.UBAfiuba
FACULTAD DE INGENIERÍA

# Taller de Programacion I - Cátedra Veiga

Resumen 2C 2020

# Índice

# 1. Memoria en C/C++

Tanto C como C++ se caracterizan como lenguajes en el cual el programador puede tener control absoluto sobre donde y como se ejecuta el codigo. Ambos tambien son caracterizados como lenguajes portables, en donde un compilador *traduce* código fuente a un código de maquina compatible con la arquitectura de la computadora. Esto tiene como consecuencia la variacion de tamaño en bytes de los tipos, los cuales pueden variar de arquitectura en arquitectura.

## 1.1. Padding

Los compiladores, en busqueda de optimizar el funcionamiento del programa, alinean las variables en posiciones de memoria multiplos de 4 (Tambien varia dependiendo la arquitectura y los flags de compilación) dado que las variables alineadas son accedidas mas rapidamente que aquellas que no lo estan. Esto genera pequeños bloques de memoria que no tienen uso mas que alinear una variable a una posición multiplo de 4. Esto se llama *padding* y es un tradeoff entre velocidad y espacio.

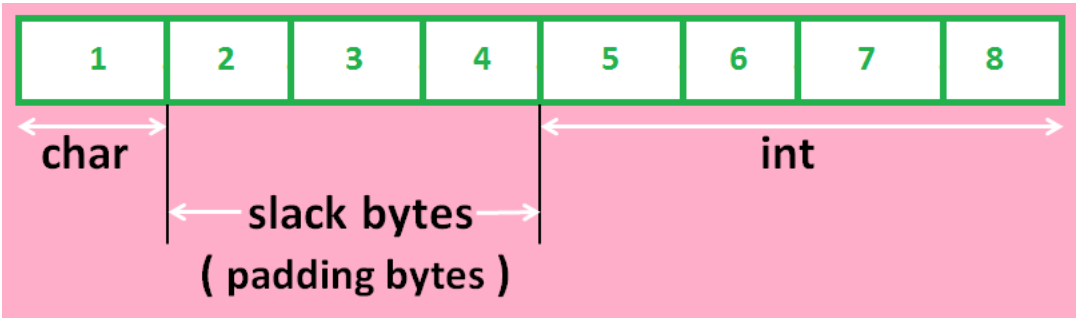



*Figura 0.1.* Gráfico de Padding Bytes.

Los paddings son mas notorios cuando se utilizan structs que contienen varias variables dentro. El compilador utiliza los llamados padding bytes para optimizar el acceso a las variables dentro del struct. Naturalmente, en un caso donde tenemos suficiente memoria este proceso sirve para optimizar la velocidad de ejecución. Sin embargo, en un sistema donde la memoria es limitada y se utilizan miles de estas estructuras podemos llegar a un caso donde no tenemos la memoria suficiente como para ejecutar el programa correctamente. Es por esto que podemos desactivar este padding dentro de los structs utilizando un atributo de GCC: *__attribute__((packed))* que le indica al compilador que no genere estos padding bytes, optimizando el uso de memoria pero bajando la velocidad de acceso.

## 1.2. Endianness

Una variable puede representarse de distintas formas dentro de una computadora. Tenemos, por un lado, **Big Endian**, en donde el byte mas significativo se lee/escribe primero (esta primero en memoria), y por otro lado **Little Endian** en donde es el byte menos significativo quien se lee/escribe primero en memoria.

