***TPN° 2: Expresiones en C. Entradas y Salidas***

| **Los ejercicios que no están incluídos en este documento, es porque la solución es únicamente un código, que pueden encontrar en el Drive junto a este documento.** |
| --- |

***Ejercicio 1***

1. Son equivalentes. La constante ‘x’ (o cualquier otro valor ASCII) es de tipo entero, por lo que el casteo (int) es innecesario.
2. No son equivalentes. En la operación “x = y / 3.0” se utiliza aritmética de punto flotante pues el divisor es un operando de punto flotante (tipo ***double***). En la operación “(float) (y/3)” la división se realiza con aritmética de enteros y luego el resultado es promovido a ***float.***
3. No son equivalentes, pues en ambos casos se utiliza aritmética de punto flotante, pero en el segundo caso x es entero y toma sólo la parte entera de la operación (trunca).
4. El resultado es equivalente, pues x termina con el valor 1. Pero las operaciones no son equivalentes, pues en el primer caso se obtiene 1.8333333 (5.5/3) y en el segundo se obtiene 1.666666 (5/3.0) y ambos son truncados a entero.
5. Son equivalentes en cuanto a su valor numérico pero no en cuanto a su tipo.
6. Son equivalentes. En el primer caso efectúa una promoción a entero y en el segundo efectúa una democión.
7. No son equivalentes, pues dependen de la plataforma. En UNIX / LINUX el entero ocupa 4 bytes, en DOS el tipo entero ocupa dos bytes. Por lo tanto en DOS en el primer caso tenemos 1 bit para el signo y los 15 restantes para representar el número.
8. Son equivalentes porque el 120 es representable en un char independientemente de que éste sea signado o no. Recordar que el lenguaje C no establece si el char es signado o no, por lo cual se lo debe declarar explícitamente.
9. Si el char es unsigned ( lo que ocurre en algunas arquitecturas ), son equivalentes. Si el char es signed no son equivalentes.
10. No son equivalentes porque el 300 no es representable en un char independientemente de que éste sea signado o no.
11. No son equivalentes. Es muy importante destacar que en el primer caso **no se puede determinar cuál de los dos términos se evalúa primero**. Si primero evalúa (x = 2) y luego (y = x) en ambos códigos las variables terminan con el mismo valor. Pero si primero se evalúa (y = x) y luego (x = 2), en el primer caso tanto x como y toman un valor incierto. El orden de evaluación de los términos depende del compilador.

***Ejercicio 2***

Ver tp2ej2.c

***Ejercicio 3***

En algunas plataformas compila correctamente, pero en otras genera un error de compilación. En caso de querer continuar una cadena de caracteres en otra línea debe hacerse como en el ejemplo siguiente:

printf ( "Hola que"

"tal \n");

En este caso, en la salida ambas cadenas aparecen en la misma línea.

***Ejercicio 4***

* printf(“\a\n”);

Imprime el caracter BELL (suena un "beep")

* printf(“%c\n”,7);

Idem al caso anterior, ya que el valor numérico del caracter BELL es 7

* printf(“%d\n”,7);

Imprime el entero 7

* printf(“%f\n”, 7);

La salida es ***impredecible***

El número 7 se encuentra almacenado en el stack, en una dirección de memoria en el formato de complemento a la base y ocupando el tamaño de un entero, la función ***printf*** tomará del stack la cantidad de bytes necesarios para completar un double (ya que la cadena de formato le indica que el dato es de ese tipo) interpretándolo como un valor expresado en notación IEEE de punto flotante de 64 bits. Como el tipo ***int*** ocupa menos bytes que el tipo ***double*** (en Linux, el ***int*** ocupa 4 bytes frente a los 8 del ***double***), en este caso la función levantará del stack los cuatro bytes que ocupa el valor 7 más los cuatros bytes que siguen (con contenido desconocido) para representar un ***double***. Otra posibilidad es que el pasaje de parámetros a la función no sea por stack sino por registros, en este caso ***printf*** tomará del registro que soporte un ***double*** un valor.

* printf(“%g\n”, 7);

La salida es ***impredecible,*** por las mismas razones que en el ejemplo anterior.

***Ejercicio 5***

#include <stdio.h>

int

main(void)

{

char c = 't';

printf("c = %c\n", c);

printf("c = %d\n", c);

printf("c = %f\n", c);

return 0;

}

La función printf recibe en el stack el valor entero 116 (valor ASCII de la letra 't'). Tener en cuenta que la asignación ***c = 't'*** es equivalente a la asignación ***c = 116***.

Con %c se imprime el caracter que se recibió: ‘t’

Con %d se imprime el ASCII correspondiente al caracter enviado, o sea, 116.

