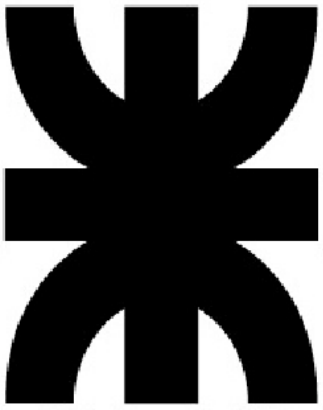


Informática I

Guía de Trabajos Prácticos 2018

Ing. Alejandro Furfaro, Profesor Titular

May 30, 2018

Part I
Programación Básica

1 Condiciones generales

A partir de a sección 3. Funciones, en Página 5, cada ejercicio debe incluir los siguientes archivos fuente:

1. **`myincludes.h`**: En él se declararán los prototipos y las macros, como también se incluirán los restantes archivos de cabecera que requiera el programa. Vale decir, que será el único archivo que necesitemos incluir en el resto de los programas fuente.

2. **`main.c`** Aquí estará la función **`main`** y desde allí se invocará a la función/funciones requerida/s en el problema. Esta/s estará/n en un archivo (**`funciones.c`**). Cada función debe incluir obligatoriamente un comentario a modo de encabezado, con el siguiente formato:

```
/**
* \file main.c
* \brief Programa principal para prueba de la función xxxxxxxx.
* \details <Aquí se explica que hace>.
* \return 0 si terminó OK
* Para compilar <Aquí se incluyen las instrucciones de
* compilación de main.c>
* Para linkear <Aquí se incluyen las instrucciones para linkear main.o y
* funciones.o>
* \author <Nombre y apellido del autor del programa y un e-mail>
* \date <Fecha de terminación de la versión funcional del programa en
* formato aa.mm.dd (año.mes.día con dos dígitos de ancho cada uno)>
*/
```

3. **`funciones.c`**: En este archivo estará escrito el cuerpo de código de la función/funciones solicitada/s en cada ejercicio. Cada función deberá contener un comentario a modo de encabezado que permita conocer que hace, como se la invoca (su prototipo) y como se la compila. El formato es el siguiente:

```
 /**
* \fn <prototipo de la función>
* \brief <en pocas palabras que hace la función. Ej: Calcula
* el factorial de un entero>.
* \details <Aquí se explica que hace con mayor detalle>.
* \arg <nombre>. Una línea de estas por cada argumento que recibe
* la función y su explicación
* \return <Explicar en este punto cuales son los posibles valores que
* retorna la función y que se hace en cada caso.>
* Para compilar <Aquí se incluyen las inst rucciones de
* compilación del archivo funciones.c>
* \author <Nombre y apellido del autor del programa y un e-mail>
* \date <Fecha de terminación de la versión funcional del programa en
* formato aa.mm.dd (año.mes.día con dos dígitos de ancho cada uno)>
*/
```

A medida que los programas se tornen mas complejos se podrán agregar mas archivos fuente a cada programa. Las funciones se distribuirán en diferentes fuentes de acuerdo con criterios de afinidad. Ej: Un archivo fuente **`sort.c`** por ejemplo tendría las diferentes funciones de ordenamiento.

2 Control de Flujo

2.1 Uso de sentencias de selección

2.1.1 Ejercicio 1

Escribir un programa que acepte por teclado el ingreso de un número entero correspondiente al radio de un círculo. Con ese número el programa calculará la superficie del círculo presentando el resultado en pantalla.

2.1.2 Ejercicio 2

Mejorar el programa anterior para que verifique que el número ingresado es positivo ya que de otro modo carece de sentido, e informe al usuario el resultado o en caso de no ser positivo el radio el mensaje de error correspondiente.

2.1.3 Ejercicio 3

Mejorar el programa anterior para que el mensaje de error salga por la salida de error estándar y el resultado por la salida estándar. **Pista**: `man perror`

2.2 Uso de sentencias de selección múltiple

2.2.1 Ejercicio 1

Escribir un programa que acepte por teclado el ingreso de tres números reales correspondientes a las dimensiones de cada lado de un triangulo. El programa deberá presentar en la pantalla si las dimensiones ingresadas corresponden a un triángulo equilátero, isósceles o escaleno, de acuerdo con los tres valores recibidos.

2.2.2 Ejercicio 2

Mejorar el programa anterior para que verifique que los tres números ingresados son positivos ya que de otro modo carece de sentido, e informe al usuario el resultado o el mensaje de error correspondiente, en caso de no ser positivo al menos un número de los ingresados.

2.2.3 Ejercicio 3

Mejorar el programa anterior para que el mensaje de error salga por la salida de error estándar y el resultado por la salida estándar.

2.2.4 Ejercicio 4

Modificar el programa anterior para comprobar que los números ingresados conformen realmente un triángulo. En caso de no conformarlo no procederá a la clasificación, presentando en su lugar un mensaje de error y terminando la ejecución.
Pista: Usar función ***`abs`*** definida en la standard library (cabecera ***`stdlib.h`***). Usar ***`man abs`*** para acceder a la documentación de ***`abs`***.

2.2.5 Ejercicio 5

Modificar el programa anterior para que en el caso de ser un triángulo, luego de indicar que tipo de triángulo es, informe a través de la salida estándar si además el triángulo es rectángulo.

2.2.6 Ejercicio 6

Realizar un programa que solicite el ingreso de un número entero del 1 al 7 y en función del número ingresado, informe el día de la semana siendo 1 el Domingo, 2 el Lunes y así sucesivamente. Si el número es mayor a 7 deberá enviar un mensaje de error.

2.2.7 Ejercicio 7

Modificar el programa anterior para que en lugar de aceptar números acepte las letras L, M, X, J, V, S y D (en mayúscula) indicando, Lunes, Martes, Miércoles, etc. respectivamente.

2.2.8 Ejercicio 8

Modificar el programa anterior para que acepte las letras en mayúscula y minúscula.

2.3 Sentencias de Iteración - Uso de for o while

2.3.1 Ejercicio 1

Escribir un programa que realice la sumatoria de los números consecutivos entre dos valores enteros ingresados por teclado (ambos extremos se incluyen en la sumatoria. Se debe aceptar números enteros signados

2.3.2 Ejercicio 2

Mejorar el programa anterior verificando que se el primer valor sea menor al segundo de modo de no incurrir en resultados incorrectos.

