Fundamentos de Software de Comunicaciones

El lenguaje C

Arrays, Cadenas, Estructuras y Punteros

Contenidos

- Primer programa y salida de datos
- Arrays y estructuras
- Punteros y memoria dinámica
- Argumentos a funciones
- Punteros a funciones
- Punteros y arrays
- Cadenas de caracteres. Funciones
- Entrada de datos
- Main con argumentos
- Algunas funciones útiles

Principales diferencias entre C y C++

- C es el lenguaje original con el que se creó Unix y la mayoría de dispositivos de comunicaciones se programan en él
  - Permite **acceso directo** a zonas de memoria
- Los lenguajes de C y C++ son muy compatibles
  - pero tienen distintas bibliotecas de funciones (y clases en el caso de C++)
  - Esta documentación está basada en el estándar **C99**. Ya hay un estándar C11 que lo acerca más a C++ (C++11/14)
- Principales diferencias al compilar:
  - ficheros .c (¡con sintaxis C pura!) -> utiliza **gcc**
  - ficheros .cpp (con C++ y/o C) -> utiliza **g++**

Primer ejemplo

```c
#include <stdio.h>
int main()
{
    printf("Hola\n");
    return 0;
}
```

- Guardar en un fichero con extensión .c (por ejemplo: programa.c)
- Compilar con:
  - **`gcc -o nombre_ejecutable programa.c -std=c99`**
  - Opciones extra: **`-Wall -WExtra`**
- Ejecutar con:
  - **`./nombre_ejecutable`**

Salida de datos en C

- Biblioteca de E/S:
  - **#include <stdio.h>**

- Salida por pantalla:

```
int x = 5;
char hexa = 0xFF;
char c = 'a';
printf(" texto con comodines: %d %x %c \n", x, hexa, c);
```

Salida de datos en C. Ejemplos

```c
printf("Hola\n");
printf("%s%c\n", "Hol", 'a');
int x = 5;
printf("%d %d\n", 35, x);
char y = 0xFF; //notación hexadecimal
printf("%x\n", y);
printf("%p\n", &x); /*imprime la dirección de memoria de x*/
```

¿Qué sale por pantalla?

Los arrays de C

- Permiten la reserva de un número de posiciones consecutivas de memoria
- Declaración:
  ***tipo nombre[tamaño]*; //EL TAMAÑO ES CONSTANTE**
- Inicialización:
  tipo nombre[T] = {c1, c2, .,. c_n};
  - Siempre que T esté definido como **#define T 50**
  - Ejemplo: **int a[5] = {1, 2, 3};**

  tipo nombre[] = {c1, c2, … c_n};
  - Tamaño automático

  NOTA: la asignación de un arrays a otro array es **ilegal** en C

```
char array1[50];
char array2[50];
array1 = array2;   //ERROR (se verá la causa más adelante)
```

- Acceso a un elemento:
  nombre[índice]
  (donde índice está entre [0 , tamaño – 1])

Ejemplo de arrays de C

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5    const int TAM_ARRAY = 10;
6    int arr[TAM_ARRAY];
7    int in;
8
9    for (int i=0; i < TAM_ARRAY ; ++i)
10         arr[i] = i; // inicializacion con valores
11
12   for (int i=0; i < TAM_ARRAY ; ++i)
13    cout << arr[i]; // impresion del array por pantalla
14
15    cout << "\nIntroduce el numero a buscar: " << endl;
16    cin >> in;
17
18    int j=0;
19    while( (j < TAM_ARRAY) && ( arr[j] != in) ){
20              j++;
21    }
22    if (j == TAM_ARRAY)
23      cout << "El numero " << in << "no esta en el array" << endl;
24    else
25      cout << "El numero " << in << "esta en la posicion " << j << endl;
26    return 0;
27 }
```

Resumen: arrays en C++ y en C

- C++:

```cpp
#include <tr1/array> // o <array> en C++11
const int TAM_ARRAY = 3;
//declaración con tamaño e inicialización
std::tr1::array<int, TAM_ARRAY> mi_array = {1,2,3};
//asignación elemento a elemento
mi_array[0] = 0;
//definición como tipo
typedef std::tr1::array<int, TAM_ARRAY> TipoArray;
TipoArray otro_array;
//consulta del tamaño del array (cuantos elementos tiene en uso)
otro_array.size();
```

- C:

