Introducción

Tema 1

Introducción

El curso
Profesores
Temario
Material de clase
Herramientas
Flujo de trabajo
Evaluación
Historia
Inicios
Influencias
¿Por qué C?
¿Para qué C?
Proyectos en C
Sumer Of Code
GSOC
Outreachy

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

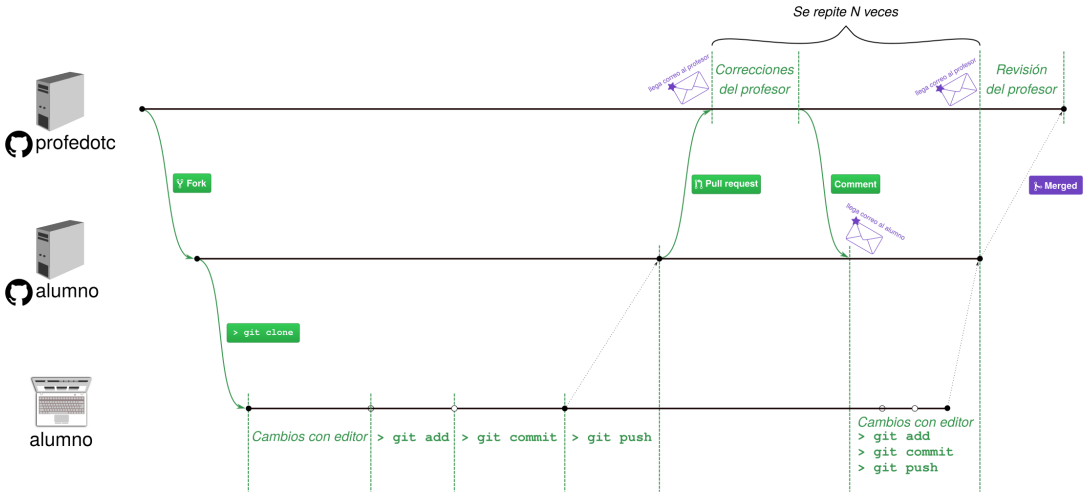
Lista de correo

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

Herramientas

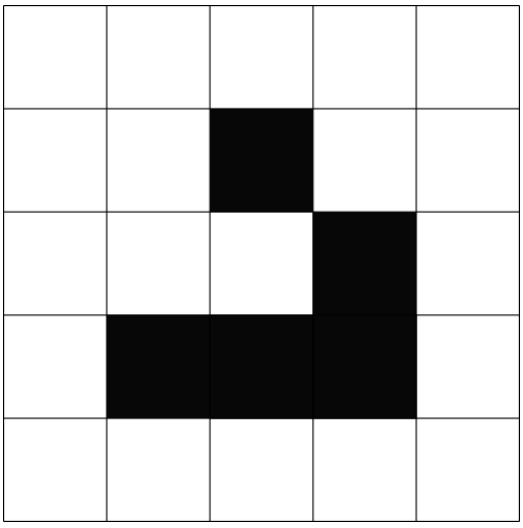
- Linux y terminal
- Editor de texto (*geany*)
- Compilador GCC
- *Make* para automatizar la compilación
- Repositorio de código (*git* y *GitHub*)

Flujo de trabajo



Evaluación

Juego de la vida de Conway



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

Inicios



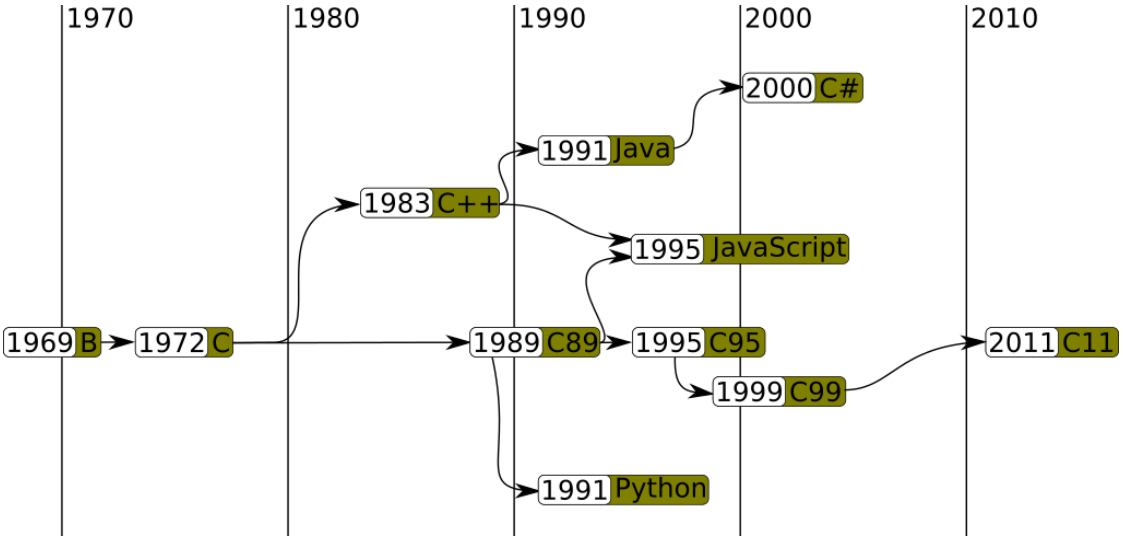
Ken Thompson
Dennis Ritchie
Brian Kernighan

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.

Introducción

El curso
Profesores
Temario
Material de clase
Herramientas
Flujo de trabajo
Evaluación
Historia
Inicios
Influencias
¿Por qué C?
¿Para qué C?
Proyectos en C
Sumer Of Code
GSOC
Outreachy

Influencias



¿Por qué C?

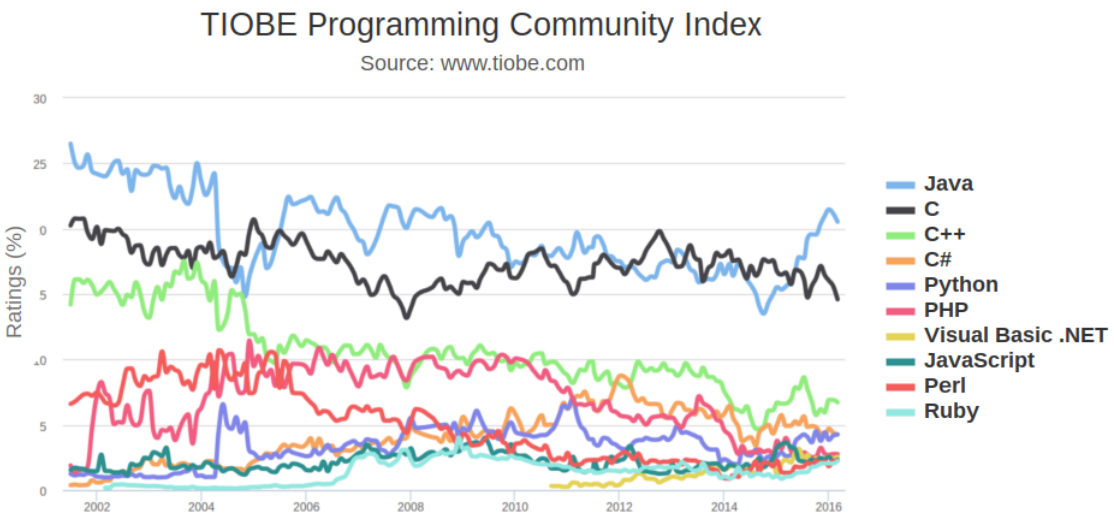
- Simpleza
- Características de bajo nivel
- Madurez
- Eficiencia
- Portabilidad
- Numerosas bibliotecas y herramientas

¿Por qué C?

Introducción

El curso
Profesores
Temario
Material de clase
Herramientas
Flujo de trabajo
Evaluación
Historia
Inicios
Influencias
¿Por qué C?
¿Para qué C?
Proyectos en C
Sumer Of Code
GSOC
Outreachy

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/

Introducción

El curso
Profesores
Temario
Material de clase
Herramientas
Flujo de trabajo
Evaluación
Historia
Inicios
Influencias
¿Por qué C?
¿Para qué C?
Proyectos en C
Sumer Of Code
GSOC
Outreachy

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$

Hasta el 22 Marzo https://gnome.org/outreachy/

Entorno de trabajo

Tema 2

Workspace

Linux
Consola
Instalando herramientas
Hola Mundo

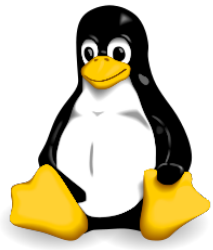
Índice

Linux

Workspace

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

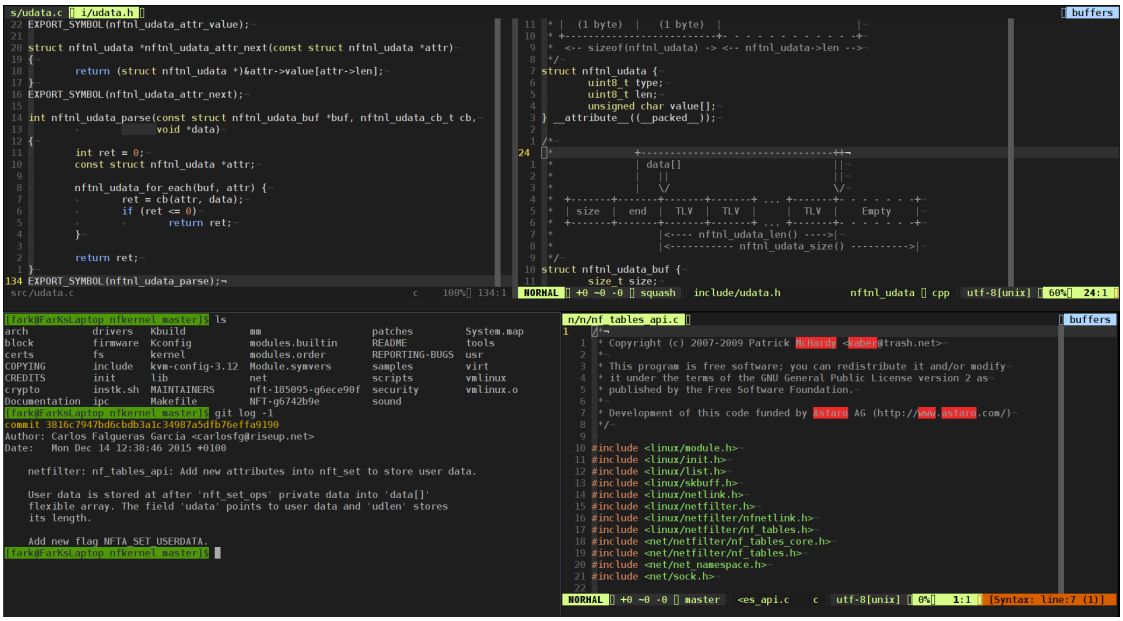
- Ofrece muchísimas facilidades al programador
- Software libre y gratuito
- Repositorios con infinidad de herramientas a nuestra disposición



Consola

Workspace

CTRL + ALT + T



Consola

Comandos básicos:

Lista directorios

Cambia a directorio

Crea directorio

Crea archivo vacío*

Borra archivo

Borra directorio y lo que hay dentro

Instalando herramientas

Comandos para instalar:

Ejecuta un comando con permisos de administrador

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

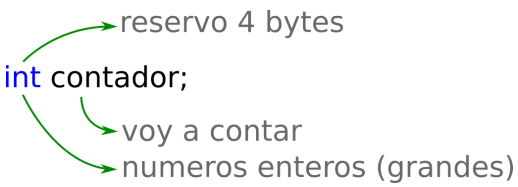
Índice

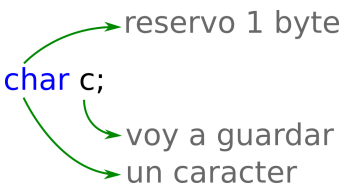
¿Qué es una variable?