2.3.3 Ejercicio 3

Mejorar el programa anterior para que el mensaje de error salga por la salida de error estándar y el resultado por la salida estándar.

2.3.4 Ejercicio 4

Repetir el programa utilizando la otra sentencia de iteración (for si utilizaste while o visceversa)

2.4 Sentencias de Iteración - Uso de do-while

2.4.1 Ejercicio 1

Ingresar pares de valores no nulos X e Y, que representan las coordenadas rectangulares de distintos puntos en el plano, y determinar e informar la cantidad de puntos que pertenecen a cada cuadrante. El fin de datos se indica con X e Y iguales a cero.

2.4.2 Ejercicio 2

Mejorar el programa anterior para que además de indicar la cantidad de puntos en cada cuadrante, indique la cantidad de puntos en el eje de la ordenadas y cuantos en el eje de las abscisas.

2.4.3 Ejercicio 3

Realizar un programa que solicite un número entero y calcule su factorial. En el caso de ingresar 0 mostrar su factorial y salir del programa.

2.4.4 Ejercicio 4

Mejorar el ejercicio anterior validando que la entrada que no sea negativa. En ese caso deberá presentar un mensaje de error y solicitar un nuevo valor.

2.4.5 Ejercicio 5

Desarrollar un programa que calcule la superficie de un depósito de mercaderías, la superficie del terreno y el porcentaje de terreno construido.
Para ello solicitará cuatro valores correspondientes al frente y fondo del terreno y del depósito respectivamente.
Si alguno de los valores ingresados es menor que cero, deberá presentar un mensaje de error y solicitar un nuevo valor. Si los 4 valores ingresados son 0, sale del programa.

2.5 Control de flujo - Ejercicios integradores

2.5.1 Ejercicio 1

Fuiste seleccionado para desarrollar el sistema de premios para el programa de televisión “Yo se cuanto cuesta”. En el mismo, se muestra un carrito de supermercado lleno de productos y los participantes deben arriesgar el valor del total de los productos que se encuentran en él. El que acierta o está mas cerca de ese valor, gana.
El sistema deberá permitir el ingreso del valor del carrito, verificar que el valor ingresado sea mayor que 0, y solicitar nuevamente el ingreso en caso de no cumplirse esta condición. Si se superan los tres ingresos fallidos consecutivos, terminará con el mensaje correspondiente al usuario por la salida de error standard.
Luego solicitará el ingreso del importe declarado por cada participante (cada participante será identificado con el número de orden de carga en el sistema, o sea, 1, 2, 3, etc).
La carga de los participantes finaliza con el ingreso de un importe igual o menor que 0.
Una vez finalizada la carga, deberá mostrar por la salida estándar cuantos participantes arriesgaron valores mayores al 15% del valor del carrito, cuántos participantes arriesgaron valores menores al 1% del valor del carrito y quien fue el participante que estuvo más cerca del valor.

2.5.2 Ejercicio 2

Dada una carga inicial de 100 valores enteros, se deberá presentar un menú de 4 opciones donde, en caso de seleccionar:

- Opción 1: Mostrará cuántos de esos valores son mayores a 50, son menores de 1000 y poseen raíz cuadrada entera.
- Opción 2: Mostrará cuántos de esos valores son menores que 0 y pares.

- Opción 3: Mostrará cuántos de esos valores son múltiplos de 8 o de 9.
- Opción 4: Sale del programa.

Si no se seleccionó ninguna de las 4 opciones presentar por la salida de error standard "Opción inválida", solicitará que se presione una tecla, borrar la pantalla[1] y presentar nuevamente el menú.

2.5.3 Ejercicio 3

Una compañía de seguros está realizando un relevamiento de las pólizas que tiene emitidas y nos piden realizar un programa que brinde las siguientes estadísticas:

- Cantidad de dominios que posean más de 10 años de antigüedad o más de 100.000 kmts.
- Cantidad de dominios que posean una póliza con más de 5 años de fidelidad a la compañía y menos de 3 siniestros.
- Cuál es el mayor kilometraje que posee un dominio.
- Cuál es el promedio de kilómetros por año del total de dominios.

Para esto, se realizará la carga de 100 pólizas. Por cada póliza se deberán cargar los siguientes 4 datos:

- Año de la primer póliza para ese dominio.
- Año del dominio.
- Cantidad de kilómetros
- Cantidad de siniestros.

El dominio es la patente del automóvil.

3 Funciones

3.1 Ejercicios sencillos

3.1.1 Ejercicio 1

Re escribir el cálculo de la superficie del círculo utilizando para este cálculo una función llamada circle () la cual devuelve en un float el resultado.

3.1.2 Ejercicio 2

Escribir una función que reciba un valor de temperatura en precisión simple, la escala de dicho valor y la escala de temperaturas de destino, y realice la conversión del valor.
Se valorará especialmente no utilizar variables en el cálculo, es decir, trabajar solo con los argumentos recibidos.

3.1.3 Ejercicio 3

Escribir una función que reciba un número entero positivo y calcule su factorial, retornándolo en un tipo long.

[1] usar éste código para borra pantalla: `printf ("\x1B[2J;\x1B[1;1H");`

3.1.4 Ejercicio 4

Escribir una función que reciba como argumento un entero positivo, que representa la cantidad de términos de la serie de Fibonacci que el usuario desea ver presentados en pantalla. La función calcula cada término de la serie y lo presenta separados por un caracter de espacio. En el **`main`** se validará que el número de términos solicitado sea entero positivo y luego se invoca a la función que con ese número ingresado como argumento calcula y presenta la serie hasta el término solicitado.
Pista: Recordar que el 1er. término es 0 y el 2do. es 1. Luego cada termino se calcula como suma de los dos anteriores.

