Repaso Teoría C

Indice

- Punteros
  - Sintaxis
  - Valor del puntero
  - El operador ‘&’
  - NULL
  - Error: punteros no inicializados
- Cadenas (C strings)
- Arrays y punteros
- Memoria dinámica

¿comprende este código?

```
typedef struct {
  int a;
  short s[2];
} MSG

MSG* mp, m={4,1,0};
char *fp, *tp;
mp = (MSG*)malloc(sizeof(MSG));
for (fp=(char*)m.s, tp=(char*)mp->s;tp<(char*)(mp+1);)
  *tp++ = *fp++;
```

¿Qué es un puntero?

- Un puntero es un valor que representa una referencia a otro valor, algunas veces conocido como *apuntado* (*pointee*).
- Sintácticamente C utiliza el asterisco '*'.
  - Un char* es un tipo de puntero que se refiere a un sólo carácter.
  - Un struct fraction* es un tipo que se refiere a un struct fraction. Si declaramos varias variables en la misma línea ponemos el '*' junto al nombre struct fraction *f1, *f2

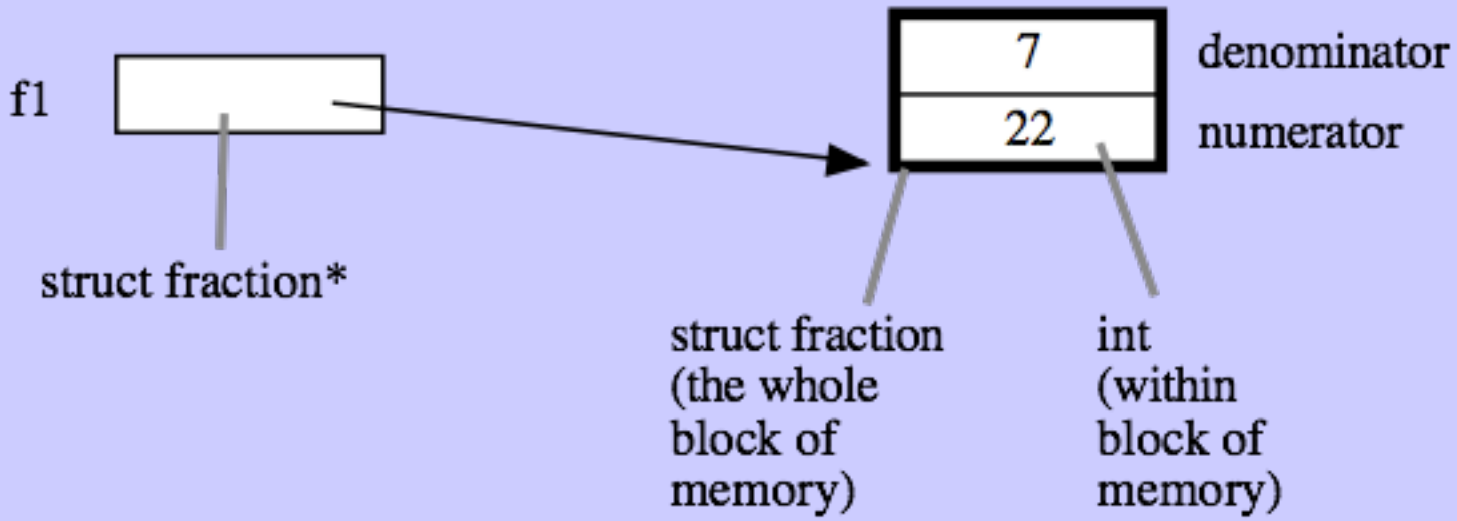
Valor del puntero

- Para extraer el valor del puntero (*pointer dereferencing*), utilizamos una expresión con el ‘*’ a la izquierda del puntero.

```
struct fraction {
        int numerator;
        int denominator;
};  // no olvidar el punto y coma
```

Ejemplo

```
struct fraction* f1;
```



Expression	Type
f1	struct fraction*
*f1	struct fraction
(*f1).numerator	int

Valor del puntero

- Otra forma de extraer el valor de un puntero en una estructura es con la flecha '→' f1→numerator (se escribe ->)

```
struct fraction** fp;                    // un puntero a un puntero de un struct fraction

struct fraction fract_array[20];         // un array de 20 struct fraction

struct fraction* fract_ptr_array[20];
                                         // un array de 20 punteros  a estructuras fracción.
```

algo más sobre sintáxis

- C evita las problemas con las definiciones circulares, como puede suceder cuando un puntero a una estructura necesita referirse a sí mismo. P ej. la definición de un nodo en una lista enlazada

```
struct node {
    int data;
    struct node* next;
};
```

