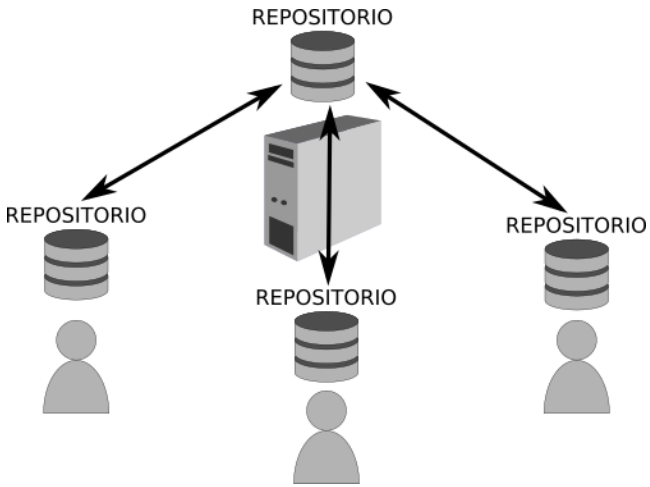
- Historial de versiones
- Visualización de cambios
- Revertir cambios
- Trabajo en equipo de forma concurrente
- Integridad de los archivos
- Sistema distribuido

Distribuido VS Centralizado

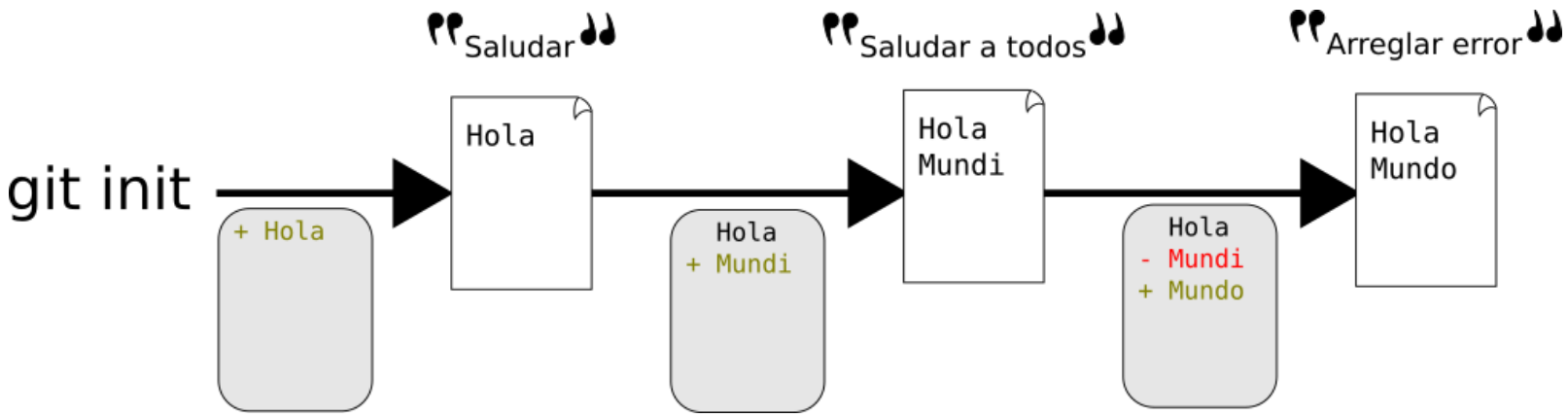
Centralizado



Distribuido



Funcionamiento



- Instantáneas del estado del repo
- Un comentario por cada instantánea
- Solo se guardan las diferencias
- Máquina de el tiempo

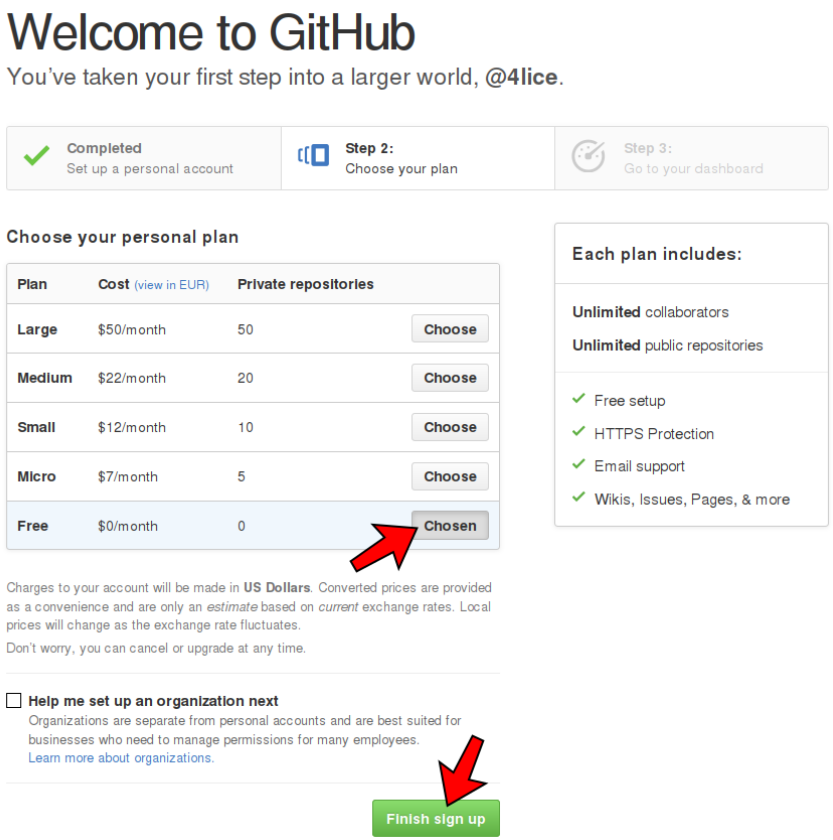
Cuenta en GitHub

github.com

Elegimos un nombre de usuario, una contraseña e introducimos nuestro correo

Plan gratuito

Nos aseguramos de que el plan gratuito está seleccionado y hacemos click en "*Finish sing up*"



Confirmar correo

Debemos confirmar la dirección de correo

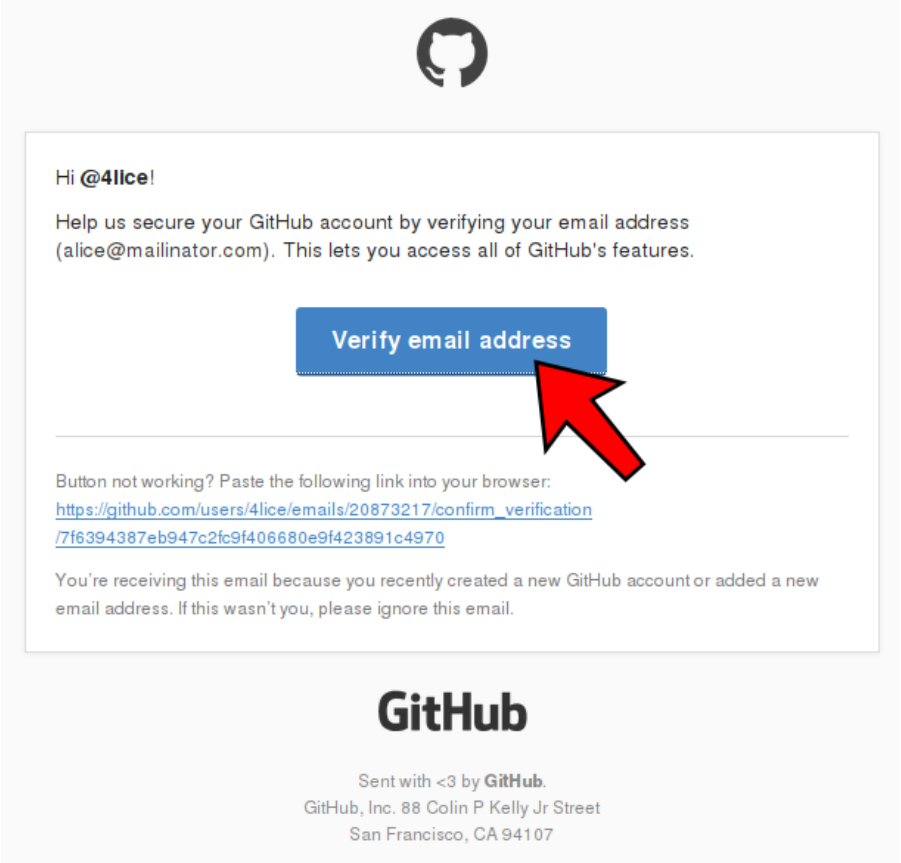
Please verify your email address

Before you can contribute on GitHub, we need you to verify your email address.
An email containing verification instructions was sent to **alice@mailinator.com**.

Didn't get the email? Resend verification email or change your email settings.

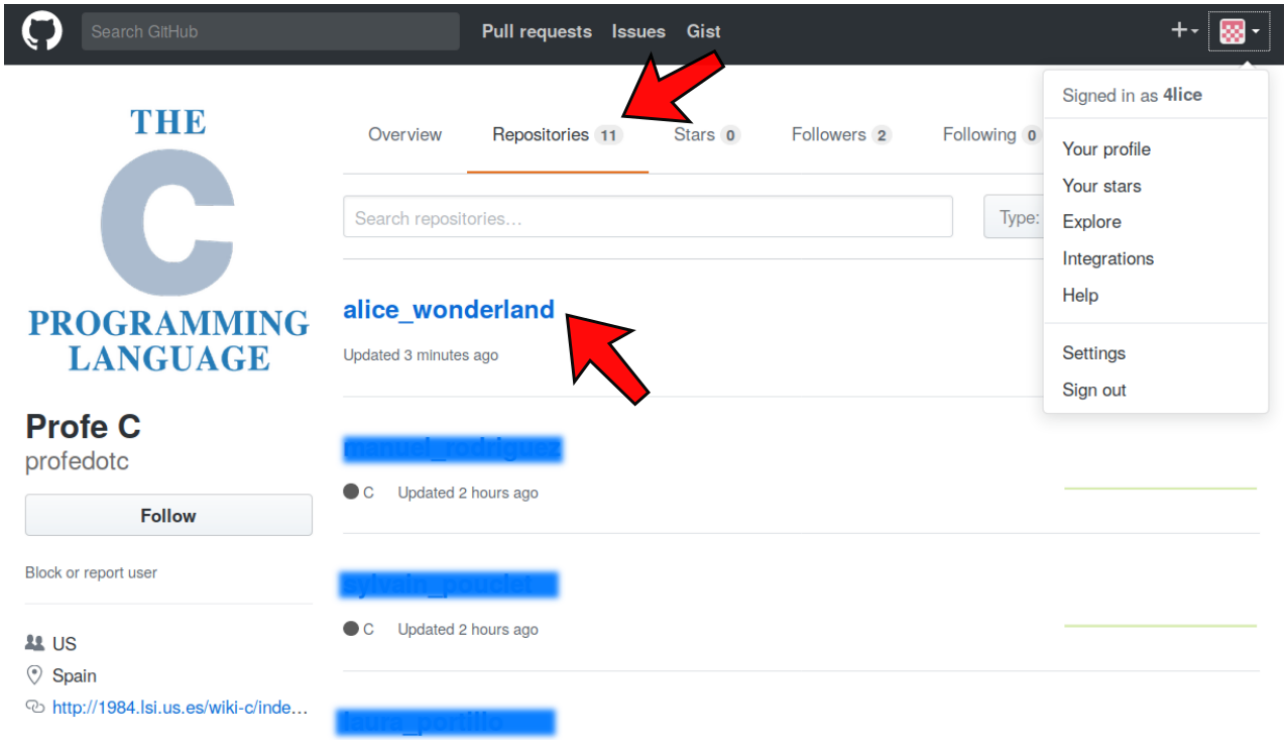
Confirmar correo

Buscamos el correo de confirmación en nuestro buzón y hacemos click en "*Verify email address*"



