Taller de Programación I (75.42)
Facultad de Ingeniería
Universidad de Buenos Aires

Contador de Palabras

Ejercicio N° 0

Objetivos	● Familiarizarse con el sistema de entregas SERCOM ● Preparación del ambiente de trabajo propio ● Nivelar conocimientos básicos en C ● Preparación de informe técnico
Instancias de Entrega	**Entrega 1:** clase 2. **Entrega 2:** clase 4.
Temas de Repaso	● Funciones para el manejo de archivos ● Manejo de memoria dinámica ● Punteros y aritmética de punteros ● Entrada y salida estándar ● Proceso de compilación ● Uso de operadores a nivel de bits ● Concepto y uso del debugger
Criterios de Evaluación	● Entrega exitosa en SERCOM dentro de los plazos estipulados. ● Correcta compilación y ejecución de casos de prueba en SERCOM ● Cumplimiento de los estándares de codificación en SERCOM ● Ausencia de errores de ejecución de Valgrind en SERCOM ● Presentación de informe impreso en la clase de entrega definida ● Presentación de carátula completa y firmada junto con el informe y el código fuente tal como fue devuelto por SERCOM.

El trabajo es personal*: debe ser de autoría completamente tuya. Cualquier forma de* ***plagio es inaceptable****: copia de otros trabajos, copias de ejemplos de internet o copias de tus trabajos anteriores (self-plagiarism).*
Si usas material de la cátedra deberás dejar en claro la fuente y dar crédito al autor (a la materia).

Índice

Introducción

En este trabajo práctico no se hará foco en la programación propiamente dicha, sino en el correcto entendimiento y uso de las herramientas de desarrollo, especialmente SERCOM. De todas formas, se espera que **el alumno repase por su propia cuenta aquellos conceptos indicados en la sección "Temas de Repaso"** , incorporados en materias anteriores, a fin de comprender el código propuesto en este ejercicio y prepararse para los próximos trabajos prácticos.

El trabajo práctico estará dividido en **pasos**, **cada uno de los cuales deberá contar con un capítulo especial en el informe** a entregar. Dicho documento incluirá capturas de pantalla, explicación y el análisis de resultados obtenidos según sea requerido en cada uno de los pasos.

Para simplificar, se provee el código fuente a ser utilizado cada paso. De todos modos, es responsabilidad del alumno su correcto análisis, entendimiento y prueba en su ambiente de trabajo local incluyendo:

- Compilación y ejecución bajo ambientes Linux
- Ejecución con Valgrind [1]
- *Debugging* mediante GDB [2]

Al finalizar, se deberá **entregar el informe junto con la versión de código fuente del último paso** formateada correctamente, siguiendo los lineamientos de entrega de trabajos prácticos indicados por la materia y **presentes en el sitio web** de la misma.

Descripción

Se requiere una aplicación de consola que cuente palabras y líneas de un archivo de texto plano determinado. El sistema debe recibir el archivo de texto por entrada estándar o como argumento de invocación por línea de comando.
El proceso del archivo consiste en analizar su contenido contando las palabras como conjuntos de caracteres separados por cierto delimitador. El total encontrado se debe imprimir por salida estándar.

Formato de Línea de Comandos

La sintaxis de ejecución del aplicativo será la siguiente

```
./tp [archivo]
```

El parámetro [**archivo**] es opcional e indica la ruta del archivo de texto a procesar. Su ausencia indica el uso de la entrada estándar.

Códigos de Retorno

Se requieren los siguientes códigos de error:

- 0 en caso de éxito

- 1 en caso de que el archivo especificado no exista o no se poseen permisos de lectura.

Entrada Estándar y Archivo de Texto

De ser requerido, el sistema leerá el contenido del archivo de entrada estándar. En cualquier caso, el formato del archivo de entrada se asume como texto plano con caracteres usuales.
Se considera una palabra a cualquier conjunto de caracteres que se encuentre delimitado por alguno de los siguientes elementos:

- inicio de archivo
- fin de archivo
- Caracteres delimitadores: espacio (‘ ‘), fin de línea (‘\n’), punto (‘.’), coma (‘,’), punto y coma (‘;’), dos puntos (‘:’).