Con %f la salida es ***impredecible****.*

***Ejercicio 6***

Se obtienen las siguientes salidas:

**num1= 53**

**num2= 0000000004**

**num1= 53**

**num1+num2= 57**

**num1+num2= ....** *(salida impredecible!!!)*

**num3= 6.87**

**num1= 53**

**num2= 4**

**num3= 6.9**

**num3(como entero)=...** *(salida impredecible!!!)*

**num1 / num2 = 13**

**num2 / num1 = 0**

**esta es la ultima prueba**

***Ejercicio 8***

#include <stdio.h>

#include "getnum.h"

int

main(void)

{

float velocidad ;

velocidad = getfloat("\nIngrese una velocidad en m/s: ");

printf ("\nEn km/h = %f\n", (velocidad \* 3600) / 1000 );

return 0;

}

***Ejercicio 11***

#include <stdio.h>

int

main(void)

{

char letra1,letra2 ;

printf("\nIngrese dos caracteres (uno al lado del otro): ");

letra1 = getchar();

letra2 = getchar();

printf ("\nEl caracter de mayor valor ASCII es %c\n",

(letra1>letra2) ? letra1 : letra2);

return 0;

}

Se deberá redireccionar con: **a.out < entrada.txt**, y las salidas serán:

1. B
2. A
3. A

***Ejercicio 14***

#include <stdio.h>

#include "getnum.h"

int

main(void)

{

int n1,n2 ;

n1 = getint("\nIngrese el numero1:");

n2 = getint("\nIngrese el numero2:");

printf("\nEl promedio es %.1f\n",(n1+n2)/2.0);

printf("La suma es %d\n",n1+n2);

printf("El menor es %d\n",(n1<n2)? n1 : n2);

printf("El mayor es %d\n",(n1<n2)? n2 : n1);

printf("%sson iguales\n",(n1==n2)? "" : "no ");

return 0;

}

***Ejercicio 17***

"3" + "4"; no compila (no se pueden sumar cadenas de caracteres)

'3' + '4'; compila (se obtiene 51+52, es decir 103)

3 + 4; compila (se obtiene 7)

'3' + 4; compila (se obtiene 51+4, es decir 55)

3 + '4'; compila (se obtiene 3+52, es decir 55)

"3" + '4'; la operación es válida, pero no es de utilidad, veremos en detalle qué significa

cuando veamos punteros.

"3" + 4; la operación es válida, pero no es de utilidad, igual que el anterior.

***Ejercicio 18***

a) FALSO. Comienza a escribir en donde se encuentra el cursor.

b) VERDADERO

c) VERDADERO. Las declaraciones de variables pueden hacerse en el medio del código, sólo deben hacerse **antes** de ser referenciadas.

d) FALSO. Se las puede imprimir en un solo ***printf***, usando la secuencia de escape **'\n'** (ver el código del ***Ejercicio 2***)

***Ejercicio 19***

La segunda opción cuenta con un mejor estilo que la primera, ya que invoca a la función **printf** una sola vez.

**La** **tercera opción es** **incorrecta** ya que, como se vio en el Ejercicio 1 k), **no se puede determinar cuál de las dos invocaciones a *getchar* se evalúa primero**.Como depende del compilador, esta opciónno es una alternativa equivalente a las dos anteriores.

***Ejercicio 20***

**Ej 1.2**: Si el valor de ***c*** hace que ***\c*** sea una secuencia de escape, se imprime el carácter correspondiente a la misma (ver ejercicio 7), sinó se imprime el mismo carácter ***c*** enviado, como un carácter común.

**Ej: 1.6**: La expresión ***c = getchar() != EOF***  primero lee un carácter desde la entrada, luego lo compara con ***EOF*** y según sean o no iguales asigna a la variable ***c*** un 1 o un 0 (verdadero o falso).

**Ej 1.7:** Para imprimir el valor EOF usamos: ***printf(“El valor de EOF es %d”, EOF);***

***Ejercicio 21***

1. INT\_MAX representa el máximo valor entero con signo. Como el límite de los enteros es de 4 bytes el valor es 2147483647 ( 32 bits de los cuales el más significativo está en cero y todos los demás en uno). Al sumarle 1 se convierte en un número de 32 bits con el bit de signo en uno seguido de 31 bits en cero, por lo que representa el menor entero posible: -2147483648, que equivale a la constante INT\_MIN. Al multiplicar por dos quedará un número de 32 bits formado por 31 bits en uno y un bit en cero, que equivale al -2 (recordar que los enteros se representan como binarios complemento a la base.
2. Explicado en el punto anterior
3. En este caso, el flag previene sobre errores de operaciones aritméticas con overflow en enteros con signo por lo que saldrán los siguientes mensajes de error

tp2\_21.c:8:6: runtime error: signed integer overflow: 2147483647 \* 2 cannot be represented in type 'int'

tp2\_21.c:9:8: runtime error: signed integer overflow: 2147483647 + 1 cannot be represented in type 'int'

indicando que en esa línea se produjo overflow, aunque la ejecución del programa sigue su curso. Esos mensajes son enviados a la salida de errores, no a la salida estándar.

***Ejercicio 22***

El problema es que al definir la variable c como char, en algunas computadoras se lo puede tomar como signed char y en otras como unsigned char. Si se lo toma como un unsigned char, c siempre será positivo y nunca será igual a EOF.