


SEMINARIO DE LENGUAJES

Opción C

Práctica 4 - 2022

1. Escriba un programa que invoque funciones definidas en un archivo `.c` diferente de donde se encuentra la función **main**. Supongamos: 

main.c

```
int main()
{
    /* ... */
    suma(x, y);
    /* ... */
}
```

matem.c

```
/* ... */
int suma(int a, int b)
{
    /* ... */
}
/* ... */
```

- (a) El programa deberá compilarse con **gcc -Wall** y no dar ningún **WARNING**.
- (b) ¿Qué está faltando?
- (c) ¿Es correcto que **main.c** agregue en su encabezado el siguiente código?

```
#include "matem.c"
```

- (d) ¿Cuál sería la solución?

2. A partir del *ejercicio 12 de la práctica 2*, analice el potencial de la directiva al preprocesador **#ifndef** para evitar inclusiones de headers recursivos en nuestro código. Por ejemplo, evite el error que se da al crear los cuatro archivos como se muestran a continuación y compilar **main.c** empleando:

```
gcc main.c -o salida
```

main.c

```
#include "uno.h"
int main()
{
    return 0;
}
```

uno.h


```
#include "dos.h"
```

dos.h


```
#include "uno.h"
#include "tres.h"
```

tres.h



```
#include "uno.h"
#include "dos.h"
```


3. Escriba una función que reciba una variable de tipo entero por referencia y la inicialice en cero.
4. Escriba una función que reciba una variable de tipo puntero a entero por referencia y aloque la memoria necesaria para alojar un entero.
5. ¿Qué alternativa sintáctica existe para la siguiente expresión? Se debe asumir que **var** es de tipo **struct** y que define un campo llamado **campo**. Indique además cuándo es posible usar esta notación: 

```
(*var).campo
```

6. Defina una estructura que represente una fecha.
 - (a) Es posible hacer la asignación de dos variables del tipo definido por esta estructura. ¿Es correcto hacerlo? 
 - (b) Implemente una función que reciba una variable del tipo creado por referencia e inicialice los valores en el 1 de enero de 1970.
7. Defina la siguiente estructura:

```
struct {
    int a;
    char b;
}
```

- (a) Analice por qué el tamaño de la estructura no coincide con la suma de los tamaños de los campos. 
 - (b) ¿De qué forma puede definir una variable del tipo definido por la estructura? 
 - (c) Haga los cambios necesarios para poder escribir un programa que imprima un **sizeof** de cada campo y un **sizeof** de la estructura completa.
8. Defina una estructura que represente un alumno con los datos: nombre, apellido, fecha de nacimiento, legajo, tipo de documento, número de documento.
 - (a) Utilice en primer instancia arreglos de tamaño fijo para todos los strings.
 1. Inicialice una variable de tipo **alumno** con datos coherentes.

2. Asigne el valor de la variable inicializada en otra variable.
 3. Imprima ambas variables con todos sus campos.
 4. Cambie el campo nombre de la variable original. Use la función **strcpy** para setear un nuevo nombre.
 5. Imprima ambas variables con todos sus campos.
 - (b) Repita los pasos del punto anterior usando punteros a **char** en vez de arreglos de tamaño fijo para todos los strings. Considere que deberá alocar memoria para los campos de tipo puntero.
 - (c) Escriba sus conclusiones. 
9. Defina una variable de tipo **union** de la siguiente forma:

```
union T_union{
    int ival;
    float fval;
    char *sval;
}
```

