Trabajo Práctico “0”

Versión 2.2

¿De qué se trata?

El TP0 es una práctica inicial para empezar a familiarizarse con algunas de las herramientas necesarias para el trabajo práctico cuatrimestral como es la configuración del entorno, el lenguaje C, etc. Es un ejercicio que sirve como base para empezar el TP luego.

El TP0 va a ser realizado en **etapas**, cada una de ellas con un entregable que servirá de base para la siguiente. La idea de este ejercicio es que lo realicen de manera individual o grupal (no más de cinco, idealmente los mismos con los que harán el TP cuatrimestral), y en unas semanas tendremos una entrega **obligatoria**. No es necesario tener el ejercicio completo y la entrega no lleva nota, **pero presentarse con lo que tengan en condición necesaria para la continuidad de la materia**. Más adelante publicaremos junto al enunciado del TP la fecha de entrega de este ejercicio.

Objetivo

El objetivo de este TP0 es empezar a familiarizarse con el entorno en el que desarrollarán el TP de la materia, aprendiendo en el proceso cómo utilizar las commons por su cuenta.
La idea es que siguiendo este documento logren completar las funciones vacías y comentarios que les dejamos en el código. Pueden hacernos cualquier pregunta que tengan sobre el enunciado,C, o cualquier otro concepto en el foro de consultas del TP, o pueden esperar a encontrarnos en algún sábado de soporte si sienten que es demasiado complejo para explicarlo ahí.

Requisitos

- La VM con Lubuntu: https://www.utnso.com.ar/recursos/maquinas-virtuales/
- Tener a mano el repo del TP0: https://github.com/sisoputnfrba/tp0

Etapa 1: Setup Inicial

El objetivo de esta etapa es tener el **entorno bien configurado** para poder desarrollar sin problemas las etapas siguientes. También te va a servir para ya tener el entorno configurado para el resto del TP.

Para ello, prendamos la Máquina Virtual e instalemos las Guest Additions. Pero, **¿qué son las Guest Additions?** Son unas herramientas extras que permiten que Virtualbox nos provea de ciertas funcionalidades; como, por ejemplo, poder **hacer pantalla completa**.

Cómo hacerlo en este documento es un poco aburrido, tenemos un video enseñando como se hace!

Una vez instaladas las Guest Additions, necesitamos instalar el repo de las "commons", la biblioteca con funciones útiles y necesarios que utilizarán a lo largo de todo el desarrollo del TP cuatrimestral, que va a ser una dependencia en nuestro tp0. A partir de acá, todo se va a hacer con la consola, así que preparen su Ctrl+T para abrirla. Protip: Por lo general, siempre vamos a estar en la "home" del usuario utnso: `/home/utnso` (o `~/`, para los amigos).

Para bajar el repo de las commons, vamos a usar git, la herramienta de versionado que usamos por defecto en la cátedra. Con ***`git clone`***, bajemos las commons e instalemoslas usando ***`make install`***.

```
git clone https://github.com/sisoputnfrba/so-commons-library
cd so-commons-library
sudo make install
cd ..
```

¡Ya tenemos la Máquina Virtual lista para poder arrancar con el TP0!

De la misma forma que bajamos el proyecto de las commons, bajemos el del tp0:

```
git clone https://github.com/sisoputnfrba/tp0
```

Recuerden bien la ubicación donde se está bajando, ¡porque la vamos a necesitar para **configurar Eclipse**!

De la misma manera que las Guest Additions, tenemos un video para saber como Importar un Proyecto con "makefile".
¿Por qué con makefile? Porque make es una *herramienta de gestión de dependencias y compilación* que vamos a estar usando en la materia y nos permite distribuirles un proyecto ya armado, sin necesidad de usar el Eclipse de antemano.

Configurado el proyecto en Eclipse, **ya tenemos la máquina virtual preparada** para programar el TP0 (y de yapa, *¡el tp mismo!*).

Etapa 2: Comandos básicos

El objetivo de esta etapa es aprender un par de funcionalidades que utilizaremos bastante durante todo el desarrollo del trabajo práctico cuatrimestral.