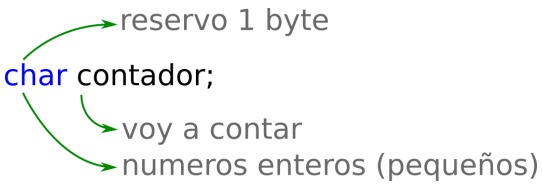
Variables como **zona de memoria** reservada de **tamaño específico**

Los tipos:

- Definen el tamaño
- Dan una idea del uso que se le van a dar a los datos guardados







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

Arrays y tipos

Tema 4

Arrays

Descripción
Ejemplo
Cadenas
Array multi-dimensional

Índice

Descripción

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

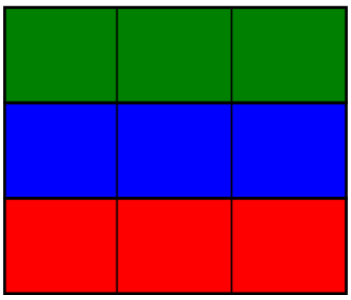

```c
#include <stdio.h>

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


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



Juego de la vida

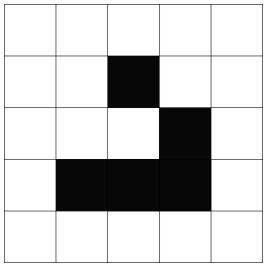
Tema 5

GOL

Introducción
Ejemplo
Enlaces de interés

Índice

Introducción



juego de 0 jugadores

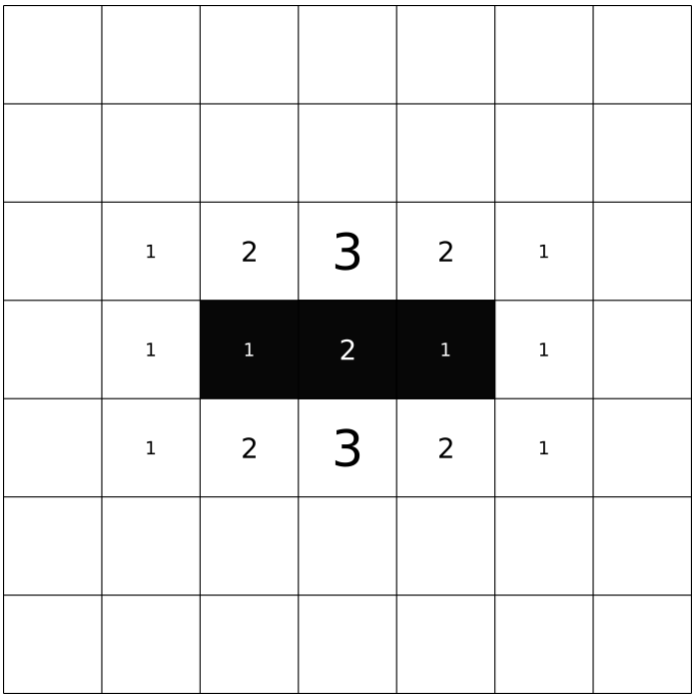
Rejilla de células cuadradas como universo bidimensional ortogonal (infinito o no)

Cada célula tiene dos estados (muerta o viva) e interactúa con sus 8 vecinas según unas reglas

La regla más común es:

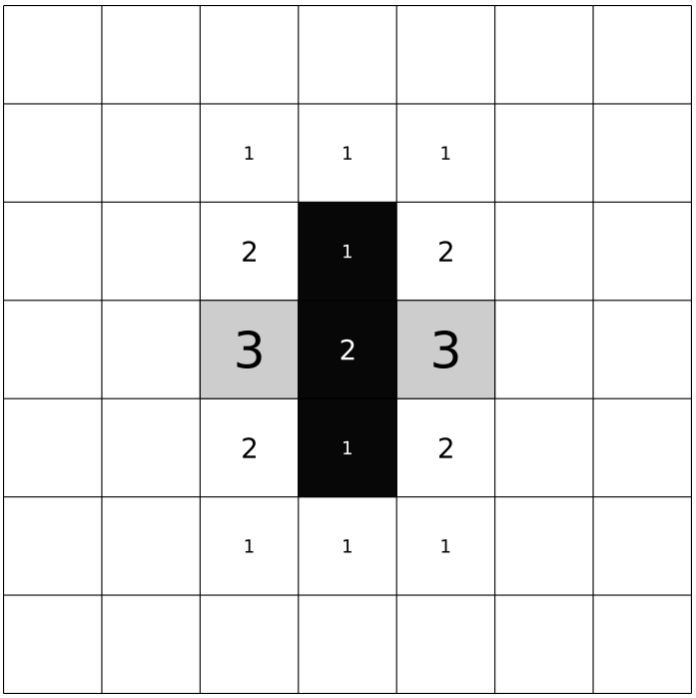
> **Nacimiento**: Una célula muerta con exactamente 3 vecinas vivas estará viva en la siguiente iteración
> **Supervivencia**: Una célula viva con 2 o 3 vecinas vivas seguirá viva en la siguiente iteración, de lo contrario morirá.

Ejemplo



Ejemplo

GOL



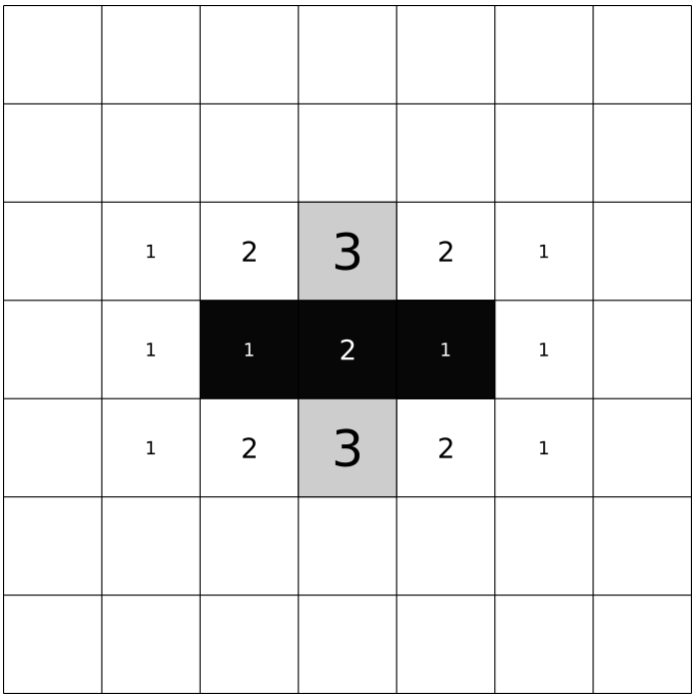
Ejemplo

GOL

Introducción

Ejemplo

Enlaces de interés



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

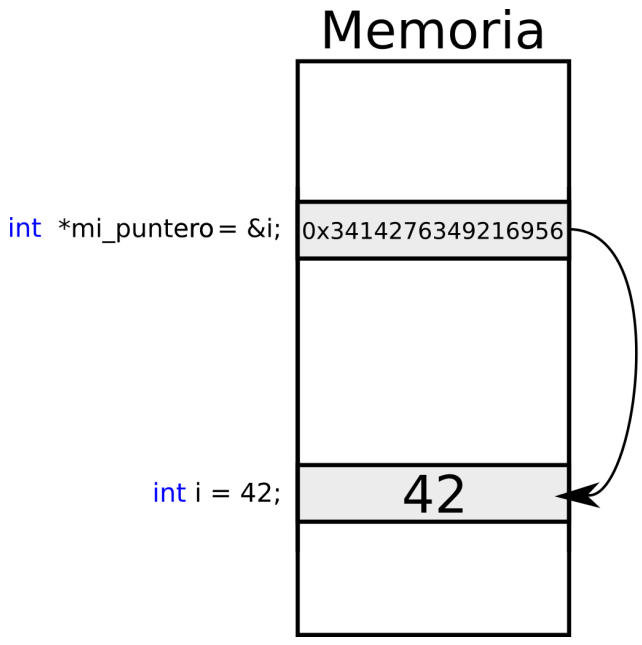
Índice

¿Qué es un puntero?

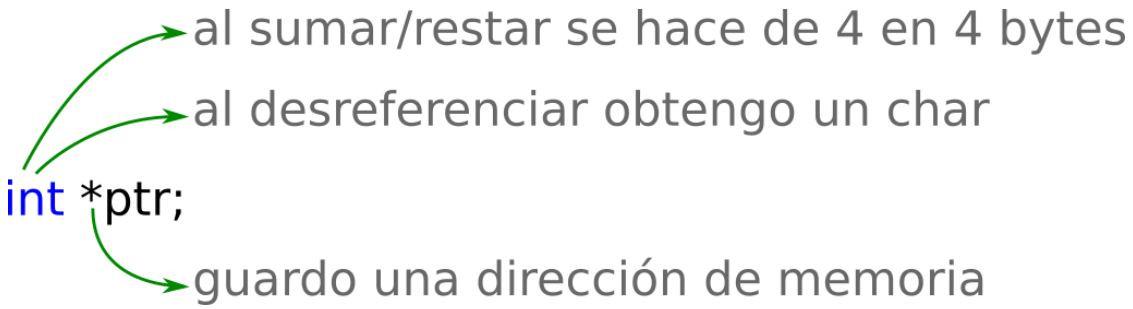
Son **variables normales** y corrientes

Pensadas para guardar una **dirección de memoria**

El tipo del puntero hace referencia al tipo de dato **al que apunta**



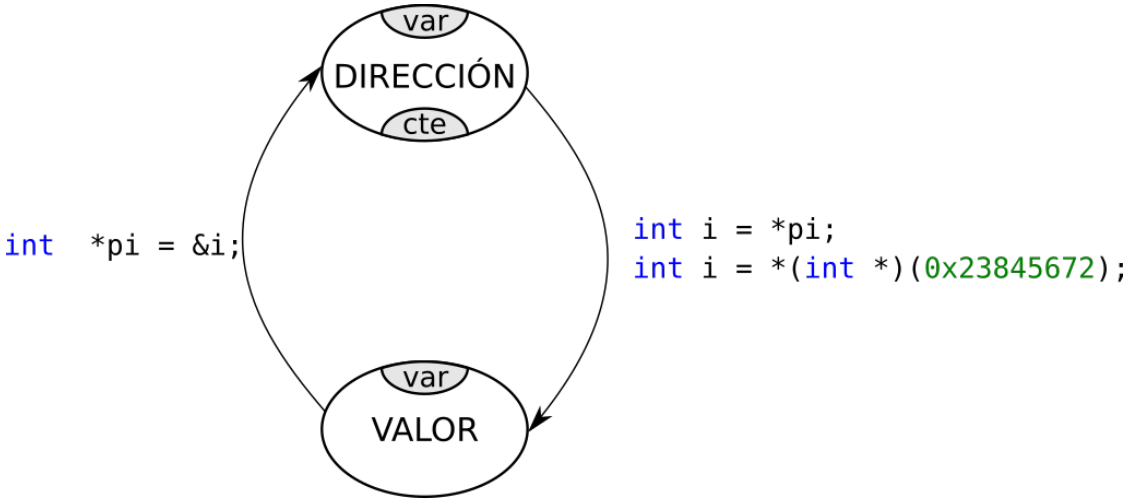
¿Qué es un puntero?



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

Ejemplo

Ejemplo:

```c
int array[3] = {1, 2, 3};
int *p = array;
int i;

