



Estructuras de Datos Dinámicas en C (Pt. 1)



Organización de Computadoras
Depto. Cs. e Ing. de la Comp.
Universidad Nacional del Sur



Copyright

- Copyright © **2011-2024** A. G. Stankevicius
- Se asegura la libertad para copiar, distribuir y modificar este documento de acuerdo a los términos de la **GNU Free Documentation License**, Versión 1.2 o cualquiera posterior publicada por la Free Software Foundation, sin secciones invariantes ni textos de cubierta delantera o trasera
- Una copia de esta licencia está siempre disponible en la página <http://www.gnu.org/copyleft/fdl.html>
- La versión transparente de este documento puede ser obtenida de la siguiente dirección:

<http://cs.uns.edu.ar/~ags/teaching>



Contenidos

- Concepto de puntero
- Pasaje por valor y por referencia
- Tipos de datos estructurados
- Gestión de la memoria dinámica
- Estructuras de datos dinámicas
- Concepto de posición directa e indirecta
- Fases de la compilación
- Parámetros en la línea de comandos



Punteros

- Las variables declaradas de tipo **puntero** representan direcciones de memoria
 - ➔ Tener bien presente la diferencia entre una variable de tipo puntero y el dato alojado en la dirección por él apuntada
- Los punteros se declaran con un **asterisco** delante del identificador de la variable:

```
int *px, y; /* px es un puntero e  
y es un entero */
```



Operaciones sobre punteros

- En relación a los punteros, existen dos operadores fundamentales:
 - ➔ Des-referenciamiento: si **px** es un puntero, la expresión ***px** denota al contenido apuntado por el puntero (es decir, el valor almacenado en la dirección referenciada por el puntero)
 - ➔ En-referenciamiento: si **x** es una variable, la expresión **&x** denota a la dirección de memoria representada por esa variable (es decir, se construye dinámicamente un puntero que apunta a esa variable)



Operaciones sobre punteros

```
int x = 10, y = 5;
```

```
int *px;
```

```
px = &x;
```

```
*px = &y;
```

```
px = 1234; // ¿tendrá sentido?
```

1000:	10	int x
1004:	5	int y
1008:	?	int *px

1000:	1004	int x
1004:	5	int y
1008:	1000	int *px



Aritmética sobre punteros

- Los punteros se comportan como cualquier otro tipo elemental, es decir, admiten toda la gama de operaciones aritméticas
 - La clave para entender esto radica en que **los punteros almacenan** en esencia direcciones de memoria, en otras palabras, **enteros positivos**
 - Las operaciones aritméticas se comportan de manera inteligente, **teniendo en cuenta el tipo base** al cual apuntan los punteros



Aritmética sobre punteros

```
int vector[5] = {10, 20, 30, 40, 50};  
int *pv = vector; // aquí, ¿qué denota *pv?  
*pv = 15; // ¿y aquí?  
*(pv + 3) += 5;
```

```
char string[250] = {'M', 'a', 'r', 'í', 'a'};  
char *ps = &string[2];  
*(ps + 2) = 'o'; // ¿en qué cambió string?
```



Pasaje por referencia

- Los punteros permiten **simular** un pasaje de parámetros por referencia:

```
int reset(int *a, int b) {  
    *a = 0; b = 0; // *a y b son reseteados  
}  
  
void main() {  
    int x = 1, y = 1;  
    reset(&x, y); // x vale 0, pero y vale 1  
}
```



Pasaje por referencia

```
int a = 10, b = 20;
void cambiar(int p, int *q) {
    p += 2;
    *q += p;
}
int main () {
    cambiar(a, &b);
    printf("a vale %i y b vale %i", a, b);
}
```



Punteros genéricos

- Para declarar un **puntero genérico**, el cual sea capaz de apuntar datos de distintos tipos, se debe apelar al tipo **void**
- Este tipo **solo puede figurar como tipo base de un punteros**, es decir, declarar a una variable común de tipo **void** carece de sentido

```
void *px; // válido
```

```
void x; // inválido
```