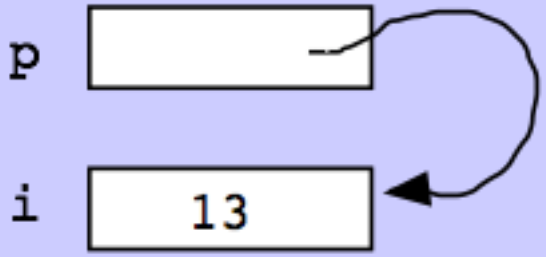
Operador ‘&’

- El operador ‘&’ es una de las formas que tienen los punteros para apuntar a *cosas*.
- El operador ‘&’ calcula un puntero para el argumento que tiene a su derecha.
- El argumento puede ser cualquier cosa que ocupe un lugar en la memoria (stack o *heap)*

```
&i;                      // Ok
&(f1->numerator);        // Ok
&6;                      // No
```

Ejemplo

```
void foo() {
   int* p; //p es un puntero a un entero
   int i; //i es un entero

   p=&i; //p apunta a i
   *p=13; //cambia lo que p apunta a --en este caso i-- a 13

   //En este punto i es 13, y tb. lo es *p. De hecho *p es i
}
```



Punteros inicializados con &i son válidos sólo mientras foo() está ejecutándose

NULL

- Asignar a un puntero el valor ‘0’ representa explícitamente que no apunta a valor alguno.
- La constante NULL está definida para ser 0.
- En una expresión lógica, NULL representa el valor *falso* (*false*).

Obtener el valor de un puntero a NULL es un ERROR, que, además puede ser difícil de detectar (depende del sistema operativo)

ERROR - Punteros no inicializados

- Cuando utilizamos punteros hay que fijarse en dos cosas: el puntero y la memoria a la que apunta (*pointee*). Hay que hacer 3 cosas para que la relación puntero\apuntado funcione:

1. El puntero debe ser declarado y situado en memoria
2. El apuntado debe ser declarado y situado en memoria
3. Debemos inicializar (1) para que apunte a (2)

ERROR - Punteros no inicializados

- Uno de los errores más frecuentes consiste en hacer el paso (1), y olvidar los pasos (2) y (3). Comenzar a utilizar el puntero como si estuviese apuntando a algo.
- Esto normalmente compila bien pero el resultado en ejecución es desastroso. Al obtener el valor con ‘*’ provoca un fallo.

Ejemplo- uso correcto de punteros

```
int* p; //(1) reserva espacio para puntero
int i; //(2) reserva espacio para el apuntado
struct fraction f1;//(2)

    p = &i;                 //(3) establece que p apunte a i
    *p = 42;                //¿podemos hacerlo?            SI

    p = &(f1.numerator);            //(3) establece que p otro int.
   *p = 22;

    p = &(f1.denominator);          //(3)
   *p = 7;
```

Strings

- Un string es un array de char, con un carácter ‘null’ (‘\0’).
- C tiene una librería de funciones para manipular strings. P.ej.

```
strcpy(char dest[], const char source[]);
```

- Copia los bytes de un string en otro
- Otra función útil es para obtener la longitud es

```
strlen(const char string[]);
```

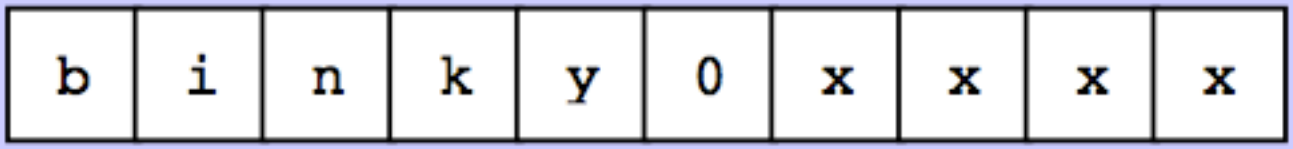
Strings

```
{
    char localString[10];

    strcpy(localString, "binky");
}
```

si quisiésemos guardar más texto en localString el programa fallaría al ejecutarse

localString

localString en realidad es un char*



Ejemplo: dar la vuelta a un string

```
{
 char string[1000]; //cadena local de 1000 char
 int len;
 strcpy(string,”binky”);
 len = strlen(string);
/*i recorre de abajo a arriba
  j recorre de arriba a abajo */
int i,j;
char temp;
for (i=0,j=len-1;i<j;i++,j--) {
    temp=string[i];
    string[i]=string[j];
    string[j]=temp;
} }
```

Array en C

- Un array está formado por un conjunto de elementos almacenados en una zona contigua de memoria.
- El array, como un todo, está relacionada con la dirección del primer elemento (dirección base del array)

Array en C

- Para referenciar cada elemento utilizo []. El programa sabe cuál es la dirección de un elemento en particular, sabiendo
  - la dirección base del array
  - el índice, y
  - el tamaño del objeto almacenado