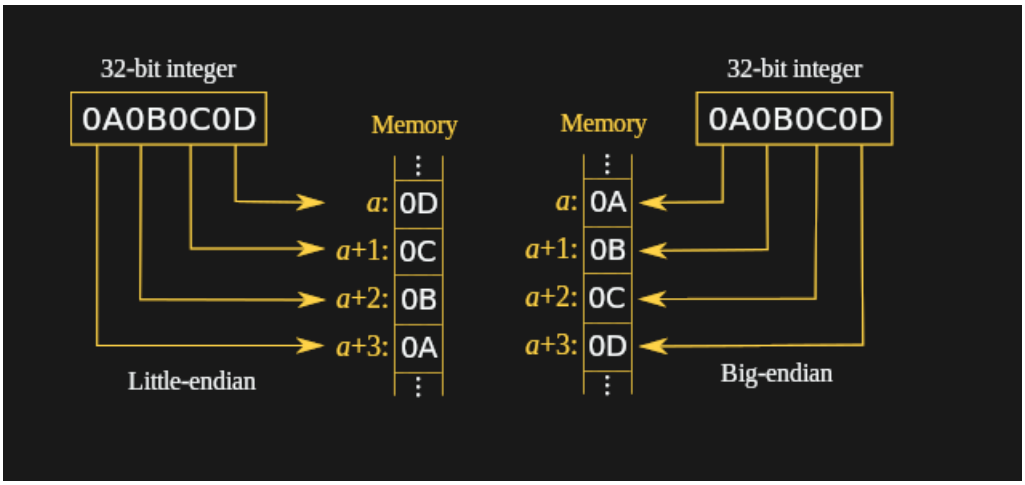



*Figura 0.2.* Distintas representaciones de una misma variable.

Normalmente, esto nos es irrelevante si no nos interesa la representación en memoria de una variable, pero es de vital importancia si queremos interpretar una variable como una tira de bytes, ya sea para escribirlo en un archivo binario o bien enviarlo por la red a través de un socket. Por lo tanto, siempre debemos especificar en que endiannes se estan guardando/enviando los datos.

¿Qué pasa cuando queremos envíar una variable entre dos computadoras que manejan arquitecturas diferentes? C/C++ nos brinda una serie de funciones con las cuales podemos cambiar el endianness de una variable short int o int del endiannes nativo de la computadora a big endian (el endianness que se utiliza en la red) y viceversa

```
#include <arpa/inet.h>.

uint32_t htonl(uint32_t hostlong); // Hostlong to network byte order
uint16_t htons(uint16_t hostshort); // Hostshort to network byte order
uint32_t ntohl(uint32_t netlong); // Netlong to host byte order
uint16_t ntohs(uint16_t netshort); // Netshort to host byte order
```

### 1.3. Segmentos de Memoria

- **Code Segment:** Solo lectura y ejecutable, donde va el código y las constantes.
- **Data Segment:** Variables creadas al inicio del programa y validas hasta que el mismo termina, pueden ser de acceso global o local.
- **Stack:** Variables creadas al inicio de una llamada a una función y destruidas automáticamente cuando esta llamada termina.
- **Heap:** Variables cuya duración esta determinada por el programador (run-time)

### 1.4. Duración y Visibilidad

- **Duración:** Tiempo desde que a la variable se le reserva memoria hasta que la misma es liberada.
- **Visibilidad:** Cuando a una variable se le puede acceder y cuando esta oculta.

```
1  int g = 1;              // Data segment; scope global
2  static int l = 1;       // Data segment; scope local (este file)
3  extern char e;          // No asigna memoria (es un nombre)
4
5  void Fa() { }           // Code segment; scope global
6  static void Fb(){}      // Code segment; scope local (este file)
7  void Fc();              // No asigna memoria (es un nombre)
8
9  void foo(int arg) {     // Argumentos y retornos son del stack
10 int a = 1;              // Stack segment; scope local (func foo)
11 static int b = 1;       // Data segment; scope local (func foo)
12
13 void * p = malloc(4);   // p en el Stack; apunta al Heap
14 free(p);                // liberar el bloque explicitamente
15
16 char *c = "ABC";        // c en el Stack; apunta al Code Segment
17 char ar[] = "ABC";      // es un array con su todo en el Stack
18 }                        // fin del scope de foo: las variables locales son liberadas
```

## 2. Sockets

### 2.1. Protocolo IP

En el protocolo IP, los mensajes son ruteados a sus destinos (hosts). Existen dos esquemas de direcciones: IPv4 (4 bytes para direcciones de máquina) e IPv6 (16 bytes para direcciones de máquina). Son redes *best effort* es decir, no garantiza que no se pierda ningun paquete en la comunicación, ni que dichos paquetes

lleguen en el orden adecuado o que no haya paquetes duplicados. Es un protocolo que fue pensado para simplificar la cuestión de hardware de la red, no para simplificar la vida del desarrollador.

## 2.2. Protocolo TCP

Como este protocolo corre sobre IP, permite el direccionamiento a nivel de servicio (port). Es un protocolo orientado a bytes, no a mensajes. Es decir, los bytes no se pierden, desordenan ni duplican, pero no garantiza *boundaries*. Ademas, es un protocolo orientado a la conexión, donde hay un participante pasivo que aguarda una comunicación, y otro participante que es quien la inicia de forma activa. Al no saber nada sobre mensaje y solos sabe de bytes, puede que un mensaje llegue incompleto.