Salida Estándar

Se espera el siguiente formato de salida estándar:

```
<cantidad-palabras>\n
```

Ejemplos de Ejecución

Ejemplo 1

A continuación se define el archivo de entrada **input.txt:**

```
El concepto de tipo de dato abstracto (TDA, Abstract Data Type), fue propuesto por primera vez hacia 1974 por John Guttag y otros, pero no fue hasta 1975 que por primera vez Liskov lo propuso para el lenguaje CLU.
```

La ejecución del programa se realiza con la siguiente línea de comando:

```
./tp input.txt
```

La salida estándar esperada es:

```
39
```

Ejemplo 2

A continuación se define el archivo de entrada **input.txt:**

```
C es un lenguaje de programación originalmente desarrollado por Dennis Ritchie entre 1969 y 1972 en los Laboratorios Bell, como evolución del anterior lenguaje B, a su vez basado en BCPL.
Al igual que B, es un lenguaje orientado a la implementación de Sistemas Operativos, concretamente Unix.
C es apreciado por la eficiencia del código que produce y es el lenguaje de
```

```
programación más popular para crear software de sistemas, aunque también se utiliza
para crear aplicaciones.
```

La ejecución del programa se realiza con la siguiente línea de comando:

```
cat input.txt | ./tp
```

La salida estándar esperada es:

```
77
```

Restricciones

La siguiente es una lista de restricciones técnicas exigidas para el sistema:

1. El sistema debe desarrollarse en C (C11) con el estándar POSIX 2008.
2. Está prohibido el uso de variables globales.
3. Se requiere un correcto uso de Tipos de Datos Abstractos (TDA).
4. Está prohibido el uso de funciones globales sin su correspondiente TDA salvo **'main**'.
5. Se deben respetar el uso de buenas prácticas de programación.

Pasos

Paso 0: Entorno de Trabajo

Si bien SERCOM es capaz de compilar y someter a prueba las entregas del alumno, usarlo como entorno de trabajo remoto es lento e ineficiente. Por eso es importante que el alumno invierta un tiempo en la configuración de un entorno de trabajo **local** que le será mucho más redituable y cómodo.

- Preparar un ambiente de trabajo local que incluya:
    a. Versión estable de Linux (Ubuntu, Debian, Fedora, o cualquier otra distribución).
    b. GCC >=5.4.0
    c. Valgrind
    d. GDB >=7.11

    Podes usar una máquina virtual y armarte el entorno o usar una imagen de docker ya armada y disponible. En https://taller-de-programacion.github.io/entorno-de-trabajo estan los detalles para ayudarte.
- En el ambiente local:
    a. Compilar y ejecutar una aplicación simple ISO C **que imprima por pantalla el mensaje "Hola Mundo"** y finalice retornando 0 (cero).
    Ej.
    ```
    gcc main.c -o tp
    ./tp
    ```
    b. Ejecutar dicha aplicación utilizando Valgrind.
    Ej.
    ```
    valgrind ./tp
    ```
- Repasar los temas de C indicados al comienzo del enunciado.
- Documentar:

a. Capturas de pantalla de la ejecución del aplicativo (con y sin Valgrind).
b. ¿Para qué sirve **Valgrind**? ¿Cuáles son sus opciones más comunes?
c. ¿Qué representa **sizeof()**? ¿Cuál sería el valor de salida de **sizeof(char)** y **sizeof(int)**?
d. ¿El **sizeof()** de una struct de C es igual a la suma del **sizeof()** de cada uno sus elementos? Justifique mediante un ejemplo.
e. Investigar la existencia de los archivos estándar: STDIN, STDOUT, STDERR. Explicar brevemente su uso y cómo redirigirlos en caso de ser necesario (caracteres **>** y **<**) y como conectar la salida estándar de un proceso a la entrada estándar de otro con un **pipe** (carácter **|**).

