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

Worms

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. 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

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

En esta PoC tendremos solamente 1 cliente y 1 servidor.

El cliente leerá de entrada estándar (**stdin**) los comandos emulando ser acciones del jugador.
La primera acción deberá ser:

- `select <scenario>`

Con `select`, el cliente le indica al servidor que quiere crear una nueva partida usando un escenario particular.

Para esta PoC el cliente tendrá un solo y único gusano cuya posición inicial será determinada por el servidor según el escenario elegido. Con el fin de simplificar el código, en esta PoC las posiciones del gusano serán números enteros `(x, y)` ***(como filas y columnas de una matriz)***.

> **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.
>
> También, en esta PoC estamos usando **posiciones enteras** y el servidor está **emulando la física**. En el TP Final usarás posiciones decimales (**float**) y usarás *Box2D* como *engine* físico.

El gusano se podrá controlar a través de las instrucciones que el cliente seguirá leyendo de entrada estándar (**stdin**). Estas pueden ser:

- `dir <direction>`
- `move`
- `jump <type>`

Con `dir` el gusano mira a una dirección u otra según el argumento `<direction>`. Este puede ser `0` (el gusano mira a la izquierda) o `1` (el gusano mira a la derecha).

Con `move` el gusano avanza **1** posición en la dirección que esté mirando. El servidor validará la posición para evitar que el gusano pueda avanzar sobre una pared.

Con `jump` el gusano dará un salto hacia adelante (`<type>` es `0`) o hacia atrás (`<type>` es `1`). El servidor validará las posiciones del salto también.

El salto hacia adelante consiste en mover el gusano en **1** hacia arriba y en **2** hacia la dirección a la que está mirando. El salto hacia atrás consiste en mover el gusano en **2** hacia arriba y en **1** hacia la dirección ***opuesta*** a la que está mirando.

Al recibir la operación, el servidor emulará y validará las posiciones intermedias y frenará ante cualquier choque del gusano con una pared o contra el suelo. El servidor también emulará la gravedad haciendo que si el gusano está en el aire, este caiga hasta encontrar el suelo.

Una vez terminada la emulación, el servidor le responde al cliente con la posición final del gusano.

Cuando el cliente encuentre el fin de la entrada estándar, el cliente deberá cerrar su socket y finalizar. Del lado del servidor, al detectar el cierre del socket deberá también finalizar.

> **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*.

Protocolo

Nota: en la descripción de abajo, `01, 02, 03,` … representan el valor de un byte. Números como `0001` y `00000001` representan un valor de 2 y 4 bytes respectivamente. La notación usada es hexadecimal.

Select

Para seleccionar y crear una nueva partida con el escenario pedido, el cliente envía:

`01 <L> <N>`

El campo `<L>` determina la longitud del nombre `<N>; <L>` es **un entero de 16 bits sin signo en big endian** y `<N>` es una secuencia de bytes **sin** el `\0`.

El servidor puede responder con un código de error (el escenario no fue encontrado):

`01`

O con un código de éxito, seguido de la posición inicial del gusano en ella:

`00 <X> <Y>`

Tanto `<X>` como `<Y>` son **enteros de 32 bits sin signo en big endian**.

Dir, move y jump

Para cambiar de dirección en la que mira el gusano: `<D>` es **un entero de 8 bits sin signo**. Vale `00` si la nueva dirección es a la izquierda o `01` si es a la derecha.

`03 <D>`

Para mover el gusano:

`04`

Para realizar un salto: `<T>` es **un entero de 8 bits sin signo**. Vale `00` si el salto es hacia adelante o `01` si el salto es hacia atrás.

`05 <T>`

En todos los casos el servidor por su parte responde con la posición final del gusano Tanto `<X>` como `<Y>` son **enteros de 32 bits sin signo en big endian**.