3.2 Uso de funciones de bibliotecas estándar

3.2.1 Ejercicio 1

Utilizando las funciones contenidas en la math library (definidas en **`math.h`**), solicitar un número x por teclado, y calcular presentando los resultados a razón de un mensaje por línea por cada función, de las siguientes: **`sqrt (x)`**, **`exp (x)`**, **`log (x)`**, **`log10 (x)`**, **`fabs (x)`**, **`ceil (x)`**, **`floor (x)`**, **`sin (x)`**, **`cos (x)`**, **`tan (x)`**.
Una vez presentados estos resultados solicite un segundo número y calcule las siguientes operaciones **`pow (x,y)`**, y **`fmod (x,y)`**

3.2.2 Ejercicio 2

Utilizando la función **`rand ()`** definida en **`stdlib.h`**, simular 50 tiros de un dado. El programa debe presentar por stdout los 50 valores (entre 1 y 6) equiespaciados comenzando un valor cada 4 columnas de código en pantalla, y a razón de 25 valores por línea (es decir que cada 25 valores generados y presentados debe enviar a stdout un caracter fin de línea.
Ejecutar varias veces y observar los resultados. ¿Que concluye acerca de la aleatoriedad de la función utilizada?

3.2.3 Ejercicio 3

Explorar la función **`srand ()`** definida en **`stdlib.h`**. Incluirla en la generación de los mismos 50 valores del Ejercicio 2 anterior, y observar su efecto al ejecutar el programa repetidas veces. Ingresar la semilla por teclado. ¿Cual es su conclusión ahora?. ¿Que ocurre si se ingresa la misma semilla?

3.2.4 Ejercicio 4

Explorar la función **`time ()`** definida en **`time.h`**. Implementarla en la generación de la semilla para asegurar que nunca se pueda producir el mismo juego de números aleatorios. ¿Porque es este método el óptimo respecto del ingreso de la semilla por teclado?.

3.3 Pila y argumentos

3.3.1 Ejercicio 1

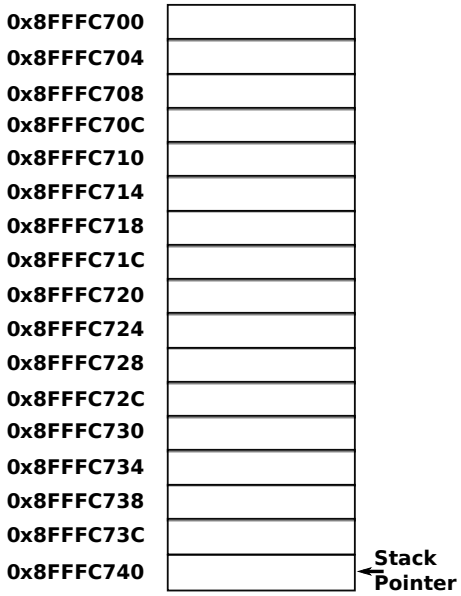
Para resolver el presente ejercicio es necesario subir a su repositorio git un archivo con el dibujo de la porción de memoria RAM correspondiente al stack, pero completo con los valores y nombres de variables correspondientes en cada dirección de memoria.

Observar que las direcciones están expresadas en hexadecimal, y asumir que la máquina en la que se está trabajando tiene un Linux de 32 bits instalado.

Considere el siguiente código:

```
int power (int, int);
int main()
{
    power (n ,exp);
    return 0;
}
int power (int num , int e)
{
 int result = num, i;
 for (i =1; i < e ; i ++)
    result*=num;
 return result;
}
```

Escriba en la Figura de la derecha como queda el contenido de la pila cuando inicia la sentencia **`for`** en la función **`power`**. Incluir todas las variables que vea en el programa: las locales de la función **`power`** y las de **`main`**.



4 Punteros - Pasaje de argumentos por referencia

4.1 Ejercicio 1

Tomar el Ejercicio 3.1.1 de la sección **3.1. Ejercicios sencillos** y reescribirlo pasando los argumentos por referencia

4.2 Ejercicio 2

Tomar el Ejercicio 3.1.2 de la sección **3.1. Ejercicios sencillos** y reescribirlo pasando los argumentos por referencia

4.3 Ejercicio 3

Tomar el Ejercicio 3.1.3 de la sección **3.1. Ejercicios sencillos** y reescribirlo pasando los argumentos por referencia

5 Arreglos

5.1 Ejercicio 1

Escribir un programa que defina un arreglo de 1024 enteros, le pase su referencia a una función que lo complete con números aleatorios, desde **`0`** a **`RAND_MAX`** (es decir el número aleatorio completo). La función recibirá dos argumentos: el puntero al inicio del arreglo y la cantidad de elementos del mismo. A la vuelta de

la función de inicialización del arreglo el programa presentará por pantalla los valores del arreglo a razón de 11 lugares de caracteres por cada valor. La función regresa la cantidad de elementos del arreglo efectivamente inicializados.

5.2 Ejercicio 2

Agregar al archivo **`funciones.c`** una función que reciba los mismos argumentos que la del ítem anterior, es decir, el puntero al inicio del arreglo, y la cantidad de elementos del mismo, calcule su promedio y lo regresa como valor de retorno.

5.3 Ejercicio 3

Agregar al archivo **`funciones.c`** una función que reciba los mismos argumentos que la del ítem anterior, es decir, el puntero al inicio del arreglo, y la cantidad de elementos del mismo, recorra el arreglo en busca del valor numérico menor y lo regrese en forma de entero como valor de retorno.

5.4 Ejercicio 4

Agregar al archivo **`funciones.c`** una función que reciba los mismos argumentos que la del ítem anterior, es decir, el puntero al inicio del arreglo, y la cantidad de elementos del mismo, recorra el arreglo en busca del valor numérico mayor y lo regrese en forma de entero como valor de retorno.

5.5 Ejercicio 5

Escribir una función denominada **`my_strlen`** que recibe como argumento un puntero a una cadena de caracteres ASCIIZ (ASCII Zero ended, es decir una secuencia de caracteres de texto finalizada en `'\0'`), y retorne un valor entero con la cantidad de caracteres contenidos en la cadena. El caracter terminador no se incluye en la cuenta.
Condición: No utilizar ninguna función auxiliar definida en el encabezamiento **`string.h`**.

