SISTEMAS OPERATIVOS
GESTIÓN DE MEMORIA

Pedro de Miguel Anasagasti

NECESIDADES DE MEMORIA DEL PROCESO

Los procesos necesitan memoria para almacenar el código y los datos.

```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
      int d;
      static int e = 2;
      d = 3;
      b = d + 5;
      .......
      return;
}

int main (int argc, char *argv[]) {
      char *p;
      p = malloc (1024);
      f(b);
      .......
      free (p);
      printf ("Hola mundo\n");
      return 0;
}
```

Tipos de memoria requeridos

1.- Código

2.- Datos declarados

- Estáticos
  - Constantes + cadenas
  - Con valor inicial
  - Sin valor inicial
- Dinámicos
  - Con valor inicial
  - Sin valor inicial

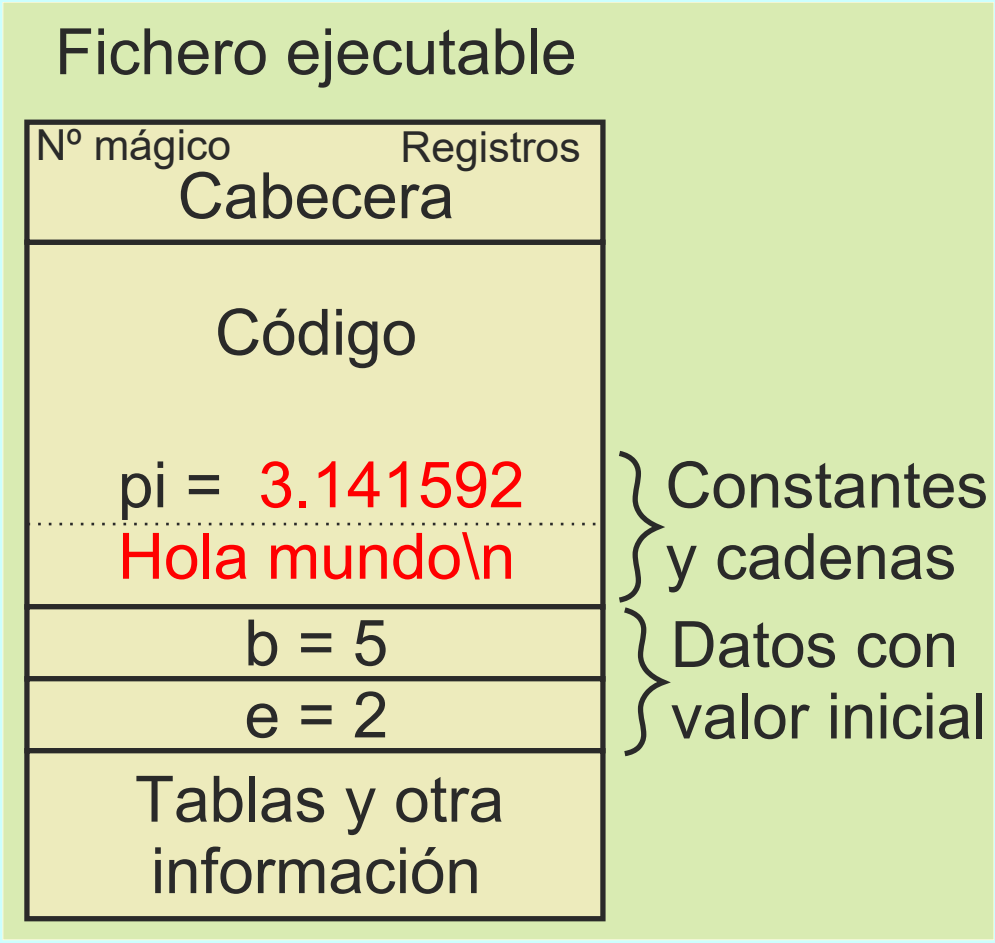
3.- Datos en bruto

- Asignación de memoria

FICHERO EJECUTABLE

Las constantes y las cadenas de caracteres deben ser inmutables, por lo que se suelen asociar al código.

```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
      int d;
      static int e = 2;
      d = 3;
      b = d + 5;
      .......
      return;
}

int main (int argc, char *argv[]) {
      char *p;
      p = malloc (1024);
      f(b);
      .......
      free (p);
      printf ("Hola mundo\n");
      return 0;
}
```