```cpp
const int TAM_ARRAY = 3;
//declaración con tamaño e inicialización (¡de todas las casillas!)
int mi_array[TAM_ARRAY] = {1,2,3};
//declaración sin tamaño (lo resuelve el compilador)
int mi_array2[] = {1,2,3,4};    //este tiene cuatro elementos
//asignación elemento a elemento
mi_array[0] = 0;
//definición como tipo
typedef int TipoArray[TAM_ARRAY];
TipoArray otro_array;
//NO es posible consultar el tamaño ocupado de un array en C (lo cual es peligroso)
```

Definición de estructuras

- Una estructura es una colección de variables (incluidas estructuras)

```
struct nombre {
        tipo nombre_campo;
        tipo nombre_campo;
    ...
};
```

- En C/C++, una estructura puede utilizarse como una variable (asignación, paso como parámetro, valor de vuelta de una función)

- Las estructuras **no se pueden comparar**

Tamaño de datos

- En C y C++ existe un operador unario ***sizeof*** que devuelve el número de bytes que ocupa una variable o un tipo de dato en memoria
  - La longitud la devuelve como natural positivo, en un tipo denominado ***size_t***

- También se puede utilizar para calcular el tamaño en memoria de una estructura
  - **CUIDADO: SU VALOR PUEDE SER MAYOR QUE LA SUMA INDIVIDUAL DEL TAMAÑO DE SUS CAMPOS**

Tamaño de datos: ejemplos

- Ejemplos:

```
size_t longitud = sizeof(int);
int var_entera; size_t longitud = sizeof(var_entera);
```

- Tamaño de datos básicos:
  - **`char`**: 1 byte
  - **`short`**: 2 bytes
  - **`int`**: 4 bytes
  - **`long`**: 8 bytes
  - **`puntero`**: 8 bytes (sistema de 64 bits)

Tamaño de datos: ejemplos

```
struct X
{
    short s;
    int   i;
    char  c;
};
```

```
struct Y
{
    int   i;
    char  c;
    short s;
};
```

```
struct Z
{
    int   i;
    short s;
    char  c;
};
```

```
int main()
{
    struct X X;
    struct Y Y;
    struct Z Z;

    int sizeX = sizeof(X);
    int sizeY = sizeof(Y);
    int sizeZ = sizeof(Z);

    printf("X = %d; Y = %d, Z = %d\n",sizeX, sizeY, sizeZ);
}
```

- ¿Qué sale por pantalla?

Tamaño de datos: alineamiento en memoria

- Reglas de alineamiento de **`structs`**
  - Antes de cada campo habrá el relleno necesario para que empiece en una dirección que es divisible por su tamaño
    - En un sistema de 64 bits como el nuestro, un **`int`** empieza en una dirección divisible por 4, **`long`** por 8 y **`short`** por 2.
  - Los **`char`** son un tipo especial que puede ir en cualquier posición porque tienen tamaño 1
  - El tamaño de un **`struct`** se alinea con respecto del tamaño de su campo más largo.

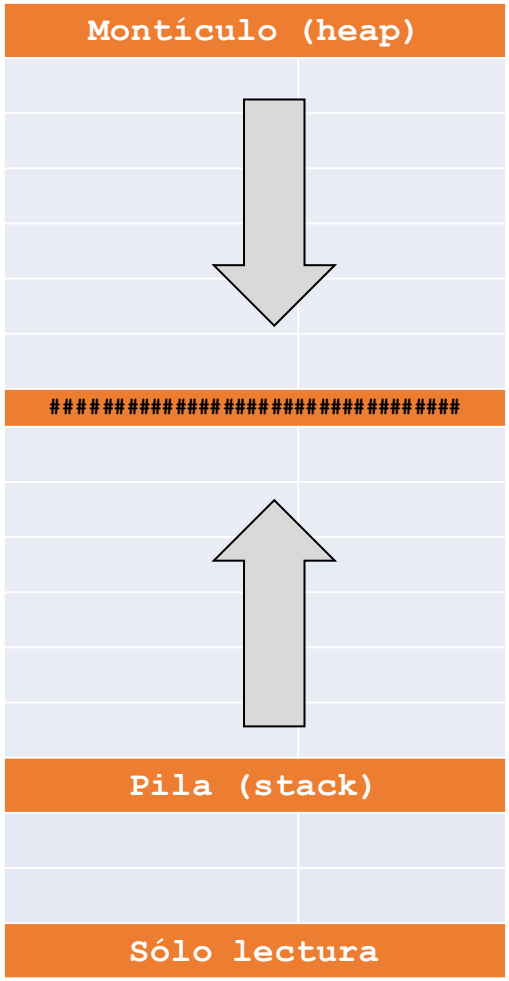
- ¿Qué tamaño tendría estas estructuras?