Nota:

*Respuestas copiadas de la documentación en línea de las herramientas, foros de internet, wikis o cualquier otra variante **quedan automáticamente descalificadas.** En todo momento **se prefiere una respuesta de realización propia, simple y concreta que resuma el conocimiento del alumno.***

Paso 1: SERCOM - Errores de generación y normas de programación

- Entregar el código fuente identificado para el paso actual subiendo un archivo .ZIP al SERCOM.
- Analizar el resultado de la ejecución. Observar que la entrega falló, debido a que SERCOM no pudo generar la aplicación correctamente y fallaron las normas de verificación de código.
- Documentar:
  a. Captura de pantalla mostrando los problemas de estilo detectados. Explicar cada uno.
  b. Captura de pantalla indicando los errores de generación del ejecutable. Explicar cada uno e indicar si se trata de errores del compilador o del *linker*.
  c. ¿El sistema reportó algún *WARNING*? ¿Por qué?

Paso 2: SERCOM - Errores de generación 2

- Volver a realizar una nueva entrega. Verificar los cambios realizados respecto de la entrega anterior utilizando el comando diff:

```
diff paso1_main.c paso2_main.c || diff paso1_wordscounter.c paso2_wordscounter.c
    || diff paso1_wordscounter.h paso2_wordscounter.h
```

- Observar que el chequeo de normas de codificación es exitoso pero aún no fue posible generar un ejecutable.
- Documentar:
  a. Describa **en breves palabras** las correcciones realizadas respecto de la versión anterior.
  b. Captura de pantalla indicando la correcta ejecución de verificación de normas de programación.
  c. Captura de pantalla indicando los errores de generación del ejecutable. Explicar **cada uno** e indicar si se trata de errores del **compilador o del *linker***.

Nota:

En la materia se usa **diff** por ser la herramienta estándar *de facto* pero existen otras herramientas similares, muchas de ellas gráficas como **meld**.

Invertí tiempo en familiarizarte con diff o meld ahora para no perder tiempo en otros TPs

Paso 3: SERCOM - Errores de generación 3

- Volver a realizar una nueva entrega. Verificar los cambios realizados respecto de la entrega anterior utilizando el comando diff:

```
diff paso2_main.c paso3_main.c || diff paso2_wordscounter.c paso3_wordscounter.c
    || diff paso2_wordscounter.h paso3_wordscounter.h
```

- Observar que aún no fue posible generar un ejecutable.
- Documentar:
    a. Describa en breves palabras las correcciones realizadas respecto de la versión anterior.
    b. Captura de pantalla indicando los errores de generación del ejecutable. Explicar **cada uno** e indicar si se trata de errores del **compilador o del *linker***.

Paso 4: SERCOM - *Memory Leaks* y *Buffer Overflows*

- Volver a realizar una nueva entrega. Verificar los cambios realizados respecto de la entrega anterior utilizando el comando diff:

```
diff paso3_main.c paso4_main.c || diff paso3_wordscounter.c paso4_wordscounter.c
    || diff paso3_wordscounter.h paso4_wordscounter.h
```

- Observar que el archivo ejecutable se generó exitosamente y se ejecutaron los casos de prueba.
- Observar que las pruebas ‘TDA’, ‘C Language’ y ‘STDIN’ corrieron exitosamente aunque fallaron al ejecutar con Valgrind.
- Observar que las pruebas ‘Invalid File’, ‘Long Filename’ y ‘Single Word’ no lograron correr exitosamente.
- Documentar:
    a. Describa en breves palabras las correcciones realizadas respecto de la versión anterior.
    b. Captura de pantalla del resultado de ejecución con **Valgrind** de la prueba ‘TDA’. Describir los errores reportados por Valgrind.
    c. Captura de pantalla del resultado de ejecución con **Valgrind** de la prueba ‘Long Filename’. Describir los errores reportados por Valgrind.
    d. ¿Podría solucionarse este error utilizando la función **strncpy**? ¿Qué hubiera ocurrido con la ejecución de la prueba?
    e. Explicar de qué se trata un **segmentation fault** y un **buffer overflow**.