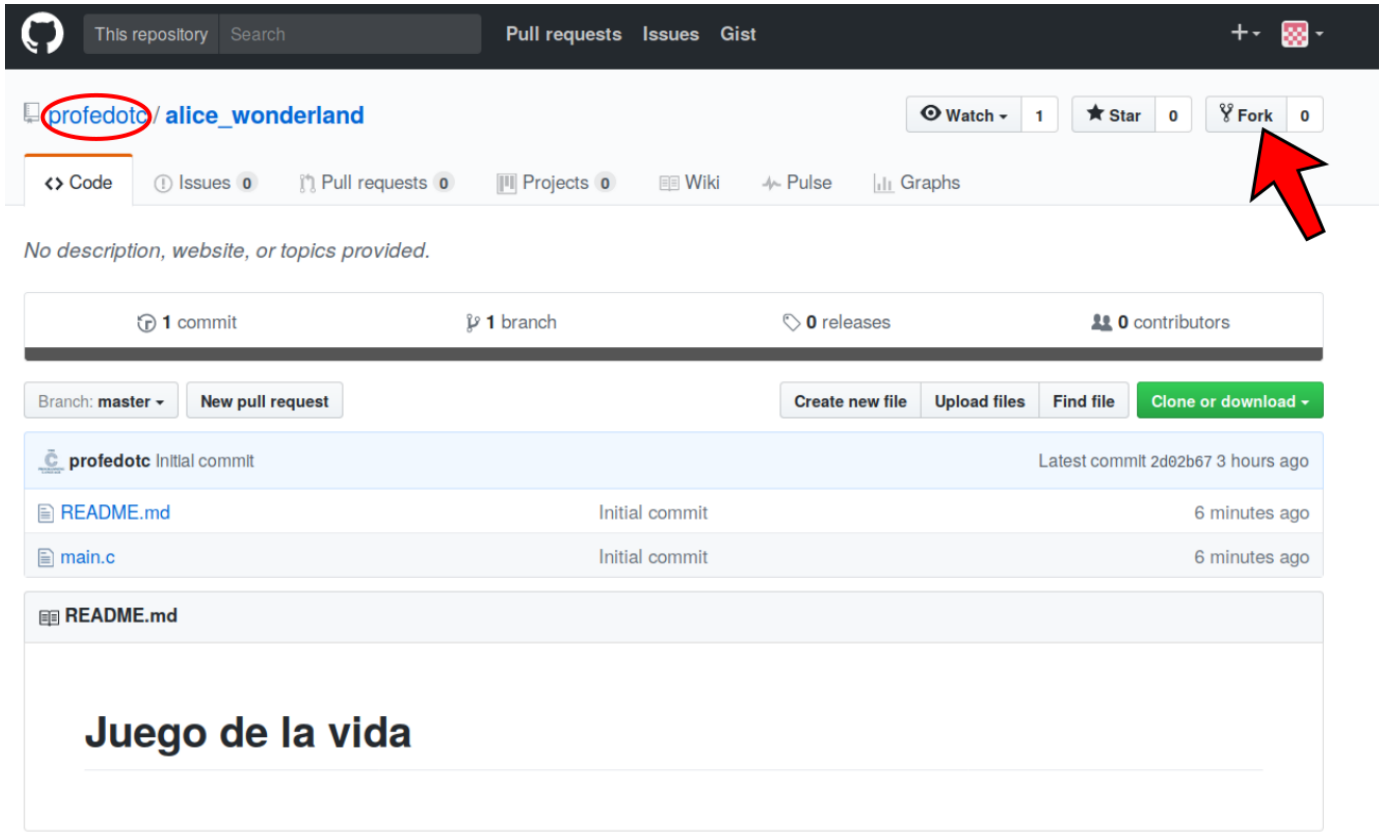
Buscar mi repositorio

Entramos en la cuenta del profesor (github.com/profedotc), y en la pestaña "*Repositories*" buscamos el repositorio que tenga nuestro nombre y apellido



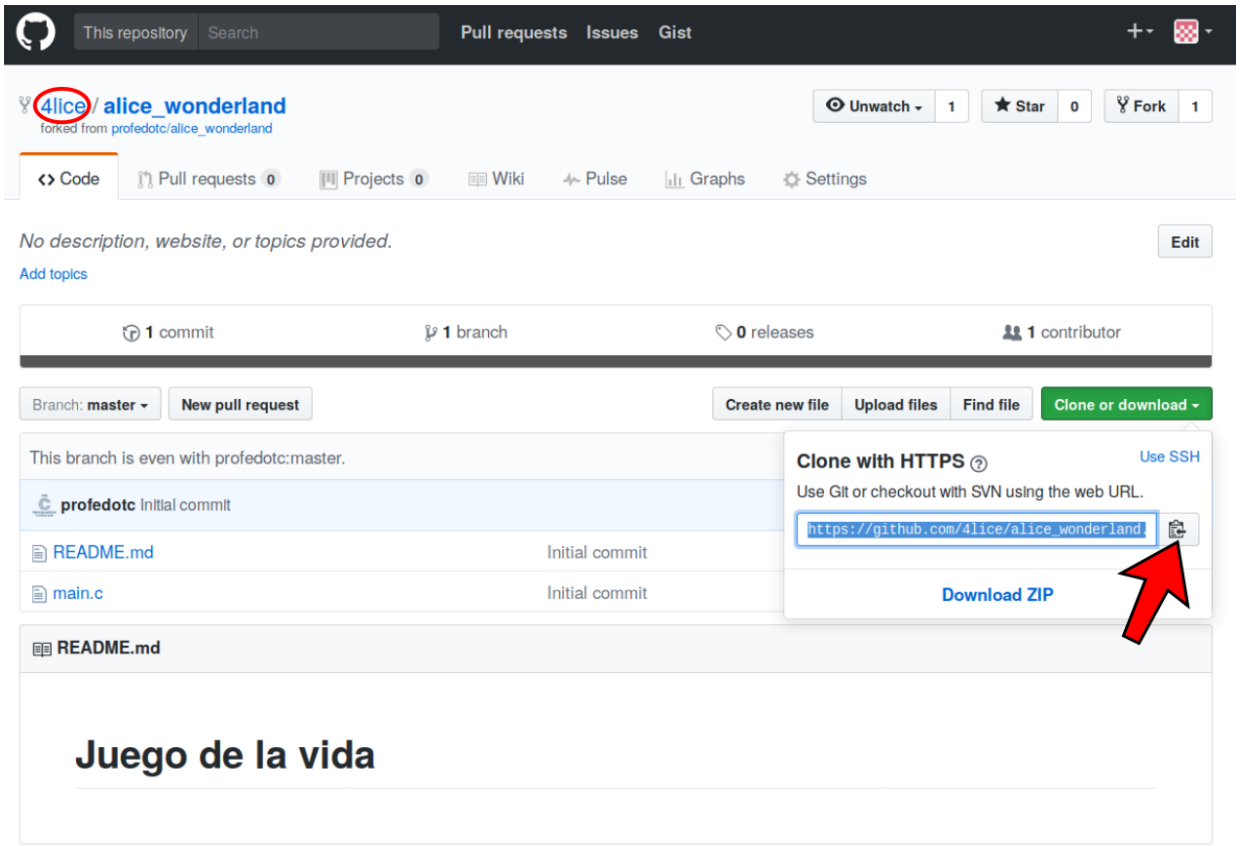
Fork

Hacemos clien en "*Fork*" para crear una copia del repositorio en nuestra cuenta



Clonar mi repositorio

En el menú "*Clone or download*" podemos encontrar la URL necesaria para clonar nuestro repositorio

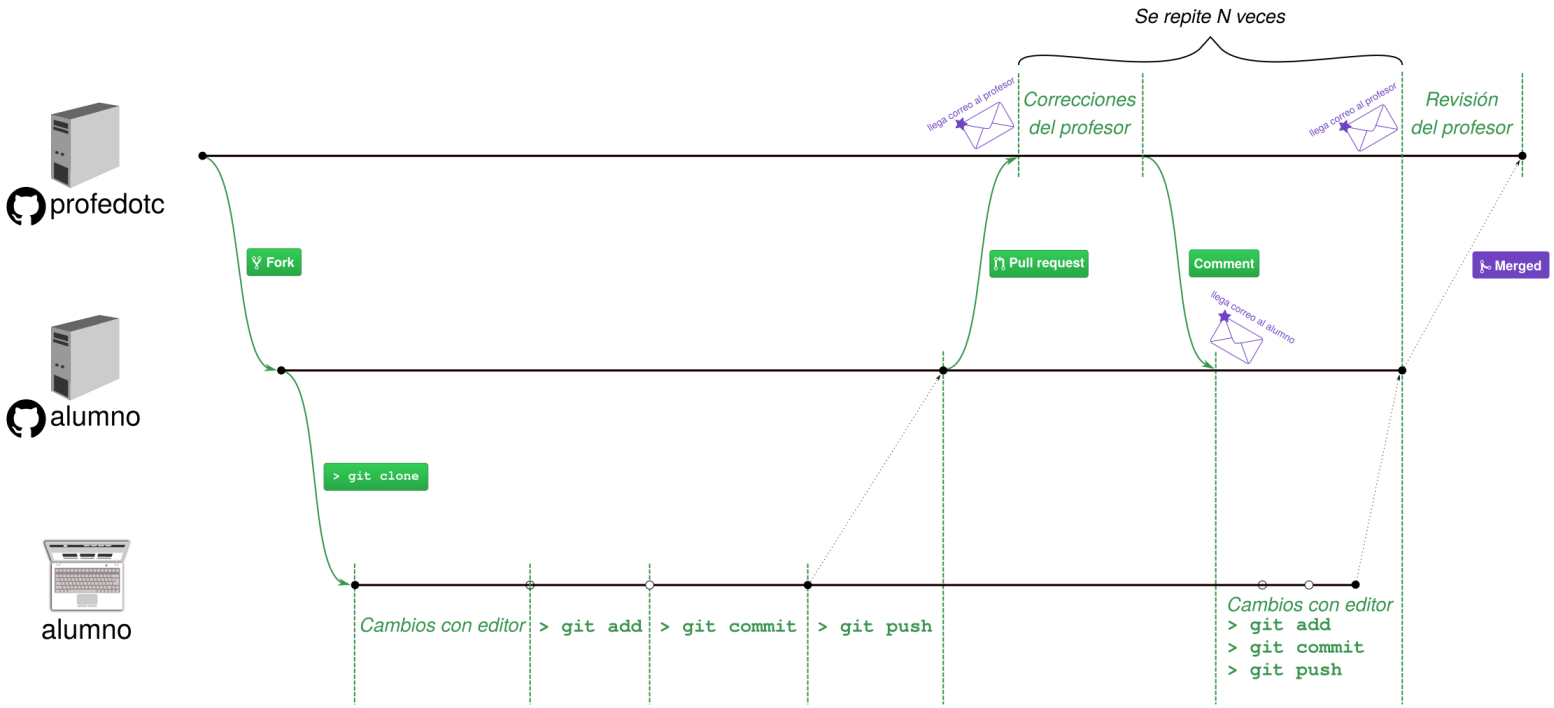


Clonar mi repositorio

Para clonar nuestro repositorio abrimos un terminal, navegamos hasta la carpeta dónde lo queramos clonar y ejecutamos el siguiente comando de git:

```
> git clone https://github.com/profedotc/
alice_wonderland.git
```

Flujo de trabajo



Crea y revisa tus cambios

> git diff

```
1  diff --git a/main.c b/main.c
2  index 7aa2631..3544b1d 100644
3  --- a/main.c
4  +++ b/main.c
5  @@ -2,10 +2,11 @@
6   #include <stdlib.h>
7   #include <stdbool.h>
8
9  -// TODO: Crea dos macros con el tamanio horizontal y vertical del
       mundo
10 +#define W_SIZE_X 10
11 +#define W_SIZE_Y 10
12
13  void world_init(/* Recibo un mundo */);
14 -void world_print(/* Recibo un mundo */);
15 +void world_print(bool w[W_SIZE_X][W_SIZE_Y]);
16  void world_step(/* Recibo dos mundos */);
17  int world_count_neighbors(/* Recibo un mundo y unas coordenadas */)
        ;
18  bool world_get_cell(/* Recibo un mundo y unas coordenadas */);
```

Crea y revisa tus cambios

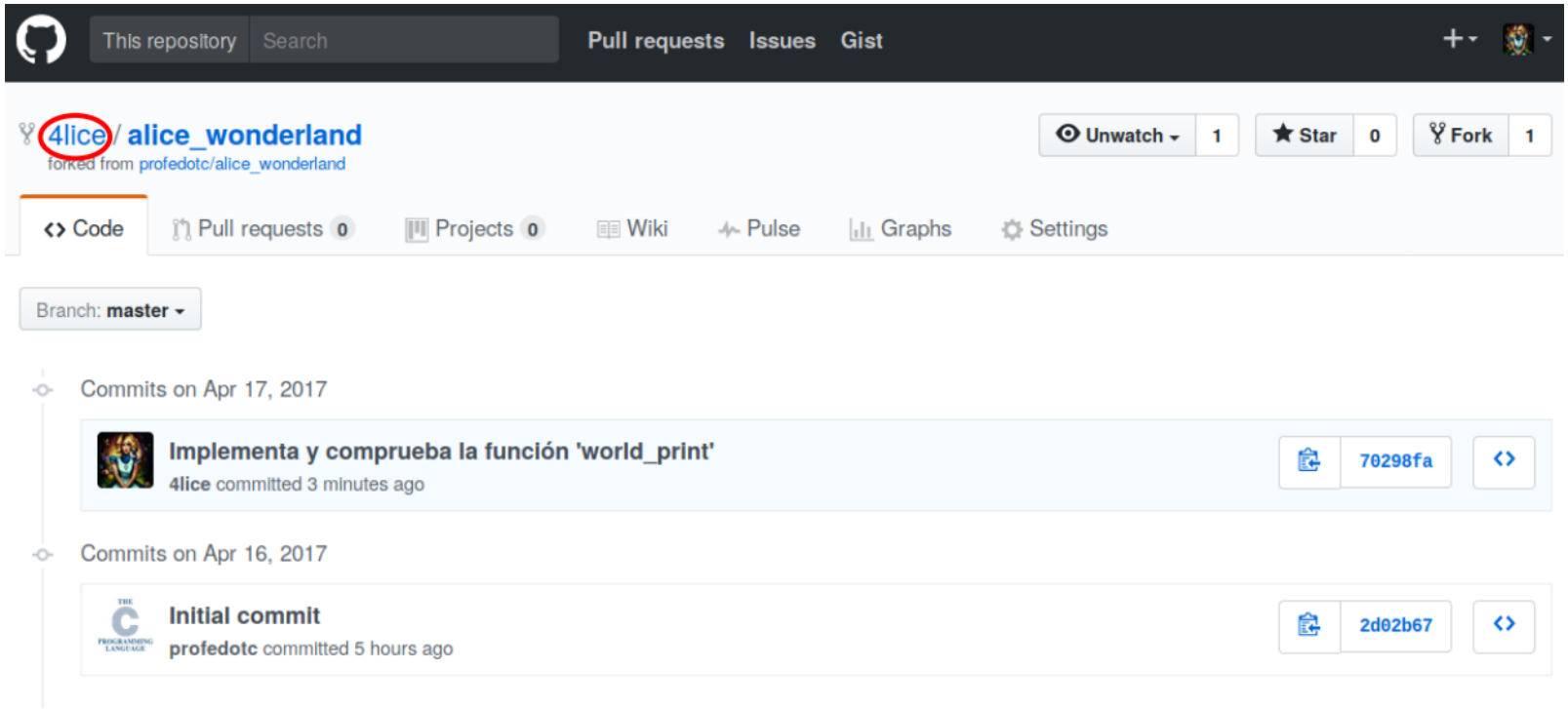

```
1  @@ -14,12 +15,13 @@ void world_copy(/* Recibo dos mundos */);
2   int main()
3   {
4     int i = 0;
5  -  // TODO: Declara dos mundos
6  +  bool world_a[W_SIZE_X][W_SIZE_Y];
7  +  bool world_b[W_SIZE_X][W_SIZE_Y];
8
9     // TODO: inicializa el mundo
10    do {
11      printf("\033clIteration %d\n", i++);
12 -    // TODO: Imprime el mundo
13 +    world_print(world_a);
14      // TODO: Itera
15    } while (getchar() != 'q');
```