5.6 Ejercicio 6

Sin utilizar ninguna función definida en **`string.h`**, implementar una función llamada **`my_strcat`** que recibe como argumentos dos punteros a cadenas de datos, y copia el segundo a continuación del primero. El caracter `'\0'` de la primer cadena desaparece siendo reemplazado por el primer caracter de la segunda cadena, y se conserva el finalizar `'\0'` de la segunda cadena como finalizador de la cadena resultado. La función retornará un valor entero equivalente a la cantidad de bytes de la cadena resultante.

6 cadenas de texto

6.1 Ejercicio 1

Escribir un programa que solicite al usuario una el ingreso de una cadena de texto desde el teclado. Utilizar la función **`gets ()`** para este fin. Para evitar sobredimensionar estáticamente la sección de datos del programa se utilizará como argumento de **`gets ()`** una variable de tipo puntero a **`char`**, al que se le asignará dinámicamente una cantidad de memoria suficiente para, a pesar del warning que de todos modos enviara el compilador, poder estar seguros que soportará la cantidad de caracteres ingresados por el usuario.
Una vez ingresada la cadena, el programa ajustará ("**`realloc`**ará") la cantidad de memoria pedida al sistema

operativo, de modo tal de utilizar para almacenar la cadena ingresada, la cantidad de memoria estrictamente necesaria.
Una vez ingresada la cadena presentará por ***stdout*** un mensaje solicitando al usuario que presiones una tecla para imprimir la cadena. El programa esperará la tecla y una vez pulsada cualquier tecla presentará por ***stdout*** la cadena ingresada.
Importante: Recordar que todo lo que le pedimos al Sistema Operativo hemos de devolverlo ...
Sugerencia: Invocar **`getchar ()`** para esperar a tecla. Utilizar caracteres nueva línea para que la cadena ingresada, el mensaje de pulsar la tecla, y la cadena presentada tengan una separación adecuada entre si constituyendo tres claros bloques en la pantalla.

6.2 Ejercicio 2

Escribir un programa que imprima en pantalla en una primer línea la cantidad de argumentos que se le proporcionaron por línea de comandos y a continuación, a razón de uno por línea, cada uno de los argumentos recibidos por línea de comando. La cantidad de argumentos que puede recibir no tiene límite.
Pista: Ensayar invocando al programa con una sucesión de palabras separadas por espacios, y luego volver a invocarlo encerrando todos los argumentos entre comillas dobles. Por ejemplo: **`./p hola mundo! Que tal?`** en primer lugar y luego **`./p "hola mundo!  Que tal?"`**. ¿Cuales son las diferencias? ¿Que conclusiones puede sacar de este experimento?

6.3 Ejercicio 3

Modificar el programa del ítem 6.1 para que, solicite al usuario el ingreso de la cadena de texto siempre que no haya recibido nada como argumento por línea de comando.
Sugerencia En base a las conclusiones del ejercicio del ítem 6.2, estime cual es la manera que resulta mas práctica a los fines de este ejercicio.

6.4 Ejercicio 4

En base al ejercicio 6.3 y empleando las funciones de definidas en **`string.h`**, trabaje con la cadena ingresada, realizando las siguientes operaciones:

- Agregar la cadena ingresada a la siguiente cadena fija: ***La cadena ingresada es:*** , de modo que al presentarla con printf, la cadena se imprima a partir del siguiente renglón del mensaje
- Insertar en la cadena ingresada la cantidad necesaria de caracteres nueva línea ('\n') para que la misma se imprima a razón de 25 caracteres por línea.

6.5 Ejercicio 5

Utilizando las secuencias de escape estándar de **ANSI**, es posible limpiar la pantalla de una consola, posicionar el cursor, escribir con colores diferentes de caracter y fondo y manejar atributos como subrayado, tachado, negrita, etc.
Esta secuencias comienzan con el ASCII de escape (***ESC***), que en la tabla de ASCII corresponde al valor 27 decimal, 33 octal o 0x1B hexadecimal. ***ESC*** es un código de control de modo que no será nunca un caracter capaz de ser impreso por ninguna terminal sino que corresponde a un código que generalmente dispara una secuencia de control.
Este ejercicio se enfocará en la ***Control Sequence Introducers*** (***CSI***). Esta secuencias comienza con ***ESC***

seguido del caracter ' *[*'.

Código	Nombre	Efecto
CSI n A	CUU (**Cu**rsor **U**p)	Mueve cursor n (default 1) lugares hacia arriba.
CSI n B	CUD (**Cu**rsor **D**own)	Mueve cursor n (default 1) lugares hacia abajo.
CSI n C	CUF (**Cu**rsor **F**orward)	Mueve cursor n (default 1) lugares hacia adelante.
CSI n D	CUB (**Cu**rsor **B**ack)	Mueve cursor n (default 1) lugares hacia atrás.
CSI n E	CNL (**C**ursor **N**ext) **L**ine)	Mueve cursor al inicio de la línea n hacia abajo (default 1).(No ANSI.SYS)
CSI n F	CPL (**C**ursor **P**revious) **L**ine	Mueve cursor al inicio de la línea n hacia arriba (default 1). (No ANSI.SYS)
CSI n G	CHA (**C**ursor **H**orizontal) **A**bsolute	Mueve el cursor a la columna n (default 1). (No ANSI.SYS)
CSI n;m H	CUP **Cu**rsor **P**osition)	Mueve cursor a fila n, col m. Los valores arrancan de 1, y el default es 1 (esquina superior izquierda)Si se omiten *n* y *m*. **CSI ;5H** equivale a **CSI 1;5H** así como CSI 17;H equivale a CSI 17H y a CSI 17;1H
CSI n J	ED (**E**rase in **D**isplay)	Limpia parte de la pantalla. S n es 0 (o no se escribe), limpia desde la posición del cursor hasta le final. Si n es 1, limpia desde el cursor hasta el inicio de la pantalla. Si n es 2, limia toda la pantalla (y mueve el cursor al borde superior izquierdo en DOS ANSI.SYS). Si n es 3, limpia la pantalla entera y borra todas las líneas salvadas en el scrollback buffer (función agregada para xterm y soportada por toda las demás aplicaciones).
CSI n K	EL (**E**rase in **L**ine)	Borra parte de la línea. Si n es 0 (o no se escribe), limpia desde el cursos hasta el final de la línea. Si n es 1, limpia desde el cursor al inicio de la línea. Si n es 2, limpia la línea completa. No cambia la posición del cursor.
CSI n S	SU **S**croll **U**p	Scroll de la pantalla completa n (default 1) líneas hacia arriba. Al fondo de la pantalla se completa con nuevas líneas. (No ANSI.SYS)
CSI n T	SD (**S**croll **D**own)	Scroll de la pantalla completa n (default 1) lineas hacia abajo. Al inicio de la pantalla se completa con nuevas líneas. (No ANSI.SYS)
CSI n;m f	HVP (**H**orizontal **V**ertical **P**osition)	Idéntico a CUP
CSI n m	SGR (**S**elect **G**raphic **R**endition)	Establece la apariencia de los caracteres que lo prosiguen, (ver en Tabla 2 los parámetros SGR).
CSI 6n	DSR (**D**evice **S**tatus **R**eport	Reporta posición del cursor (CPR) a la aplicación como (como si se escribiese en el teclado) **`ESC[n;mR`**, donde n es la file y m la columna.)
CSI s	SCP (**S**ave **C**ursor **P**osition)	Salva el estado/posición actual del cursor.
CSI u	RCP (**R**estore **C**ursor **P**osition	Recupera el estado/posición actual del cursor.