Logging

Durante todo el tp iremos logueando en un archivo de texto las diferentes acciones que el programa vaya realizando, tanto las correctas, como los errores. Para ello utilizaremos las funciones de logging que proveen las commons.

Parados en el archivo **`tp0.c`**, si revisamos el header de logs de las commons, habla de la función **`log_create()`**, que nos devuelve un logger listo para usar. Usaremos esa función para configurarlo para que:

- Loguee en el archivo tp0.log
- Muestre los logs por pantalla y no solo los escriba en el archivo.
- Muestre solo los logs del nivel info para arriba.

Creado nuestro logger, usemos **`log_info`** para loggear el string `"Soy un Log"` y cerremos el logger al final del programa con el **`log_destroy`**. Compilamos usando el símbolo del martillo (o apretamos ctrl + B) y démosle Run al programa (o apretamos ctrl + F11).

Archivos de Configuración

Estaría bueno loggear un valor que no sea un literal en el código, por lo que vamos a leer el valor de un archivo de configuración y lo vamos a loggear usando nuestro logger.

Para ello vamos a usar las config de las commons. Usando el link, creemos una config sobre el archivo `"tp0.config"` y obtengamos el valor de la key CLAVE, en formato string.

Usemos el logger anterior para mostrar el valor que obtuvimos. Compilamos, corramos el programa y evaluemos los resultados.

Nota: no se olviden de destruir el config al final del programa!

Leer de Consola

De los comandos básicos, nos queda leer de consola. Si bien existen muchas formas de hacerlo, vamos a usar la biblioteca **`readline`**.

Necesitamos incluirla usando `#include<readline/readline.h>`. Una vez incluida, al llamar la función `readline("")`, va a hacer que el programa espere a que se ingrese una línea y devolverla en un string ya listo para loggear. La misma ya se encuentra agregada en el TP0, por lo que no hace falta hacer este include

Recuerden que readline no te libera la memoria que devuelve, por lo que es necesario liberarla usando **`free(1)`**.

Terminando con esta etapa, nos gustaría que el tp0 **lea de consola** todas las líneas que se ingresen, las **loggee** y si se ingresa una línea vacía, terminar el programa.

Si ejecutamos el comando `man readline` en la consola, podemos ver que el valor de retorno de la función, ante una línea vacía, es un string vacío. Pero cómo hacemos para revisar eso?

Más adelante hablamos un poquito de que los strings son cadenas de caracteres terminadas con `'\0'`. Esto implica que un string vacío va a tener, en su contenido, como primer valor ese caracter, por lo que podemos usar una comparación de ese valor como **condición de corte**.

Haciendo uso de la función strcmp de la biblioteca estándar de C para strings, podemos comparar lo que nos devuelva readline contra el string vacío (“”) para saber si debemos salir del bucle.

Etapa 3: Programar el Cliente-Servidor

A partir de esta etapa, vamos a plantear una arquitectura cliente-servidor. Para esta sección, tanto el cliente como el servidor en sus respectivas carpetas tiene un archivo **utils.c**, que tienen las funciones vacías con los comentarios con lo que debemos hacer para poder conectar dos procesos mediante la red. Para poder ayudar con los conceptos y aspectos técnicos de estos tienen disponible la guía de sockets.

El entregable de esta etapa es enviar al servidor el valor de CLAVE en el archivo de configuración, y luego enviar al servidor todas juntas, las líneas que se ingresen por consola.

Simplificando un poco, una conexión por socket hacia otro programa va a requerir de alguna manera realizar lo siguiente:

- Iniciar el servidor en la función **iniciar_servidor** del utils del servidor
- Esperar que el cliente se conecte en la función **esperar_cliente**
- Crear una conexión contra el servidor en la función **crear_conexion** del utils del cliente
- Enviar los datos
- Cerrar la conexión

Para simplificar el TP0, tenemos ya pre implementadas un par de funciones para comunicarnos con el servidor en el archivo fuente utils, que deberán consultarlo de manera similar al log y config de la etapa anterior:

- **enviar_mensaje(3)**: Recibe el socket, el tamaño de los datos y los datos a mandar.

