Cadenas

- Lenguaje C no maneja cadenas (strings) como un elemento del lenguaje, sino que lo asimila a secuencia de caracteres con un “centinela” (`'\0'`) que indica el final de la secuencia.
- Reservar memoria para las cadenas es tarea del programador. Los métodos habituales son declarar un vector de caracteres o usar la función `malloc()`.
- La lógica para manipular cadenas es recorrerlas y operarlas carácter a carácter.
- La biblioteca estándar string.h provee varias funciones para operar con cadenas.

Definir como vector

- Si declaro como un vector
  - `char palabra[10];`
    - Permite guardar cadenas de hasta 9 caracteres, ya que debemos contemplar el `'\0'` final.
  - `char frase[] = "Hola mundo";`
    - El vector frase se dimensiona automáticamente de 11 posiciones (10 caracteres + `'\0'` final)
  - `char frase[100] = "Hola mundo";`
    - Similar al caso anterior, con las posiciones no inicializadas explícitamente en cero, como cualquier vector.
    - **OJO**: El estándar no permite inicializar con cadena de más elementos que la dimensión del vector. Si pongo un literal del mismo largo que la dimensión del vector, **NO** pone el `'\0'` final.

Definir como vector

- El uso de literal cadena es equivalente a haber inicializado cada posición :

```
char palabra[5] = "Hola";
char palabra[5] = {'H', 'o', 'l', 'a', '\0'};
```

- Como con cualquier vector **solo sirve para inicializar**, la siguiente secuencia es **incorrecta**
  - `char palabra[10];`
  - `palabra = "Hola";`
- El modo **correcto** sería
  - `char palabra[10];`
  - `strcpy(palabra, "Hola"); //strcpy en string.h`

Definir con punteros

- Puedo simplemente definir un puntero a char, pero eso no reserva memoria para los datos, debo hacerlo aparte

```
char *cadena;
cadena = malloc(20);
strcpy(cadena, "Hola");
```

- Si la zona de memoria ya está reservada puede apuntarla directamente

```
char buffer[512];
char *cadena = buffer;
char *saludo = "Hola";
```

- El tipo de dato de un literal cadena es arreglo de caracteres, que decae a `const char*`. Puede que no sea modificable

Funciones de string.h

- `size_t strlen(const char *s);`
  - Devuelve la cantidad de caracteres en la cadena s (sin incluir el '\0')
- Ejemplo

```
char pais[20] = "Argentina";
char *p = malloc(strlen(pais) + 1);
strcpy(p, pais);
```

- Más simple:

```
char pais[20] = "Argentina";
char *p = strdup(pais);
//strdup NO estandar de C (es posix)
```

Duplicar cadena

- `char *strdup(const char *s);`
  - Duplica la cadena s, pide memoria con malloc y copia la cadena. Devuelve puntero a la cadena duplicada o NULL si no hay suficiente memoria
- `char *strndup(const char *s, size_t n);`
  - Similar a la anterior pero copia a lo sumo n caracteres y luego agrega `'\0'`
- Usar `malloc` y `strcpy` si `strdup` no está disponible (p.ej: en mingw)

Copiar cadena

- `char *strcpy(char * restrict s1, const char * restrict s2);`
  - Copia la cadena apuntada por s2 en la memoria apuntada por s1. Si al copiar hay solapamiento el resultado es indefinido. Devuelve el valor de s1
- `char *strncpy(char * restrict s1, const char * restrict s2, size_t n);`
  - Similar a la anterior pero copiando a lo sumo n caracteres. Esto implica que puede **NO** copiar el `'\0'` final!!!. Si largo de la cadena copiada es menor a n, el resto de las posiciones se llenan con `'\0'`.

Concatenar cadena

- ```
  char *strcat(char * restrict s1,
               const char * restrict s2);
  ```
  - Agrega una copia de la cadena apuntada por s2 al final de la cadena s1. Si hay solapamiento el resultado es indefinido. Devuelve el valor de s1
- ```
  char *strncat(char * restrict s1,
                const char * restrict s2,
                size_t n);
  ```
  - Similar a la anterior pero agrega a los sumo n caracteres de s2. Si al copiar n caracteres el último NO es un `'\0'`, lo **agrega**. Significa que la cadena resultante puede llegar a ser de largo strlen(s1)+n+1

Comparar cadenas

- `int strcmp(const char *s1, const char *s2);`
  - Compara las cadenas apuntadas por s1 y s2. El entero devuelto será:
    - `> 0` si `s1 > s2`
    - `< 0` si `s1 < s2`
    - `= 0` si `s1 = s2`
- `int strncmp(const char *s1, const char *s2, size_t n);`
  - Similar al anterior pero compara solo hasta el carácter n, o menos si alguna de las dos cadenas termina antes.

