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

Duck Game

Ejercicio / Prueba de Concepto N° 1
Sockets

Objetivos	● Modularización del código en clases Socket, Protocolo, Cliente y Servidor entre otras. ● Correcto uso de recursos (memoria dinámica y archivos) y uso de RAII y la librería estándar STL de C++. ● Encapsulación y manejo de Sockets en C++
Entregas	● **Entrega obligatoria:** clase 4. ● **Entrega con correcciones:** clase 6.
Cuestionarios	● Sockets - Recap - Networking
Criterios de Evaluación	● Criterios de ejercicios anteriores. ● Resolución completa (100%) de los cuestionarios *Recap.* ● Cumplimiento de la **totalidad** del enunciado del ejercicio incluyendo el **protocolo** de comunicación y/o el **formato** de los archivos y salidas. ● Correcta **encapsulación** en clases, ofreciendo una interfaz que **oculte** los detalles de implementación (por ejemplo que **no** haya un *get_fd()* que exponga el *file descriptor* del Socket).. ● Código **ordenado**, separado en archivos .cpp y .h, con **métodos y clases cortas** y con la **documentación** pertinente. ● Empleo de memoria dinámica (*heap*) justificada. Siempre que se pueda **hacer uso del *stack*,** hacerlo antes que el del *heap*. Dejar el *heap* sólo para reservas grandes o cuyo valor sólo se conoce en *runtime* (o sea, es dinámico). Por ejemplo hacer un *malloc(4)* está mal, seguramente un *char buf[4]* en el *stack* era suficiente. ● Acceso a información de archivos de forma ordenada y moderada.

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

Índice

- Introducción
- Descripción
- Formato de Línea de Comandos
- Códigos de Retorno
- Entrada y Salida Estándar
- Ejemplos de Ejecución
  - Ejemplo 1
  - Ejemplo N
- Restricciones
- Referencias

Introducción

En este trabajo haremos una *prueba de concepto* del TP enfocándonos en el uso de *sockets*.

Puede que parte o incluso el total del código resultante de esta PoC ***no*** te sirva para el desarrollo final de tu juego. – Y está bien.

En una PoC el objetivo es familiarizarse con la tecnología, en este caso, con los sockets. Familiarizarse ***antes*** de embarcarse a desarrollar el TP real te servirá para tomar mejores decisiones.

Descripción

Duck Game cuenta con un amplio arsenal de armas para eliminar al pato enemigo. En este trabajo desarrollarán un mercado por el cual el cliente podrá preparar su armamento para la fase de juego. Deberán implementar un protocolo de comunicación para poder acceder a las armas y municiones disponibles. El servidor contará con un registro de todas las transacciones realizadas.

> **Nota:** en esta PoC el servidor atiende a **un solo cliente y nada más**. En la PoC de *threads* aprenderás cómo atender a múltiples clientes en paralelo.
>
> **Podes usar el socket provisto por la cátedra** (usa el último commit): https://github.com/eldipa/sockets-en-cpp . Solo no te olvides de **citar** en el readme la fuente y su licencia.
>
> Podes implementar **tu** propio socket o reescribir algunos métodos de la clase socket provista para practicar, pero hacelo cuanto tengas el TP ya andando así tenes *peace of mind*.
>
> **Recordar** que tenes que tener el código de parsing, impresión y lógica del juego **separado** del código de armado de protocolo así como tener separado este del código del socket. No tengas métodos que parseen, armen un mensaje y lo envíen por socket todo en un solo lugar, así como no tengas la lógica del juego llamando directamente a `std::cout`.

En esta PoC te parecerá un overkill pero cuando te enfrentes al TP Final el *"separar"* te permitirá organizarte mejor en tu equipo, trabajar en paralelo y testear por separado. Es tu ***única arma para mitigar la complejidad y el gran tamaño del TP.***

Nota: los archivos son **archivos de texto**, están **libres de errores** y cumplen el formato especificado.

Recomendación: puede que te interese repasar algún **container de la STL** de C++ como `std::vector` y `std::map`.

