Seminario de lenguajes
Opción C

Práctica 3 - 2016

1. Escriba un programa que implemente la función:

```
void convertir(int, char[ ] s, int b);
```

   (a) La función debe convertir el entero **i**, en el string **s** en su representación en base **b**.

   (b) La base debe ser mayor que 1 y menor que 37 dado que iría de base-2 hasta base-36, que usaría los dígitos: 0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ

   (c) Implemente la función considerando que **i** puede ser negativo, y se empleará el mismo símbolo (**-**) para su representación en base-**b**.

2. ¿Qué imprime el siguiente fragmento de código?

```
char *arr1 = "Hola mundo";
char arr2[20];

printf("%d ", sizeof(arr1));
printf("%d ", sizeof(arr2));
```

3. Describa las diferencias esenciales entre un arreglo multidimensional y un arreglo de punteros. Por ejemplo:

```
char arr1[ ][15] = {"uno", "dos", "tres"};
char arr2[5][15];
char *arr3[ ] = {"uno", "dos", "tres"}
char *arr4[4];
```

   (a) Para cada caso mencionado muestre cómo se representarían físicamente en memoria. Puede emplear un gráfico de la representación.

   (b) Indique el tamaño de las variables **arr1**, **arr2**, **arr3**, **arr4**.

4. Realizar un programa que dado un arreglo de enteros lo muestre ordenado de menor a mayor. *No es necesario leer el arreglo desde el teclado, use arreglos de prueba.*

5. Realizar una función que reciba dos arreglos de enteros y almacene en un tercer arreglo la suma de sus elementos.

6. Realizar una función que dado un string, lo imprima en orden inverso. Haga dos implementaciones de la función: iterativa y recursiva.

7. Realizar una función que dado un string, devuelva la cantidad de palabras que contiene.

8. Analice lo que hacen las funciones de la librería *string.h*:

```
char *strcat(char *dest, const char *src);
int strcmp(const char *s1, const char *s2);
```

```c
char *strcpy(char *dest, const char *src);
size_t strlen(const char *s);
char *strstr(const char *haystack, const char *needle);
```

9. Implemente las funciones del punto anterior. En cada caso puede usar otra de las funciones a implementar. Los prototipos deben ser iguales, pero los nombres de las funciones a implementar deber comenzar con **my_**, por ejemplo **my_strlen**.

10. Implemente las siguientes funciones:

```c
/*
 * Retorna si la palabra recibida es capicua.
 */
int es_palindromo(const char *word)

/*
 * Encripta el string src en la variable dest y, retorna el mismo
 * valor que dest. Para encriptar debe aplicar el complemento a 1 de
 * cada char.
 */
char* encripto(char *dest, const char *src)
```

11. Escriba un programa que cuente la cantidad de palabras capicúa. Debe emplear la función **es_palindromo** del ejercicio anterior. *Para probar el ejercicio, inicialice una variable con una cadena de caracteres que contenga palabras.*

12. Escriba un programa que analice un texto indicando el número de aquellas líneas que tienen palabras capicúas e indique la cantidad de palabras capicúas en la línea. Debe emplear la función **es_palindromo** del ejercicio 10. *Para probar el ejercicio, inicialice una variable con una cadena de caracteres que contenga palabras. El fin de línea es \n.*

13. Escriba un programa que reciba parámetros desde su ejecución e imprima todos los parámetros recibidos en pantalla.

14. Depure el siguiente código para descubrir por qué falla:

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

void capitalizar(char *cadena){
      printf("Forma original: '%s'\n", cadena);
      // Pasamos el primer caracter a mayúsculas
      cadena[0] = toupper(cadena[0]);
      printf("Capitalizado: '%s'\n", cadena);
      puts("--");
}

int main(int argc, char **argv){
      char cadena1[] = "foo";
      char *cadena2 = "bar";

      capitalizar(cadena1);
      capitalizar(cadena2);

      return 0;
}
```

Ejercicio adicional

Investigue usando GDB el motivo por el cuál el siguiente programa deja acceder a cualquier usuario que ponga un password largo (probar con 7 o 13 chars) pero funciona correctamente para passwords más cortos:

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

int login(char *user, char *passwd){
   int entrar = 0;
   char p[6];
   char u[6];
   printf("Usuario: "); gets(u);
   printf("Password: "); gets(p);
   if (strcmp(user, u) == 0 && strcmp(passwd, p) == 0){
      entrar = 1;
   }
   return entrar;
}

int main(){
   char user[] = "admin";
   char passwd[] = "12345";
   if (login(user, passwd)){
      puts("Pudo ingresar con el usuario admin");
   }
   else{
      puts("Nombre de usuario o password incorrecto");
   }
   return 0;
}
```

Ejemplo de uso:

```
$ ./login
Usuario: a
Password: aaaaaaaaaaaa
Nombre de usuario o password incorrecto
$ ./login
Usuario: a
Password: aaaaaaaaaaaaa
Pudo ingresar con el usuario admin
```

¿Cómo solucionaría este bug? ¿Por qué no falla con 6 o 12 chars? ¿Por qué el programa lanza "Violación de segmento" para passwords demasiado largos?

Si está compilando en un sistema de 64 bits también pruebe compilar con la opción -m32 y observe cuantos chars son necesarios en el password para acceder.

Ayudas, usando GDB:

- Establecer un breakpoint en la línea 4 (`break 4`)
- Ejecutar el programa (`run`)
- Ver las direcciones de memoria de las variables:

```
printf "%p\n", &entrar
printf "%p\n", p
```