Crea y revisa tus cambios


```
1  @@ -37,21 +39,15 @@ void world_init(/* Recibo un mundo */)
2     */
3   }
4
5  -void world_print(/* Recibo un mundo */)
6  +void world_print(bool w[W_SIZE_X][W_SIZE_Y])
7   {
8  - // TODO: Imprimir el mundo por consola. Sugerencia:
9  - /*
10 -  *      . # . . . . . . .
11 -  *      . . # . . . . . .
12 -  *      # # # . . . . . .
13 -  *      . . . . . . . . .
14 -  *      . . . . . . . . .
15 -  *      . . . . . . . . .
16 -  *      . . . . . . . . .
17 -  *      . . . . . . . . .
18 -  *      . . . . . . . . .
19 -  *      . . . . . . . . .
20 -  */
21 + for (int i = 0; i < W_SIZE_X; i++) {
22 +   for (int j = 0; j < W_SIZE_Y; j++) {
23 +     printf("%s", w[i][j] ? " #" : " .");
24 +   }
25 +   printf("\n");
26 + }
27 + printf("\n");
28  }
29
30  void world_step(/* Recibo dos mundos */)
```

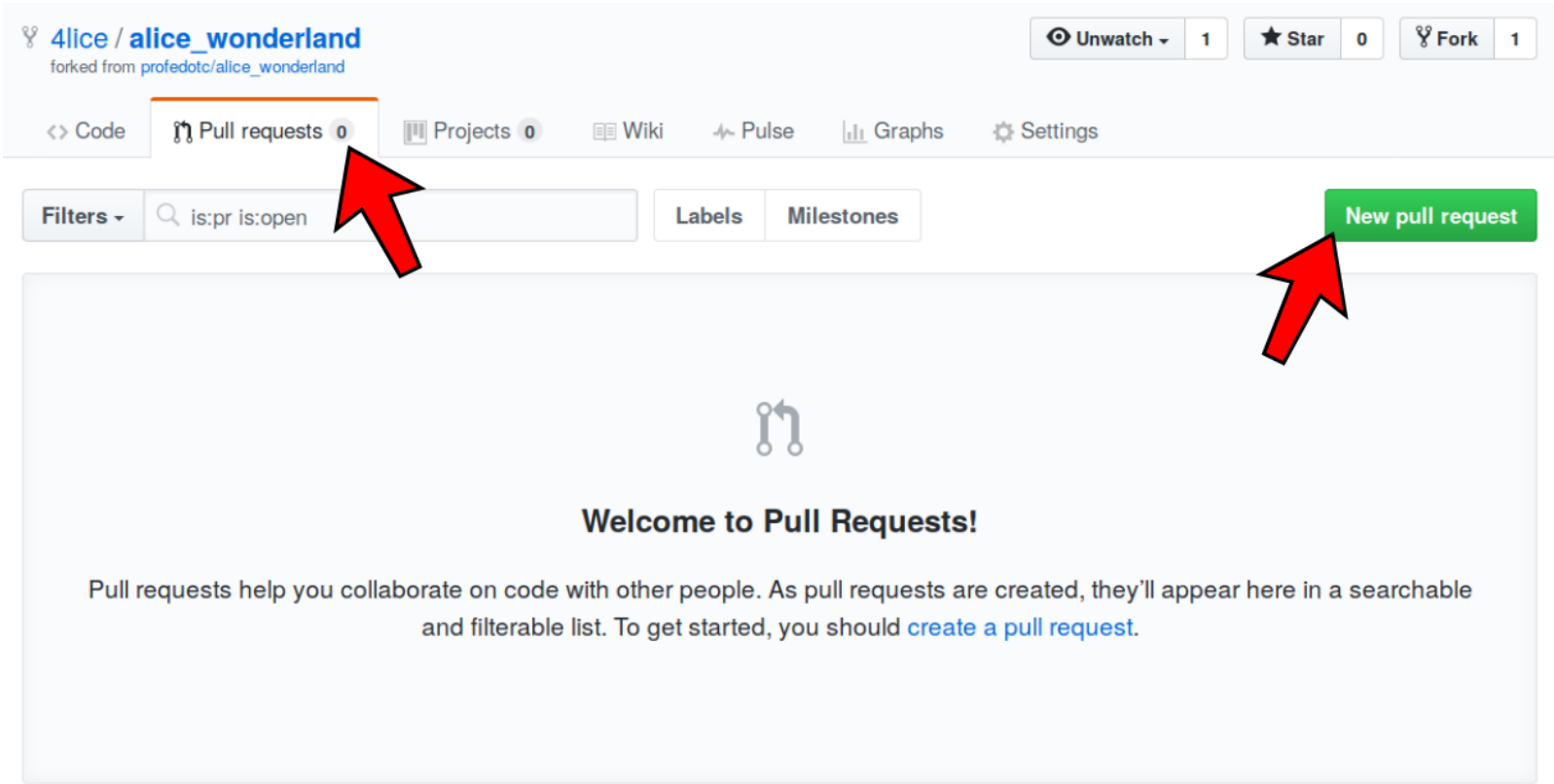
Sube los cambios a tu repositorio

Subimos los cambios con `git push` y observamos que aparecen en GitHub



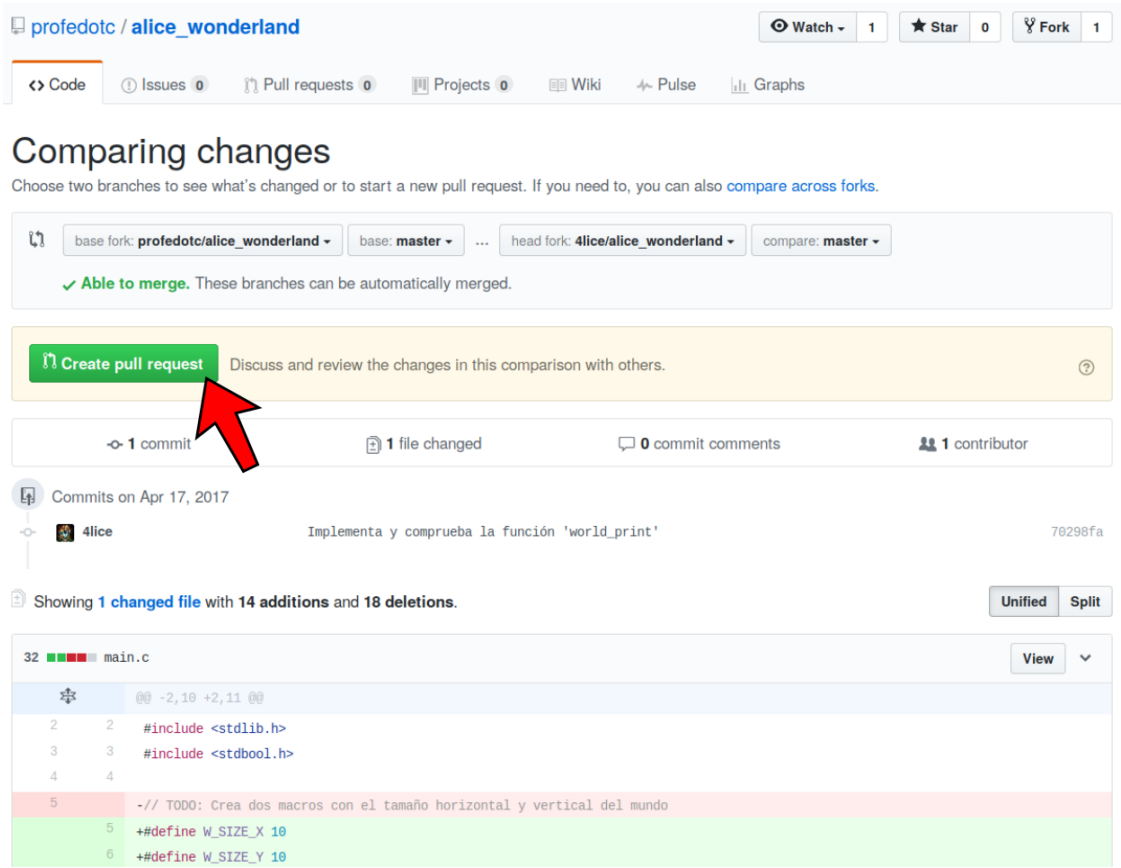
Crea un nuevo pull request

En la pestaña "*Pull requests*"pulsamos "*New pull request*" para crear un nuevo Pull Request



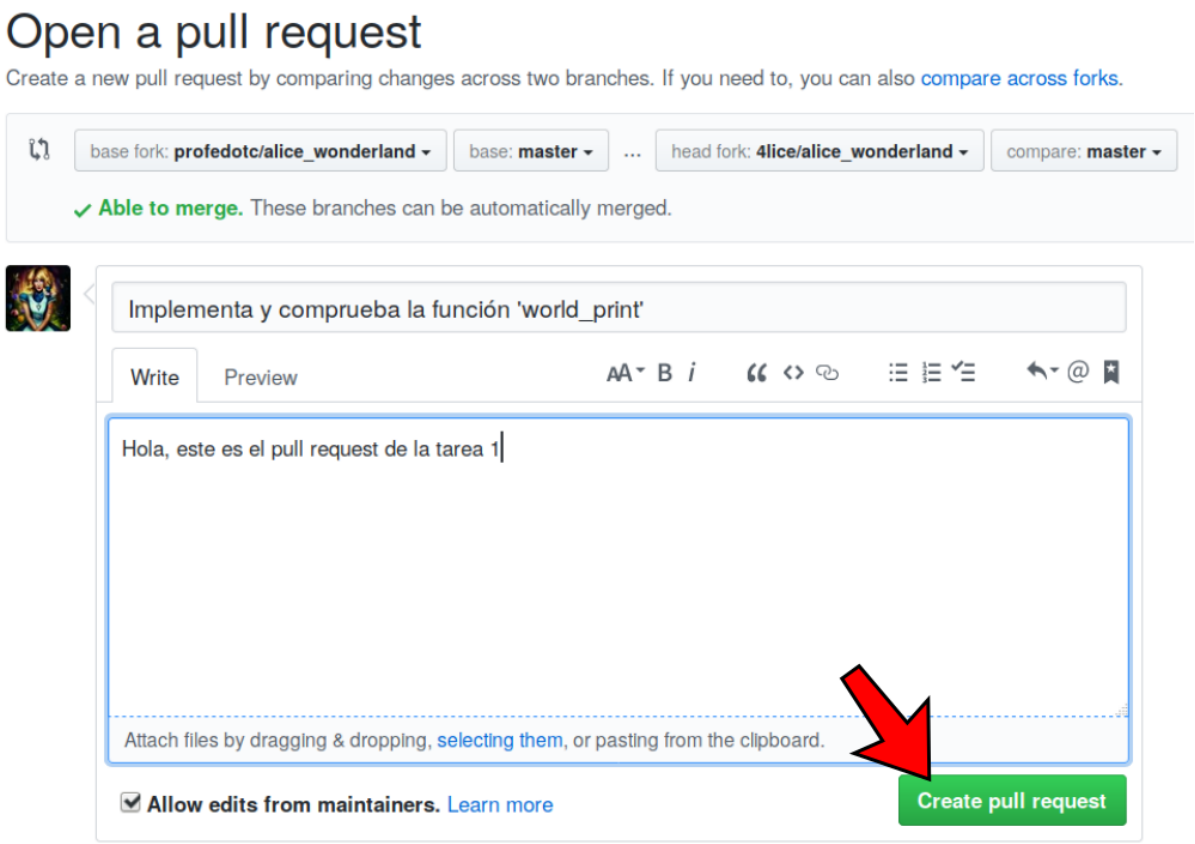
Comprobar los cambios

Volvemos a comprobar los diffs y pulsamos "*Create pull requests*"