Definición de arreglos

- La definición de arreglos y matrices se realiza indicando las dimensiones entre corchetes:

// un arreglo de 25 enteros

```
int a[25]
```

// una matriz de reales de 4x4

```
float vx[4][4];
```

// un arreglo de caracteres

```
char string[250];
```



Indexado de arreglos

- A diferencia de otros lenguajes, en **C** la primer componente de un arreglo ocupa la posición 0:

```
int i, vector[10];
```

```
for (i = 0; i < 10; i++) { // ¡empieza en 0  
    vector[i] = i;          // y termina en 9!  
}
```



Inicialización de arreglos

- Los arreglos en C pueden inicializarse al mismo tiempo que son definidos:

```
int vector[5] = {0, 1, 2, 3, 4};
```

```
int matrix[2][3] = { {1, 2, 3},  
                    {4, 5, 6} };
```

```
char cadena1[250] =  
    {'H', 'o', 'l', 'a', '!'};
```

```
char cadena2[250] = "Hola!";
```

```
// ¿será cadena1 == cadena2?
```



Cadenas de caracteres

- En este lenguaje las **cadenas de caracteres** son simples arreglos de caracteres que terminan en un caracter nulo (' \0 '):

```
char cadena1[250] =  
    {'H', 'o', 'l', 'a', '!', '\0'};  
char cadena2[250] = "Hola!";
```

```
/* ahora cadena1 y cadena2  
   resultan equivalentes */
```



Asignación de arreglos

- El operador de asignación no puede utilizarse con arreglos ni con cadenas de caracteres

```
int a[5], b[5];
```

```
a[0] = 1;
```

```
a[2] = 3;
```

```
b = a; // ¡Error!
```

- No confundir con la inicialización de arreglos, la cual si está permitida:

```
int c[5] = {0, 3, 2, 0, 0};
```



Funciones de librería

● C dispone de una gran cantidad de **funciones de librería sobre cadenas de caracteres** (presentes en la librería **strings**):

- **strcat()**: para concatenar strings
- **strcpy()**: para copiar strings
- **strcmp()**: para comparar dos strings lexicográficamente (no es una función booleana)
- **strlen()**: para determinar la longitud de un string
- **strstr()**: para buscar un string en otro



¡Acá los LLMs brillan!

A explicame con un ejemplo cómo funciona strcmp()

Edit

Claro, te explicaré cómo funciona la función `strcmp()` con un ejemplo. Esta función se utiliza en C para comparar dos cadenas de caracteres.

`strcmp()` compara las cadenas carácter por carácter y devuelve un valor entero basado en el resultado de la comparación:

- Devuelve 0 si las cadenas son iguales.
- Devuelve un valor negativo si la primera cadena es lexicográficamente menor que la segunda.
- Devuelve un valor positivo si la primera cadena es lexicográficamente mayor que la segunda.

Aquí tienes un ejemplo de código que muestra cómo se usa `strcmp()`:

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

int main() {
    char str1[] = "hola";
    char str2[] = "hola";
    char str3[] = "adios";
    char str4[] = "zelda";

    int result1 = strcmp(str1, str2);
    int result2 = strcmp(str1, str3);
    int result3 = strcmp(str1, str4);

    printf("Comparación de '%s' y '%s': %d\n", str1, str2, result1);
    printf("Comparación de '%s' y '%s': %d\n", str1, str3, result2);
    printf("Comparación de '%s' y '%s': %d\n", str1, str4, result3);

    return 0;
}
```

Este código comparará las siguientes cadenas:

1. "hola" con "hola"
2. "hola" con "adios"
3. "hola" con "zelda"

La salida será algo como:

Copy