Recomendación: usar el `operador >>` de `std::fstream` para la lectura de la entrada estándar y de los archivos del servidor por que simplifica enormemente el parsing. Puede que también quieras ver los métodos `getc` y `read` de `std::fstream` para leer ciertas partes de los archivos.

Que sea C++ quien parsee por vos.

Formato de Línea de Comandos

El servidor recibirá por línea de comandos el puerto o servicio donde escuchará una **única conexión**.

```
./server <puerto>
```

Por su parte, el cliente recibirá la ip o host y el puerto o servicio

```
./client <hostname> <servicio>
```

y leerá de entrada estándar las instrucciones que va a querer realizar.

El servidor comenzará la comunicación presentándose, enviando el mensaje:

“`What arya buying?`”

Y luego se quedará esperando a que le lleguen órdenes de compra.
Los patos tienen buena memoria, por lo tanto se memorizaron los códigos de las armas disponibles. Para realizar una compra, el pato debe introducir la letra ‘B’ seguida por los códigos de los productos que está buscando.

```
B27
```

El servidor, al recibir la orden le informa al cliente cual es el estado de su armamento, comunicando que armas tiene equipadas y cuanta munición cuenta para cada una. El servidor imprimirá por consola el nombre del arma comprada y en el caso de la munición la cantidad que estaría llevando de la misma.

Magnum
7

El pato puede enviar varias órdenes de compra, pero solamente puede cargar 1 arma primaria y 1 arma secundaria. Si el pato ya tiene una Banana equipada y envía una orden por una Magnum, perderá las municiones que tenía para la Banana y se quedará con la Magnum equipada con 0 balas.
Además el pato puede decidir si lleva equipado un Knife o no.

Código	Equipamiento
1	Banana (Secondary)
2	Magnum (Secondary)
3	Old Pistol (Secondary)
4	Pew Pew Laser (Primary)
5	Phaser (Primary)
6	Chaindart (Primary)
7	Secondary Ammo (+7)
8	Primary Ammo (+10)
9	Knife

El cliente, al recibir la respuesta del servidor imprime cuál es el estado de su armamento en ese momento, con el siguiente formato:

Primary Weapon: **Chaindart** | Ammo: **0**
Secondary Weapon: **Old Pistol** | Ammo: **35**
Knife Equipped: **No**

Protocolo

El protocolo cuenta con 3 tipos de mensajes:

- Las órdenes de compra, que consisten en un arreglo de caracteres en formato ascii de 1 byte finalizado por un byte en null(0x00).
- El envio de strings, con el siguiente formato:
  - 2 **bytes sin signo en formato big endian**, indicando el largo del string.
  - La tira de bytes del string, sin contar el terminador de palabra (‘\0’).

  <largo del payload> <payload>
- **Un entero de 8 bits sin signo** para comunicar las municiones de cada arma
- **Un entero de 8 bits sin signo,** 0 si no está equipado el cuchillo, 1 si lo esta.

Ejemplo de Ejecución

Server:
./server 8080

00 11 57 68 61 74 20 61 72 79 61 20 62 75 79 69
6e 67 3f

Banana
00 0c 4e 6f 74 20 45 71 75 69 70 70 65 64
00
00 06 42 61 6e 61 6e 61
00
00

Secondary Ammo: 7
Secondary Ammo: 14
Pew Pew Laser
00 0d 50 65 77 20 50 65 77 20 4c 61 73 65 72
00
00 06 42 61 6e 61 6e 61
0e
00

Cliente:

./client localhost 8080

What arya buying?
B1
42 31 00

Primary Weapon: Not Equipped | Ammo: 0
Secondary Weapon: Banana | Ammo: 0
Knife Equipped: No

B774
42 37 37 34 00

Primary Weapon: Pew Pew Laser | Ammo: 0
Secondary Weapon: Banana | Ammo: 14
Knife Equipped: No

Primary Ammo: 10
Secondary Ammo: 21
Magnum
Equipped knife
00 0d 50 65 77 20 50 65 77 20 4c 61 73 65 72
0a
00 06 4d 61 67 6e 75 6d
00