/* Todas las direcciones iguales */
printf("%p\n%p\n%p\n", array, p, &array[0]);

p[2] = 22;
p += 1;
*p = 33;
*(p - 1) = 11;

/* Que imprimira? */
for (i = 0; i < 3; i++)
  printf("%d\n", array[i]);
```

UNIVERSIDAD DE SEVILLA

Formas de recorrer un array


```c
int main() {
  int i;
  p = array;

  // forma 1: Contador e incremento de puntero
  for (i = 0; i < 5; i++)
    printf("%d\t", *p++);

  // forma 2: Incremento de puntero y comparacion de
      direcciones
  for (p = array; p <= &array[4]; p++)
    printf("%d\t", *p);

  // forma 3: Contador y puntero como array
  for (i = 0, p = array; i < 5; i++)
    printf("%d\t", p[i]);

  return 0;
}
```

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

```c
#include <stdio.h>

void f(int a)
{
  a = 33;
}

int main()
{
  int a = 3;
  f(a);
  printf("%d\n", a);

  return 0;
}
```


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**

```c
#include <stdio.h>

void f(int *a)
{
  *a = 33;
}

int main()
{
  int a = 3;
  f(&a);
  printf("%d\n", a);

  return 0;
}
```

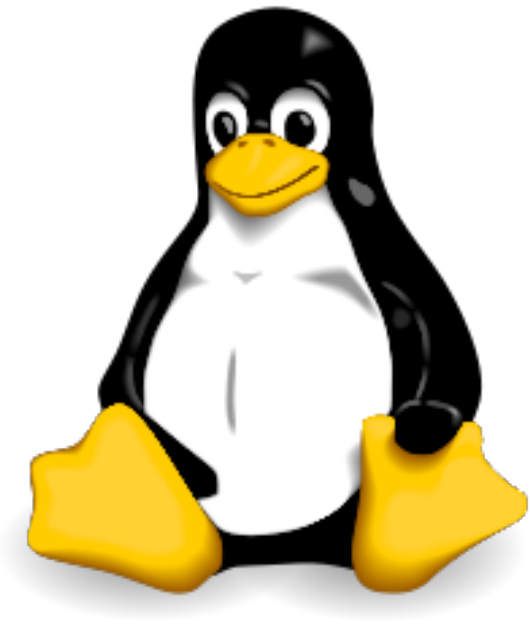
GIT

Tema 8

GIT

Índice

Git
Características
Distribuido VS Centralizado
Funcionamiento
Cuenta en GitHub
Plan gratuito
Confirmar correo
Fork de mi repositorio
Buscar mi repositorio
Fork
Clonar mi repositorio
Workflow
Cambios
git push
Pull request
Comprobación
Descripción
Solicitado
Revisiones
Correcciones



Características

GIT

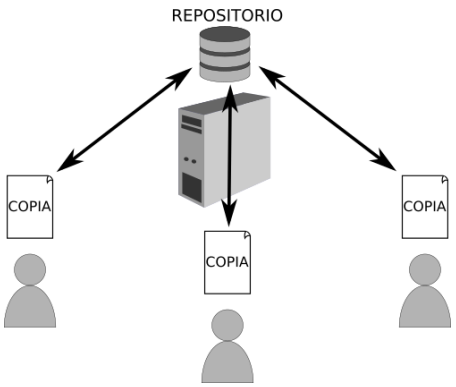
Git
Características
Distribuido VS Centralizado
Funcionamiento
Cuenta en GitHub
Plan gratuito
Confirmar correo
Fork de mi repositorio
Buscar mi repositorio
Fork
Clonar mi repositorio
Workflow
Cambios
git push
Pull request
Comprobación
Descripción
Solicitado
Revisiones
Correcciones

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

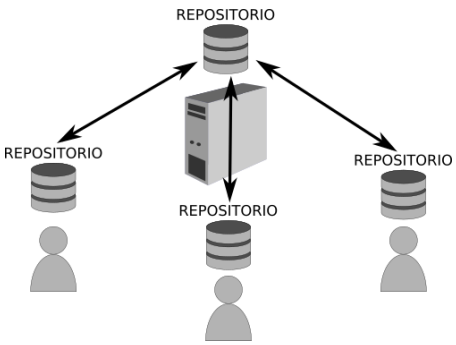
Distribuido VS Centralizado

GIT

Centralizado

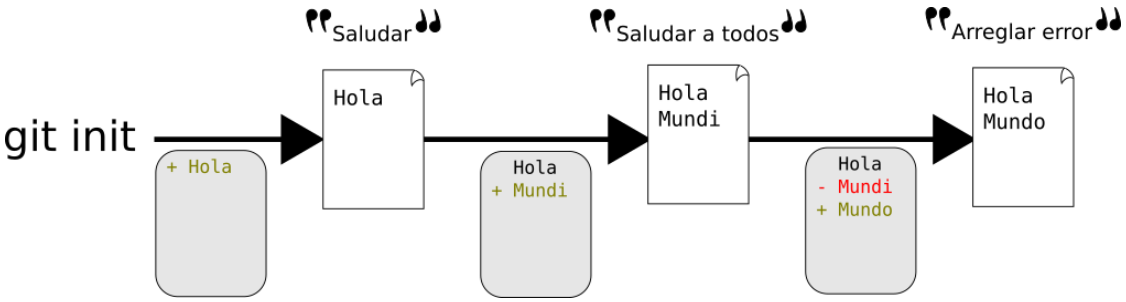


Distribuido



Funcionamiento

GIT



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

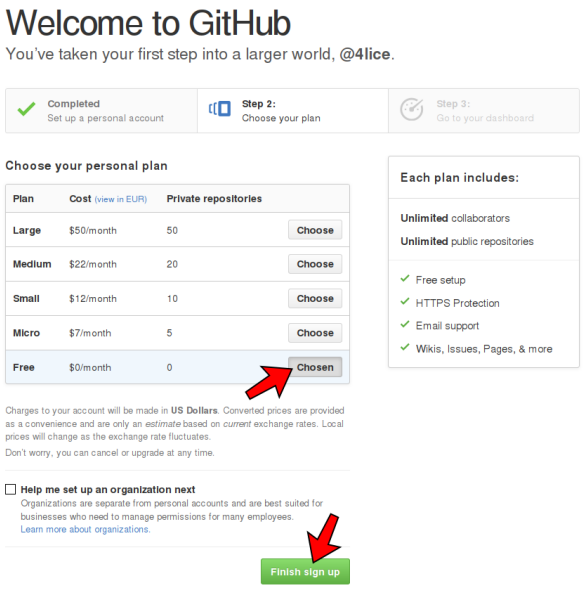
UNIVERSIDAD DE SEVILLA

GIT

Plan gratuito

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

Git
Características
Distribuido VS Centralizado
Funcionamiento
Cuenta en GitHub
Plan gratuito
Confirmar correo
Fork de mi repositorio
Buscar mi repositorio
Fork
Clonar mi repositorio
Workflow
Cambios
git push
Pull request
Comprobación
Descripción
Solicitado
Revisiones
Correcciones



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**.

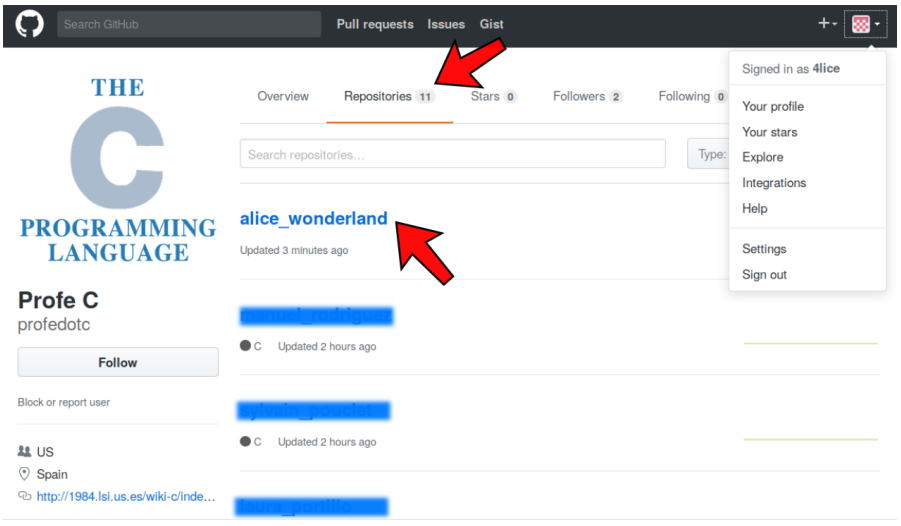
Didn't get the email? Resend verification email or change your email settings.

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

GIT

Buscar mi repositorio

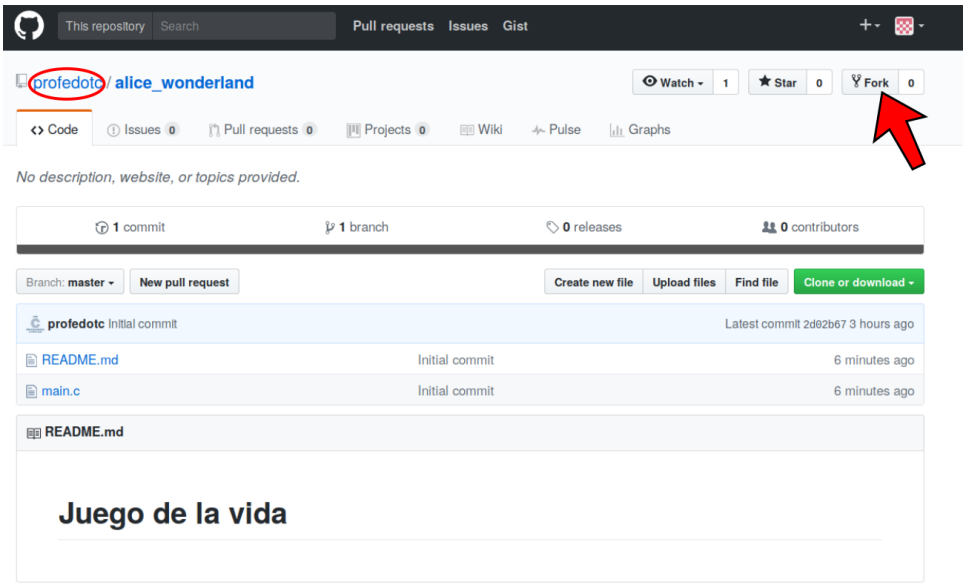
Entramos en la cuenta del profesor (github.com/profedotc), y en la pestaña "*Repositories*" buscamos el repositorio que tenga nuestro nombre y apellido



Git
Características
Distribuido VS Centralizado
Funcionamiento
Cuenta en GitHub
Plan gratuito
Confirmar correo
Fork de mi repositorio
Buscar mi repositorio
Fork
Clonar mi repositorio
Workflow
Cambios
git push
Pull request
Comprobación
Descripción
Solicitado
Revisiones
Correcciones

Fork

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

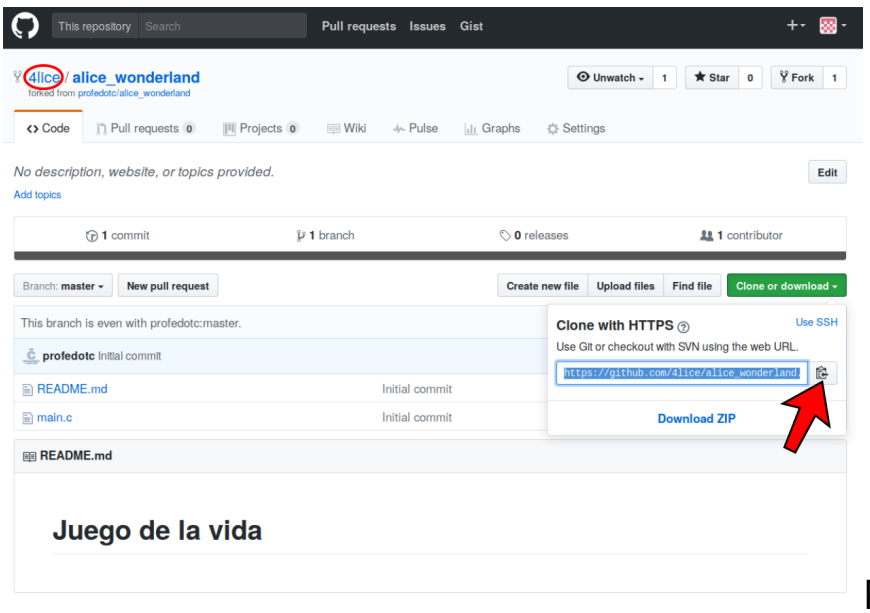


UNIVERSIDAD DE SEVILLA

GIT

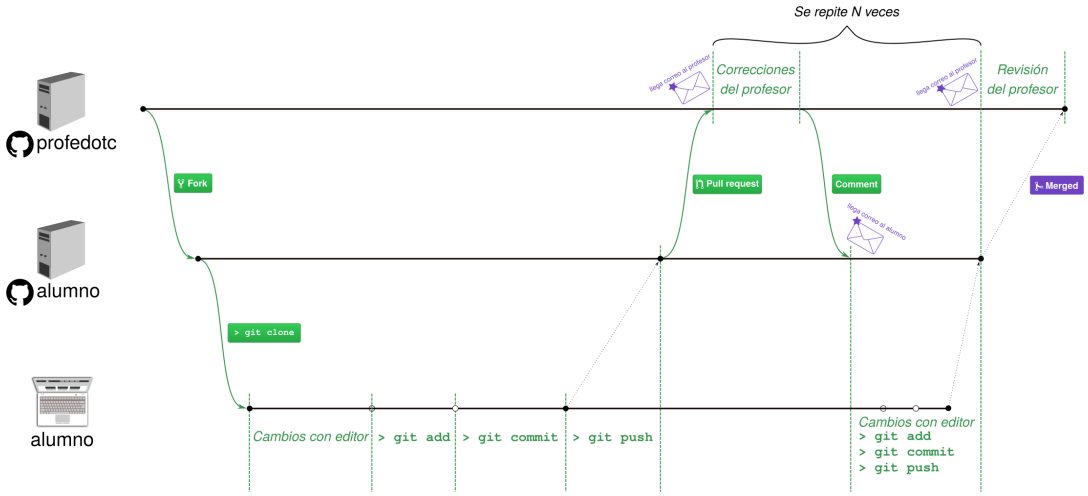
Clonar mi repositorio

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



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

Flujo de trabajo



Crea y revisa tus cambios

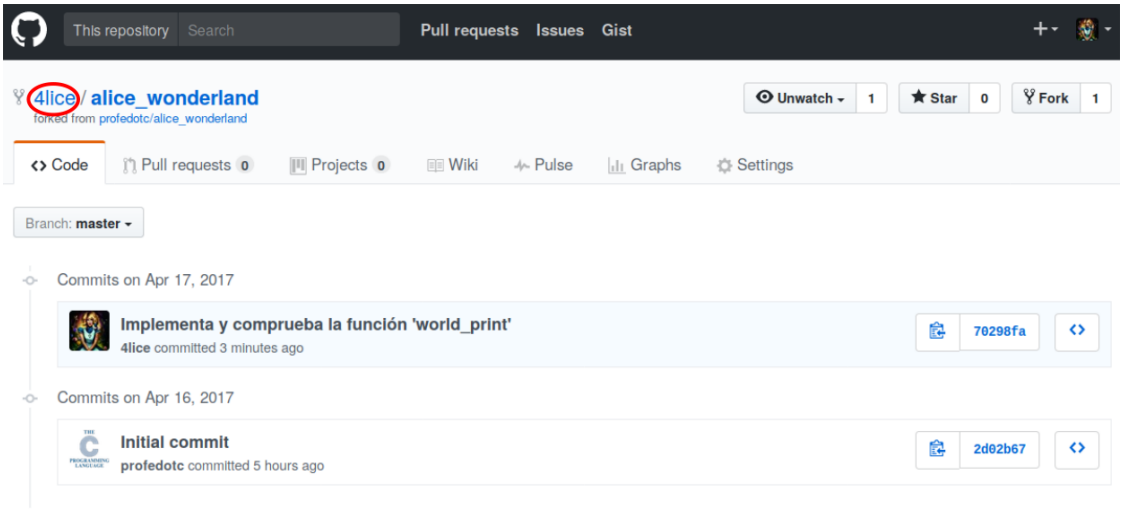
`> git diff`