- **liberar_conexion(1)**: Termina la conexión y libera los recursos que se usaron para gestionar la misma.

El único problema, es que estas funciones no nos sirven para enviar las líneas de consola todas juntas, por lo que vamos a **crear un paquete**. Este paquete nos va a asegurar que toda la información que mandemos se envíe junta. Para ello les proveemos otro conjunto de funciones o "API" para crear, rellenar y enviar paquetes:

- **crear_paquete**(): Nos crea el paquete que vamos a mandar.
- **agregar_a_paquete(2)**: Dado un string y su tamaño, lo agrega al paquete.
- **enviar_paquete(2)**: Dada una conexión y un paquete, lo envía a través de ella.
- **eliminar_paquete(1)**: Elimina la memoria usada por el paquete.

Es importante recalcar, que hay que enviar un string (y no un stream). La diferencia entre ambos radica en que, si bien ambos son una seguidilla de "bytes", **el string termina en el caracter \0**.

Dado que nuestra función recibe un **string**, es necesario incluir este caracter (denominado centinela) al calcular el tamaño del string a mandar. Esto a que viene? Si consultan la documentación de una función amiga llamada `man strlen` verán que **nos devuelve el tamaño de un string, sin contar el caracter centinela**, por lo que hay que "sumarle 1".

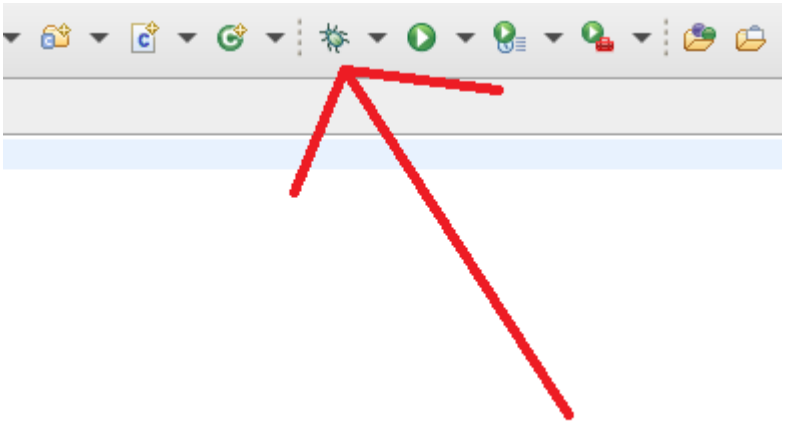
Usando **enviar_mensaje** para enviar nuestro valor de config y usando **enviar_paquete** para enviar las líneas de consola, deberíamos poder llegar al entregable.

Etapa 4: Debugger

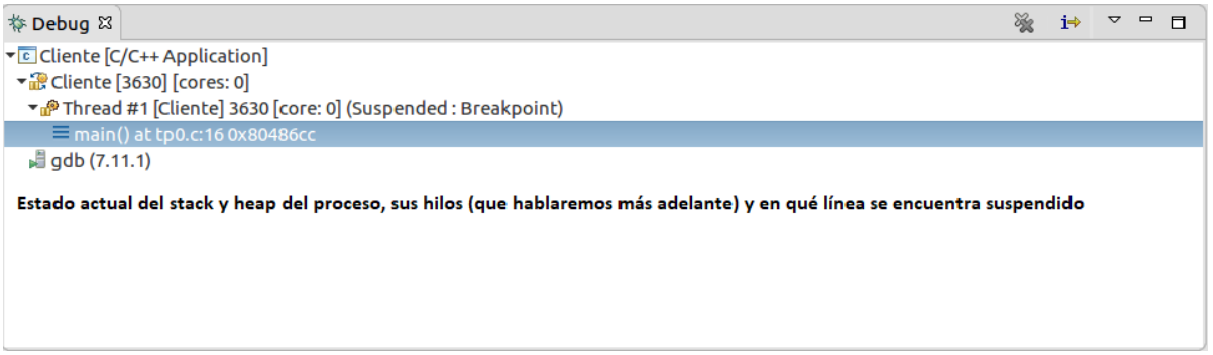
En esta etapa vamos a aprender a usar el debugger del IDE, **por qué?** Porque es una herramienta que nos va a ayudar a probar el sistema y encontrar errores. Para esto, necesitamos que algo rompa. Para ello vamos a cambiar el único uso de la función **crear_paquete()**, por su homónima **crear_super_paquete()**.