FICHERO EJECUTABLE

Los datos estáticos y con valor inicial aparecen en la zona de datos del ejecutable.

```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
      int d;
      static int e = 2;
      d = 3;
      b = d + 5;
      .......
      return;
}

int main (int argc, char *argv[]) {
      char *p;
      p = malloc (1024);
      f(b);
      .......
      free (p);
      printf ("Hola mundo\n");
      return 0;
}
```

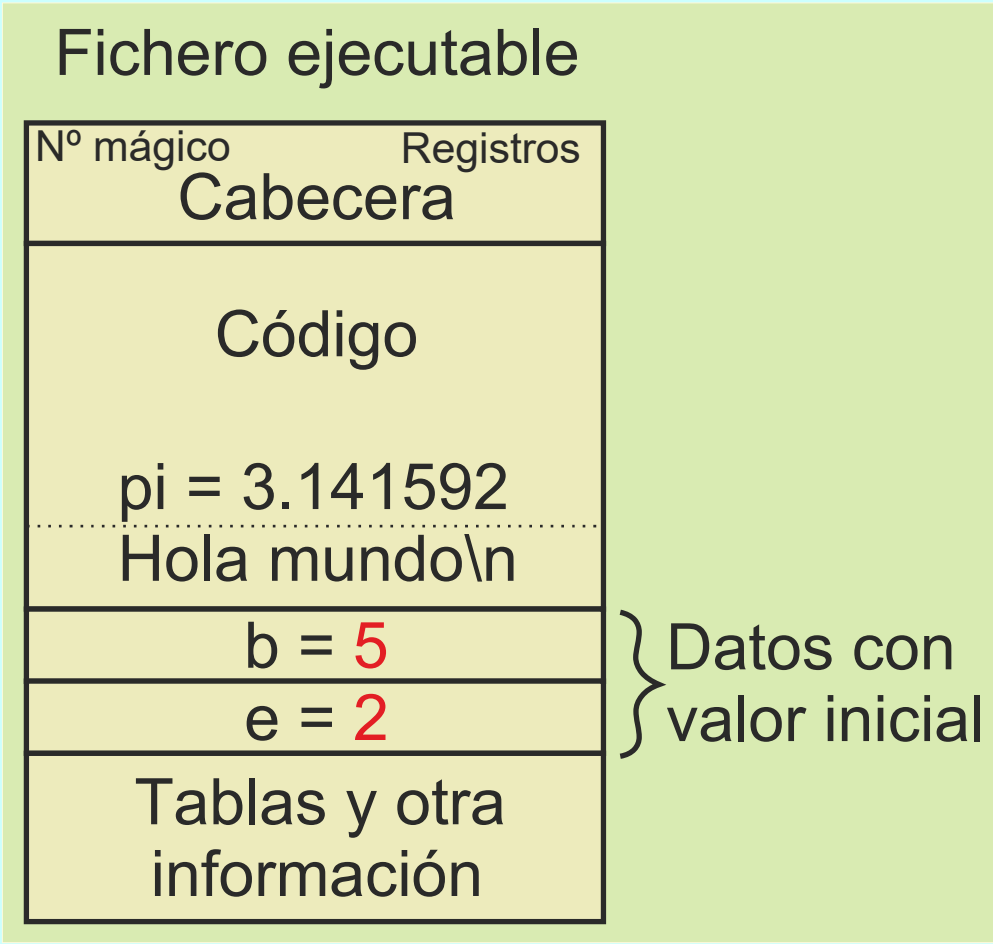


IMAGEN DE MEMORIA AL INICIO DE LA EJECUCIÓN

La imagen de memoria inicial incluye el código, los datos estáticos con y sin valor inicial, y la pila inicial.

```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
      int d;
      static int e = 2;
      d = 3;
      b = d + 5;
      .......
      return;
}

int main (int argc, char *argv[]) {
      char *p;
      p = malloc (1024);
      f(b);
      .......
      free (p);
      printf ("Hola mundo\n");
      return 0;
}
```