```
diff --git a/main.c b/main.c
index 7aa2631..3544b1d 100644
--- a/main.c
+++ b/main.c
@@ -2,10 +2,11 @@
 #include <stdlib.h>
 #include <stdbool.h>

-// TODO: Crea dos macros con el tamanio horizontal y vertical del
-    mundo
+#define W_SIZE_X 10
+#define W_SIZE_Y 10

 void world_init(/* Recibo un mundo */);
-void world_print(/* Recibo un mundo */);
+void world_print(bool w[W_SIZE_X][W_SIZE_Y]);
 void world_step(/* Recibo dos mundos */);
 int world_count_neighbors(/* Recibo un mundo y unas coordenadas */)
     ;
 bool world_get_cell(/* Recibo un mundo y unas coordenadas */);
```

```
@@ -14,12 +15,13 @@ void world_copy(/* Recibo dos mundos */);
 int main()
 {
   int i = 0;
- // TODO: Declara dos mundos
+ bool world_a[W_SIZE_X][W_SIZE_Y];
+ bool world_b[W_SIZE_X][W_SIZE_Y];
```

Sube los cambios a tu repositorio

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

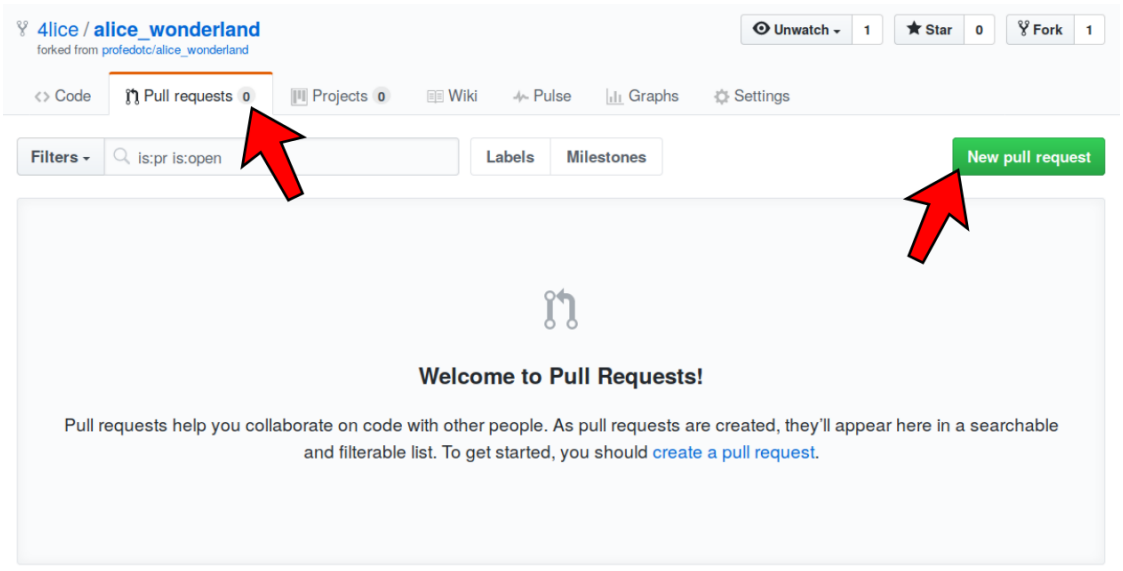


UNIVERSIDAD DE SEVILLA

GIT

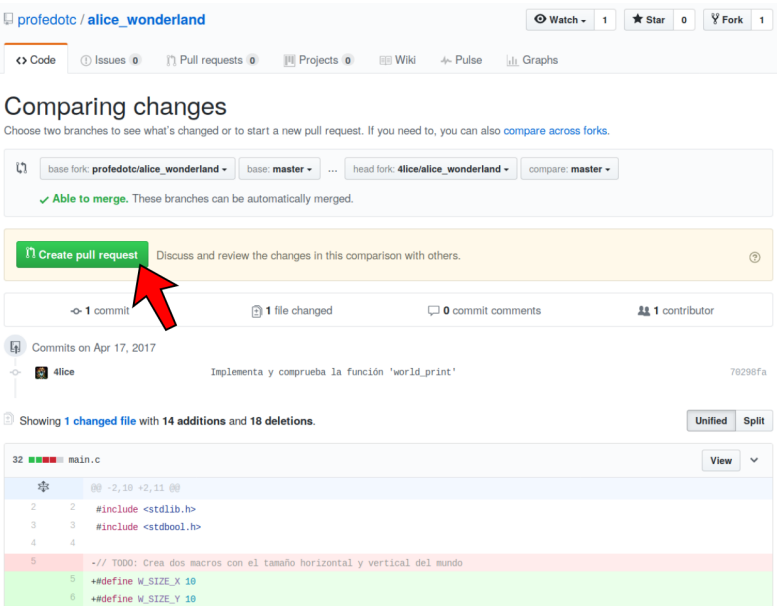
Crea un nuevo pull request

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*"

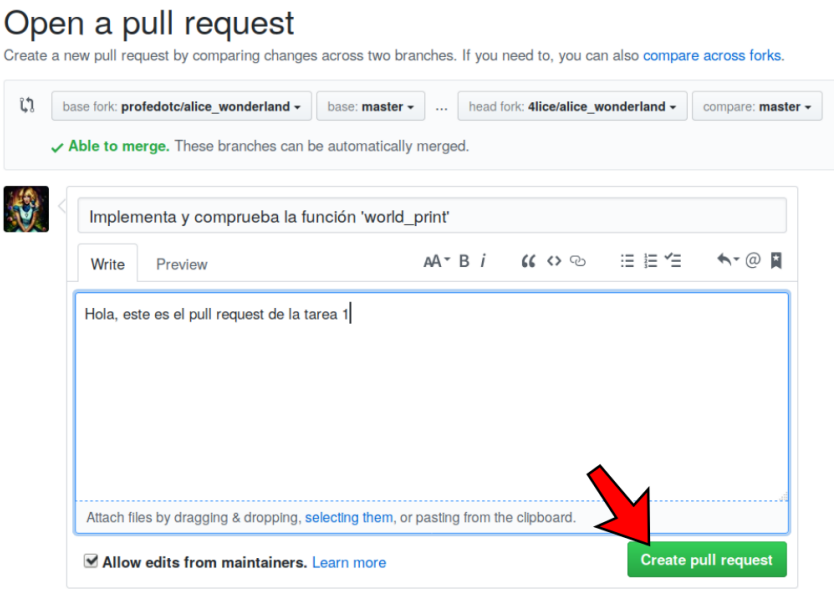


GIT

Git
Características
Distribuido VS Centralizado
Funcionamiento
Cuenta en GitHub
Plan gratuito
Confirmar correo
Fork de mi repositorio
Buscar mi repositorio
Fork
Clonar mi repositorio
Workflow
Cambios
git push
Pull request
Comprobación
Descripción
Solicitado
Revisiones
Correcciones

Descripción

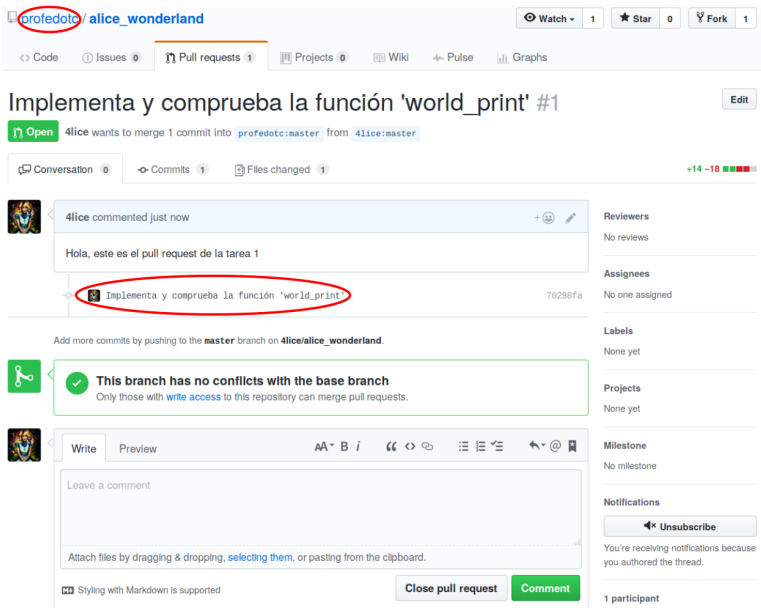
Escribimos un pequeño texto indicando la tarea que se entrega y los cambios realizados



Solicitud terminada

GIT

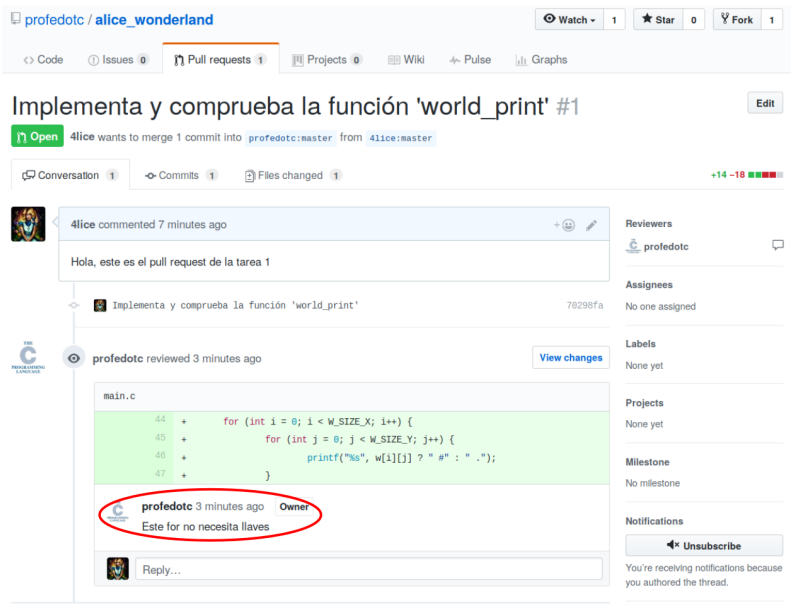
Si todo ha ido bien, deberíamos ver una pantalla parecida a la siguiente:



Esperar las revisiones del profesor

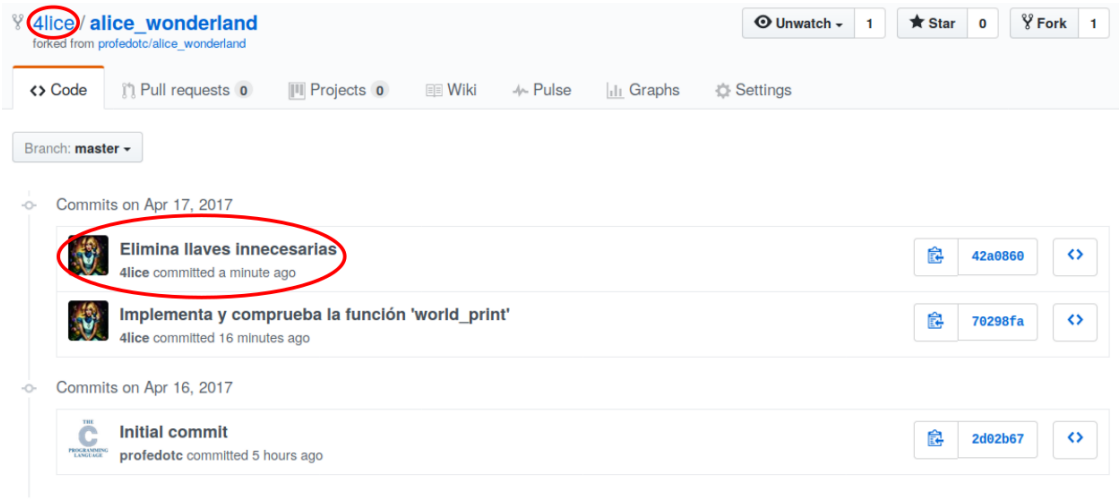
GIT

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

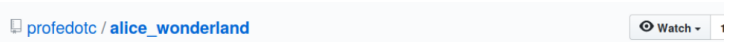


Correcciones

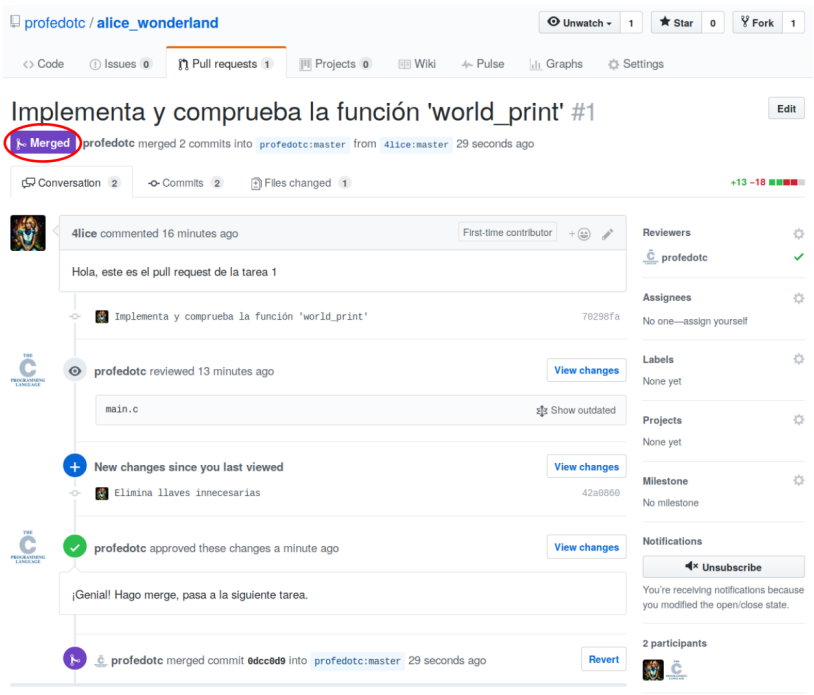
Crea un nuevo commit (o varios) para solucionar las correcciones que te hayan pedido



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

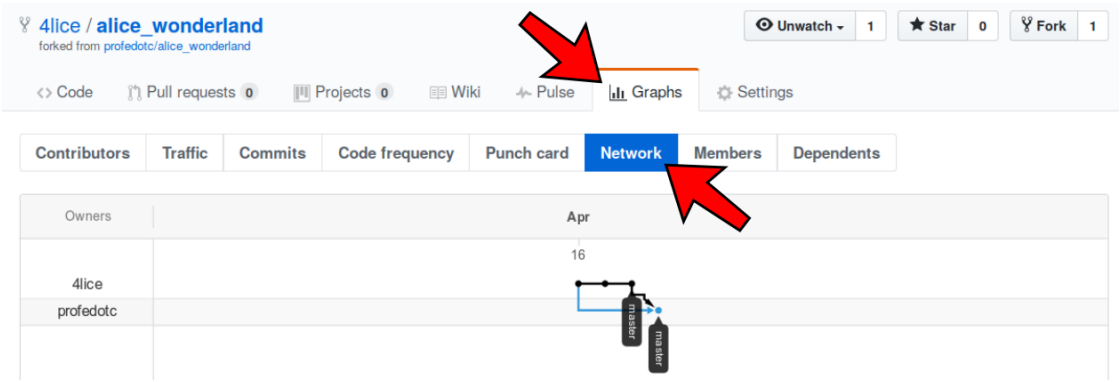