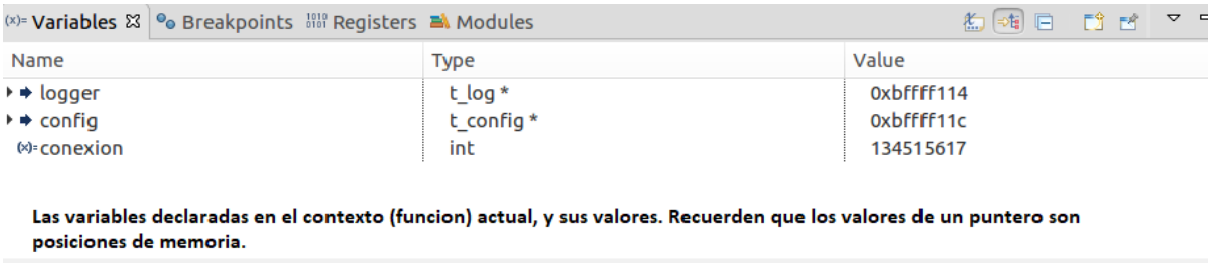
A diferencia de su prima-hermana, al correr el programa, **crear_super_paquete** hace que se genere un Segmentation Fault. Vamos a arreglar esta función para que tenga el mismo comportamiento anterior.

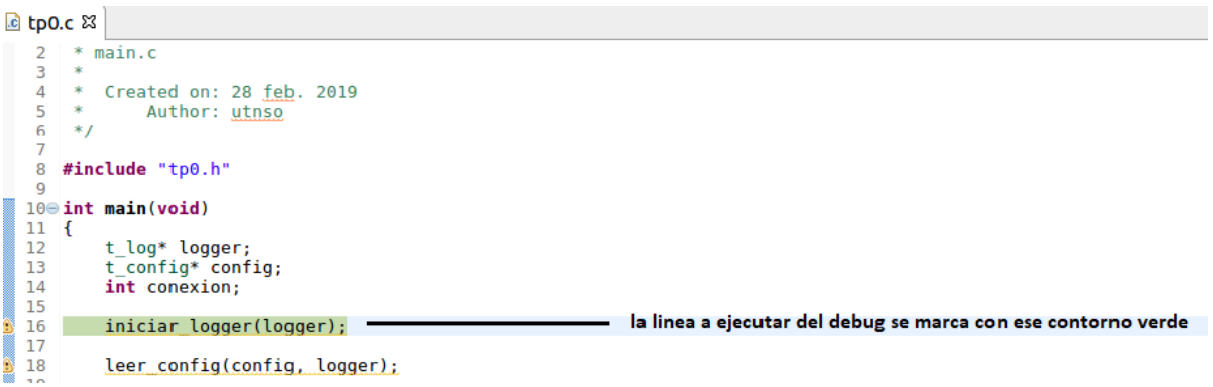
Para ello vamos a entrar a la pantalla de debugger con el símbolo del "bichito":

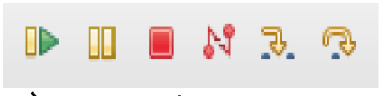


Allí vamos a tener la siguiente pantalla, que explicamos por partes:







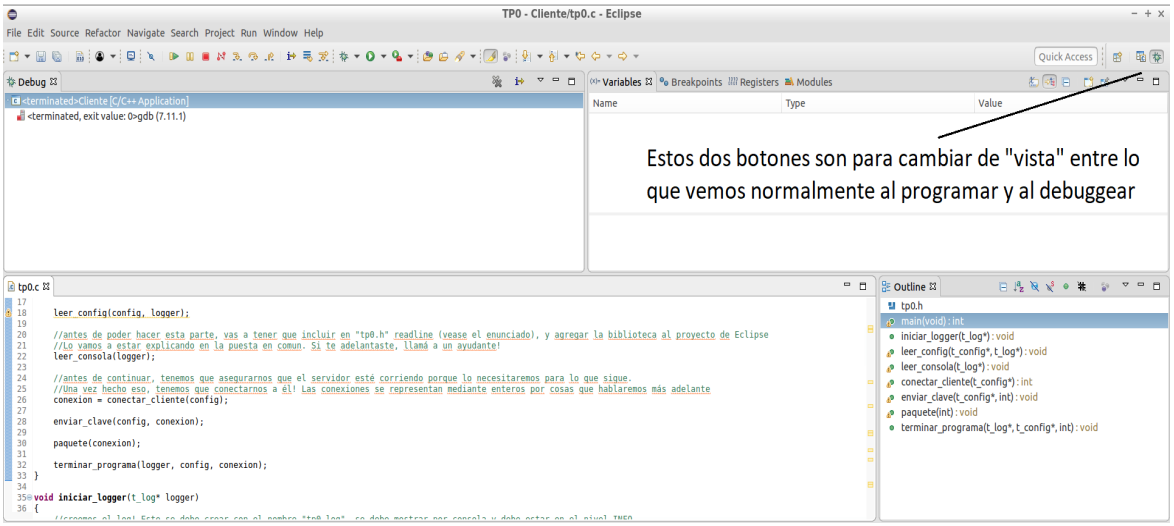


Step over(F6): Si la linea a ejecutar fuese una función, continúa a la linea que le sigue a su llamado y ejecuta la función sin debuggearla

Step into (F5): Si la linea a ejecutar fuese una función, abre su definición y continua debuggeando desde su primera línea

Continuar el programa normalmente hasta el siguiente break point

Corta la ejecucion. Muy importante porque no se puede corregir codigo y re ejecutar sin hacer esto primero



Vamos a ir avanzando con el comando **step over**, hasta llegar a una línea que falle.

Si evaluamos por qué falló la ejecución, podemos ver que la variable paquete **contiene basura**. Esto nos indica que a nuestra super función, le faltó algo: **reservar memoria para el paquete**.

paquete	t_paquete *	0xbffff0b4
codigo_operacion	op_code	(PAQUETE | unknown: 2588164930)
buffer	t_buffer *	0xa1aead53
size	int	
stream	void *	

Para reservar memoria, existen una familia de funciones que se pueden ver haciendo `man malloc`. La función que nosotros vamos a usar es **`malloc(1)`**, que recibe por parámetro el tamaño de memoria a pedir (osea, el tamaño de nuestro paquete) y devuelve un puntero a la posición de memoria que te reservó.

Pero cómo obtenemos el tamaño del paquete? Peguemosle una revisada a **crear_paquete** y veamos cómo lo resuelve! Podemos acceder a la definición de una función haciendo ctrl + click sobre su llamado.

Con la función arreglada, probemos volver a correr el programa cliente y veamos como, al debuggear, pudimos arreglar el problema.

Nota: Si se perdieron en algún paso, también pueden ver este vídeo.

Notas Finales

Fue un poco largo, pero aprendimos un montón! Recapitulemos un poco:

- Pudimos configurar nuestro entorno de desarrollo.
- Aprendimos a usar funciones de las commons que nos van a ser muy útiles.
- Aprendimos sobre reservar memoria, liberarla y leer por consola!
- Pudimos mandar mensajes por red a otro programa.
- Mostramos un poco del proceso de debuggeo.

Esto fue todo, pero recuerden que el TP0 es solo una introducción a todas las herramientas que podemos usar. Por lo que les pedimos que hagan uso de los links de Links Útiles, del redireccionario, de las Guías y los Videos Tutoriales para mejorar constantemente y llegar bien holgados a fin de cuatrimestre!

Hasta la próxima amigos!