- (a) Escriba un programa que lea un valor entero y si coincide con:
 - La macro **T_INT**: entonces se lee el valor de **ival**.
 - La macro **T_FLOAT**: entonces se lee el valor **fval**.
 - La macro **T_STR**: se lee una serie de caracteres que se almacenan en **sval** (previa alocaión de la memoria necesaria)
 - (b) Luego, imprima el valor de la variable de tipo **union** dependiendo del tipo leído.
10. Implemente una “librería” para manejo de alumnos que contenga lo siguiente:
- (a) Encapsule la definición del tipo **T_alumno** (use **typedef** para definir el tipo).
 - (b) Provea funcionalidad para:
 1. Inicializar una variable de tipo alumno.
 2. Convertir el alumno a un string (útil para usar dentro de un **printf**). Analice cómo solucionará el problema de una función que retorna un string (si lo aloca la función, ¿quién lo desaloca? ¿conviene usar una constante?).
 3. Comparar dos alumnos por:
 - nombre
 - apellido
 - fecha de nacimiento
 4. Destruya la variable en caso de no usarse más (puede que no haga nada dependiendo del tipo que definió para **T_alumno**)
 - (c) Escriba un programa (en un fuente diferente la librería) que testee la funcionalidad. *Debe usar archivos de header.*
11. Implemente una lista de datos genérica **T_lista_generica**, es decir, que en vez de que cada elemento sean de un tipo específico (como **int**, **char ***, **T_alumno**, etc), se utilice un puntero a **void**. Las operaciones a implementar sobre la lista serían:

Inicializar la lista: crea una nueva lista vacía

Destruir la lista: destruye la estructura de la lista, no los elementos en la lista. *Liberar los elementos de la lista es tarea de quien use la librería*

Está vacía la lista: indica si tiene elementos o no

Agregar a la lista: agrega un nuevo elemento a la lista

Eliminar de la lista: recibe un elemento perteneciente a la lista, lo encuentra y elimina de ella. Luego lo retorna. *No libera al elemento eliminado, eso es tarea de quien usa la librería.*


Existe elemento en la lista: retorna verdadero si existe el elemento, falso en caso contrario.

(a) Analice cuál es el problema de usar la lista genérica en el siguiente ejemplo de código:

```
void mal_uso(t_lista_generica *lista)
{
    int i, arr_int[10] = {10, 21, 31, 42, 52, 62, -10, 2, 5, 6};
    listag_inicializar(lista);
    for(i = 0; i < 10; i++){
        listag_agregar(lista, &arr_int[i]);
    }
}
```

(b) ¿Cuál sería el uso correcto?

(c) Analice el buscar y eliminar de la lista genérica. ¿Encuentra casos donde no funcionaría de la forma esperada?

12. Indique en cada caso qué es lo que hace la palabra clave **static**: 

Caso 1


```
#include <...>
/*
 * FUERA DE CUALQUIER
 * FUNCION
 */
static int mi_variable;
...
```

Caso 2

```
#include <...>
static int sumar(int a, int b)
{
    return a + b;
}
...
```

Caso 3

```
#include <...>
int hacer_algo()
{
    static int a = 10;
    ....
}
```

13. Dado los siguientes códigos: 

main-1.c

```
int main()
{
    return suma(1,2);
}
```

main-2.c

```
int main()
{
    return prod(2, 3);
}
```

suma.c

```
int suma(int a, int b)
{
    return a + b;
}


static int prod(int a, int b)
{
    return a * b;
}
```

(a) Indique por qué funciona:

```
gcc -o funciona.exe -Wall main-1.c suma.c
```

(b) Y por qué no funciona:

```
gcc -o no-funciona.exe -Wall main-2.c suma.c
```

14. Analice el siguiente código y saque sus propias conclusiones. 

```
#include <stdio.h>

int var_1 = 3;

void alter_var_1_A()
{
    int var_1;
    var_1++;
}

void alter_var_1_B()
{
    extern int var_1;
    var_1++;
}
```

```
void alter_var_1_C()
{
    var_1++;
}

int main()
{
    printf("El valor inicial de var_1 es: %d\n", var_1);

    alter_var_1_A();
    printf("El valor luego de alter_var_1_A es: %d\n", var_1);

    alter_var_1_B();
    printf("El valor luego de alter_var_1_B es: %d\n", var_1);

    alter_var_1_C();
    printf("El valor luego de alter_var_1_C es: %d\n", var_1);

    return 0;
}
```

15. ¿Cómo funcionan las variables **extern** en diferentes archivos *.c*? Ejemplifique con un programa compuesto por varios fuentes. 