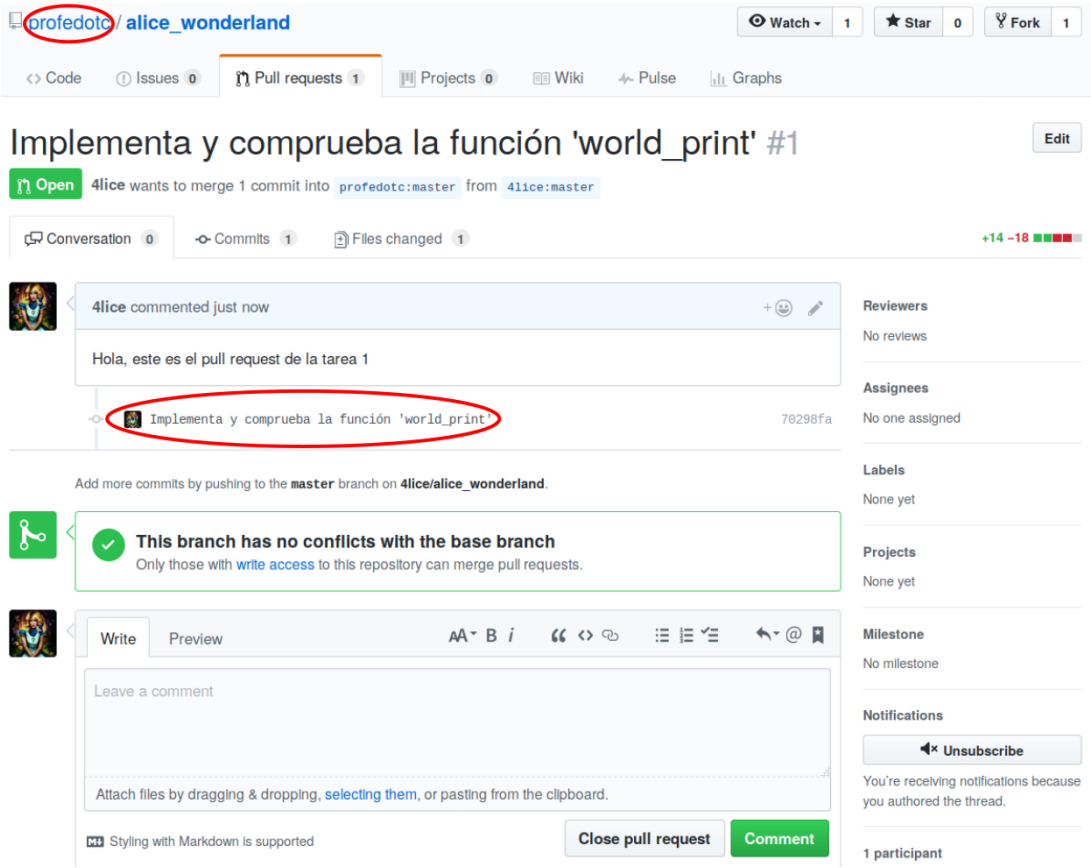
Descripción

Escribimos un pequeño texto indicando la tarea que se entrega y los cambios realizados



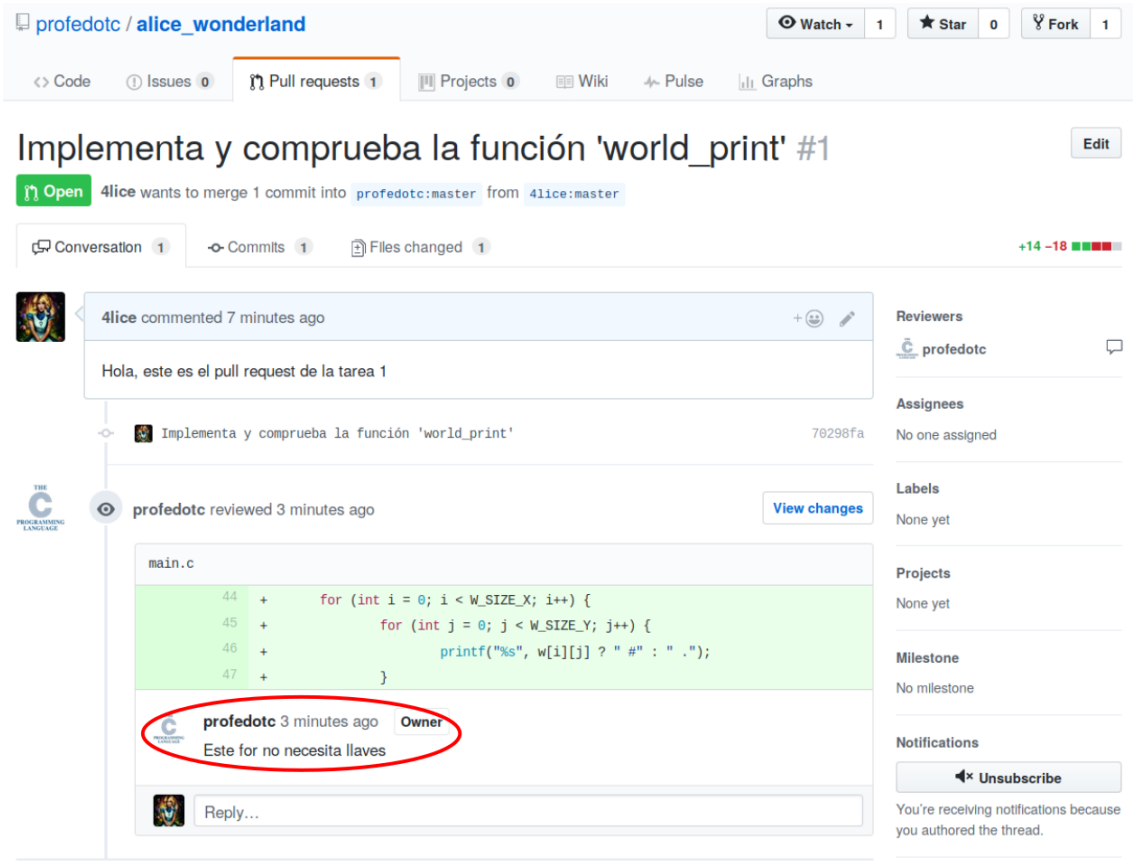
Solicitud terminada

Si todo ha ido bien, deberíamos ver una pantalla parecida a la siguiente:



Esperar las revisiones del profesor

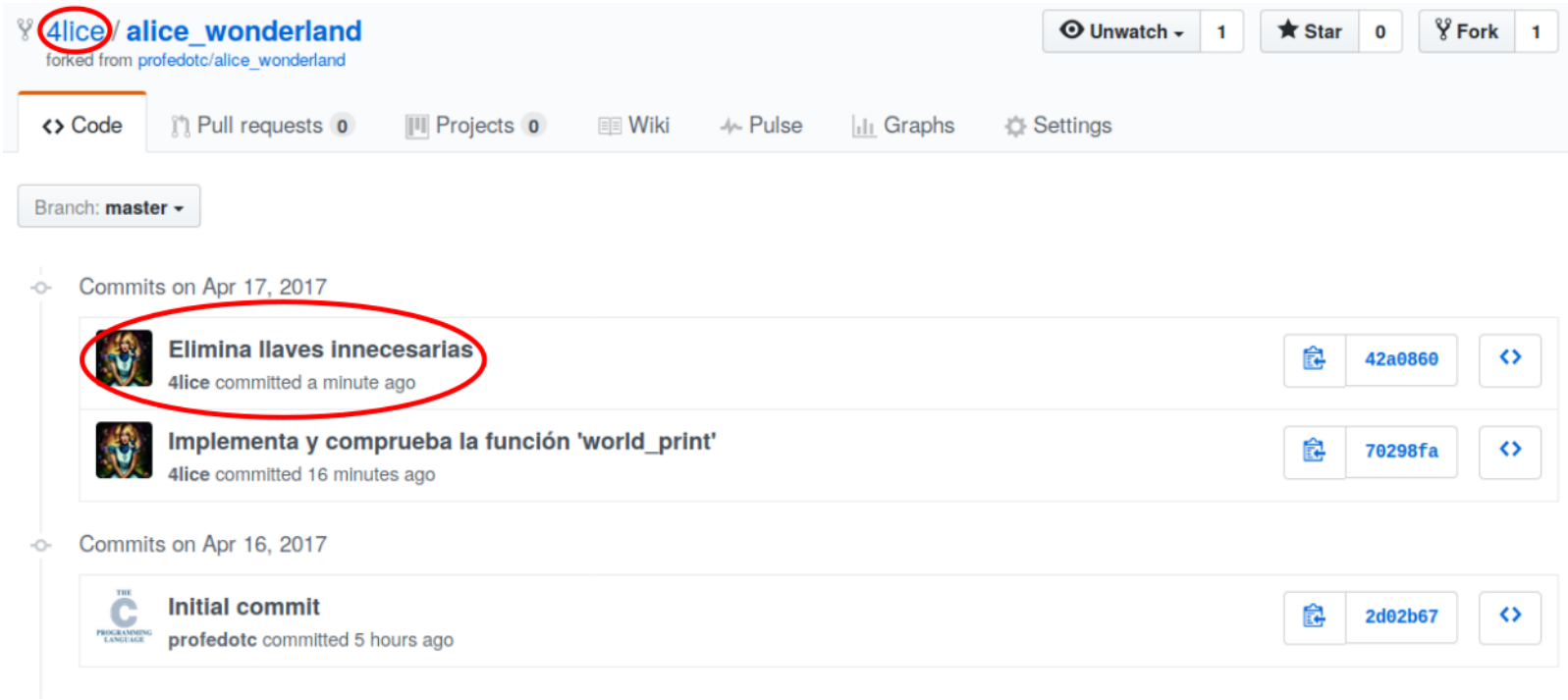
Cuando el profesor te haga correcciones, te llegará un correo y te aparecerán en la página del pull request



Correcciones

Crea un nuevo commit (o varios) para solucionar las correcciones que te hayan pedido