Table 1: Secuencias de escape

La Tabla 1 contiene los códigos de escape de interés y su efecto, y la Tabla 2 los valores enteros que corresponden al parámetro de la secuencia de escape ***SGR*** de la Tabla 1. Observar que en las secuencias definidas en la Tabla 1 tanto `n` como `m` son números de una o mas cifras pero expresados en ASCII.

Código	Nombre	Efecto
0	Reset / Normal	Apaga todos los atributos.
1	Negrita	Aumenta la intensidad de los caracteres.
2	Faint	Merma la intensidad del los caracteres (no está ampliamente difundido).
3	Italic	No es ampliamente soportada, puede resultar en Video Inverso.
4	Subrayado	
5	Slow Blink	Menos de 150 blinks por minuto.
6	Rapid Blink	No es ampliamente soportado. Mas de 150 blinks por minuto.
7	Video Inverso	Intercambia colores entre fuente y fondo
9	Tachado	No está ampliamente soportado.
30	Negro	Color de foreground.
31	Rojo	Color de foreground.
32	Verde	Color de foreground.
33	Amarillo	Color de foreground.
34	Azul	Color de foreground.
35	Magenta	Color de foreground.
36	Cyan	Color de foreground.
37	Blanco	Color de foreground.
40	Negro	Color de background.
41	Rojo	Color de background.
42	Verde	Color de background.
43	Amarillo	Color de background.
44	Azul	Color de background.
45	Magenta	Color de background.
46	Cyan	Color de background.
47	Gris	Color de background.

Table 2: Parámetros de la Secuencia de Escape SGR (Select Graphics Rendition

Nota Importante: Observar que tanto ***ESC*** como ' `[` ' son dos tipos char (1 byte).
En base a los contenidos de las 1, y 2, se pide:

1. Escribir un archivo frmconsole.h, en donde se tenga definidos los comandos de las secuencias de escape default (ESC+'['+ASCII), es decir los que respondan al comportamiento default cuando no se escriben los argumentos numéricos denotados n y m.

2. Escribir en el mismo archivo las macros correspondientes a la secuencia de escape SGR (estos datos son constantes enteras)

3. Escribir una función que permita armar una string con las secuencias de escape personalizadas con argumentos de color de foreground o background, o comandos con valores de n y m ASCII.

4. Escribir un programa que pruebe ampliamente cada secuencia de escape

7 Punteros a puntero. Manejo avanzado de strings y punteros

7.1 Ejercicio 1

Asumiendo que realizó el Ejercicio 6.2, nos proponemos procesar el contenido de un archivo sin necesidad de conocer la interfaz de programación por ahora (solo por ahora ...). Para ello aprovecharemos la siguiente propiedad del shell de consola, que consiste en poder reemplazar un comando por su resultado. ¿Como se logra?. Simple: el comando se encierra entre apóstrofos[2], es decir, **`` `comando` ``**. Si además a la cadena entre apóstrofos la encerramos entre comillas dobles, vale la misma conclusión que en el ejercicio 6.2. pero en lugar de texto entre las comillas dobles llegará el contenido del archivo.
Una vez obtenido el texto debe desarrollar en un archivo separado (funciones.c) las siguientes funciones:
`int sgetline (char * , char ** );`
Lee una string hasta el siguiente '\n' y apunta al inicio de la línea con una variable puntero que recibe por referencia en el segundo argumento. Devuelve la cantidad de bytes que contiene la línea de texto.
`char * sgetword (char *);`
Recibe el puntero a la línea obtenida de la función anterior, y devuelve un puntero a una string con la primer palabra ingresada. Con ayuda de estas dos funciones hay que contar:

- La cantidad total de líneas del texto ingresado desde la línea de comandos.
- La cantidad total de palabras del texto.
- La cantidad total de caracteres (contando los '\n').

Nota: Evaluar pensando en el scope de las variables porque el segundo argumento es un puntero a puntero.

7.2 Ejercicio 2

Escribir un programa que implemente un conversor de base para números enteros con signo. Se debe requerir al usuario el ingreso de un número por teclado, y presentar un menú que permita seleccionar con una opción numérica la base de origen y a que base se quiere convertir.
Cada opción debe corresponder a un puntero a la función que implementa la conversión.
La función de conversión devuelve un puntero a una string con el resultado listo para ser presentado en pantalla, es decir no devuelve el número sino su representación ASCII terminada en '\0'.
Las funciones reciben un puntero a la string con el número ingresado y dos enteros con los códigos que representen la base de origen y destino respectivamente (Los valores numéricos de dichos códigos quedan a su criterio)

8 Estructuras

8.1 Ejercicio 1

A partir de una estructura denominada punto, compuesta por dos miembros de tipo double correspondientes a un par de coordenadas cartesianas ortogonales, definir una estructura que describa un rectángulo.
Escribir un set de funciones que permitan calcular:

- La superficie dle rectángulo
- La longitud de la base y de la altura
- La longitud de la diagonal que lo divide en un par de triángulos

[2] No confundir apóstrofo con comilla simple, `'`. El apostrofo es el que está inclinado a la izquierda `` ` ``.