## 2.3. Protocolo DNS

No es necesario recordar la direccion IP del destino (4 a 16 bytes dependiendo si utiliza IPv4 o IPv6) si no su nombre de dominio. Estos nombres de dominio se registran a nivel gubernamental, por ejemplo **.com.ar (Manejado por Cancillería Argentina)**. La resolución del dominio a una o varias direcciones IP las maneja un servidor DNS.


*Figura 0.3.* Protocolo DNS.

## 2.4. Resolución de nombres

```
Cliente
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_INET; /* IPv4 */
hints.ai_socktype = SOCK_STREAM; /* TCP */
hints.ai_flags = 0;

```

```
status = getaddrinfo("fi.uba.ar", "http", &hints, &results);

Servidor
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_INET; /* IPv4 */
hints.ai_socktype = SOCK_STREAM; /* TCP */
hints.ai_flags = AI_PASSIVE;

status = getaddrinfo(0 /* ANY */, "http", &hints, &results);

 freeaddrinfo(results); // Cliente y Servidor
```

## 3. Structs y Clases en C++

Por convención, todas las funciones de un TDA deben tener como prefijo de su nombre el nombre del TDA. Esto ocurre dado que todas las funciones terminan en un mismo espacio global, por lo tanto el metodo *get* de vector crearía conflicto con el *get* de una matriz. C++ nos brinda mejores formas de resolver estos conflictos. Ademas, en C, las funciones de los TDA recibian como primer parametro un puntero al struct. C++ tambien nos brinda formas para solucionar esto. Observando el siguiente código de un struct en C++:

```
struct Vector {
int *_data; /*private*/
int _size; /*private*/

void vector_create(int size) {
    this->_data = malloc(size*sizeof(int));
    this->_size = size;
}

int vector_get(int pos) {
    return this->_data[pos];
}

void vector_destroy() {
    free(this->_data);
}
};
```

Podemos notar dos cosas: Primero, las funciones y los datos del TDA estan integrados en una sola unidad. A los datos se les llama atributos, mientras que a las funciones se las llama métodos. Segundo, notemos que los métodos no tienen como prefijo de su nombre el nombre del TDA, ni tampoco reciben por

parametro un puntero al struct. Esto es porque es un parametro implicito, que contiene un puntero a la instancia, y se lo llama *this*. Este puntero es un puntero constante a la instancia, por lo que no se puede modificar. Ahora la pregunta es: ¿Como se evitan las colisiones de los nombres de los métodos? Los métodos del TDA no entran en conflicto con otros aunque se llamen igual, dado que en rigor, el método get del vector se llama *Vector::get* mientras que el get de una matriz se llama *Matriz::get*

## 3.1. Permisos de Acceso

Por defecto, un struct tiene sus atributos y métodos publicos, es decir que pueden accederse desde cualquier lado. Esto se puede cambiar especificando los permisos de acceso. Por ejemplo, **Private** hace que solo los métodos internos puedan acceder a dicho método o atributo. Observemos el siguiente código para un struct en C++:

```
struct Vector {
private:
    int* data;
    int size;

public:
    void vector_create(int size) {
        this->data = malloc(size*sizeof(int));
        this->size = size;
    }

    int vector_get(int pos) {
        return this->data[pos];
    }

    void vector_destroy() {
        free(this->data);
    }
};

void f(){
    Vector v;
    v.data; // Lanza un error, el atributo es privado.
}
```

Todo lo visto para structs en C++ aplica para las clases, solo que en las clases, por default, todos sus métodos y atributos son privados.

## 3.2. Herencia y Polimorfismo

Una clase o struct puede heredar de otra clase o struct sus atributos y métodos publicos o protegidos, utilizando la siguiente sintaxis:

```
class A : public B{
    ...
};
```

Notese que no se heredan los constructores de la clase padre. La clase padre puede especificar que ciertos métodos pueden ser *overrideados*, es decir, que su implementación va a diferir de aquella en la clase padre. Para especificar esto, la clase padre debe utilizar la keyword **virtual** la cual crea una entrada en la **vtable** (Tabla de métodos virtuales) para dicha clase. Podemos ejemplificar esto en el siguiente código:

```
class B{

private:
    int atributo0;

public:
    int atributo1;
    int atributo2;

    virtual void funcionQueHaceAlgo(){
        std::cout<<"Llamo desde clase B"<<std::endl;
    }
};

class A: public B{

    void funcionQueHaceAlgo() override {
        std::cout<<"Llamo desde clase A"<<std::endl;
    }
};
```

La clase A tendrá acceso a los atributos 1 y 2 ya que son públicos, pero no al atributo 0 ya que esta marcado como privado. Ademas, la keyword *override* no es obligatoria pero se considera buena práctica utilizarla. Si queremos evitar que una clase que herede de A tenga la posibilidad de hacerle override al metodo funcionQueHaceAlgo podemos marcarlo utilizando la keyword **final**. Se considera buena práctica marcar al destructor de una clase base como virtual, en el eventual caso de que una clase que herede de ella pueda liberar recursos en su destructor si es que lo necesita.