Paso 5: SERCOM - Código de retorno y salida estándar

- Volver a realizar una nueva entrega. Verificar los cambios realizados respecto de la entrega anterior utilizando el comando diff:

```
diff paso4_main.c paso5_main.c || diff paso4_wordscounter.c paso5_wordscounter.c
    || diff paso4_wordscounter.h paso5_wordscounter.h
```

- Observar que las pruebas ‘TDA’, ‘C Language’, ‘STDIN’ y ‘Long Filename’ corrieron exitosamente sin fallas de Valgrind.
- Observar que las pruebas ‘Invalid File’ y ‘Single Word’ no lograron correr exitosamente.
- Ubique en el SERCOM el **caso de prueba** ‘Single Word’, descargue y descomprima sus archivos de entrada.
- Ubique en el SERCOM el **Makefile** usado y descárguelo.
- Asegúrese que todos los archivos de código fuente, el archivo ‘input_single_word.txt’ y el archivo

Makefile se encuentran **en el mismo directorio**.

- Ejecute el comando **hexdump** para examinar el contenido del archivo 'input_single_word.txt' en modo binario:

  ```
  hd input_single_word.txt
  ```

  O bien

  ```
  hexdump -C input_single_word.txt
  ```

- Coloque el archivo 'input_single_word.txt' junto con todos los archivos de código fuente. **Compile el código** utilizando el comando make:

  ```
  make
  ```

- Ejecute el comando **gdb** para examinar el depurar la ejecución del programa:

  ```
  gdb ./tp
  info functions
  list wordscounter_next_state
  list
  break 45
  run input_single_word.txt
  quit
  ```

- Documentar:
  a. Describa en breves palabras las correcciones realizadas respecto de la versión anterior.
  b. Describa el motivo por el que fallan las prueba 'Invalid File' y 'Single Word'. ¿Qué información entrega SERCOM para identificar el error? Realice una captura de pantalla.
  c. Captura de pantalla de la ejecución del comando **hexdump**. ¿Cuál es el último carácter del archivo input_single_word.txt?
  d. Captura de pantalla con el resultado de la ejecución con **gdb**. Explique brevemente los comandos utilizados en **gdb**. ¿Por qué motivo el debugger no se detuvo en el breakpoint de la línea `45: self->words++;`?

Nota:
Debuggear o depurar un programa haciendo uso **printf**, aunque simple, es método cuanto mucho ineficiente. El uso de un debugger como **gdb** facilita enormemente la tarea de *debugging*: aprender a usar un debugger será un *habilidad invaluable* a la hora de detectar y corregir errores en los trabajos prácticos posteriores.
Si bien en el presente trabajo se requiere el uso de **gdb**, en los siguientes trabajos el alumno puede optar por usar su *debugger* de su preferencia (DDD, Eclipse, etc).

> ***Invertí tiempo en buscarte un debugger con el que te sientas cómodo ahora. Lo vas a necesitar.***

Paso 6: SERCOM - Entrega exitosa

- Volver a realizar una nueva entrega. Verificar los cambios realizados respecto de la entrega anterior utilizando el comando diff:

  ```
  diff paso5_main.c paso6_main.c || diff paso5_wordscounter.c paso6_wordscounter.c
  || diff paso5_wordscounter.h paso6_wordscounter.h
  ```

- Observar que todas las pruebas fueron exitosas.
- Compilar el código y ejecutar localmente la prueba 'Single Word' utilizando distintas variantes:

  ```
  make
  ```

```
./tp input_single_word.txt
./tp <input_single_word.txt
./tp <input_single_word.txt >output_single_word.txt
```

- Documentar:
    a. Describa en breves palabras las correcciones realizadas respecto de la versión anterior.
    b. Captura de pantalla mostrando **todas las entregas realizadas**, tanto exitosas como fallidas.
    c. Captura de pantalla mostrando la ejecución de la prueba 'Single Word' **de forma local** con las distintas variantes indicadas.