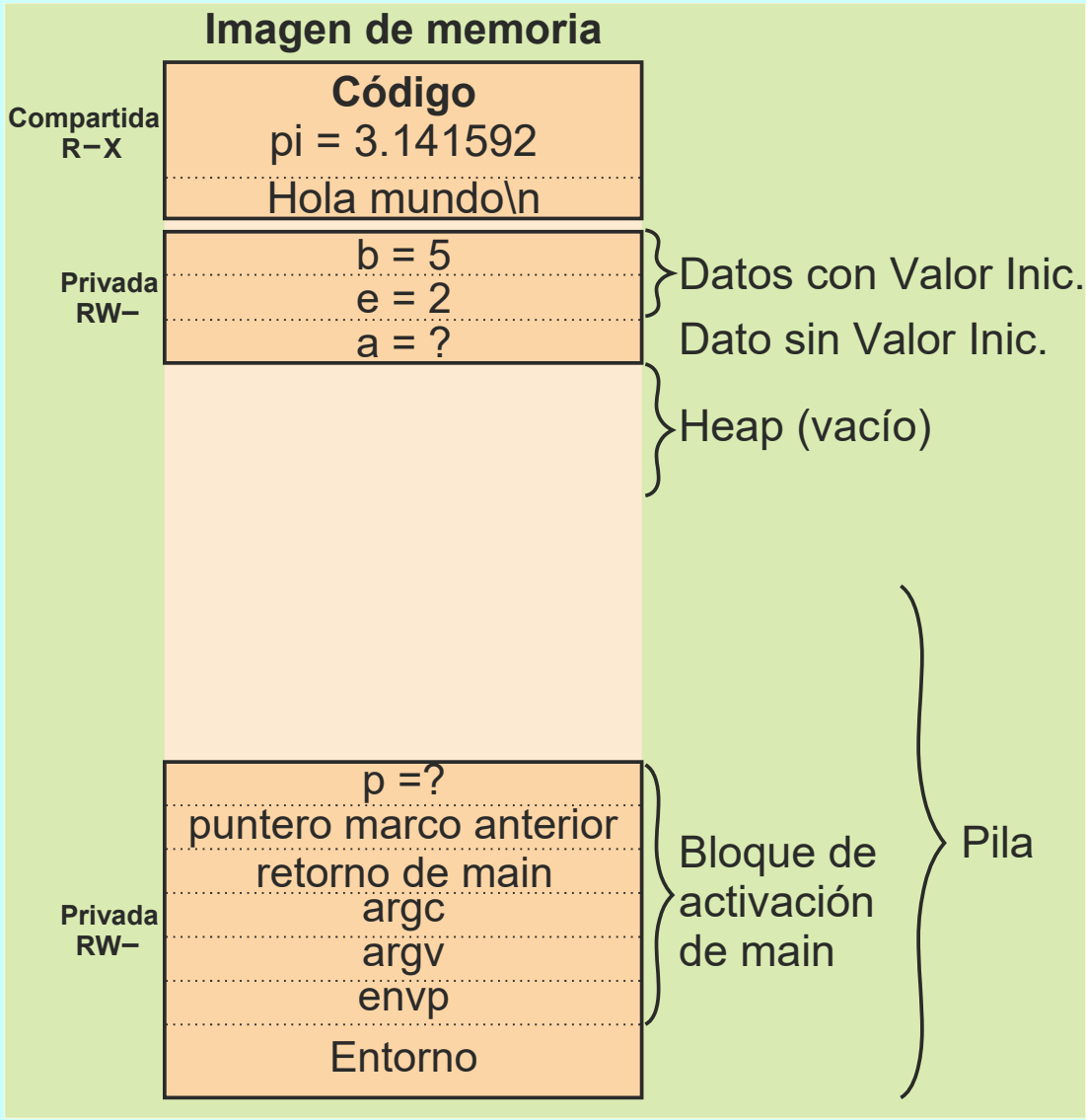


IMAGEN DE MEMORIA AL EJECUTAR MALLOC

El malloc lo resuelve la biblioteca de C. Si la región de datos no tiene libre los 1024 bytes solicitados hará una llamada al SO para aumentar la región.

```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
      int d;
      static int e = 2;
      d = 3;
      b = d + 5;
      .......
      return;
}

int main (int argc, char *argv[]) {
      char *p;
      p = malloc (1024);
      f(b);
      .......
      free (p);
      printf ("Hola mundo\n");
      return 0;
}
```