```
Comparación de 'hola' y 'hola': 0
Comparación de 'hola' y 'adios': 4
Comparación de 'hola' y 'zelda': -18
```

Explicación de los resultados:

1. `strcmp(str1, str2)` devuelve 0 porque las cadenas son idénticas.
2. `strcmp(str1, str3)` devuelve un valor positivo porque "hola" es lexicográficamente mayor que "adios".
3. `strcmp(str1, str4)` devuelve un valor negativo porque "hola" es lexicográficamente menor que "zelda".

Los valores exactos pueden variar dependiendo de la implementación, pero el signo (positivo, negativo o cero) es lo importante.

¿Te gustaría que explique algún aspecto específico de `strcmp()` o que proporcione más ejemplos?

Entrada estándar de strings

- El modificador “%s” permite el ingreso de cadenas de caracteres por parte del usuario:

```
char str[25];
```

```
scanf(“%s”, str); // ¿¿no falta algo??
```

```
/* lee hasta encontrar un separador
```

```
(blancos, fin de líneas, etc.) */
```

- Para leer un string hasta el fin de línea se debe usar el modificador “%[^\n]”:

```
scanf(“%[^\n]”, str); // lee hasta el '\n'
```



Arreglos y punteros

- Una variable de tipo arreglo se asimila a un puntero a su primer componente
- En consecuencia, puede ser utilizada como tal:

```
int *pb, *pc;
```

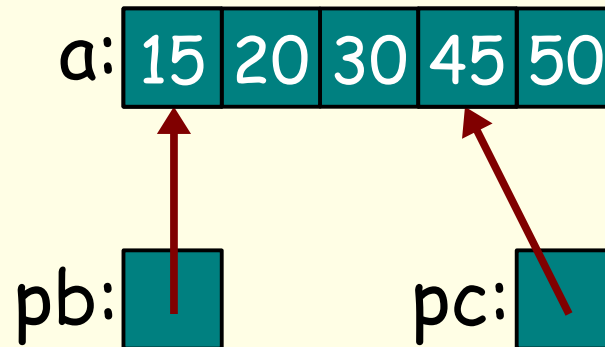
```
int a[5] = {10, 20, 30, 40, 50};
```

```
pb = a;
```

```
*pb = 15;
```

```
pc = &a[3];
```

```
*pc += 5;
```



Pasaje de arreglos

- Al pasar un arreglo como argumento, se debe dejar su primera dimensión sin especificar:

```
void times(int vector[], int matrix[][4]) {  
    ...  
}  
  
int main() {  
    int a[12], b[4][4];  
    times(a, b);  
}
```



Pasaje de arreglos

- Producto de esta omisión, no resulta del todo trivial determinar la longitud de los arreglos recibidos como argumento
 - Una alternativa bastante directa consiste en **agregar un parámetro adicional** que informe el valor de la dimensión faltante
 - Otra posibilidad es **usar alguna variable global** para comunicar ese dato
 - Una tercer propuesta es **acordar que el arreglo tenga un tamaño predeterminado**, haciendo uso por caso de una constante para denotar ese tamaño



Usando un parámetro extra

```
void mostrar(int vx[], int size) {  
    int i;  
    for (i = 0; i < size; i++)  
        printf("Elto nro. %i = %i\n", i, vx[i]);  
}  
  
int main() {  
    int a[5] = {10, 20, 30, 40, 50};  
    mostrar(a, 5);  
}
```



Usando una variable global

```
int size = 5;
void mostrar(int vx[]) {
    int i;
    for (i = 0, i < size, i++)
        printf("Elto nro. %i = %i\n", i, vx[i]);
}
int main() {
    int a[5] = {10, 20, 30, 40, 50};
    mostrar(a);
}
```



Registros

- En ocasiones hace falta manipular **múltiples datos de una cierta entidad**
 - ➔ Por caso, nombre, apellido y número de registro de un alumno
- También con frecuencia hay que retener datos de **múltiple entidades**
 - ➔ ¿Qué estructura de datos resulta más conveniente usar?
 - ➔ ¿Múltiples arreglos, uno para cada “atributo” de las entidades?



Definición de un registro

- Los **registros** son un tipo de dato estructurado compuesto de un conjunto de campos, los que son de otros tipos (básicos o complejos) y que se les asocia una etiqueta a cada uno:

```
struct etiqueta {  
    tipo1 campo1;  
    tipo2 campo2;  
    ...  
} variable1, variable2, ...;
```



Definición de un registro

```
struct persona {  
    char nombre[20];  
    int edad;  
    float altura;  
} profesor, alumnos[10];
```

```
struct persona unprofesor =  
    {"Juan Perez", 32, 1.82};  
struct persona *palumno;
```



Acceso a los campos

- Los campos se acceden de dos maneras:
 - Usando el operador punto (.), si es un registro
 - O bien usando el operador flecha (->), si se trata de un puntero a un registro

```
struct persona el, *ella, todos[20];  
printf("Nombre: %s\n", el.nombre);  
todos[2].edad = 20;  
ella = &todos[2];  
printf("Su edad es %d\n", ella->edad);
```



Pasaje de registros

- En general, ninguna estructura de datos compleja debe ser pasada por valor
 - La razón es que esto implica el copiado en tiempo de ejecución de mucha información
 - Por ende, los registros no deben ser pasados como argumentos, al menos no de forma directa
- En otras palabras, siempre conviene **pasar por valor un puntero a la estructura** en cuestión
 - En todas las arquitecturas, los punteros ocupan apenas unos bytes



Definición de enumerados

- Los **enumerados** son conjuntos de constantes numéricas definidas por el usuario

```
enum colores {rojo, verde, azul};
```

```
enum colores fondo, borde = verde;
```

```
enum booleano { falso = 0,  
                verdadero = 1 };
```

```
enum booleano condicion = falso;
```



Definición de tipos de datos

- Para definir nuevos tipos de datos a partir de otros ya definidos se usa la sentencia **typedef**:

```
typedef int booleano;  
typedef struct persona tPersona;  
typedef struct punto {  
    int coordenadas[3];  
    enum colores color;  
} tPunto;  
tPunto plano[3];
```



Modificadores de variables

- La declaración de variables acepta los siguientes modificadores:
 - ➔ **static**: el valor de la variable se debe conservar entre las llamadas a la función
 - ➔ **register**: la variable es almacenada, de ser posible, en un registro del procesador, en vez de hacerlo en memoria principal
 - ➔ **volatile**: la variable puede ser modificada por un proceso exterior
 - ➔ **const**: la variable no puede ser modificada, sólo inicializada



Modificadores de variables

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

```
void count() {
```

```
    static int acc = 0;
```

```
    printf("%d\n", acc++);
```

```
}
```

```
int main() { // ¿qué valores se imprimen?
```

```
    count(); count(); count();
```

```
    return 0;
```

```
}
```



Modificadores de variables

```
const int max = 10;
```

```
int letra(const char *text, char l) {  
    int i, acc = 0;  
    for (i=0; i < max && text[i]; i++)  
        if(text[i] == l)  
            acc++;  
    return acc;  
}
```



Declaraciones constantes

- Existen dos formas de declarar una variable como constante:

```
const int x = 5;
```

```
int const x = 5;
```

- Pero, al tratarse de punteros no da lo mismo:

```
const char * origen;
```

```
char * const origen;
```

- Consejo práctico: ¡leer la declaración siempre de atrás para adelante!



Modificadores de funciones

- Las funciones también pueden ser declaradas con otros modificadores:
 - **static**: esta es una restricción de enlace. Denota que sólo se puede usar dentro del archivo de código fuente actual
 - **extern**: la variable o función en cuestión será declarada pero no será definida (su definición será provista en otro archivo fuente)
 - **inline**: la función debe ser expandida íntegramente al ser invocada (es decir, no se va a generar un salto a la función)



Modificadores de funciones

uno.c

```
static void f() {  
    ...  
}  
  
void g() {  
    f();  
}
```

dos.c

```
extern void f();  
extern void g();  
  
int main() {  
    g();  
    f(); // ¡Error!  
}
```



¿Preguntas?