En C++, polimorfismo significa que la misma llamada a función tiene distintos comportamientos dependiendo del tipo del objeto. La keyword virtual mencionada anteriormente crea una entrada en la tabla virtual, y el método al que se llama es decidido en **tiempo de ejecución**, a esto se lo conoce como *late binding* (Decrementa la performance)

En C++, no existe un modficador *abstract*, si no que se considera clase abstracta a aquella que tenga un metodo *puramente virtual*. Esto se consigue de la siguiente manera:

```
1
2 class A{
3
4 public:
5     A(){}
6     virtual void funcionQueHaceAlgo() = 0;
7 };
```

Se puede simular una interfaz al estilo Java teniendo una clase abstracta sin atributos y con todos sus metodos puramente virtuales.

## 3.3. RAII: Resource Acquisition is Initialization

### 3.3.1. Conceptos Previos

El constructor es un código que se ejecuta al momento de crear un nuevo objeto en C++. Siempre se llama a un constructor, si un objeto no tiene constructor, se le crea uno por default. Lo mismo ocurre con los destructores, a los cuales se los llama cuando se destruye un objeto. Sin embargo, solo puede existir un destructor, y en el caso de que no se cree uno, C++ crea uno por default.

### 3.3.2. RAII

RAII es un patrón en el cual adquirimos recursos en el constructor de un objeto y los liberamos en el destructor del mismo. Los constructores nos reducen la probabilidad de error en nuestro código, ya que nos provee una forma en la que no podamos utilizar un objeto sin inicializar. Ademas, podemos utilizar convenientemente los constructores y destructores de un objeto para utilizar RAII y asi evitar leaks de memoria cuando necesitemos reservar memoria en el heap (Recordar que es el programador quien controla cuando se libera esa memoria)

C++ siempre llama al destructor de un objeto cuando este tiene que ser destruido. Por lo tanto, si utilizamos el stack en conjunto con un destructor bien escrito, evitaremos tener leaks de memoria. Observemos el siguiente codigo C++ que ejemplifica esto:

```
1 struct Vector {
```

```
    int *data;
    int size;

    Vector(int size) {
        this->data = new int[size]();
        this->size = size;
    }

    int get(int pos) {
        return this->data[pos];
    }
    ~Vector() {
        delete[] this->data;
    }
};
```

Notemos que el recurso se pide en el constructor, y se libera en el destructor por lo cual si utilizaramos el stack para instanciar un objeto de clase Vector, no tenemos que explicitamente llamar al delete para liberar la memoria que utiliza el vector. Por lo tanto, el siguiente codigo no tendria leaks:

```
int main(){

    int vectorSize = 5;
    Vector v(vectorSize);

    v.get(0);

    return 0;
}
```

Una vez que salgo del scope, C++ automaticamente llama al destructor de v, lo cual libera el recurso data que tiene dentro.

RAII no es util unicamente para cuando necesitemos memoria del heap, si no que nos sirve para una amplia variedad de recursos (archivos, sockets, locks, etc). Por lo tanto, estos recursos deberían encapsularse en un objeto C++ cuyo constructor adquiera el recurso y lo inicialize y su destructor lo libere. Es un diseño simetrico constructor-destructor.

# 4. Programación Multihilo

La programación multihilo es un tipo de programación en la cual el programa cuenta con multiples hilos de ejecución. Normalmente, un programa tiene un unico *thread* sobre el cual se ejecutan instrucciones.

Sin embargo, podemos crear nuevos hilos de ejecución que realizen otras tareas mientras el hilo principal hace otra. Esto resulta particularmente util cuando hablamos de servidores, dado que el esperar conexiones entrantes es una funcion *bloqueante*, es decir, bloquea al hilo de ejecución hasta que entre una conexión. Por lo tanto, si un servidor no fuese multihilo, estaría constantemente bloqueado y no podría servir su proposito con los clientes que ya fueron aceptados. Es por eso que se utilizan hilos para atender estos clientes mientras que el hilo principal queda a la espera de conexiones, y cuando se recibe una, instancia un hilo que pueda atender las necesidades de ese cliente.