8.2 Ejercicio 2

Definir un Tipo de Dato "Complex" que permita operar con números complejos. Implementar las funciones suma, resta, producto, cociente, y conjugado.

8.3 Ejercicio 3. Funciones de bibliotecas que utilizan estructuras

Basándose en las descripciones de las funciones de la biblioteca standard cuyos prototipos se definen en el header ***time.h***, (consultar como referencia, por ejemplo `https://www.tutorialspoint.com/c_standard_library/time_h.htm`, y luego acceder a man para obtener todos los detalles de la llamada), escribir un programa que realice las siguientes actividades:

- Presentar la fecha y hora actual en pantalla en formato dd:mm:yyyy hh:mm:ss
- Solicitar el ingreso de una fecha en ese formato y calcular la diferencia en segundos, en minutos, en hora y en días.

Part II
Programación avanzada

9 Condiciones Generales

Los programas entregables se componen de la estructura de archivos utilizada hasta ahora en la Parte I. Sin embargo en esta parte es obligatorio que cada entrega además de componerse de un archivo header personalizado, un ***main.c***, y por lo menos un archivo de funciones (***funciones.c*** o como se lo quiera denominar) debe ir acompañado de un ***Makefile*** de modo tal que los docentes simplemente ejecuten en una consola ***make***, y se construya en forma automática el código. Por otra parte el makefine debe tener reglas para limpiar objetos y ejecutable (clean).

10 Archivos / Streams

10.1 Ejercicio 1

Escribir un programa que abra su propio archivo fuente y lo presente en pantalla. No utilizar printf para acceder a stdout.

10.2 Ejercicio 2

Modificar el programa del **??** para que en lugar de imprimir su propio archivo fuente, reciba el archivo a presentar en pantalla como argumento por la línea de comandos.

10.3 Ejercicio 3

En el repositorio público se provee un archivo con números en punto flotante doble precisión. Se pide escribir un programa que abra dicho archivo, lea los valores y entregue en pantalla por cada grupo de 100 valores double leídos los siguientes valores y los imprima por stdout:

- Valor medio (o promedio), dado por la expresión: $\overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$
- Varianza, calculada mediante la siguiente expresión: $\sigma_n^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \overline{x})$

10.4 Ejercicio 4

Repetir el Ejercicio 10.2, pero utilizando las funciones de manejo de streams de entrada salida buffereada definidas en ***stdio.h***.

10.5 Ejercicio 5

Repetir el Ejercicio 10.3, pero utilizando las funciones de manejo de streams de entrada salida buffereada definidas en ***stdio.h***.

10.6 Ejercicio 6

Escribir un programa que inicialice con números reales representados en doble precisión normalizada, un array de estructuras punto con la utilizada en el Ejercicio **??**. La cantidad de elementos del array, se recibe como argumento por línea de comandos. Una vez terminada la generación del set de números, almacenarlos en un archivo ***./puntos.dp***.
Una vez que el archivo se haya generado, e lprograma deberá presentar un menú con las siguientes opciones cada una de las cuales deberá ser resuelta por una función separada:

- Presentar cada par de valores generados en el archivo
- Ordenar los valores mediante la coordenada x de cada punto, contenido en la estructura.

11 Desarrollo de Librerías de código

11.1 Ejercicio 1

Tomar los archivos **`sock-lib.c`** y **`sock-lib.h`** que se encuentran en el repositorio público del curso (en el path **`publico/Clases/Clase20-2017-09-18/sources`**), y aplicando los conceptos vistos en clase en el slide de Bibliotecas de código construir una biblioteca **`libInfo1sock.a`** y **`libinfo1sosk.so`**.
En su computador los archivos de las bibliotecas que deben referenciar las aplicaciones deben estar en el directorio `/Info1libs`. En el caso de los archivos de la biblioteca shared el archivo presente en este directorio debe ser el link por el cual se accede. El binario de la versión puede estar en otro path (siempre que se ebncuentre correctamente referenciado por el archivo.
Se pide además escribir un Makefile (puede trabajar a partir del Makefile que se distribuyó para compilar los ejemplos de clase), que permita crear ambas Bibliotecas y copiarlas al directorio `/Info1libs`. Para subirlo al git se pide un directorio para los archivos que se distribuyen de la biblioteca estática y el otro con los que se distribuyen de la biblioteca dinámica.

11.2 Ejercicio 2

Utilizando las bibliotecas almacenadas en el path `/Info1libs`, y ubicando los headers de las bibliotecas en el path `/Info1Includes`, retocar los fuentes y el Makefile para que los programas `cliente.c` y `servidor.c` compilen en las nuevas condiciones de operación.

11.3 Ejercicio 3

Repetir el procedimiento especificado en el Ejercicio 11.1 para la biblioteca de listas desarrollada en el Ejercicio **??**. Los archivos de las bibliotecas deben ir en el mismo path: `/Info1libs`.

11.4 Ejercicio 4

Del mismo modo que para el caso de los ejemplos de sockets, modificar el Makefile y los fuentes para poder compilar lso programas de listas bajo las nuevas condiciones de operación.

12 Procesos y Señales

12.1 Ejercicio 1

Escribir un programa que cree 10000 instancias child. Cada una debe dormir durante 5 segundos y finalizar. Mientras tanto el proceso padre deberá dormir 30 segundos o mas y terminar. Utilizar en una segunda consola el comando **`ps -elf | grep [nombre del programa ejecutable] | grep defunct | wc -l`**. El resultado debe ser 10000. ¿Porque?. Calibrar el tiempo que duerme el programa padre de modo de poder ver el estado de los 10000 childs luego de terminar. Tal vez la visualización del estado (simplemente eliminando el ultimo comando de la lista de comandos encolados con el pipe, le permita visualizar el detalle del estado de los procesos child y obtener la respuesta. Si le resulta mas cómodo ejecutar el **`sleep`** del proceso padre dentro de un **`while`** infinito. Para normalizar e equipo en caso de demora infinita en el padre, pulse **`CTRL-C`** en la consola donde el se está ejecutando el proceso.