Ejemplo a

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

int main()
{
        char s1[30] = "Hola";
        char *s2 =", que tal!";
        if (strlen(s1) + strlen(s2) < sizeof(s1))
                strcat(s1, s2);
        printf("%s\n", s1);
        //Imprime: Hola, que tal!
        return EXIT_SUCCESS;
}
```

Ejemplo b

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

char *strmin(const char* str1, const char*str2)
{
        if (strcmp(str1, str2) < 0) {
                //cuando está disponible posix
                return strdup(str1);
        } else {
                // Cuando posix no está disponible
                char *temp = malloc(strlen(str2) + 1);
                strcpy(temp, str2);
                return temp;
        }
}
```

Ejemplo b (continuación)

```
int main()
{
        char *smin;
        char cand1[50];
        char cand2[50];
        printf("Nombre del primer candidato: ");
        scanf("%[^\n]%*c",cand1);
        printf("Nombre del segundo candidato: ");
        scanf("%[^\n]",cand2);

        smin = strmin(cand1, cand2);
        printf("Debe listarse primero a: %s\n", smin);
        free(smin);
        return EXIT_SUCCESS;
}
```

Estructuras

- Una estructura, conocida en otros lenguajes como registro, permite agrupar en una unidad un conjunto, posiblemente heterogéneo, de variables relacionadas.
- Para poder definir una variable de tipo estructura primero debemos declarar el tipo completo, con sus miembros. Luego podremos definir la variable de este “tipo estructura”

```
struct empleado {//declaro el tipo
        int legajo;
        char nombre[30];
};

struct empleado e1, e2; //defino variables
```

Declaración y definición

- Puedo declarar la estructura y definir las variables en un solo paso

```
struct punto {
        double x;
        double y;
} pto1, pto2;
```

- Puedo definir una variable de una estructura anónima

```
struct { //anónima (falta nombre del struct)
        int i, j; //posible pero desaconsejado
} anon1;
```

Inicialización

- Puedo incializar valores al declarar la variable

```
struct punto pto3 = { 1.0, 2.0};
struct empleado e3 = { 525, "Pablo"};
```

- Puedo hacer todo junto (declarar e inicializar)

```
struct ejemplo {
        int a;
        int b;
} vejemplo = {1, 5};
```

- A partir de C99 se puede incializar en cualquier orden si “designo” el campo al que asigno.

```
struct punto pto = {.y = 2.0, .x = 1.0};
/* idéntico a pto = {1.0, 2.0} */
```

Uso

- Para acceder a un miembro de una estructura uso el operador . (punto) si lo que tengo es una variable

```
e1.legajo = 123;
strcpy(e1.nombre, "Juan");
```

- Puedo asignar para copiar TODOS los campos

```
e2 = e1;
```

- Puedo pasar una estructura como parámetro o devolverla como resultado de una función

Punteros

- Si declaro un puntero a una estructura y quiero acceder a un campo, por tema de precedencias debo usar paréntesis

```
struct punto *p;
p = &pto;
(*p).x = 5.0;
```

- Como esto es engorroso se definió el operador -> que es lo que se acostumbra utilizar.

```
p->x = 5.0; //equivalente a (*p).x = 5.0;
```

typedef y autoreferencias

- Se puede declarar una estructura dentro de otra y es posible tener un miembro de tipo puntero a la estructura que lo contiene

```
struct enlazada {
    int clave;
    struct {
        double medida;
        char *descrip;
    } datos;
    struct enlazada *siguiente;
};
typedef struct enlazada *ptr2en;
```

```
struct enlazada enl01, enl02;
enl01.clave = 1;
enl01.datos.medida = 25.6;
enl01.siguiente = &enl02;

ptr2en primero = &enl01;
primero->siguiente->clave = 2;
/*equivale a: enl02.clave = 2 */

struct enlazada enl03 = {1, {3.0}};
/*descrip y siguiente puestos en
NULL */
```

Modo de almacenamiento

- Así como un vector calcula un desplazamiento para cada elemento, en una estructura hay un desplazamiento para cada miembro, pero este no se puede calcular (el compilador arma una tabla)
- El sizeof de una estructura es mayor o igual a la suma de los sizeof de sus miembros. Esto se debe a las conveniencias de alineación, pudiendo dejar espacios (en inglés gap o filler) entre campo y campo o al final de la estructura (para que en un arreglo la siguiente estructura esté bien alineada).
- En stddef.h se define la macro offsetof(type, member-designator) que permite conocer el desplazamiento de un miembro en particular.

Uniones

- Al declarar una estructura se asigna memoria para todos sus miembros, uno detrás de otro, en el orden en que fueron declarados, con posibles espacios intermedios por cuestiones de alineación
- Una unión es similar pero asigna espacio para el miembro más grande ya que todos se alinean al principio. Es decir, en la unión solo uno de los miembros puede estar almacenado en un momento dado.
- Salvo por el modo de almacenaje, en todos los demás aspectos se comporta igual que una estructura

Uniones

```c
enum  modos_venta
{XUNIDAD, XPESO, OTROS};

struct pedido {
    int tipo_pedido;
    union {
        int unidades;
        double peso;
        char descrip[20];
    } dato;
} vped;
```

```c
/* Supongamos que ya leímos y
cargamos vped.tipo_pedido */

switch (vped.tipo_pedido) {
case XUNIDAD:
    scanf("%d"
          , &vped.dato.unidades);
    break;
case XPESO:
    scanf("%lf"
          , &vped.dato.peso);
    break;
case OTROS:
    fgets(vped.dato.descrip, 20
          , stdin);
}
```

Licencia