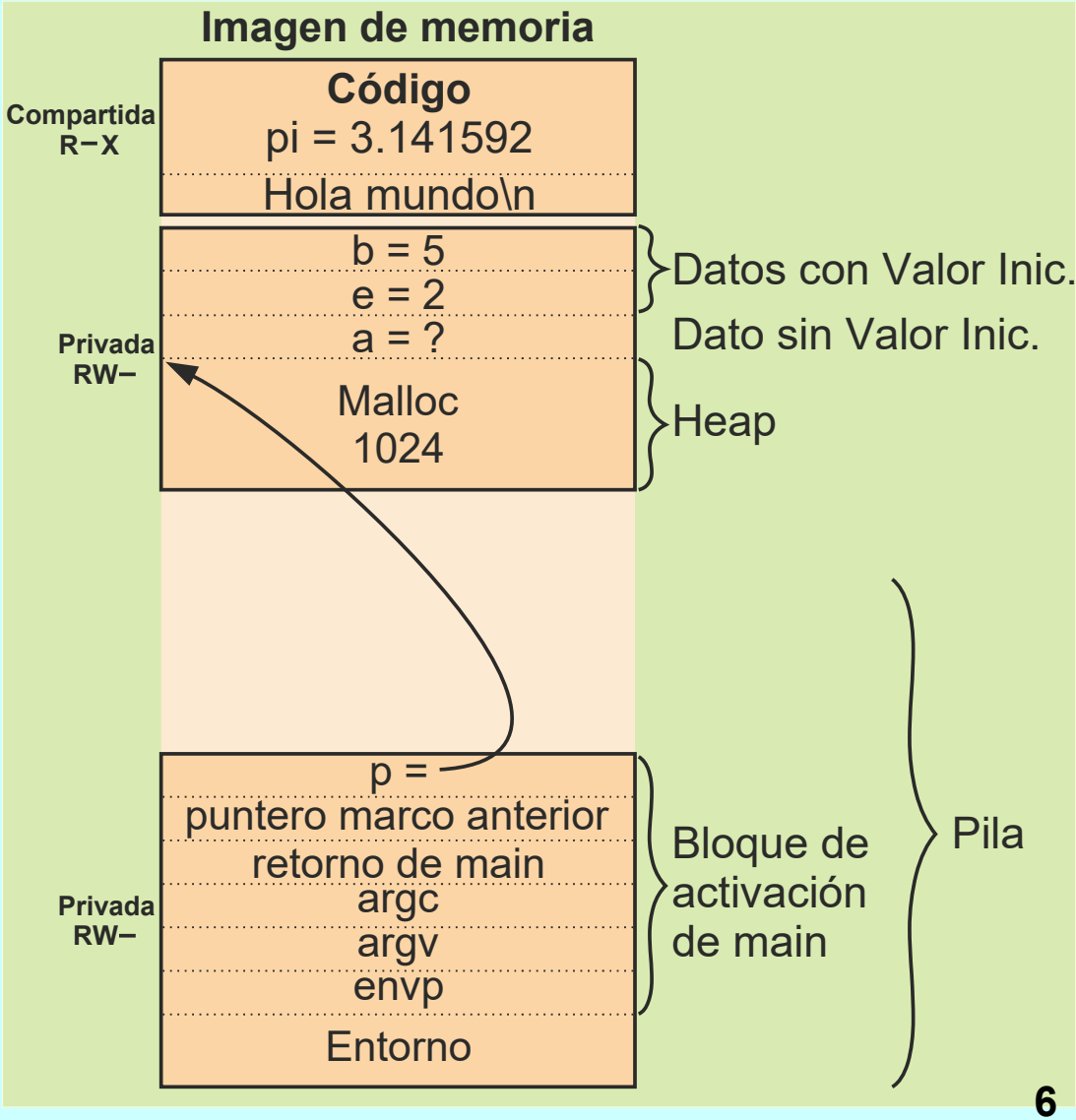