Espera a nuevas revisiones o a que sea aceptado



Git
Características
Distribuido VS Centralizado
Funcionamiento
Cuenta en GitHub
Plan gratuito
Confirmar correo
Fork de mi repositorio
Buscar mi repositorio
Fork
Clonar mi repositorio
Workflow
Cambios
git push
Pull request
Comprobación
Descripción
Solicitado
Revisiones
Correcciones
Aceptado

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?~~ →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


```c
#include <stdio.h>

struct cell {
  int state;
  int size;
};

int main()
{
  struct cell culture[5] = {{1,10}, {0,342}, {1,7},
      {1,50}, {1,77}};
  struct cell *c;

  for (c = culture; c <= &culture[4]; c++)
    printf("{%d, %d}\n", c->state, c->size);

  return 0;
};
```

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

```c
union float_int {
  float f;
  int   i;
};

int main()
{
  union float_int fi;

  fi.f = 2.7182;
  printf("%X\n", fi.i);

  return 0;
}
```

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;
}
```

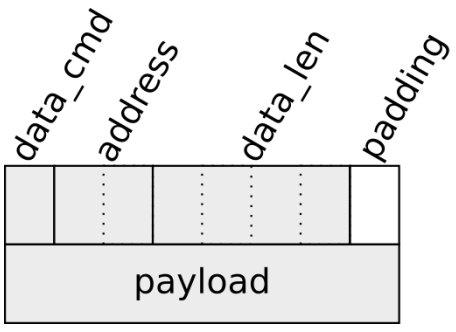
Campos de bits

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

```c
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


```c
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


```c
struct bug {
  unsigned int legs;
  unsigned char color[3];
  const char *name;
} bugs[30] = {
  [0] = {
    .legs = 6,
    .color = {100, 100, 100},
    .name = "ant",
  },
  [1] = {
    .legs = 8,
    .color = {[0] = 200},
    .name = "spider",
  },
  [2 ... 10] = { // Only GNU
    .legs = 100,
    .color = {[1] = 255},
    .name = "centipede",
  },
};
```

C Modular

Tema 10

C Modular

Cabeceras
Ejemplo
Compilación
Make
Estructura
Ejemplo

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

```c
#ifndef _PERSON_H_
#define _PERSON_H_

#define MAX_NAME 256

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

void print_person(const struct person *p);

#endif
```

person.c

```c
#include <stdio.h>
#include "person.h"

void print_person(const struct person *p)
{
  printf("%s, %s\n", p->surname, p->name);
  printf("\t- age:   %d\n", p->age);
  printf("\t- phone: %d\n", p->phone);
}
```