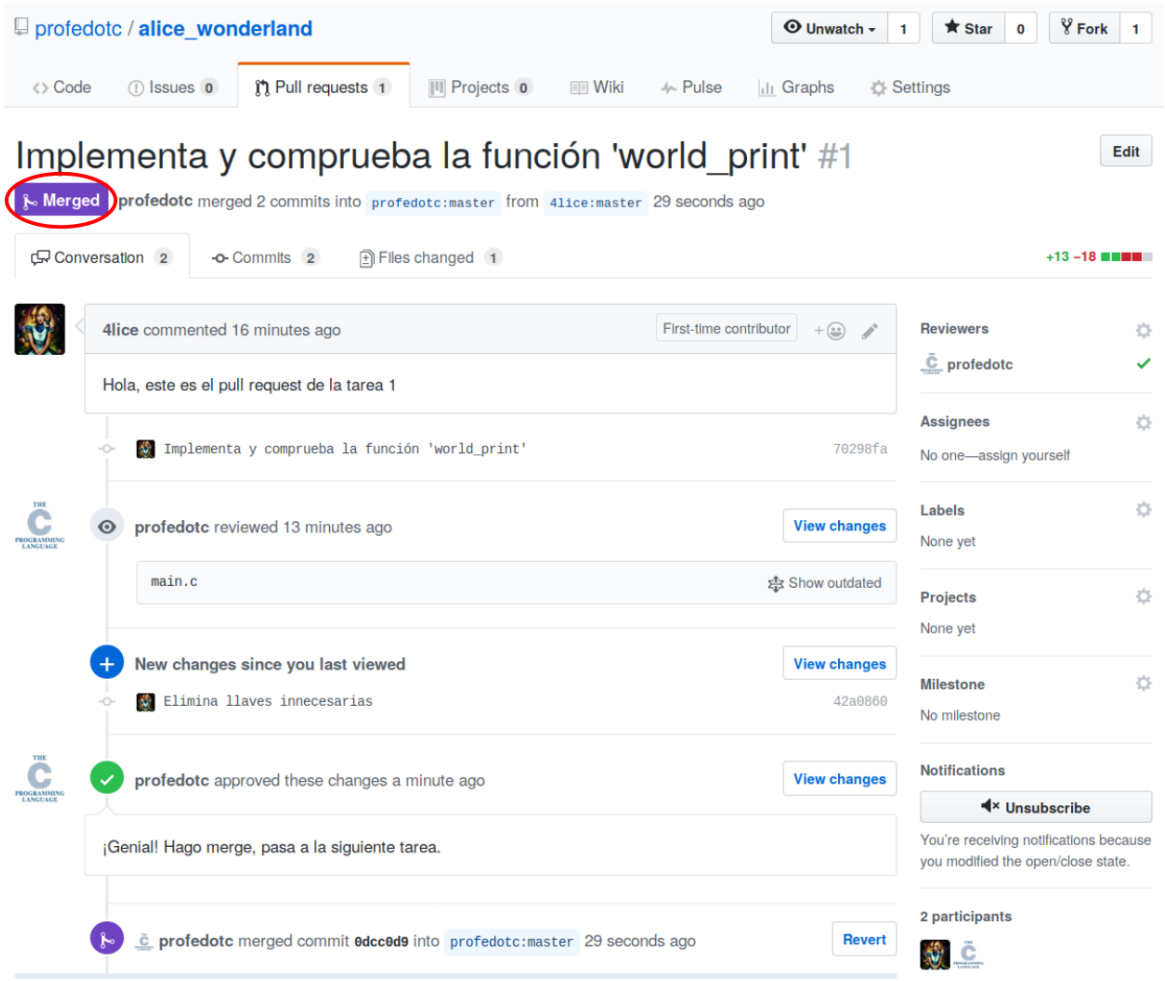
---



Correcciones

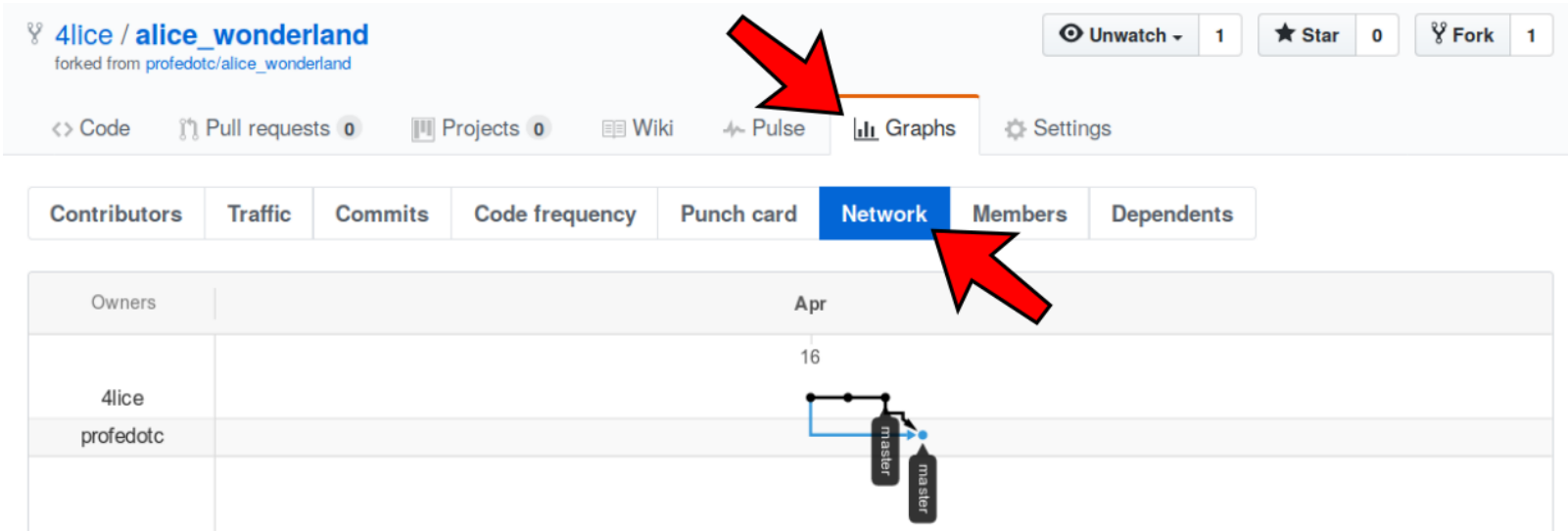
En la página del pull request deben aparecer los nuevos commits automáticamente

Espera a nuevas revisiones o a que sea aceptado



Network

Puedes ver tu pull request gráficamente en la sección "*Network*" de la pestaña "*Graphs*"



Estructuras de datos

Tema 9

Índice

Struct

Lista de variables agrupadas físicamente en un mismo bloque de memoria.

```c
struct person {
  char name[256];
  char surname[256];
  unsigned char age;
  unsigned int phone;
};

int main()
{
  struct person p = {"Man", "Bat", 35, 69813244};

  printf("%s%s\n", p.surname, p.name);

  return 0;
}
```

Alineación y tamaño


```
struct ejemplo
{
  uint8_t  v1; /* 1 */
  uint32_t v2; /* 4 */
  uint32_t v3; /* 4 */
};
```

sizeof(struct ejemplo);¿5 bytes?

Alineación y tamaño


```
struct ejemplo
{
  uint8_t  v1; /* 1 */
  uint32_t v2; /* 4 */
  uint32_t v3; /* 4 */
};
```

`sizeof(struct ejemplo);` ~~¿5 bytes?~~ →8 bytes

Anidamiento


```
1  struct A {
2    int a;
3    struct B {
4      int b;
5    } stb;
6  };
7
8  int main()
9  {
10   struct A a = {1, {2}};
11
12   printf("a = {%d, {%d}}\n", a.a, a.stb.b);
13
14   return 0;
15 }
```

Estructuras anónimas


```c
struct A {
  int a;
  struct {
    int b;
  };
};

int main()
{
  struct A a = {1, {2}};

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

  return 0;
}
```

Arrays y punteros


```
1  #include <stdio.h>
2
3  struct cell {
4    int state;
5    int size;
6  };
7
8  int main()
9  {
10   struct cell culture[5] = {{1,10}, {0,342}, {1,7},
         {1,50}, {1,77}};
11   struct cell *c;
12
13   for (c = culture; c <= &culture[4]; c++)
14     printf("{%d, %d}\n", c->state, c->size);
15
16   return 0;
17 };
```

Union

- Una unión es un valor que tiene varias representaciones o formatos
- Estructura que permite guardar varios tipos de datos en la misma zona de memoria

```
1  union float_int {
2    float f;
3    int   i;
4  };
5
6  int main()
7  {
8    union float_int fi;
9
10   fi.f = 2.7182;
11   printf("%X\n", fi.i);
12
13   return 0;
14 }
```

Ejemplo 1


```c
union char_int {
  struct {
    char c1;
    char c2;
    char c3;
    char c4;
  };
  int   i;
};

int main()
{
  union char_int ci;

  ci.i = 1701999205;
  printf("%c%c%c%c\n", ci.c1, ci.c2, ci.c3, ci.c4);

  return 0;
}
```

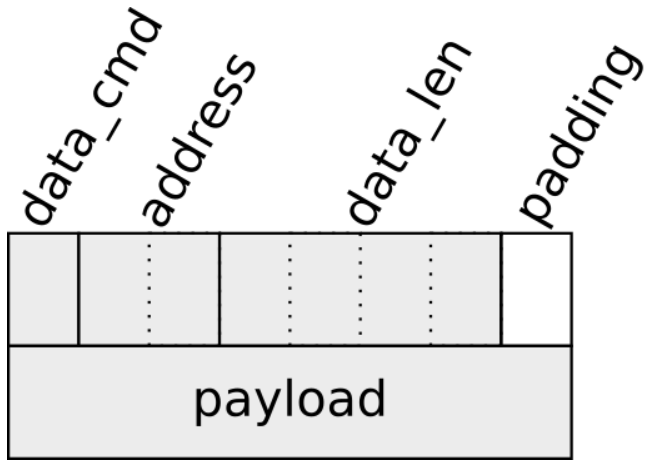
Ejemplo 2


```c
#include <stdio.h>

struct gtype {
  int type;

  union {
    char char_t;
    int int_t;
    float float_t;
  };
};

int main()
{
  struct gtype gt;

  gt.int_t = 3;
  gt.type = 1;

  switch (gt.type) {
  case 0:
    printf("%c\n", gt.char_t);
    break;
  case 1:
    printf("%d\n", gt.int_t);
    break;
  case 2:
    printf("%f\n", gt.float_t);
    break;
  default:
    printf("error: invalid type
        \n");
    break;
  };

  return 0;
}
```

Campos de bits

- Característica de las estructuras y uniones que nos permite declarar campos de hasta un bit de longitud
- La memoria reservada es la que indica el tipo del campo
- Para acceder a nivel de bit se realizan numerosas operaciones por debajo

```
struct frame {
  uint16_t data_cmd  : 1;
  uint16_t address   : 2;
  uint16_t data_len  : 4;
  uint16_t           : 1;
  uint16_t payload   : 8;
};
```



Ejemplo


```c
union float_t{
  float f;
  struct {
    uint32_t fra : 23;
    uint32_t exp : 8;
    uint32_t sig : 1;
  };
};

int main()
{
  union float_t f;

  f.f = -3.1416;

  printf("signo      = %u\n", f.sig);
  printf("exponente = %d\n", f.exp);
  printf("fraccion   = %d\n", f.fra);

  return 0;
}
```

Macros: El preprocesador de C

- Preprocesador: Se ejecuta antes de compilar
- Lenguaje de **macros**
- Multiples usos:
  - Declaración de constantes
  - Pequeñas funciones y utilidades
  - Compilación condicional de código
  - Depuración

Ejemplos


```c
#include <stdio.h>

#define TAM_ARRAY 20
#define POW2(x) ((x)*(x))
#define PRINT 0

int main()
{
   int array[TAM_ARRAY];
   int i;

   for (i = 0; i < TAM_ARRAY; i++)
     array[i] = POW2(i);

#if PRINT == 1
   for (i = 0; i < TAM_ARRAY; i++)
     printf("%i ", array[i]);
#elif PRINT == -1
   printf("Array initialized\n");
#else
  #warning PRINT may be 1 or -1
   printf("Error in %s:%d\n", __FILE__, __LINE__);
#endif

   return 0;
}
```

Enum


```
enum estado_coche {ARRANCADO, PARADO, EN_MARCHA,
DETENIDO};
```

- **Tipo** formado por una lista de macros
- Las macros toman valores enteros de forma consecutiva

Ejemplo


```c
#include <stdio.h>

struct person {
  char name[256];
  char surname[256];
  unsigned char age;
  unsigned int phone;
};

enum person_attr {
  NAME,
  SURNAME,
  AGE,
  PHONE
};

int main ()
{
  person p = {"Alice", "Smith",
        25, 12434321};
  enum person_attr choice =
       NAME;

  switch (choice) {
  case NAME:
    printf("%s\n", p.name);
    break;
  case SURNAME:
    printf("%s\n", p.surname);
    break;
  case AGE:
    printf("%d\n", p.age);
    break;
  case PHONE:
    printf("%d\n", p.phone);
    break;
  default:
    printf("error: %s:%d",
           __FILE__, __LINE__);
  }

  return 0;
}
```

Designated initializers


```
1  struct bug {
2    unsigned int legs;
3    unsigned char color[3];
4    const char *name;
5  } bugs[30] = {
6    [0] = {
7      .legs = 6,
8      .color = {100, 100, 100},
9      .name = "ant",
10   },
11   [1] = {
12     .legs = 8,
13     .color = {[0] = 200},
14     .name = "spider",
15   },
16   [2 ... 10] = { // Only GNU
17     .legs = 100,
18     .color = {[1] = 255},
19     .name = "centipede",
20   },
21 };
```

C Modular

Tema 10

Índice

Cabeceras

- Podemos crear nuestros propios ficheros **.h*
- Un fichero de cabecera suele tener únicamente declaraciones y no definiciones
- En el fichero de código (**.c*) se definen las funciones declaradas en la cabecera
- Podemos tener una cabecera y varios tipos de definiciones
- Nos permite crear una interfaz que aisle al usuario de la definición de las funciones