12.2 Ejercicio 2

Tomar el código desarrollado en el Ejercicio 12.1 y agregarle un handler a la señal **`SIGCHLD`** dentro del cual se invoque la función **`wait ()`**, pasándole un puntero a **`NULL`** de modo de ignorar la información de estado del proceso. Repetir la comprobación del estado de los zombies, por medio del comando **`ps -elf | grep [nombre del programa ejecutable] | grep defunct | wc -l`** en otra terminal. ¿Solucionó el problema observado en el Ejercicio anterior?. ¿Puede evaluar a partir del resultado el comportamiento de las señales cuando se dirigen en forma masiva y cuasi simultánea a un mismo proceso?. Escribir un pequeño reporte en un archivo de texto plano a modo de reseña.

12.3 Ejercicio 3

Tomar el código desarrollado en el Ejercicio 12.2 y mejore el handler utilizando la función **`waitpid ()`**, utilizando **`WNOHANG`** para el argumento opciones (3er. Argumento), de modo de poder cerrar la totalidad de procesos childs que están solicitando **`exit ()`**, por cada ciclo de uso del handler de **`SIGCHLD`**. Repita el comando **`ps -elf | grep [nombre del programa ejecutable] | grep defunct | wc -l`** en otra consola y observe el resultado. Escribir un pequeño reporte en un archivo de texto plano a modo de reseña.

13 sockets

13.1 Ejercicio 1

Tomar el código de los archivos **`cliente.c`** y **`servidor.c`**, que figuran como ejemplos de base de clase ubicados en el público del curso (en el path **`publico/Clases/Clase20-2017-09-18/sources`**), y convertir a **`servidor.c`** en un servidor concurrente, que por cada pedido de un cliente responda el típico mensaje "Hola Mundo!" (ya habrá tiempo de complicar esta operatoria funcional).
Utilizar la syscall **`fork ()`** para este propósito creando una instancia hijo (child) que responda el pedido mientras que el proceso principal (el proceso padre, regrese a ejecutar la función **`Aceptar_pedidos ()`**). Tal como se dijo en clase, el proceso padre, *una vez creada la instancia hijo que resolverá el pedido*, deberá cerrar el socket duplicado regresado por la función **`Aceptar_pedidos ()`**, para luego volver a la línea en que invoca a dicha función una y otra vez.Por su parte el proceso hijo recientemente creado cerrará el socket original que el proceso padre utiliza para recibir pedidos de conexión, y utilizará solamente su copia del socket duplicado utilizando a éste para enviar el mensaje al cliente remoto.

Utilizar para desarrollar la biblioteca shared bajo las condiciones de uso descriptas en el Ejercicio 11.1 Ejecutar 10 veces el programa cliente para comprobar el correcto funcionamiento. Una vez ejecutado 10 veces el programa cliente inspeccionar con el comando **`ps -elf`**, el estado del proceso servidor. Anotar sus conclusiones en un archivo de texto plano **`result.README`** el que deberá subir junto con los fuentes de cliente y servidor y el Makefile correspondiente. En dicho archivo deberá explicar el resultado y proponer una solución en el caso de encontrar a su criterio algún problema.

13.2 Ejercicio 2

Tomar el código del programa **`servidor.c`** utilizado en el ejercicio 13.1 y modificarlo de modo que el proceso principal (el padre) espere por la finalización de cada proceso hijo creado por el programa principal sin bloquearse, de modo de ir a esperar mas pedidos de conexión con la función **`Aceptar_pedidos ()`**.
Utilizar la función **`waitpid ()`** del mismo modo que en el Ejercicio 12.3
Repetir las ejecuciones del programa cliente y seguidamente verificar con el comando **`ps -elf`**, el estado del proceso servidor. Anotar sus conclusiones en un archivo de texto plano **`result.README`** el que deberá subir junto con los fuentes de cliente y servidor y el Makefile correspondiente. En dicho archivo deberá explicar el resultado y proponer una solución en el caso de encontrar a su criterio algún problema.

13.3 Ejercicio 3

Tomar ahora el código del programa **`cliente.c`** utilizado en el Ejercicio anterior y modificarlo para solicitar 10000 pedidos consecutivos al mismo programa **`servidor.c`** resultante del Ejercicio anterior.
Al finalizar el proceso cliente, verificar con el comando **`ps -elf`**, el estado del proceso servidor. Anotar sus conclusiones en un archivo de texto plano **`result.README`** el que deberá subir junto con los fuentes de cliente y servidor y el Makefile correspondiente. En dicho archivo deberá explicar el resultado y proponer una solución en el caso de encontrar a su criterio algún problema.

13.4 Ejercicio 4

Tomar el programa **`servidor.c`** del Ejercicio 13.3, y limitar la cantidad de procesos child que pueden estar ejecutando en forma simultánea, de modo de limitar la sobrecarga de equipo en el que ejecuta el server.
Utilice la función **`sleep ()`** para simular la duración del proceso. Al finalizar el proceso cliente, verificar con el comando **`ps -elf`**, el estado del proceso servidor. Anotar sus conclusiones en un archivo de texto plano **`result.README`** el que deberá subir junto con los fuentes de cliente y servidor y el Makefile correspondiente.

13.5 Ejercicio 5

Tomar el programa del ejercicio 13.4, y agregar un archivo de configuración en el cual podamos escribir parámetros generales de uso. Por el momento configuraremos la cantidad de childs. El nombre del archivo es **`/etc/servidor.conf`** (usar sudo para editarlo con permisos de modificación, de otro modo lo abrirá read only).
El archivo de configuración es ascii, y la sintaxis es:
parámetro1=valor1

parámetro2=valor2
...
...
parámetron=valorn
El programa deberá tener una función para leer el archivo de configuración y establecer los valores globales de esas variables. Esa función deberá estar en un archivo llamado **`aux.c`**.

13.6 Ejercicio 6

Tomar el programa del Ejercicio 13.5 y reemplazar el manejador de la señal **`SIGHUP`**, de modo tal que cuando el proceso reciba esa señal vuelva a leer el archivo de configuración y recargar en tiempo real las variables globales con el/los eventual/es nuevo/s valor/es definido/s para el/los parámetro/s.
Reiniciar el nuevo servidor con una configuración dada en el archivo **`/etc/servidor.conf`**. Una vez activo el servidor editar el archivo de configuración y modificar su contenido (recordar invocar al editor con el comando sudo para poder ganar permisos de modificación en el directorio **`/etc`**). Salvar los cambios y enviar desde el shell la señal **`SIGHUP`** al proceso servidor para verificar que los parámetros modificados han sido efectivamente tomados por el servidor.