```
struct Datos1
{
    short s;
    char  c;
};
```

```
struct Datos2
{
    long l;
    char c;
};
```

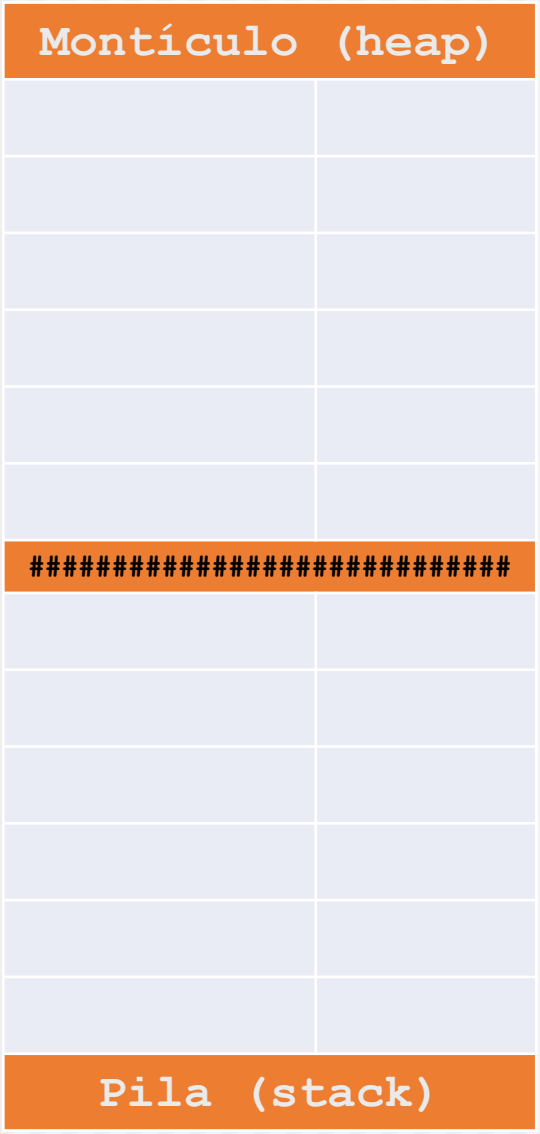
Repaso de punteros

- La memoria de un proceso en Linux se organiza en segmentos
  - Data segment (heap o montículo)
    - Memoria dinámica
  - Stack (pila)
    - Memoria estática (tiempo de compilación)
  - Sólo lectura
    - Text segment
      - Código del programa
      - Nombres de las variables
      - …
    - Bss segment



Repaso de punteros

```
1 int x = 10;
2 int *p;
3 p = &x;
4 *p = 20
5
```



p guarda la dirección de memoria de **x**

Repaso de punteros (III)

Declara un puntero a un entero

```
int x = 10;
int *p;
```

```
p = &x;
```

`&` es el operador **dirección**,
que obtiene la dirección de x

```
*p = 20;
```

Es el operador **desreferencia**,
que obtiene el valor apuntado por `p`

Memoria dinámica en C

- **`malloc()`** es el equivalente en C a **`new`** en C++
- **`free()`** es el equivalente en C a **`delete`** en C++
- Ejemplo:

```
#include <stdlib.h>
/*se reserva memoria para un entero en el heap */
int * p = (int *)malloc(sizeof(int));
*p = 7;
free(p);
```

- Si se están utilizando **`malloc()`** y **`free()`** en C++, no hay que mezclar nunca **`new`** con **`free`** ni **`malloc`** con **`delete`**

El operador flecha "->"

```
struct MisDatos{
        int dato1;
        int dato2;
};

MisDatos *p = new MisDatos;

// Acceso a campos:
(*p).dato1 = 5;

// Es equivalente a:
p->dato1 = 5;
```

Heap (montón) vs. Stack (pila)

En el Heap /
Reserva dinámica

```
void f()
{
  MisDatos *p = (MisDatos *)
    malloc(sizeof(MisDatos));
  p->dato1 = 5;
  //...
}
```

En la Pila /
Reserva automática

```
void f()
{
  MisDatos p;
  p.dato1 = 5;
  //...
}
```

¿Qué sucede cuando **p** queda fuera de alcance?