`<X> <Y>`

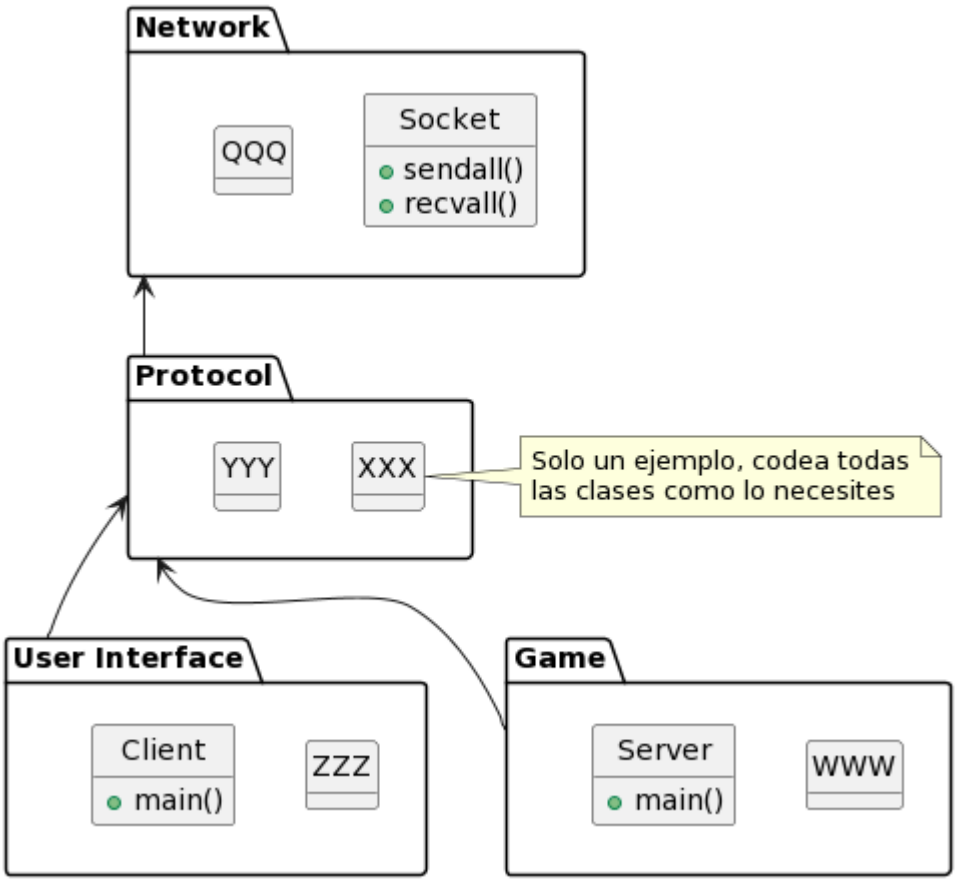
Finalización

Una vez que el cliente termina de leer de la entrada estándar, este cierra la conexión con el servidor y finaliza. El servidor por su parte, detecta el cierre por parte del cliente y finaliza también.

> **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.***
>
> *Ya sabes lo que dicen, la mezcla es lo que hace mal.*

Diagramas

En el **Readme** del proyecto deberás hacer un **diagrama de clases** que muestre las clases y métodos más importantes (no, no queremos ver todas las clases). Agrupa las clases según sean de la capa de *network*, *protocol, user interface o game*. Podes poner **tus propias categorías**, lo que buscamos acá es que **separes tu código y que no mezcles** sockets con parsing y con un `std::cout` metido por el medio.



Formato de Línea de Comandos

Cliente:

```
./client <ip/hostname server> <puerto/servicename>
```

Servidor:

```
./server <puerto/servicename> <escenarios>
```

El archivo `<escenarios>` tiene la definición de los escenarios que el cliente podrá elegir y crear una nueva partida con él.

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).

Archivos de Entrada y Salida

El cliente leerá de la entrada estándar las acciones a simular `select, dir, move, jump` (explicadas más arriba). Si la entrada estándar tiene una línea vacía o una línea que comienza con el símbolo **# debe ignorarse**.

Ante cada acción de movimiento `select, dir, move, jump`, el cliente debe imprimir por salida estándar la posición del gusano que obtuvo por parte del servidor. Por ejemplo, si el servidor le respondió con la posición `(x, y) = (3, 2)`, el cliente debe imprimir en una línea la coordenada `X` seguida por la `Y` separadas por un espacio:

```
3 2\n
```

Si el `select` falla, el cliente deberá imprimir el mensaje "`Fallo`" y finalizar.

El servidor por su parte por cada `select, dir, move y jump` recibido imprimirá por salida estándar la posición final del gusano con el mismo formato que el cliente. Por ejemplo, si las posiciones intermedias fueron `(1, 2) y (1, 3)` y la final `(1, 4)`, el servidor imprimirá:

```
1 4\n
```

El archivo `<escenarios>` es un **archivo de texto** y define 1 o más escenarios donde **cada escenario** tiene el siguiente formato:

```
<W> <H> <N>\n
<cell><cell><cell><cell>....<cell>\n
<cell>....
….
<cell><cell>...           ..<cell>\n
```

Los campos `<W>` y `<H>` marcan las dimensiones del escenario (ancho y alto); `<N>` el nombre del escenario. Estos campos están separados por 1 o más espacios.

En las siguientes `<H>` líneas habrán en cada una `<W>` caracteres representando una celda (`<cell>`) del escenario. Los valores posibles son:

- `X`: representa una posición de un suelo/pared (viga).
- `G`: la posición inicial de gusano cuando una partida es creada. La dirección inicial en la que mira el gusano es siempre a la izquierda.
- *un espacio*: representa un lugar libre.