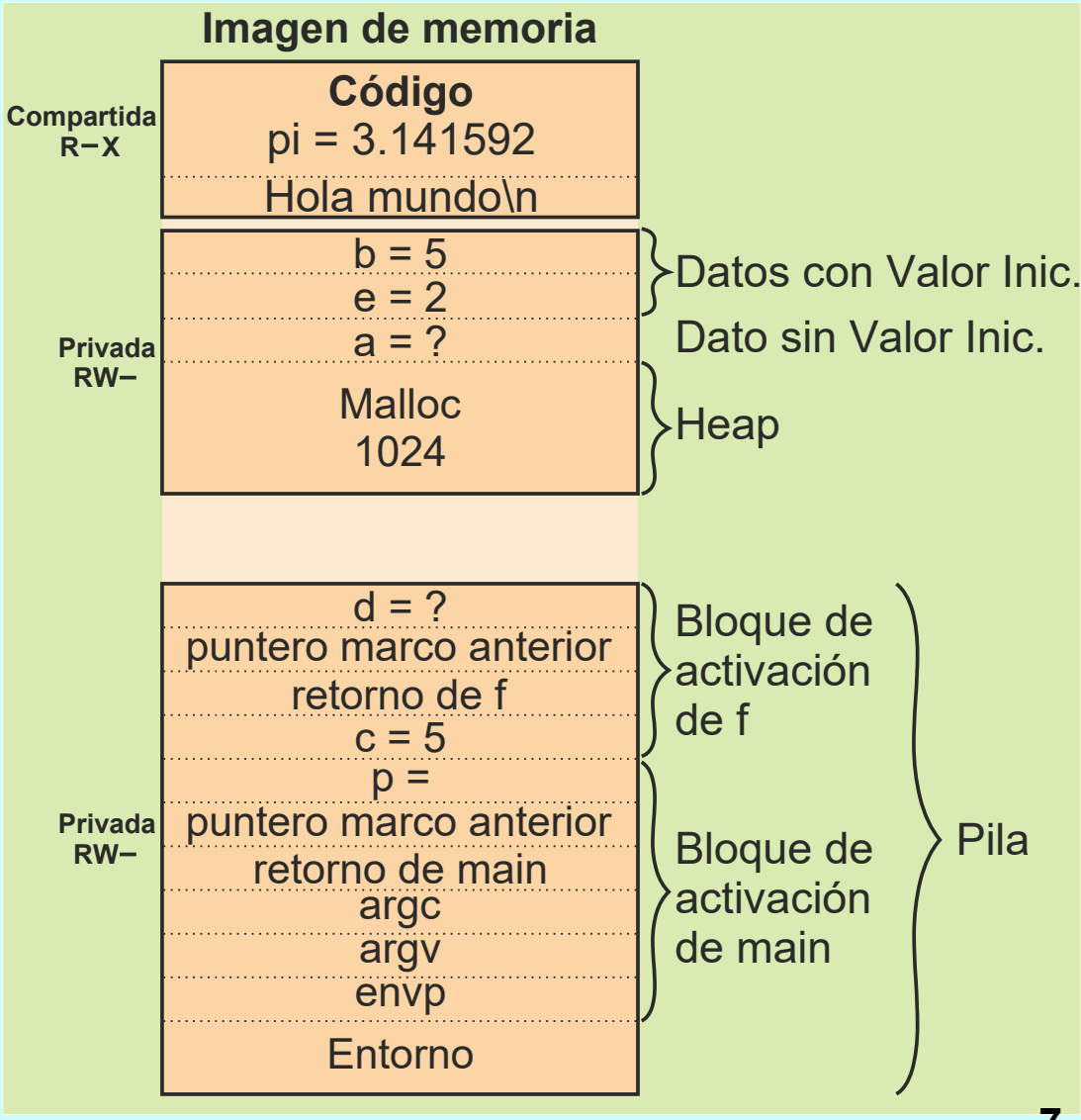