Ejemplo

person.h

```
1  #ifndef _PERSON_H_
2  #define _PERSON_H_
3
4  #define MAX_NAME 256
5
6  struct person {
7    char name[MAX_NAME];
8    char surname[MAX_NAME];
9    unsigned int age;
10   int phone;
11 };
12
13 void print_person(const struct person *p);
14
15 #endif
```

person.c

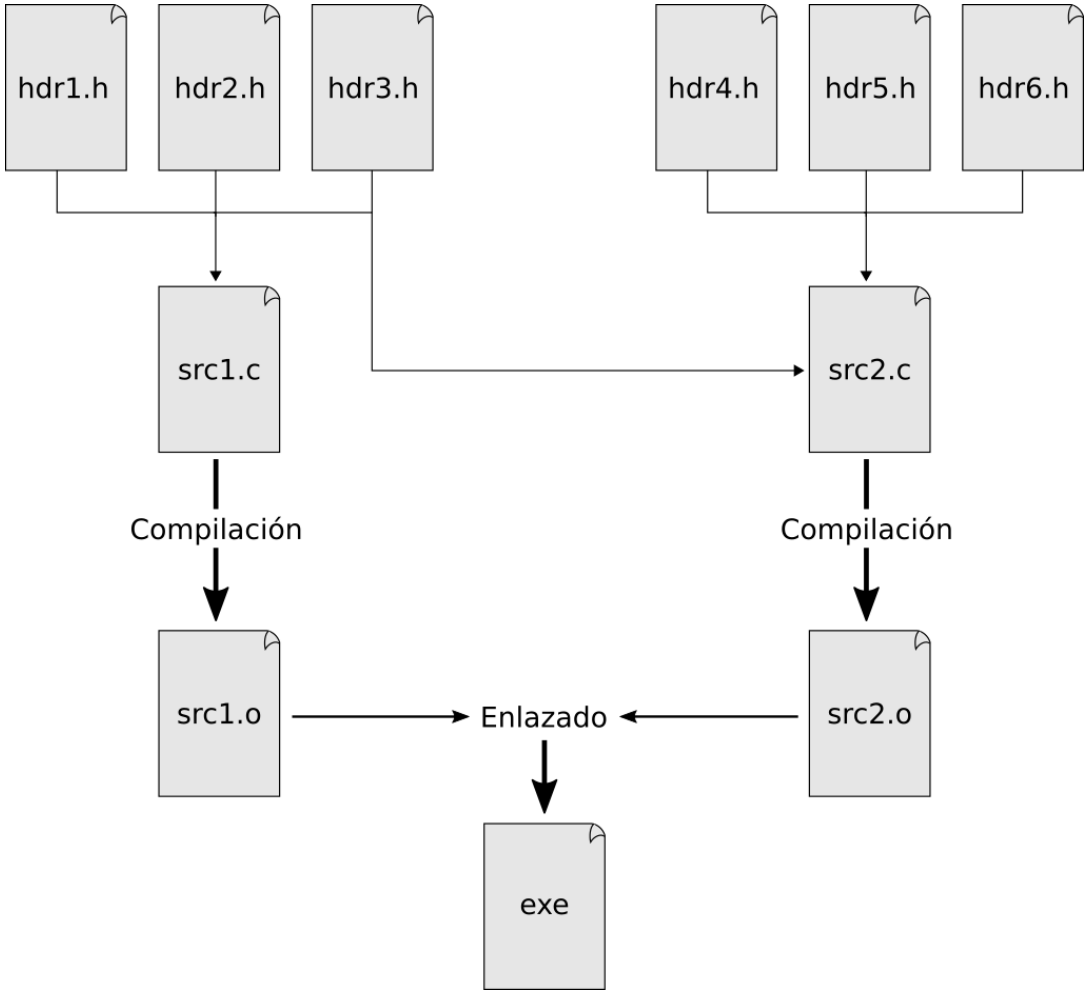
```
1  #include <stdio.h>
2  #include "person.h"
3
4  void print_person(const struct person *p)
5  {
6    printf("%s, %s\n", p->surname, p->name);
7    printf("\t- age:   %d\n", p->age);
8    printf("\t- phone: %d\n", p->phone);
9  }
```

Compilación por bloques

Razones:

- Tiempo:
  - La compilación es un proceso complejo y costoso
  - Proyectos muy grandes pueden tardar mucho tiempo en compilar
  - Cuando se está desarrollando esto se vuelve prohibitivo
- Organización:
  - Dividir el código en bloques lógicos es una buena práctica
  - Mejora:
    - El mantenimiento
    - La legibilidad
    - La portabilidad
    - La escalabilidad
    - etc

Compilación por bloques



Compilación por bloques

Solución:

- Cada fichero de código puede compilarse de manera independiente, creando un **fichero objeto** (**.o*)
- El **linker** se encarga de enlazar todos los ficheros objetos para crear el ejecutable
- Un módulo objeto puede tener referencias a símbolos definidos en otro módulo

Cómo se genera: `gcc -c persona.c`

Make y Makefile

Herramienta para automatizar la compilación del código (y mucho más)

- Gestiona dependencias para no compilar innecesariamente
- Facilita enormemente el trabajo
- También se suele utilizar para instalación y desinstalación

Estructura


```
objetivo: prerequisito1 prerequisito2 ...
  ordenes para generar "objetivo"
```

- **Objetivos**: Un objetivo suele ser un archivo que se desea generar (por ejemplo, un ejecutable)
- **Prerequisitos**: Lista de objetivos
- **Órdenes**: Instrucciones a realizar para generar el objetivo. Siempre van precedidas de una tabulación

Ejemplo

makefile

```
1  all: mi_prog
2
3  mi_prog: main.o persona.o
4    gcc main.o persona.o -o mi_prog
5
6  main.o: main.c
7    gcc -c main.c
8
9  persona.o: persona.h persona.c
10   gcc -c persona.c
```

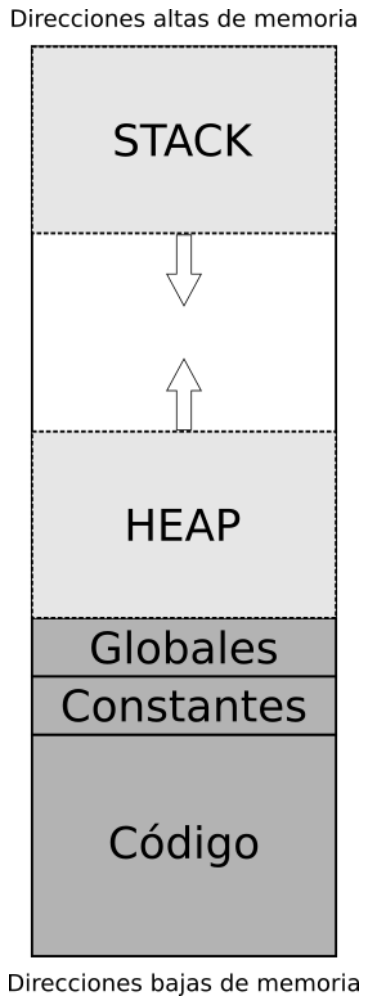
Reserva dinámica de memoria

Tema 11

Índice

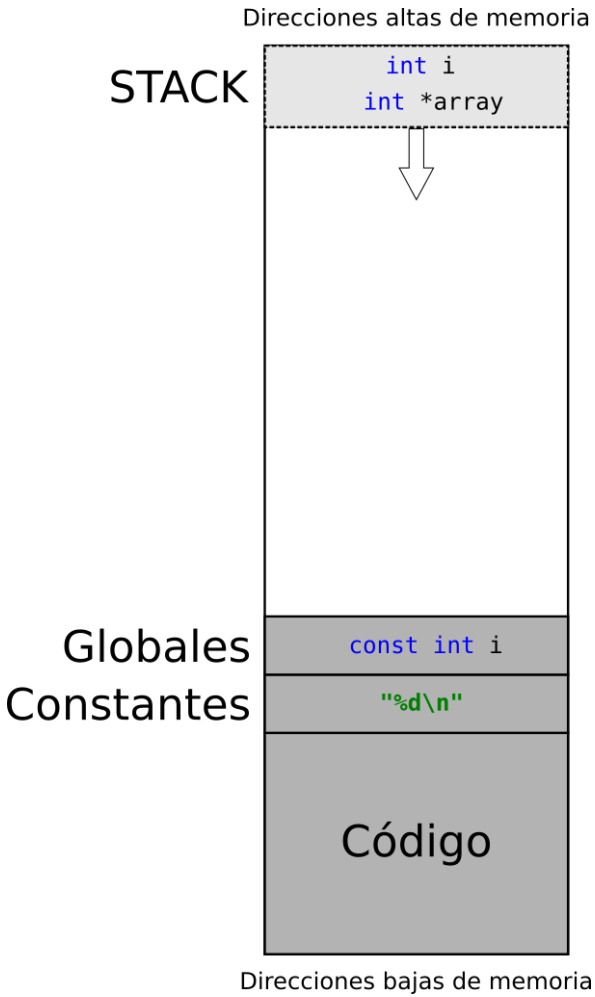
Mapa de memoria

- **Heap**: Se almacena la memoria reservada **dinámicamente** con **malloc**
- **Stack**: Se almacenan las **variables locales** de cada llamada a función
- **Globales**: Todas las variables globales
- **Constantes**: Todas las constantes (números, cadenas, etc)
- **Código**: El programa en sí



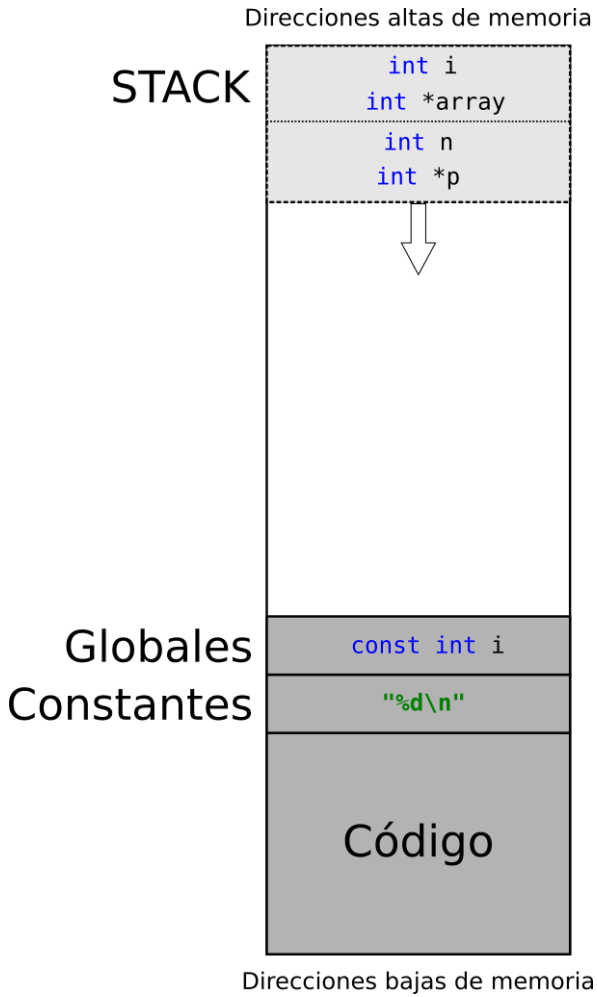
Ejemplo


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

int *array_alloc(int n)
{
  int *p = (int *)malloc(n * sizeof(int))
  return p;
}

static const int tam = 3;

int main()
{
  int i;
  int *array;

  array = array_alloc(tam);

  /* Inicializamos */
  for (i = 0; i < tam; i++)
    array[i] = i;

  /* Imprimimos */
  for (i = 0; i < tam; i++)
    printf("%d\n", array[i]);

  free(array);

  return 0;
}
```



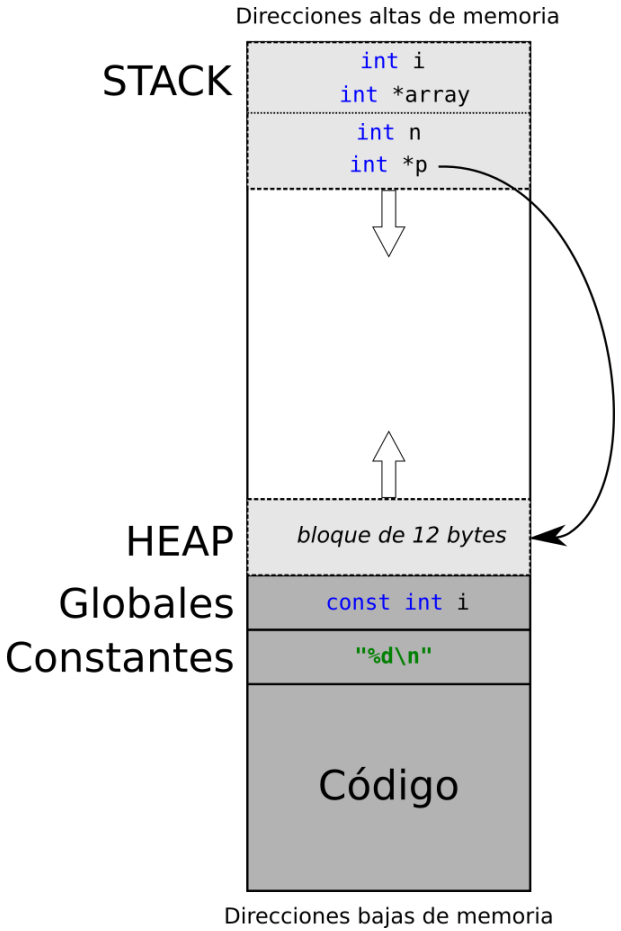
Ejemplo


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

int *array_alloc(int n)
{
  int *p = (int *)malloc(n * sizeof(int))
  return p;
}

static const int tam = 3;

int main()
{
  int i;
  int *array;

  array = array_alloc(tam);

  /* Inicializamos */
  for (i = 0; i < tam; i++)
    array[i] = i;

  /* Imprimimos */
  for (i = 0; i < tam; i++)
    printf("%d\n", array[i]);

  free(array);

  return 0;
}
```