Compilación por bloques

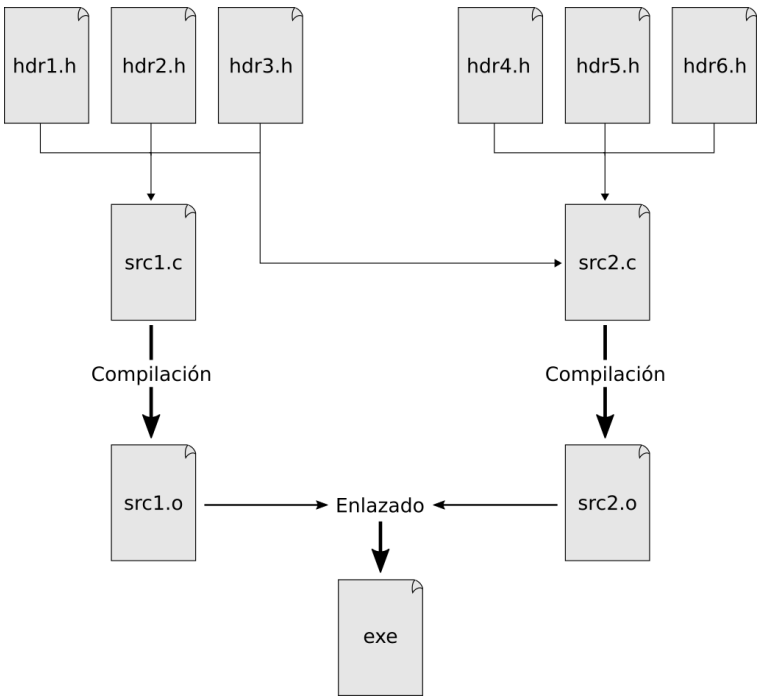
Razones:

- 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

C Modular

Cabeceras
Ejemplo
Compilación
Make
Estructura
Ejemplo



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

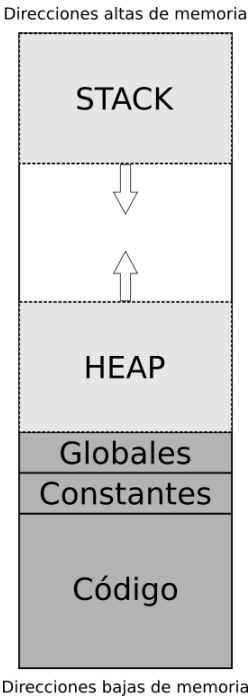
Heap: Se almacena la memoria reservada **dinámicamente** con **malloc**

Stack: Se almacenan las **variables locales** de cada llamada a función

Globales: Todas las variables globales

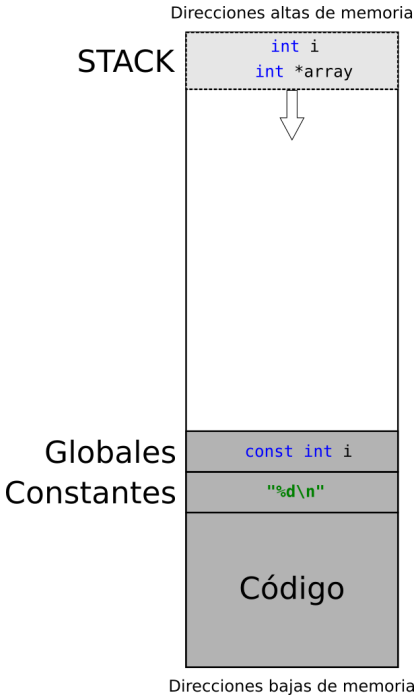
Constantes: Todas las constantes (números, cadenas, etc)

Código: El programa en sí



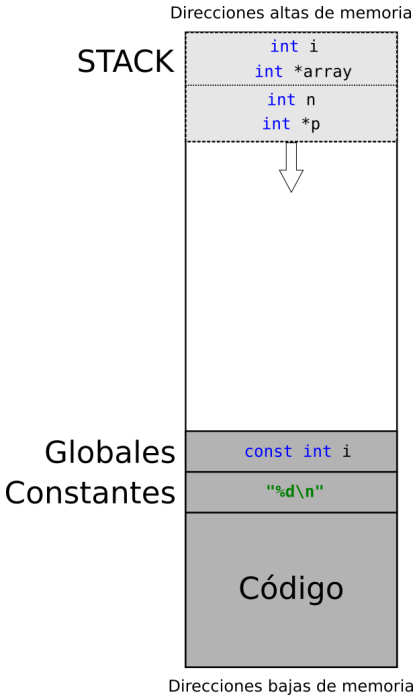
Ejemplo


```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

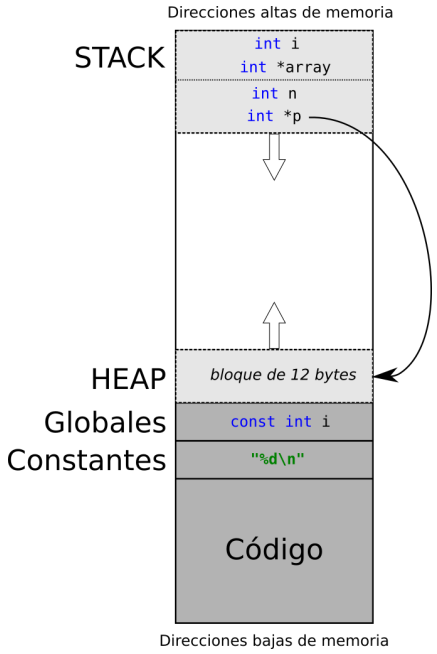

```c
#include <stdio.h>
#include <stdlib.h>

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

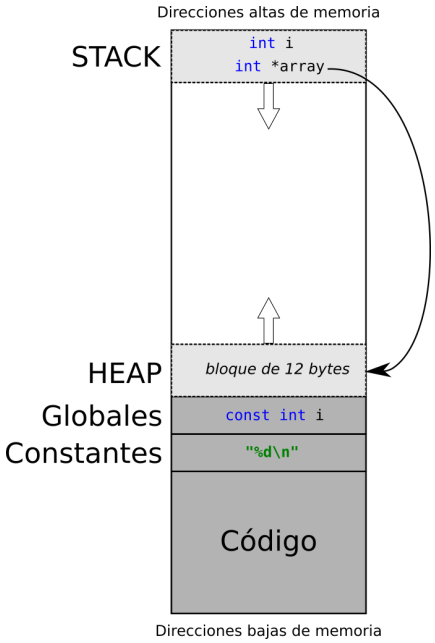
Dyn memory
Mapa de memoria
Ejemplo
Stack Overflow
Fragmentación
arrays multiD
Forma 1
Forma 2
Uso tras liberación
valgrind
Ejemplo


```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

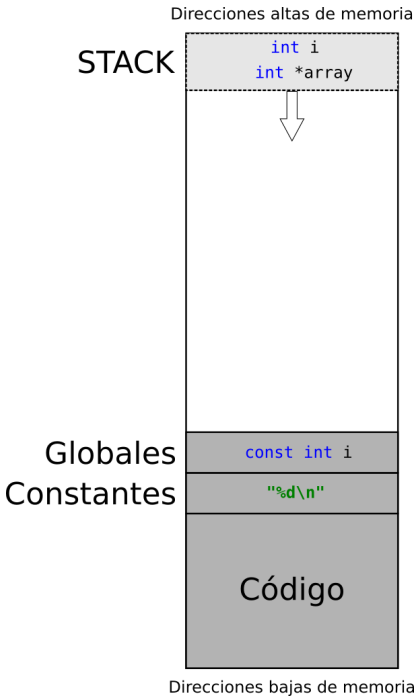

```c
#include <stdio.h>
#include <stdlib.h>

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

Dyn memory
Mapa de memoria
Ejemplo
Stack Overflow
Fragmentación
arrays multiD
Forma 1
Forma 2
Uso tras liberación
valgrind
Ejemplo

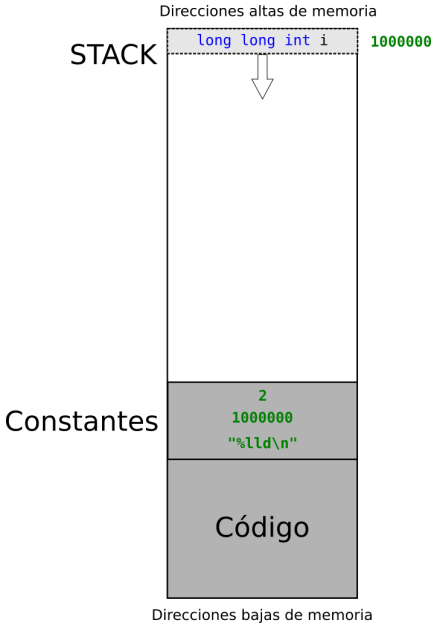

```c
#include <stdio.h>
#include <stdlib.h>

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

```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;
}
```

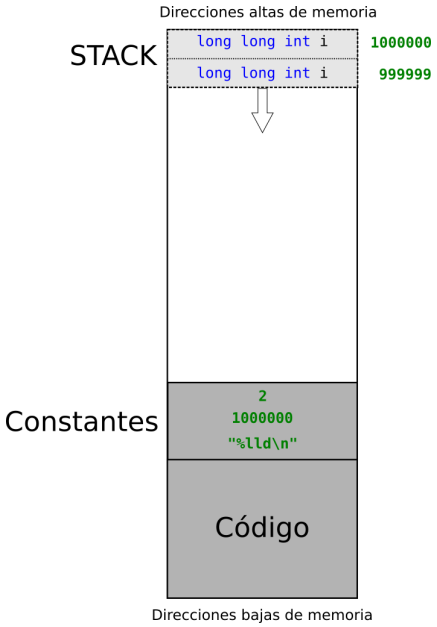


Stack Overflow

Dyn memory
Mapa de memoria
Ejemplo
Stack Overflow
Fragmentación
arrays multiD
Forma 1
Forma 2
Uso tras liberación
valgrind
Ejemplo

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

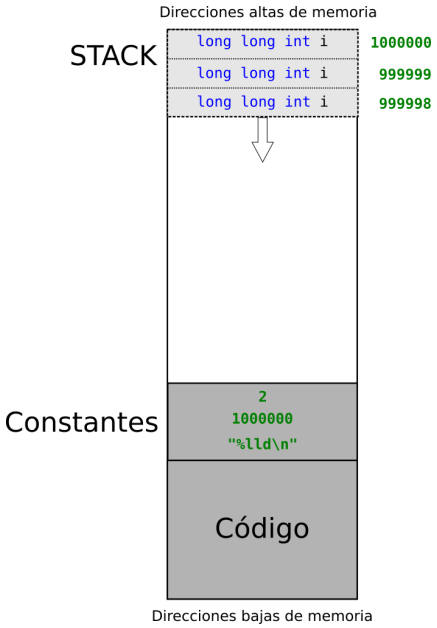
```c
#include <stdio.h>

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

```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;
}
```



UNIVERSIDAD DE SEVILLA

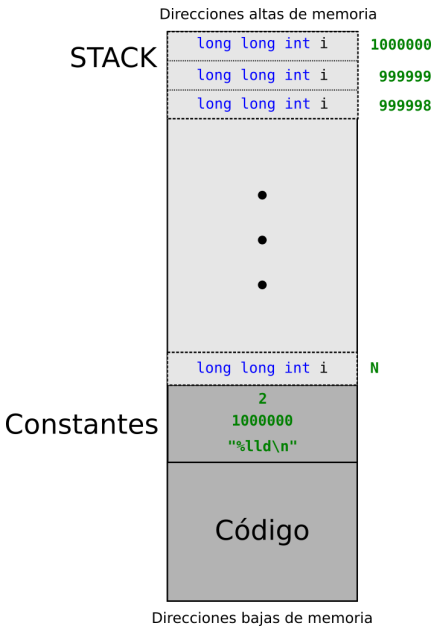
Dyn memory

Mapa de memoria
Ejemplo
Stack Overflow
Fragmentación
arrays multiD
Forma 1
Forma 2
Uso tras liberación
valgrind
Ejemplo

Stack Overflow

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

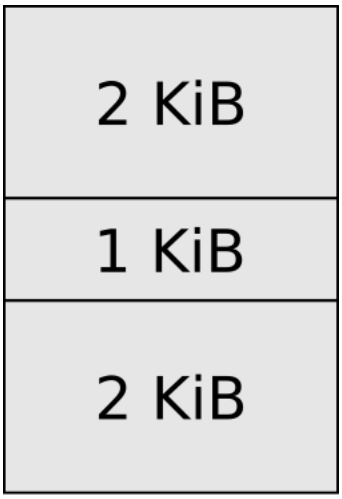
```c
#include <stdio.h>

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;
}
```