Sintaxis ‘+’

- La expresión (intArray + 3) es del tipo (int*), y apunta al entero intArray[3].

```
(intArray+3); //¿otra expresión equivalente?
(&(intArray[3])); //las dos compilan igual

intArray[3]; //¿otra expresión equivalente?
*(intArray +3);
```

Ejemplos (1 y 2): Strcpy

strcpy(char dest[], const char source[]);

```
void strcpy1(char dest[], const char source[]) {
   int i = 0;

   while (1) {
      dest[i] = source[i];
      if (dest[i] == '\0') break;      // hecho
      i++;
   }
}
```

- La asignación se realiza en el propio test

```
void strcpy2(char dest[], const char source[]) {
   int i = 0;

   while ((dest[i] = source[i]) != '\0') {
      i++;
   }
}
```

Ejemplos (3 y 4): Strcpy

```
  //eliminando i,y solo moviendo los punteros
void strcpy3(char dest[], const char source[])
{
  while((*dest++ = *source++)!='\0'){
  }
}

    //tiene en cuenta que '0' es equivalente a FALSE
void strcpy4(char dest[], const char source[])
{
  while((*dest++ = *source++);
}
```

Consideraciones adicionales a la suma de punteros

- Al compilar, tanto [] como +, utilizan el tipo de puntero para calcular el tamaño del elemento. Mirando el código podemos imaginar que todo está en uds. de byte.

```
int *p;
p = p+12;
```

¿qué resultado provoca al ejecutarse?¿añade 12 a p?

El código incrementa p en 12 int. Cada int son 4 bytes, luego el incremento es de 48 bytes. El compilador toma esta decisión basándose en el tipo de puntero.

Casting

- Podemos hacer que el código sume 12 a la dirección del puntero p.
- Podemos decir al compilador que el puntero apunta a un char en lugar de un int. El tamaño de un char es 1 byte, sizeof(char) es 1.

```
p = (int*)(((char*)p) + 12);
```

Array y punteros

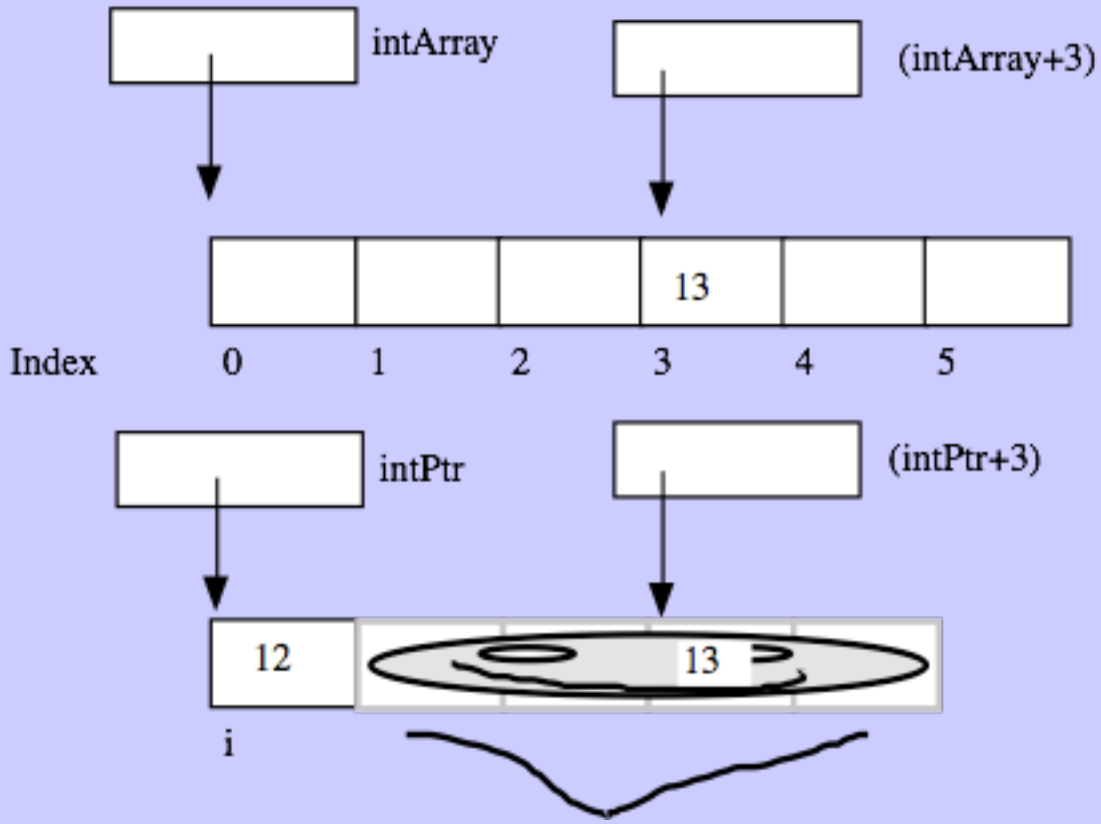
```
{
    int intArray[6];
    int *intPtr;
    int i;

    intPtr = &i;

    intArray[3] = 13;
    intPtr[0] = 12;
    intPtr[3] = 13;
}
```

//OK

//¿cambia i? SI

//¿es correcto?