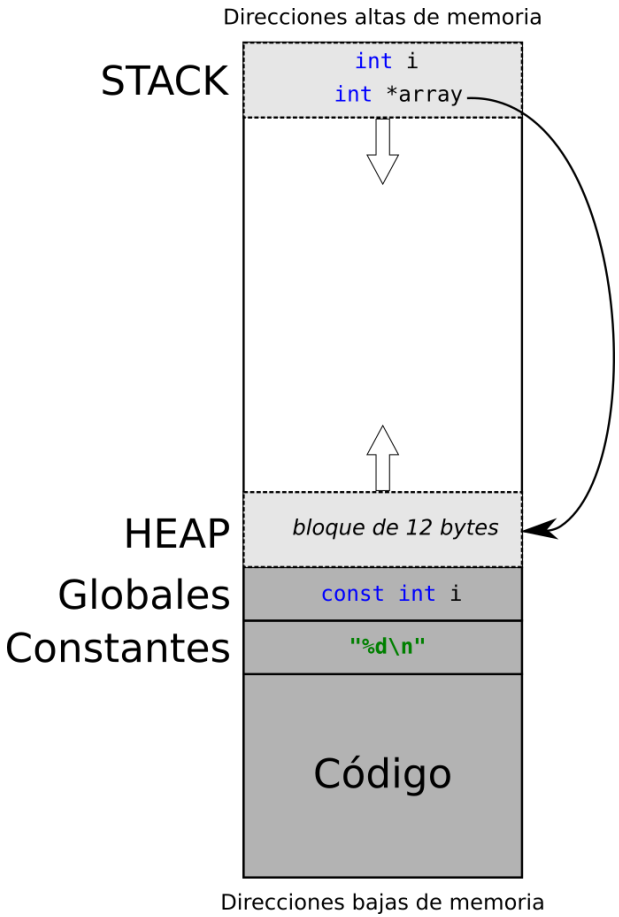
Ejemplo


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

int *array_alloc(int n)
{
  int *p = (int *)malloc(n * sizeof(int))
  return p;
}

static const int tam = 3;

int main()
{
  int i;
  int *array;

  array = array_alloc(tam);

  /* Inicializamos */
  for (i = 0; i < tam; i++)
    array[i] = i;

  /* Imprimimos */
  for (i = 0; i < tam; i++)
    printf("%d\n", array[i]);

  free(array);

  return 0;
}
```



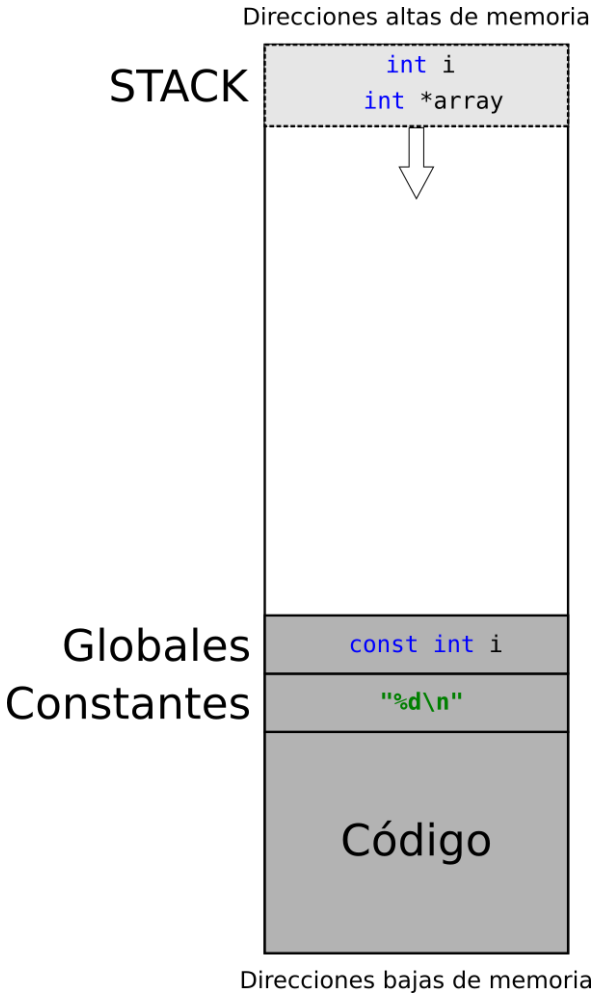
Ejemplo


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

int *array_alloc(int n)
{
  int *p = (int *)malloc(n * sizeof(int))
  return p;
}

static const int tam = 3;

int main()
{
  int i;
  int *array;

  array = array_alloc(tam);

  /* Inicializamos */
  for (i = 0; i < tam; i++)
    array[i] = i;

  /* Imprimimos */
  for (i = 0; i < tam; i++)
    printf("%d\n", array[i]);

  free(array);

  return 0;
}
```



Ejemplo

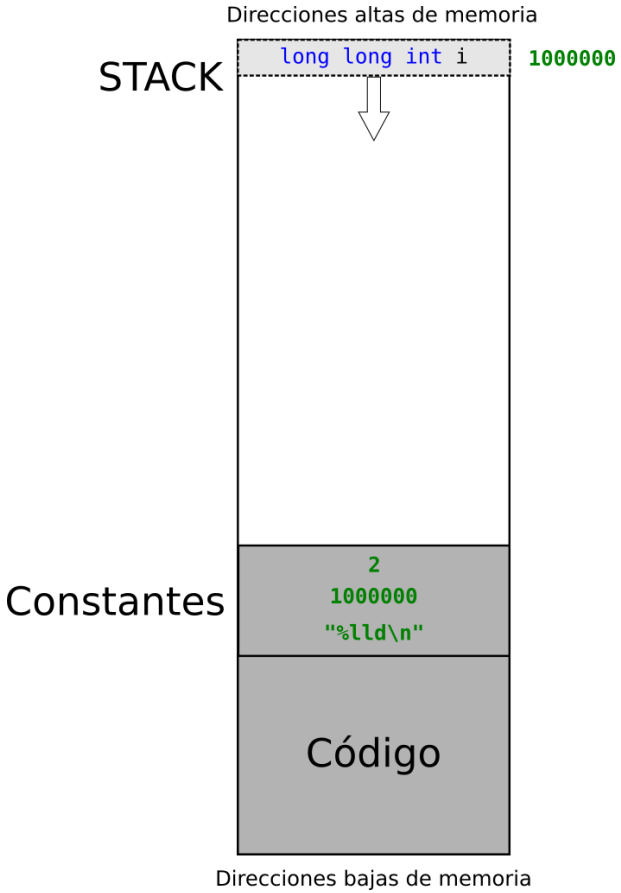

```c
#include <stdio.h>
#include <stdlib.h>

int *array_alloc(int n)
{
  int *p = (int *)malloc(n * sizeof(int))
  return p;
}

static const int tam = 3;

int main()
{
  int i;
  int *array;

  array = array_alloc(tam);

  /* Inicializamos */
  for (i = 0; i < tam; i++)
    array[i] = i;

  /* Imprimimos */
  for (i = 0; i < tam; i++)
    printf("%d\n", array[i]);

  free(array);

  return 0;
}
```



Stack Overflow

El uso de funciones recursivas puede ocasionar un agotamiento de la pila

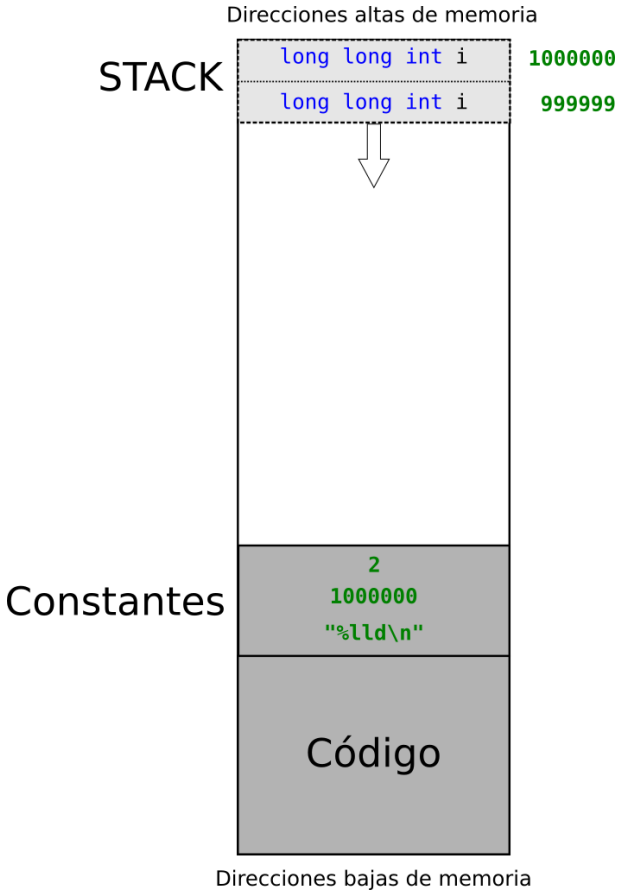
```
1  #include <stdio.h>
2
3  long long int sum_all(long long int i)
4  {
5    if (i >= 2)
6      return i + sum_all(i - 1);
7    else
8      return i;
9  }
10
11 int main()
12 {
13   printf("%lld\n", sum_all(1000000));
14
15   return 0;
16 }
```



Stack Overflow

El uso de funciones recursivas puede ocasionar un agotamiento de la pila

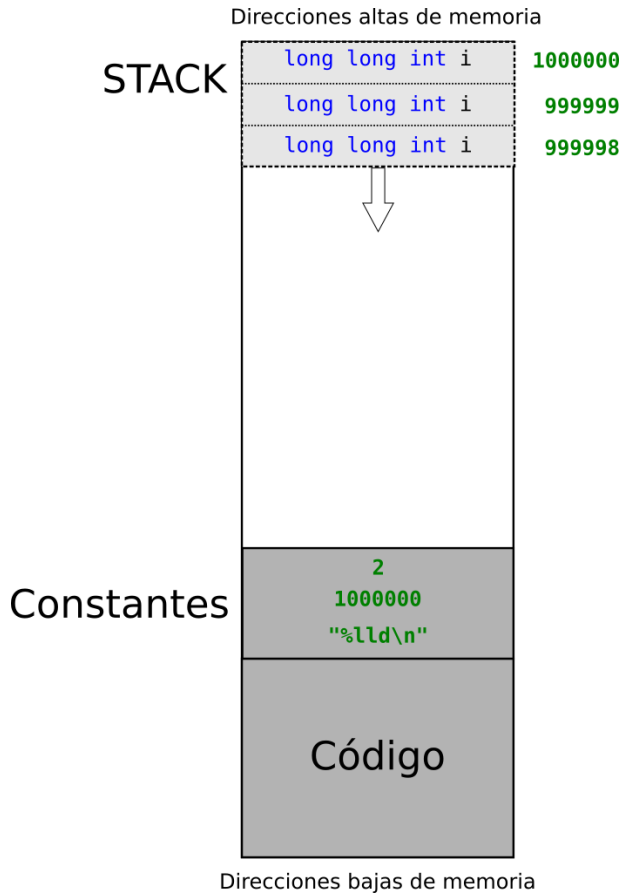
```
1  #include <stdio.h>
2
3  long long int sum_all(long long int i)
4  {
5    if (i >= 2)
6      return i + sum_all(i - 1);
7    else
8      return i;
9  }
10
11 int main()
12 {
13   printf("%lld\n", sum_all(1000000));
14
15   return 0;
16 }
```



Stack Overflow

El uso de funciones recursivas puede ocasionar un agotamiento de la pila

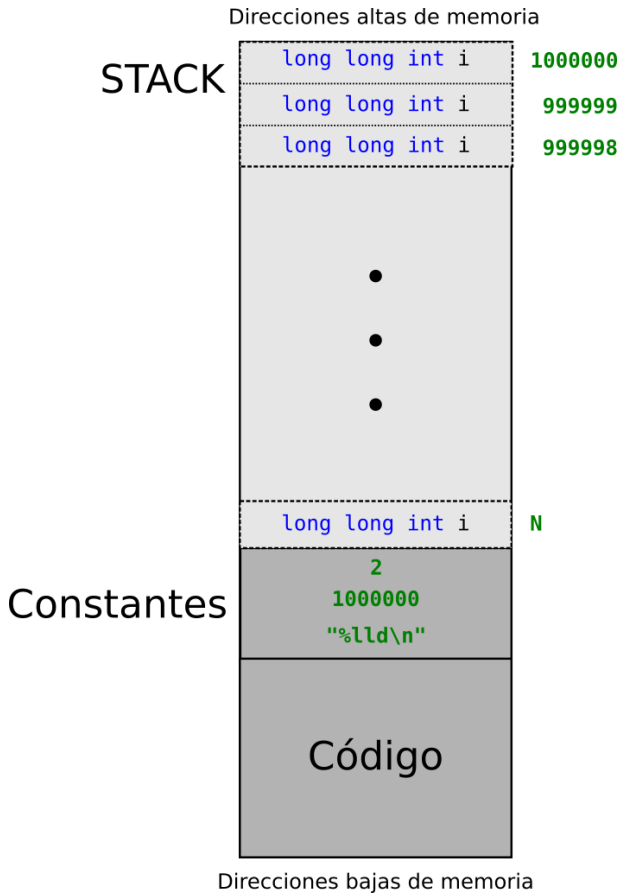
```c
#include <stdio.h>

long long int sum_all(long long int i)
{
  if (i >= 2)
    return i + sum_all(i - 1);
  else
    return i;
}

int main()
{
  printf("%lld\n", sum_all(1000000));

  return 0;
}
```



Stack Overflow

El uso de funciones recursivas puede ocasionar un agotamiento de la pila

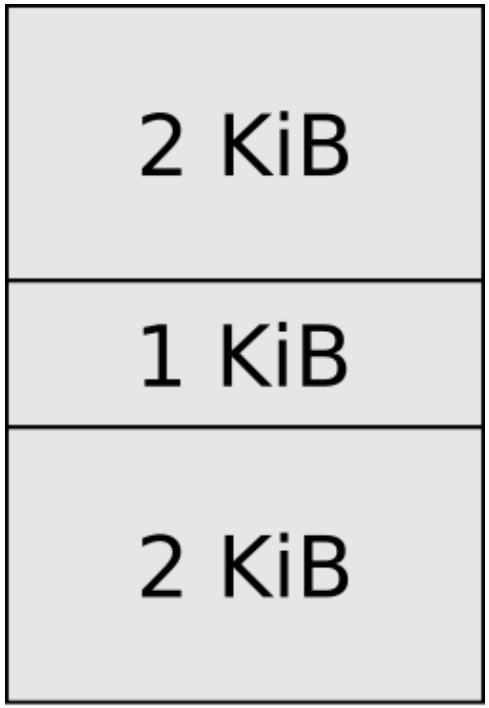
```
1  #include <stdio.h>
2
3  long long int sum_all(long long int i)
4  {
5    if (i >= 2)
6      return i + sum_all(i - 1);
7    else
8      return i;
9  }
10
11 int main()
12 {
13   printf("%lld\n", sum_all(1000000));
14
15   return 0;
16 }
```



Fragmentación

Fragmentos pequeños de memoria libre entre bloques de memoria reservada.