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;
}
```

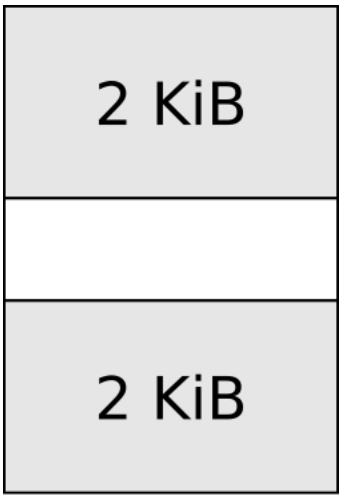


Fragmentación

Dyn memory
Mapa de memoria
Ejemplo
Stack Overflow
Fragmentación
arrays multiD
Forma 1
Forma 2
Uso tras liberación
valgrind
Ejemplo

Fragmentos pequeños de memoria libre entre bloques de memoria reservada.

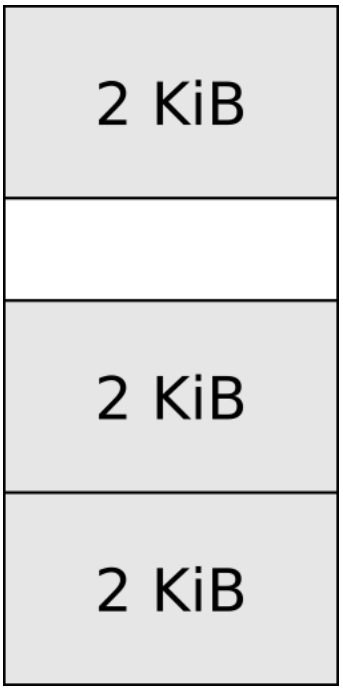
```c
#include <stdio.h>
#include <stdlib.h>

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;
}
```



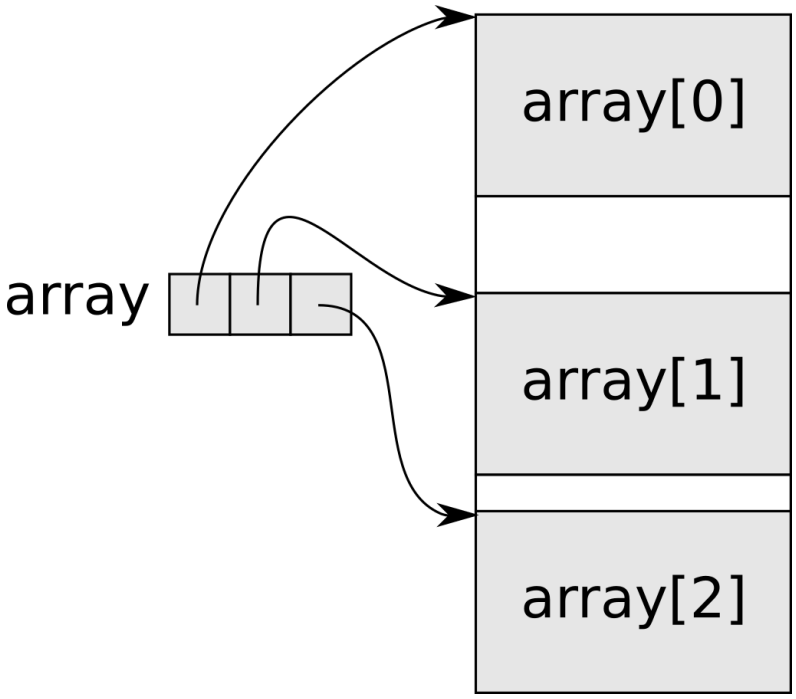
Forma 1 (la mala)

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

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

  /* Reservamos */
  array = (int **)malloc(ROWS * sizeof(int *));
  for (i = 0; i < ROWS; i++)
    array[i] = (int *)malloc(COLS * sizeof(int)); // Falta comprobar

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

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

  /* Liberamos */
  for (i = 0; i < ROWS; i++)
    free(array[i]);

  return 0;
}
```

Forma 1 (la mala)

No se garantiza continuidad entre los bloques reservados



Dyn memory

Mapa de memoria
Ejemplo
Stack Overflow
Fragmentación
arrays multiD
Forma 1
Forma 2
Uso tras liberación
valgrind
Ejemplo

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:

- Compilar con símbolos de depuración (`gcc -g`)
- 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

Dyn memory

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

Mapa de memoria
Ejemplo
Stack Overflow
Fragmentación
arrays multiD
Forma 1
Forma 2
Uso tras liberación
valgrind
Ejemplo

```
Memcheck, a memory error detector
....
Command: valg_ej

La direccion de p es 0x51d7040

HEAP SUMMARY:
    in use at exit: 20 bytes in 1 blocks
  total heap usage: 1 allocs, 0 frees, 20 bytes allocated

20 bytes in 1 blocks are definitely lost in loss record 1 of 1
   at 0x4C2ABD0: malloc (in /usr/lib/...)
   by 0x4004F7: main (valgrind.c:6)

LEAK SUMMARY:
   definitely lost: 20 bytes in 1 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 0 bytes in 0 blocks

For counts of detected and suppressed errors, rerun with: -v
ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Objetos

Tema 12

Índice

¿Qué es un objeto?

Un objeto es una *entidad* con:

- Un **estado**: datos que guarda
- Un **comportamiento**: métodos que interactúan con ese estado.

Un objeto es una **instancia** de una **clase** específica

Un clase puede **heredar** propiedades de otras

Ejemplo de objeto en C

Ejemplo de objeto
persona

Ocultación

person.h

```h
#ifndef _PERSON_H_
#define _PERSON_H_

struct person;

#endif
```

person.c

```c
#include "person.h"

#define DNI_LEN 9

struct person {
  char *name;
  char dni[DNI_LEN];
  int  age;

  uint32_t flags;
};

enum person_attr {
  PERSON_NAME,
  PERSON_DNI,
  PERSON_AGE,
};
```

main.c

```c
#include "person.h"

int main()
{
  struct person *p;

  return EXIT_SUCCESS;
}
```

Flags

person.c

```c
#define ATTR_SET(flags, attr)    (flags) |= (1 << (attr))
#define ATTR_IS_SET(flags, attr) ((flags) & (1 << (attr)))
```

Necesitamos saber si un atributo ha sido o no establecido para:

- Saber si el valor que guarda o no es válido
- Saber si se ha reservado memoria y hay que liberarla
- Imprimir o no los atributos
- ...

Reserva

person.c

```
1  struct person *person_alloc()
2  {
3    struct person *p;
4
5    p = (struct person *)malloc(sizeof(struct person));
6    p->flags = 0;
7
8    return p;
9  }
10
11 void person_free(struct person *p)
12 {
13   if (ATTR_IS_SET(p->flags, PERSON_NAME))
14     free(p->name);
15   free(p);
16 }
```

main.c

```
1  int main()
2  {
3    struct person *p;
4
5    p = person_alloc();
6    person_free(p);
7
8    return EXIT_SUCCESS;
9  }
```

Setters


```
1  void person_set_name(struct person *p, const char *name)
2  {
3    if (ATTR_IS_SET(p->flags, PERSON_NAME))
4      free(p->name);
5
6    p->name = strdup(name);
7    ATTR_SET(p->flags, PERSON_NAME);
8  }
9
10 int person_set_dni(struct person *p, const char *dni)
11 {
12   if (strlen(dni) != DNI_LEN)
13     return 0;
14
15   strcpy(p->dni, dni);
16   ATTR_SET(p->flags, PERSON_DNI);
17
18   return 1;
19 }
20
21 int person_set_age(struct person *p, int age)
22 {
23   if (age < 0 || age > 150)
24     return 0;
25
26   p->age = age;
27   ATTR_SET(p->flags, PERSON_AGE);
28
29   return 1;
30 }
```

Getters

```
1  const char *person_get_name(const struct person *p)
2  {
3    if (ATTR_IS_SET(p->flags, PERSON_NAME))
4      return p->name;
5    else
6      return NULL;
7  }
8
9  const char *person_get_dni(const struct person *p)
10 {
11   if (ATTR_IS_SET(p->flags, PERSON_DNI))
12     return p->dni;
13   else
14     return NULL;
15 }
16
17 int person_get_age(const struct person *p)
18 {
19   if (ATTR_IS_SET(p->flags, PERSON_AGE))
20     return p->age;
21   else
22     return -1;
23 }
```


Print

```
1  void person_print(const struct person *p)
2  {
3    printf("Person {");
4
5    if (ATTR_IS_SET(p->flags, PERSON_NAME))
6      printf("\n\tname = \"%s\"", p->name);
7
8    if (ATTR_IS_SET(p->flags, PERSON_DNI))
9      printf("\n\tdni  = \"%s\"", p->dni);
10
11   if (ATTR_IS_SET(p->flags, PERSON_AGE))
12     printf("\n\tage  = %d", p->age);
13
14   printf("\n}\n");
15 }
```

Argumentos de main()

Tema 13

main() args

argc y argv
Ejemplo
getopt
getopt short
getopt long

Índice

argc y argv

$$\texttt{int main(int argc, char *argv[])}$$

La función main admite dos parámetros que están relacionados con los argumentos que pasamos a nuestro programa al ejecutarlo:

- **argc**: Número de argumentos pasados
- **argv**: Vector de argumentos

Cada argumento es una cadena de texto

Siempre se pasa al menos un argumento con el nombre del programa (incluida la ruta relativa de llamada)

Ejemplo

código:

```
1  int main(int argc, char *argv[])
2  {
3    int i;
4
5    printf("Argumento i = <argumento>\n");
6    for (i = 0; i < argc; i++)
7      printf("%d = %s\n", i, argv[i]);
8
9    return 0;
10 }
```

salida:

```
#>./args_example foo bar
Argumento i = <argumento>
0 = ./args_example
1 = foo
2 = bar
```

getopt

Es una utilidad de GNU C Library

Sigue el estándar POSIX sobre argumentos

Permite analizar los argumentos fácilmente

getopt short

```
> gopts -a -b 3 -c hola
```

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

int main(int argc, char **argv)
{
  int aflag;
  int bvalue;
  char *cvalue;
  int c;

  while ((c = getopt(argc, argv, "ab:c:")) != -1) {
    switch (c)
    {
    case 'a':
      aflag = 1;
      break;
    case 'b':
      bvalue = strtol(optarg, NULL, 0);
      break;
    case 'c':
      cvalue = optarg;
      break;
    default:
      return EXIT_FAILURE;
    }
  }
```

getopt short


```
30    printf("bvalue = %d\n", bvalue);
31    printf("cvalue = \"%s\"\n", cvalue);
32
33    for (int i = optind; i < argc; i++)
34      printf("Argumento desconocido \"%s\"\n", argv[i]);
35
36    return 0;
37  }
```

```
> gopts -a -b 3 -c hola
aflag  = 1
bvalue = 3
cvalue = "hola"
```

getopt long


```
> goptl -u -w 10 -h 20 --random=52432
```



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4  #include <getopt.h>
5
6  int main(int argc, char **argv)
7  {
8    int usage = false;
9    size_t width = 0, height = 0;
10   bool random = false;
11   int rseed = 0;
12
13   int option_index = 0;
14   int c;
15
16   static struct option long_options[] =
17   {
18     {"usage",  no_argument,       NULL, 'u'},
19     {"width",  required_argument, NULL, 'w'},
20     {"height", required_argument, NULL, 'h'},
21     {"random", optional_argument, NULL, 'r'},
22     {0, 0, 0, 0}
23   };
```

getopt long