IMAGEN DE MEMORIA A LA LLAMADA A f

Al realizarse la llamada el programa crea el bloque de activación con los parámetros de invocación y las variables locales de la función.

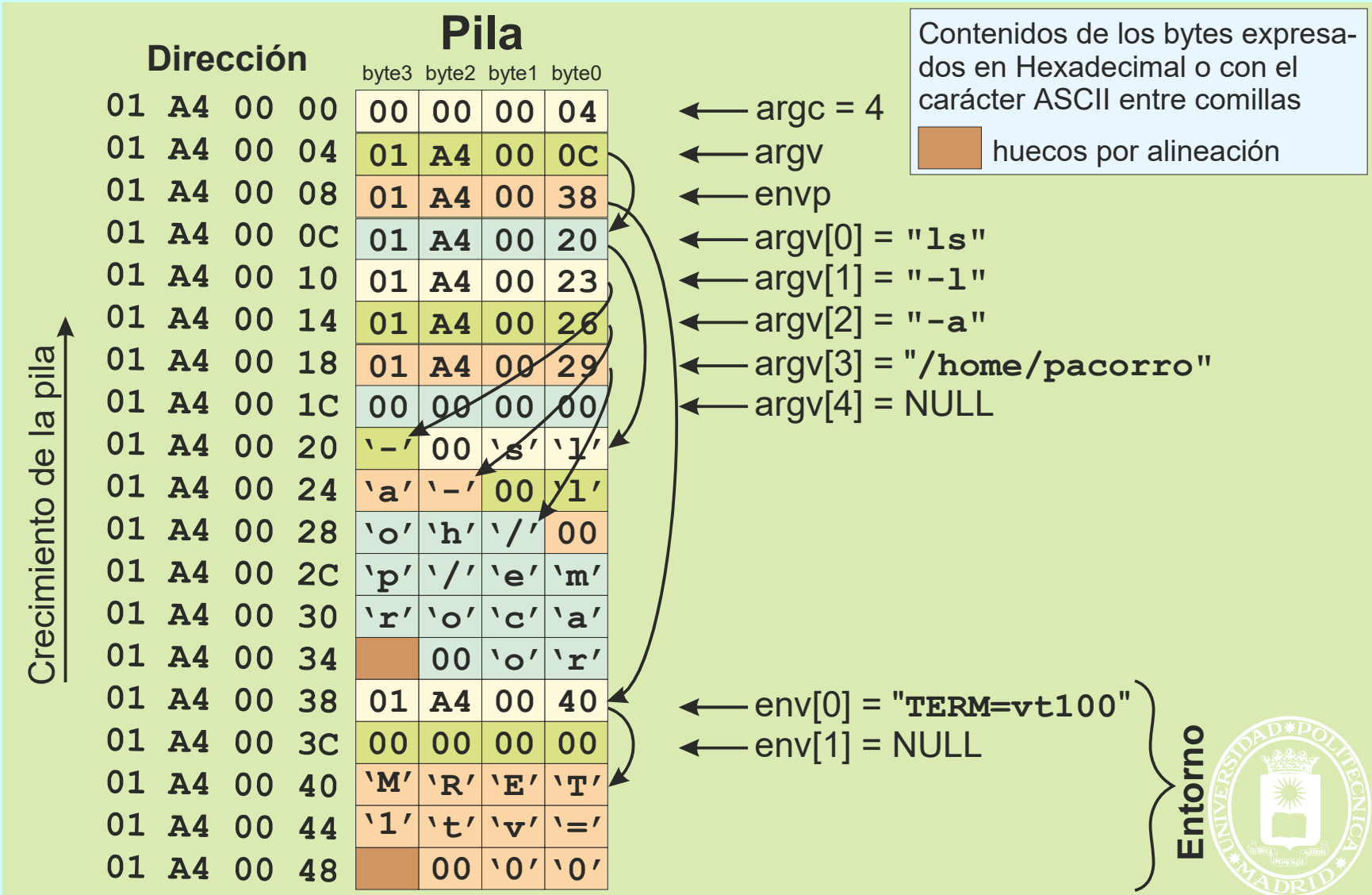
```
int a;
int b = 5;
const float pi = 3.1416;

void f(int c) {
      int d;
      static int e = 2;
      d = 3;
      b = d + 5;
      .......
      return;
}

int main (int argc, char *argv[]) {
      char *p;
      p = malloc (1024);
      f(b);
      .......
      free (p);
      printf ("Hola mundo\n");
      return 0;
}
```



EJEMPLO ARGUMENTOS DEL MAIN

Invocación: `ls -l -a /home/pacorro`

Prototipo: `int main(int argc, char* argv[], char* envp[]);`



HEAP versus BLOQUE DE ACTIVACIÓN


```
struct s {int d; char a[2000];};
struct s * fun(int a) {
        struct s *p;
        p = malloc(sizeof (struct s));
        .......
        return p;}
```

La función puede devolver p (la zona de memoria sobrevive el retorno de f)

Idóneo para funciones que crean estructuras dinámicas (listas, árboles)

```
struct s {int d; char a[2000];};
char * fun(int a) {
        struct s p;
        .....

        // no ponerreturn p.a;
```

Mucho menor coste computacional que el malloc o el new.

La zona de memoria se recupera en el retorno de f (no devolver dirección p.a).