B8729
42 38 37 32 39 00

Primary Weapon: Pew Pew Laser | Ammo: 10
Secondary Weapon: Magnum | Ammo: 0
Knife Equipped: Yes

q

En negrita se encuentra la entrada estándar, y en cursiva los mensajes del protocolo.
El server comienza la comunicación enviando el mensaje "`What arya buying?`" respetando el protocolo para enviar strings
El cliente manda una orden de compra.
El servidor imprime cada operación, y al finalizar envía el estado del inventario al cliente con el siguiente formato:

- El arma principal equipada en formato de string. Si no hay arma equipada envía "`Not Equipped`"
- La munición primaria disponible en 1 byte sin signo
- El arma secundaria equipada en formato de string. Si no hay arma equipada envía "`Not Equipped`"
- La munición secundaria disponible en 1 byte sin signo
- Si está equipado el cuchillo o no, en 1 byte sin signo.

El cliente imprime este resultado con el formato indicado en la sección anterior, deja una línea en blanco y vuelve a esperar por otra orden de compra.

Si el cliente ingresa una 'q' este debe cerrarse ordenadamente.

Si el pato intenta agregar munición cuando no tiene un arma equipada, la munición no varía pero se registra el intento de compra igualmente, se imprime la cantidad (0) de munición en ese momento.

Códigos de Retorno

Tanto el cliente como el servidor retornarán 0 en caso de éxito o 1 en caso de error (argumentos inválidos, archivos inexistentes o algún error de sockets).

Recomendaciones

Los siguientes lineamientos son claves para acelerar el proceso de desarrollo sin pérdida de calidad:

1. **¡Repasar los temas de la clase!** Los videos, las diapositivas, los handouts, las guías, los ejemplos, los tutoriales, los recaps. **Todo**. ***Muchas soluciones y ayudas están ahí.***
2. **¡Programar por bloques!** No programes todo el TP y esperes que funcione. Debuggear un TP completo es más difícil que probar y debuggear sus partes por separado. No te compliques la vida con diseños complejos. **Cuanto más fácil sea tu diseño, mejor**.
   Dividir el TP en bloques, codearlos, testearlos por separada y luego ir construyendo hacia arriba.
   ¡Si programas así, podés hasta hacerles tests fáciles antes de entregar!. Teniendo cada bloque, es fácil armar un bloque de más alto nivel que los use. La **separación en clases** es crucial.
3. **Usa las tools!** Corre *cppcheck* y *valgrind* a menudo para cazar los errores rápido y usa algun **debugger** para resolverlos (GDB u otro, el que más te guste, lo importante es que **uses** un debugger)
   Usa la librería estándar de C++ y las clases que te damos en la cátedra. Cuando más puedas reutilizar código oficial mejor: menos bugs, menos tiempo invertido.
4. **Escribí código claro**, sin saltos en niveles de abstracción, y que puedas leer entendiendo qué está pasando. Si editás el código "*hasta que funciona*" y cuando funcionó lo dejás así, **buscá la explicación de por qué anduvo**.

Restricciones

La siguiente es una lista de restricciones técnicas exigidas por el cliente:

1. El sistema debe desarrollarse en C++17 con el estándar POSIX 2008.
2. Está prohibido el uso de **variables globales**, **funciones globales** y **goto** (salvo código dado por la cátedra). Para este trabajo no es necesario el uso de excepciones (que se verán en trabajos posteriores).
3. Todo socket utilizado en este TP debe ser **bloqueante** (es el comportamiento por defecto).
4. Deberá haber una clase **Socket** tanto el socket aceptador como el socket usado para la comunicación. Si lo preferís podes separar dichos conceptos en 2 clases.
5. Deberá haber una clase **Protocolo** que encapsule la serialización y deserialización de los mensajes entre el cliente y el servidor. Si lo preferís podes separar la parte del protocolo que necesita el cliente de la del servidor en 2 clases pero asegurate que el código en común no esté duplicado.
   - La idea es que ni el cliente ni el servidor tengan que armar los mensajes *"a mano"* sino que le delegan esa tarea a la(s) clase(s) **Protocolo**.