```
25    while ((c = getopt_long(argc, argv, "uw:h:r::", long_options,
26          &option_index)) != -1) {
27      switch (c) {
28      case 'u':
29        usage = true;
30        break;
31      case 'w':
32        width = strtol(optarg, NULL, 0);
33        break;
34      case 'h':
35        height = strtol(optarg, NULL, 0);
36        break;
37      case 'r':
38        random = true;
39        if (optarg)
40          rseed = strtol(optarg, NULL, 0);
41        break;
42      case '?':
43        /* getopt_long imprime un mensaje de error*/
44        break;
45      default:
46        printf("Error\n");
47        exit(EXIT_FAILURE);
48      }
49    }
```

getopt long


```
51    printf("usage  = %s\n", usage? "TRUE" : "FALSE");
52    printf("width  = %lu\n", width);
53    printf("height = %lu\n", height);
54    printf("random = %s\n", random? "TRUE" : "FALSE");
55    printf("rseed  = %d\n", rseed);
56
57    for (int i = optind; i < argc; i++)
58      printf("Argumento desconocido: \"%s\"\n", argv[i]);
59
60    exit(0);
61  }
```

```
> goptl -u -w 10 -h 20 --random=52432
usage  = TRUE
width  = 10
height = 20
random = TRUE
rseed  = 52432
```

Depuración

Tema 14

Índice

Introducción

¿Qué es depurar (debugging)?: El proceso por el cual buscas y/o eliminas fallos (bugs) de tu programa

¿Qué es un depurador (debugger)?: Una herramienta que facilita la depuración al permitir detener tu programa en cualquier instante y mirar que está pasando

¿Por qué es importante usarlo?: . El método tradicional de los `printf` de depuración está bien, pero se queda corto en cuanto el programa crece en complejidad. En este punto, el depurador se vuelve la manera más eficiente y fácil de depurarlo.

GDB

Depurador del proyecto GNU

Un proyecto muy maduro (30 años)

Repleto de utilidades para depurar

Capaz de depurar varios lenguajes (**C**, C++, Ada, Objective-C, Free Pascal, Fortran, Java . . .)

Soporta muchas arquitecturas (x86, x86-64, ARM, AVR, . . .)

Sistema cliente-servidor que permite depuración remota

Base de muchos depuradores gráficos actuales (CodeBlocs, Eclipse, NetBeans, VisualStudio, QtCreator, . . .)

Cómo utilizar GDB

Compilar con símbolos de depuración `gcc -g main.c -o miprog` (`-ggdb` para símbolos específicos de gdb)

Arrancar nuestro programa con gdb: `gdb miprog`

Utilizar los comandos de gdb para depurar

Comandos

COMANDOS

Inicio

El programa se inicioa con el comando `run`. Si nuestro programa recibe parámetros, se los pasamos a `run`:

```
(gdb) run -w 10 -h 15
```


Breakpoints

Un **breakpoint** define el momento en el que se va a detener nuestro programa para permitirnos examinarlo.

Podemos elegir que se detenga cuando:

- Alcance cierta línea de código:
  ```
  (gdb) break main.c:15
  ```
- Entre en una función determinada:
  ```
  (gdb) break world_alloc
  ```
- Se modifique cierta variable:
  ```
  (gdb) watch contador
  ```

Breakpoints

Otros comandos:

Listar los breakpoints (y watchpoints):

```
(gdb) info breakpoints
```

Elimiar breakpoint:

```
(gdb) delete <num que sale al listar>
```

Elimiar todos los breakpoint:

```
(gdb) delete breakpoints
```


Paso a paso

Tenemos varios comandos para realizar una ejecución "paso a paso" de nuestro programa:

Ejecutar hasta el siguiente breakpoint:

```
(gdb) continue
```

Ejecutar una línea (sin entrar en funciones):

```
(gdb) next
```

Ejecutar una línea (entrado en funciones):

```
(gdb) step
```

Salir de la función actual:

```
(gdb) finish
```

Estado del programa

Ver una variable:

```
(gdb) print contador
```

Ver una estructura:

```
(gdb) print *world
```

Ver el resultado de una expresión:

```
(gdb) print size_x * size_y
```

Ver el resultado de una función:

```
(gdb) print world_get_cell(w, 1, 2)
```

Ver las variables locales:

```
(gdb) info locals
```

Ver algunas línas de código de alrededor:

```
(gdb) list
```

Ver la pila de llamadas:

```
(gdb) backtrace
```


Otros comandos útiles

Ver clases de comandos:

```
(gdb) help
```

Ver comandos de un clase (por ejemplo, breakpoints):

```
(gdb) help breakpoints
```

Ver ayuda sobre un comando (por ejemplo, break):

```
(gdb) help break
```

Para salir:

```
(gdb) quit
```

Puedes abreviar un comando siempre que no sea ambiguo:

```
(gdb) help n == (gdb) help next
```

Para ejecutar el comando anterior pulsar ENTER

Para autocompletar (comandos, nombres de funciones y variables, etc) pulsar TAB

Chuleta:

http://lab46.corning-cc.edu/_media/opus/fall2014/mgardne8/70120637.png

Entrada/Salida

Tema 15

Entrada/Salida

Streams
Funciones
fopen
fwrite
fread
fputs
fgets
fprintf

Índice

¿Qué es un stream?

Flujo de bytes de longitud indeterminada, al que se accede de forma **secuencial**.

En un *stream*:

Al **leer** uno o más bytes, en la próxima lectura obtendremos los siguientes.

Al **escribir** uno o más bytes, en la próxima escritura los añadiremos a continuación.

Streams estándar:

`stdout`: *Salida estándar* (normalmente por consola)

`stdin`: *Entrada estándar* (normalmente por teclado)

`stderr`: *Salida estándar de errores* (normalmente por consola)

Funciones básicas

Apertura y cierre de ficheros:

```
FILE *f = fopen("file.txt", "r");
fclose(f);
```

Lectura/Escritura en crudo:

```
size_t read = fread(buf, sizeof(char), bufsize, f);
size_t written = fwrite(buf, sizeof(char), bufsize, f);
```

Lectura/Escritura sin formato:

```
fputs("Hola Mundo", f);
fgets(buf, bufsize, f);
```

Lectura/Escritura con formato:

```
fprintf(f, "Hola", name);
fscanf(f, "%10s %d", str, &i);
```

Para más información ver:

http://es.cppreference.com/w/c/io

fopen


```
FILE *fopen(const char *fname, const char *mode);
int fclose(FILE *stream);
```

Modos:

modo	significado	si ya existe	si no existe
"r"	**Abre** para **leer**	lee desde el **principio**	error
"w"	**Crea** para **escribir**	**descarta** el contenido	lo crea
"a"	**Crea** para **añadir**	escribe al **final**	lo crea
"r+"	**Abre** para **leer/escribir**	lee/escribe desde el **principio**	error
"w+"	**Crea** para **leer/escribir**	**descarta** el contenido	lo crea
"a+"	**Crea** para **leer/añadir**	lee*/escribe desde el **final**	lo crea

fwrite

size_t fwrite(const void *buf, size_t size, size_t count,FILE *strm);


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

#define FNAME "file"

int main() {
  FILE *f;
  char buf[] = "Hola Mundo";

  f = fopen(FNAME, "w+");
  if (!f) {
    perror("Error al abrir");
    exit(EXIT_FAILURE);
  }

  fwrite(buf, sizeof(char), strlen(buf), f);
  if (ferror(f)) {
    perror("Error al escribir");
    exit(EXIT_FAILURE);
  }

  fclose(f);
  return 0;
}
```

fread

Entrada/Salida

```
size_t fread(void *buf, size_t size, size_t cnt, FILE *strm);
```

Streams

Funciones

- fopen
- fwrite
- fread
- fputs
- fgets
- fprintf

```c
#define FNAME "file"
#define BUFSIZE 256

int main() {
  FILE *f;
  char buf[BUFSIZE];
  int read;

  f = fopen(FNAME, "r");
  if (!f) {
    perror("No se ha podido abrir el fichero");
    exit(EXIT_FAILURE);
  }

  read = fread(buf, sizeof(char), BUFSIZE - 1, f);
  if (ferror(f)) {
    perror("Error al leer");
    exit(EXIT_FAILURE);
  }

  buf[read] = '\0';
  printf("%s", buf);

  fclose(f);
  return 0;
}
```

fputs

Entrada/Salida

```
int fputs(const char *str, FILE *strm);
```

Streams

Funciones

- fopen
- fwrite
- fread
- fputs
- fgets
- fprintf

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

#define FNAME "file"

int main()
{
  FILE *f;
  char buf[] = "Hola Mundo";

  f = fopen(FNAME, "w+");
  if (!f) {
    perror("Error al abrir");
    exit(EXIT_FAILURE);
  }

  fputs(buf, f);
  if (ferror(f)) {
    perror("Error al escribir");
    exit(EXIT_FAILURE);
  }

  fclose(f);
  return 0;
}
```

fgets

Entrada/Salida

```
char * fgets(char *buf, int bufsize, FILE *strm);
```

Streams

Funciones

fopen
fwrite
fread
fputs
fgets
fprintf

```c
#define FNAME "file"
#define BUFSIZE 256

int main()
{
  FILE *f;
  char buf[BUFSIZE];

  f = fopen(FNAME, "r");
  if (!f) {
    perror("Error al abrir");
    exit(EXIT_FAILURE);
  }

  int i = 0;
  while (fgets(buf, BUFSIZE, f))
    printf("%d: \"%s\"\n", i++, buf);
  if (ferror(f)) {
    perror("Error al leer");
    exit(EXIT_FAILURE);
  }

  fclose(f);
  return 0;
}
```

fprintf

Entrada/Salida

```
int fprintf (FILE *strm, const char *frmt, ...) ;
```

Streams

Funciones

fopen
fwrite
fread
fputs
fgets
fprintf

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

#define FNAME "file"

int main()
{
  FILE *f;

  f = fopen(FNAME, "w+");
  if (!f) {
    perror("Error al abrir");
    exit(EXIT_FAILURE);
  }

  fprintf(f, "Hola Mundo\n 2 + 2 = %d", 2 + 2);
  if (ferror(f)) {
    perror("Error al escribir");
    exit(EXIT_FAILURE);
  }

  fclose(f);
  return 0;
}
```

fscanf

Entrada/Salida

```
int fscanf(FILE *strm, const char *frmt, ...);
```

Streams

Funciones

fopen
fwrite
fread
fputs
fgets
fprintf

```c
int main()
{
  FILE *f;
  char str1[10], str2[10];
  int a, b, c, ret;

  f = fopen(FNAME, "r");
  if (!f) {
    perror("Error al abrir");
    exit(EXIT_FAILURE);
  }

  while ((ret = fscanf(f, "%10s %10s %d + %d = %d",
                       str1, str2, &a, &b, &c)) == 5) {
    printf("%s %s\n %d + %d = %d\n", str1, str2, a, b, c);
  }
  if (ferror(f)) {
    perror("Error al leer");
    exit(EXIT_FAILURE);
  }
  else if (ret != EOF) {
    fprintf(stderr, "Error al analizar: %d/%d\n", ret, 5);
  }

  fclose(f);
  return 0;
}
```

Funciones peligrosas

Entrada/Salida

Streams

Funciones

fopen
fwrite
fread
fputs
fgets
fprintf

Todas las funciones que lean un número indefinido de bytes sin comprobar el tamaño del buffer de destino.

- `scanf` y sus variantes si no especificamos el tamaño al escanear una cadena: `"%s"`
- `gets`, que lee una cadena de `stdin` sin posibilidad de especificar ningún límite para su tamaño. Eliminada en el estándar de 2011.