NO, no hay un entero reservado ahí


intArray
(intArray+3)
13
Index
0
1
2
3
4
5
intPtr
(intPtr+3)
12
13
i

Los nombres de arrays son constantes

- La diferencia entre un puntero y el nombre de un array es que éste representa la dirección base del array y no puede cambiarse en el código.

```
{
 int ints[100];
 int *p;
 int i;

 ints = NULL;          //¿podemos hacerlo?      NO
 ints = &i;            //¿podemos hacerlo?      NO
 ints =ints + 1;       //¿podemos hacerlo?      NO
 ints++;               //¿podemos hacerlo?      NO
```

Los nombres de arrays son constantes

p = ints; //¿podemos hacerlo? Sí, p es un puntero normal

p++; //¿podemos hacerlo? Sí, pero no con ints

p = NULL; //¿podemos hacerlo? Sí

Memoria Dinámica

- Las ventajas son:
  - control de la vida de las variables que utilizamos en el programa. P ej. con memoria local las variables existen mientras ejecutamos la función.
  - control preciso sobre el tamaño. Por ejemplo, podemos guardar un string con el tamaño adecuado; con memoria local, el código reservaría un buffer de 1000 y esperaría que no se desborde.

Memoria Dinámica

- Las desventajas son:
  - Mayor trabajo: hay que detallarlo en el código
  - Más errores: provocadas por la misma razón.

<stdlib.h>

- Los prototipos de las funciones para el manejo de memoria dinámica están en <stdlib.h>.
  - void* malloc(size_t size); Solicita un bloque contiguo de memoria de tamaño determinado. malloc() devuelve un puntero al bloque, o NULL si la petición no puede realizarse. Al devolver un puntero tipo void*, habrá que realizar un casting para guardarlo en un puntero normal (con algún tipo).
  - void* free(void* block); Lo inverso a malloc(). Libera un bloque de memoria reservado por malloc(). Una vez hecho el free de un bloque de memoria, esta no puede ser accedida ni tampoco liberada por segunda vez

<stdlib.h>

- void* realloc(void* block, size_t size); Toma un bloque ya existente de memoria dinámica, e intenta cambiar su tamaño, que puede ser mayor o menor. Devuelve un puntero al nuevo bloque, o NULL. Debemos comprobar el resultado que devuelve realloc() para evitar seguir utilizando el mismo bloque de memoria. Para ello podemos utilizar assert(expr. lógica).

- Una vez que termina el programa, también se libera la memoria.

Arrays dinámicos

- Podemos guardar un array en la memoria dinámica utilizando malloc(). P. ej. para un array de 1000 enteros, podemos hacerlo en memoria local o dinámica.
- Los dos arrays se acceden igual, pero las reglas de mantenimiento son diferentes.

Arrays dinámicos

```
{
  int a[1000];

  int *b;
  b = (int*) malloc(sizeof(int) *1000);
  assert(b!=NULL); //comprobamos que reservamos
                   //un área para 1000 enteros
  a[123]; //ok
  b[123]; //ok
  free(b);
}
```

Ventajas de la memoria dinámica

- El tamaño del array se define durante la ejecución del programa. No sucede lo mismo con a[].
- Podemos reajustar el tamaño del array durante la ejecución con realloc(). Realloc () se encarga de mover la memoria del antiguo bloque al nuevo.

```
...
b = realloc(b, sizeof(int) *2000);
assert(b!=NULL);
```

Desventajas de la memoria dinámica

- Hay que recordar cómo reservar memoria y hacerlo bien.
- Hay que acordarse de liberar, *sólo una vez*, la memoria una vez que hemos terminado de usarla.
- Si no hacemos bien lo anterior, el código parece *bueno*, compila bien, y se ejecuta bien, sólo para algunos casos. Pero en otros casos puede que no, al acceder zonas de memoria inadecuadas.

Ejemplo - String dinámico

- La siguiente función coge un string como entrada, realiza una copia en la memoria dinámica. El que llama la función es responsable del nuevo string y de liberar la memoria dinámica que utiliza.

```
#include <string.h>
char* MakeStringInHeap(const char* source) {
    char* newString;
    newString = (char*)malloc(strlen(source+1);
                    //+1 por el ‘/0’
    assert(newString!=NULL);
    strcpy(newString, source);
    return(newString);
}
```