  Nota: tanto la entrada estándar del cliente como el archivo `<escenarios>` son **archivos de texto**, están **libres de errores** y cumplen el formato especificado. Se garantiza además que todos los escenarios tendrán un techo, piso y paredes (habrá vigas alrededor de todo el escenario)

  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 `get` y `read` de `std::fstream` para leer **ciertas** partes de los archivos.

  Que sea C++ quien parsee por vos.

Ejemplos de Ejecución

Supongamos que el archivo `<escenarios>` tiene:

```
9 5 basic
XXXXXXXXX
X       X
X       X
X G X   X
XXXXXXXXX
4 4 small
XXXX
X  X
X GX
XXXX
```

Como puede verse hay 2 escenarios, `basic` y `small`.

Supongamos que el cliente recibe por entrada estándar el siguiente mensaje:

```
select basic
```

El cliente entonces le envía al servidor:

```
01 0005 basic
```

El servidor verifica que tiene dicho escenario y responde

```
00 00000002 00000003
```

Donde `(2, 3)` es la **posición inicial** del gusano en dicho escenario. Notar que el `(0,0)` está en la esquina superior izquierda, como en una matriz. El gusano mira inicialmente hacia la derecha.

El servidor y el cliente imprimen por salida estándar:

```
2 3
```

Solo a efectos ilustrativos voy a mostrar el escenario, en ningún momento se pide imprimirlo. El servidor debería tener en memoria el siguiente escenario:

```
XXXXXXXXX
X       X
X       X
X G X   X
XXXXXXXXX
```

Si el cliente *lee* un `dir 1`, le enviará al servidor el mensaje:

```
03 01
```

El servidor cambia la dirección del gusano para que mire hacia la derecha. Luego le responde al cliente con:

```
00000002 00000003
```

Tanto el servidor como el cliente imprimen

```
2 3
```

Si el cliente *lee* un `move` de la entrada estándar y le enviará al servidor el siguiente mensaje:

```
04
```

El servidor hace avanzar al gusano y le responde al cliente con la nueva posición `(3,3)` con el siguiente mensaje:

```
00000003 00000003
```

El escenario quedaría así:

```
XXXXXXXXX
X       X
X       X
X  GX   X
XXXXXXXXX
```

El servidor deberá imprimir por salida estándar

```
3 3
```

Y el cliente también

```
3 3
```

Si el cliente envía otro `move`, el servidor detecta que el gusano no puede avanzar y le responde con `(3,3)`.

Tanto el servidor como el cliente imprimen por salida estándar:

```
3 3
```

Si el cliente lee un `jump 0` , esto es un `jump` *hacia adelante*, le envía al servidor:

```
05 00
```

El servidor mueve el gusano a las posiciones `(3, 2)`, `(4, 2)` y finalmente `(5, 2)`. Observá que el salto hizo mover al gusano en **1** hacia arriba y en **2** hacia adelante pero el servidor debe verificar las **posiciones intermedias** para evitar que un gusano pueda atravesar una pared.

```
XXXXXXXXX
X       X
X    G  X
X   X   X
XXXXXXXXX
```

Dado que el gusano **está en el aire** (no hay un piso debajo de él), el servidor emula la gravedad moviéndose hacia abajo. El servidor le responde al cliente con la posición final del gusano, que debería ser `(5, 3)`.
El mensaje sería:

```
00000005 00000003
```

El escenario quedaría así:

```
XXXXXXXXX
X       X
X       X
X   XG  X
XXXXXXXXX
```

El servidor debería imprimir:

```
5 3
```

El cliente deberá imprimir:

```
5 3
```

Si el cliente ahora envía un `jump` *hacia atrás*, el servidor movería al gusano a las posiciones `(5, 2)`, `(5, 1)` y `(4, 1)`.

```
XXXXXXXXX
X   G   X
X       X
X   X   X
XXXXXXXXX
```

Luego, por gravedad el gusano terminará en la posición `(4, 2)`:

```
XXXXXXXXX
X       X
X   G   X
X   X   X
XXXXXXXXX
```

El servidor debería imprimir:

```
4 2
```

Y el cliente:

```
4 2
```

Si el cliente ahora envía otro `jump` *hacia atrás*, el servidor movería al gusano a las posiciones `(4, 1)`, `(4, 0)` y `(3, 0)`. Sin embargo, el gusano solo se moverá hasta la posición `(4,1)` porque la siguiente posición `(4,0)` está ocupada por una `X`. En ese momento el salto se *suspende* y resta simular la gravedad.

Luego, por gravedad el gusano terminará otra vez en la posición `(4, 2)`:

```
XXXXXXXXX
X       X
X   G   X
X   X   X
XXXXXXXXX
```

El servidor debería imprimir:

```
4 2
```

Y el cliente:

```
4 2
```

Si ahora el cliente detecta el fin de archivo en la entrada estándar, deberá cerrar su socket y finalizar y el servidor hará lo mismo,

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**.