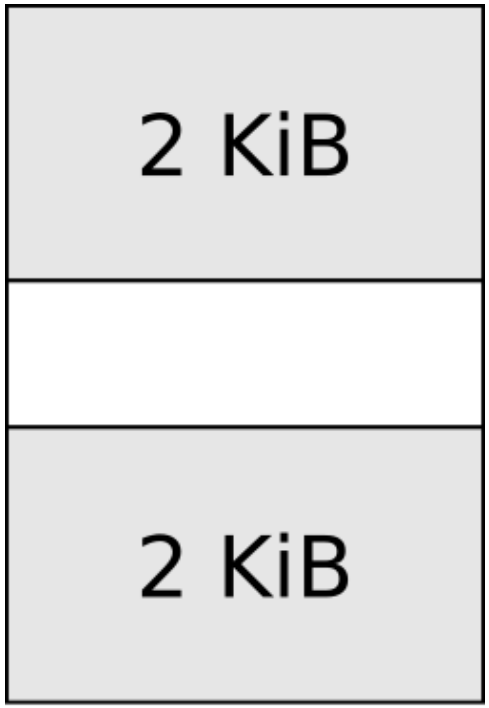
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   void *p1 = malloc(2 * 1024);
   void *p2 = malloc(1 * 1024);
   void *p3 = malloc(2 * 1024);

   void *p;

   printf("%ld\n", (long int)p1 / 1024);
   printf("%ld\n", (long int)p2 / 1024);
   printf("%ld\n", (long int)p3 / 1024);

   free(p2);
   p = malloc(2 * 1024);

   printf("\n%ld\n", (long int)p / 1024);

   return 0;
}
```



Fragmentación

Fragmentos pequeños de memoria libre entre bloques de memoria reservada.

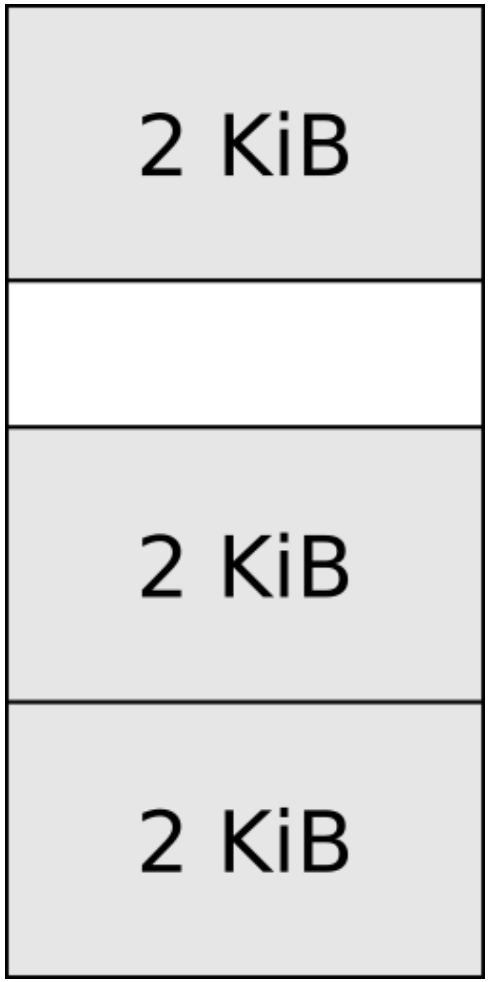
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
   void *p1 = malloc(2 * 1024);
   void *p2 = malloc(1 * 1024);
   void *p3 = malloc(2 * 1024);

   void *p;

   printf("%ld\n", (long int)p1 / 1024);
   printf("%ld\n", (long int)p2 / 1024);
   printf("%ld\n", (long int)p3 / 1024);

   free(p2);
   p = malloc(2 * 1024);

   printf("\n%ld\n", (long int)p / 1024);

   return 0;
}
```



Fragmentación

Fragmentos pequeños de memoria libre entre bloques de memoria reservada.

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

int main()
{
   void *p1 = malloc(2 * 1024);
   void *p2 = malloc(1 * 1024);
   void *p3 = malloc(2 * 1024);

   void *p;

   printf("%ld\n", (long int)p1 / 1024);
   printf("%ld\n", (long int)p2 / 1024);
   printf("%ld\n", (long int)p3 / 1024);

   free(p2);
   p = malloc(2 * 1024);

   printf("\n%ld\n", (long int)p / 1024);

   return 0;
}
```



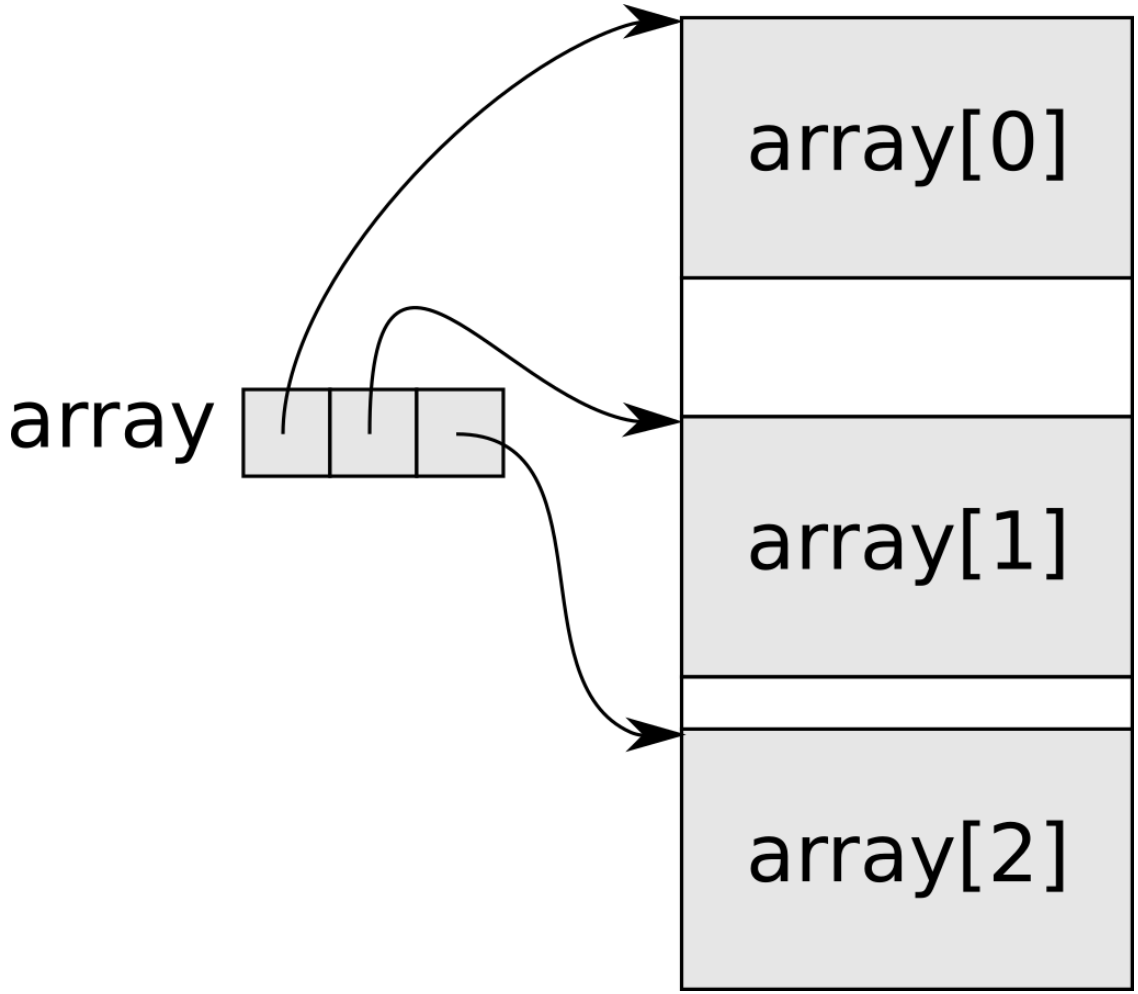
Forma 1 (la mala)

Un malloc por cada fila del array (array de punteros)

```
1  int main()
2  {
3    int i, j;
4    int **array;
5
6    /* Reservamos */
7    array = (int **)malloc(ROWS * sizeof(int *));
8    for (i = 0; i < ROWS; i++)
9      array[i] = (int *)malloc(COLS * sizeof(int)); // Falta comprobar
10
11   /* Inicializamos */
12   for (i = 0; i < ROWS; i++)
13     for (j = 0; j < ROWS; j++)
14       array[i][j] = i * 10 + j;
15
16   /* Imprimimos */
17   for (i = 0; i < ROWS; i++) {
18     for (j = 0; j < ROWS; j++) {
19       printf("%02d ", array[i][j]);
20     }
21     printf("\n");
22   }
23
24   /* Liberamos */
25   for (i = 0; i < ROWS; i++)
26     free(array[i]);
27
28   return 0;
29 }
```

Forma 1 (la mala)

No se garantiza continuidad entre los bloques reservados



Forma 2 (la buena)


```c
#define ROWS 3
#define COLS 3

int main()
{
  int i, j;
  int *array;

  /* Reservamos */
  array = (int *)malloc(ROWS * COLS * sizeof(int));
  if (!array) {
    printf("Can't allocate the array\n");
    return -1;
  }

  /* Inicializamos */
  for (i = 0; i < ROWS; i++)
    for (j = 0; j < COLS; j++)
      *(array + i * COLS + j) = i * 10 + j;

  /* Imprimimos */
  for (i = 0; i < ROWS; i++) {
    for (j = 0; j < COLS; j++) {
      printf("%02d ", *(array + i * COLS + j));
    }
    printf("\n");
  }

  free(array); /* Liberamos */
  return 0;
}
```

Uso tras liberación


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

struct person
{
  char name[200];
  char surname[200];
  unsigned char age;
};

int main()
{
  struct person *p;

  p = (struct person *)malloc(sizeof(struct person));

  strcpy(p->name, "Bob");
  strcpy(p->surname, "Smith");
  p->age = 20;

  free(p);

  printf("name    = %s\n", p->name);
  printf("surname = %s\n", p->surname);
  printf("age     = %d\n", p->age);

  return 0;
}
```

Depuración con valgrind

- Valgrind es una herramienta que nos avisa de los errores cometidos en el manejo de memoria
- Un **leak** o fuga de memoria es un error de programación que hace que la memoria reservada no se libere cuándo ya no se usa. Un programa que no libere correctamente la memoria reservada puede acabar consumiendo toda la memoria del sistema y dejarlo inutilizado.
- Para que valgrind nos muestre más información podemos hacer dos cosas:
  1. Compilar con símbolos de depuración (`gcc -g`)
  2. Ejecutar valgrind con el flag `-leak-check=full`

Ejemplo


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

int main()
{
  int *p = (int *)malloc(20);

  if (!p)
    return EXIT_FAILURE;

  printf("La direccion de p es %p\n", p);

  /* free(p) */

  return 0;
}
```

Ejemplo


```
gcc -g valgrind.c -o valg_ej
valgrind --leak-check=full valg_ej
```

```
1  Memcheck, a memory error detector
2  ....
3  Command: valg_ej
4
5  La direccion de p es 0x51d7040
6
7  HEAP SUMMARY:
8      in use at exit: 20 bytes in 1 blocks
9    total heap usage: 1 allocs, 0 frees, 20 bytes allocated
10
11 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
12    at 0x4C2ABD0: malloc (in /usr/lib/...)
13    by 0x4004F7: main (valgrind.c:6)
14
15 LEAK SUMMARY:
16    definitely lost: 20 bytes in 1 blocks
17    indirectly lost: 0 bytes in 0 blocks
18      possibly lost: 0 bytes in 0 blocks
19    still reachable: 0 bytes in 0 blocks
20         suppressed: 0 bytes in 0 blocks
21
22 For counts of detected and suppressed errors, rerun with: -v
23 ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```