13.7 Ejercicio 7

Tomar el Ejercicio 13.6 y realizar las siguientes modificaciones:

1. Dejar en el fuente del programa principal solamente la función **`main ()`**, las variables globales y la línea **`#include "servidor.h"`**.
2. Crear el archivo **`seniales.c`** en el que se tengan las funciones manejadoras (handlers) de las diferentes señales que se reemplacen para el sistema en desarrollo, así como una función que reemplace los manejadores de las mismas por dichas funciones, cuyo prototipo se escribe a continuación:
   `int sig_trap (void);`
   La función devuelve 0 si todas las señales se reemplazaron exitosamente, y devuelve **`SIG_ERR`** en caso que al menos una no se haya podido reemplazar.
3. Crear un archivo child.c que contendrá la aplicación que debe ejecutarse en cada instancia child creada por el programa principal en respuesta a un pedido de conexión de un cliente. La función principal tiene el mismo prototipo, que la existente:
   `void child_process (int s);`
   En donde **`s`** es el file descriptor del socket obtenido por el proceso padre producto de invocar a la función **`Aceptar_conexiones ()`**. La función **`child_process`** debe implementar la siguiente operatoria:
   (a) Esperar por el socket **`s`** que el cliente envíe una string con el nombre de un archivo, con su ruta (path) completa o relativa.
   (b) Recibida esa ruta, intentará abrir el archivo en la ruta especificada (si recibió el nombre del archivo solamente, obviamente lo abrirá en el directorio actual).
   (c) Si el archivo no existe o no puede abrirse enviará por el socket al cliente la string "ERROR!: [string de descripción del error devuelta por **`strerror()`**] y finaliza invocando a la syscall **`exit ()`**.

(d) Si el archivo existe, el child y lo puede abrir correctamente enviará al cliente la string "OK", y a continuación le transmitirá el archivo solicitado por el socket.

(e) Finalizada la transmisión deberá terminar su ejecución invocando la syscall **`exit ()`**. En caso de no tener éxito en la transmisión también ejecutará **`exit ()`** pero con un código de finalización que indique esta situación.

4. Tomar el programa fuente del cliente y modificarlo para que interactúe con la instancia child del servidor de acuerdo con la operatoria descripta en el ítem 3. Como adicional el cliente tendrá una función que en caso de transcurrir 30 segundos desde que envía el nombre del archivo al server sin respuesta finalizará su ejecución, presentando en pantalla el mensaje de error correspondiente.

`Sugerencias:`
Trabajar en pequeños pasos incrementales. En especial al trasladar las funciones existentes a los archivos fuente nuevos que se requieren. Una por una. Modificar el **`Makefile`**, y ejecutar **`make`**. Pequeños pasos producen pocos o errores (o ninguno).
Para el caso del punto 4 explorar la syscall alarm ().

14 Directorios y otros nodos ("archivos") especiales

14.1 Ejercicio 1

Explorar la syscall open (**`man 2 open`**). Utilizando el prototipo

```
int open(const char *path, int oflags, mode_t mode);
```

Experimentar con el tercer argumento de la función, (**`mode`**), creando tres archivos, **`arch1`**, **`arch2`** y **`arch3`**, cuyos permisos en octal resulten en 742, 564, y 413 respectivamente. Verificar los permisos con el comando **`ls`** en el directorio en el que se crearon los archivos.
Considerar que hay una máscara por default que conviene modificar para tener amplia disponibilidad de establecer permisos. Para ello explorar la syscall **`umask ()`**, con **`man 2 umask`**. Tener en cuenta que el valor de máscara que se utiliza en una and con los permisos solicitados es **`mask`**, es decir e complemento a 1 de la máscara que generamos con **`umask ()`**.
Las macros para los diferentes atributos de permisos se encuentran en la misma página de **`man`** que describe la función.

14.2 Ejercicio 2

Ejecutar el comando **`stat`** desde una consola pasándole como argumento cualquiera de los tres archivos generados en el Ejercicio 14.1. Observar el resultado.
Explorar la syscall **stat ()** mediante `man 2 stat`. Escribir un comando similar a **`stat`** llamado **`my_stat`**, que permita presentar toda la información que la syscall **`stat ()`** reúne en la estructura:

```
struct stat {
               dev_t     st_dev;         /* ID of device containing file */
               ino_t     st_ino;         /* inode number */
               mode_t    st_mode;        /* protection */
```

```
    nlink_t    st_nlink;      /* number of hard links */
    uid_t      st_uid;        /* user ID of owner */
    gid_t      st_gid;        /* group ID of owner */
    dev_t      st_rdev;       /* device ID (if special file) */
    off_t      st_size;       /* total size, in bytes */
    blksize_t  st_blksize;    /* blocksize for filesystem I/O */
    blkcnt_t   st_blocks;     /* number of 512B blocks allocated */
    struct timespec st_atim;  /* time of last access */
    struct timespec st_mtim;  /* time of last modification */
    struct timespec st_ctim;  /* time of last status change */
};
```

El tipo de dato de cada miembro de **`struct stat`** está explicado en la página de man junto con las macros que definen los diferentes valores para cada caso.

14.3 Ejercicio 3

Desde la consola crear un hard link para cualquiera de los archivos y ejecutar **`my_stat`** para el archivo "original"y para el link creado mediante el comando **`ln`**.
Comparar la salida respecto de la salida del archivo original antes de crear el link.
Elaborar en un archivo de texto un breve informe con las conclusiones. Debe detallarse que valores cambian en el archivo original luego de creado el link, así como la correspondencia de valores al ejecutar **`my_stat`** en ambos archivos (original y enlace). ¿Que nos sugiere el número de i-nodo del archivo original y del enlace creado con **`ln`**?.¿Como explica el valor de este metadato en ambos archivos?

14.4 Ejercicio 4