Paso 7: SERCOM - Revisión de la entrega

- Revisar el estado de todas las pruebas ejecutadas en SERCOM con el código del paso 6.
- Abrir cada una de las salidas de **Valgrind** y controlar que no hay errores reportados que no fueran detectados como un fallo en la ejecución. Revisar con atención el listado de archivos abiertos al finalizar el programa.
- Controlar el código final entregado. Verificar el uso de buenas prácticas de programación y el cumplimiento del enunciado del trabajo.
- Proceder con la impresión del código fuente entregado por el SERCOM en la página de entregas y adjuntar al informe.
- Recordar utilizar la carátula brindada por la materia en el sitio de la cátedra.

Paso 8: Netcat, ss y tiburoncin

Para las prácticas de sockets usaremos algunas herramientas adicionales con las que te conviene familiarizarse.

- `**netcat**` o `**nc**` te permite establecer una comunicación TCP emulando ser un cliente o un servidor.
- `**ss**` te mostrará que comunicaciones están "*en espera*" o "*listen*" y cuáles están activas.
- `**tiburoncin**` es una herramienta open source que tendrás que compilar e instalar a partir del código fuente. Con `**tiburoncin**` podrás espiar que bytes se envían entre un cliente y un servidor. Muy útil para debuggear! Source: https://github.com/eldipa/tiburoncin

En una consola ejecutá:

```
nc -l -p 9081
```

Verás que `**netcat**` se queda bloqueado: está "*en espera*" de un cliente. Dejalo así.

a) ¿Qué hacen los flags "-l" y "-p"?

En otra consola ejecutá:

```
ss -tuplan
```

b) Verás muchas cosas dependiendo de tu sistema. Sacá un screenshot y resaltá la línea que hace referencia al **netcat** que dejaste "en espera".

En una tercer consola ejecutá:

```
nc 127.0.0.1 9081
```

Verás que `**netcat**` se queda bloqueado esperando ***tu input***.

Escribí algunas frases finalizadas con un enter (pero no presiones Ctrl-C ni Ctrl-D).

c) Qué ves en la **primera** consola (en el primer **netcat**)?

Ejecutá nuevamente `ss -tuplan`.

d) Cuantos **netcats** vez? ¿Cuáles son sus estados? Sacá un screenshot y resaltá esas líneas.

Cerrá ambos **netcats** tipeando o Ctrl-C o Ctrl-D.

En una consola abrí un nuevo **netcat**:

```
nc -l -p 9091
```

En otra consola lanzá `**tiburoncin**`:

```
tiburoncin -o -A 127.0.0.1:9095 -B 127.0.0.1:9091
```

Verás que **tiburoncin** se conectará al primer **netcat** y quedará "*en espera*" a que otro programa se conecte a él.

En una tercer consola lanza un segundo **netcat:**

```
nc 127.0.0.1 9095
```

Escribí algunas frases finalizadas con un enter (pero no presiones Ctrl-C ni Ctrl-D) en ambos **netcats**. Dejalos corriendo.

e) ¿Qué ves en la consola donde está corriendo **tiburoncin** ? ¿Qué relación hay con `**hexdump**`? Por qué se dice que `**tiburoncin**` es un ****man in the middle***? (Nota: **tiburoncin** debería seguir corriendo, sino, hiciste algo mal en alguno de los pasos anteriores, como cerrar uno de los **netcats** por error.)

Cerrá ambos **netcats**. Verás que `**tiburoncin**` también se cerró y que escribió 2 archivos que representan los bytes enviados de un **netcat** a otro.

f) Mostrá el contenido de uno de ellos tal como está (haciendo **cat**). Luego mostrá el mismo archivo pero corriendo el siguiente comando:

```
xxd -p -c 16 -r ElArchivoQueEscribioTiburoncin | hexdump -C
```

Referencias

[1] http://valgrind.org/
[2] https://www.gnu.org/software/gdb/
[3] https://github.com/eldipa/tiburoncin