## 4.1. Recursos Compartidos

Los recursos compartidos son aquellos recursos que pueden ser accedidos desde multiples hilos de ejecución, por ejemplo, una variable global que es visible para todo el programa. Esto nos puede traer problemas ya que un hilo puede leer una variable, y en el transcurso de ejecutar una instruccion con esa variable, la misma pudo haber sido modificada por otro hilo de ejecución. Ejemplicando con el siguiente código C++:

```
1  int counter = 0;
2
3  void inc() {
4      ++counter;
5  }
6
7  int main(int argc, char* argv[]) {
8      std::thread t1 {inc};    // Creamos un hilo que ejecute la funcion inc
9      std::thread t2 {inc};    // Creamos otro hilo que ejecute la funcion inc
10
11     t1.join(); t2.join();
12     return counter;
13 }
```

El valor de counter en este caso es indefinido, puede tener valor 1 o valor 2. Esto se debe a que existe la posibilidad de que el segundo thread haya leido el valor 0 antes de que el primer thread haya podido guardar el valor 1, por lo tanto el segundo thread agarraría el 0 y le sumaría uno, guardando el valor definitivo en la variable counter. Esto se llama *Race Condition o Condición de Carrera*

## 4.2. Sincronización: Mutual Exclusion

Para evitar las condiciones de carreras, debemos encontrar la forma que los hilos de ejecución se sincronizen a la hora de ejecutar instrucciones sobre variables mutables. Esto se puede conseguir de varias maneras, pero una de las mas comunes es utilizando lo que se llama un mutex (proveniente de

mutual exclusion) que nos asegura que la ejecución de código sea exclusivamente en un hilo a la vez. En C++ se puede crear un mutex utilizando *std::mutex*. El siguiente código C++ ejemplifica esto:

```
int counter = 0;
std::mutex m;

void inc() {
    m.lock();
    ++counter;
    m.unlock();
}

int main(int argc, char* argv[]) {
    std::thread t1 {inc};   // Creamos un hilo que ejecute la funcion inc
    std::thread t2 {inc};   // Creamos otro hilo que ejecute la funcion inc

    t1.join(); t2.join();
    return counter;
}
```

Ahora con certeza podemos decir que el valor de counter va a ser 2. Cuando el segundo thread intente bloquear el mutex m, no podra ya que ese mutex esta bloqueado por otro hilo, y este segundo hilo no podra adquirir el mutex hasta que el primero lo desbloquee. Podemos pensar al mutex como una variable booleana que es seteada en 1 y 0 de forma atómica que solo podemos adquirir cuando dicha variable este seteada en 0. Hay que recordar de desbloquear el mutex, dado que si no se libera el recurso se llega a una situación de dead-lock (Los otros hilos no pueden continuar su ejecución porque estan bloqueados por un mutex)

Como vimos anteriormente, al ser un recurso que necesita ser liberado explicitamente, podemos utilizar RAII, encapsulando tanto el recurso como su mutex en un objeto. Podemos modificar el código anterior para que respete RAII.

```
class Counter{
    std::mutex mutex;
    int counter = 0;

public:
    void inc(){
        // Objeto RAII que encapsula el bloqueo y desbloqueo del mutex
        std::unique_lock<std::mutex> l(mutex);

        ++counter;
        // Notese que no hace falta liberar el mutex explicitamente
        // de eso se encarga el destructor de unique_lock
```

```
    }

    int getCounter(){
        return counter;
    }
};

Counter c;

void inc() {
   c.inc();
}

int main(int argc, char* argv[]) {
    std::thread t1 {inc};   // Creamos un hilo que ejecute la funcion inc
    std::thread t2 {inc};   // Creamos otro hilo que ejecute la funcion inc

    t1.join(); t2.join();
    return c.getCounter();
}

}
```

## 5. Templates y Programación génerica

Muchas veces llegamos a escenarios en la que tenemos muchas funciones que son prácticamente iguales, solo que se modifica el tipo de dato que recibe o devuelve. Por ejemplo, una función de bubble-sort en C para int y long son prácticamente idénticas salvo el tipo de dato que recibe. Los templates surgen como una solución a estas problematicas dandonos la posibilidad de generar código génerico, evitando de esta manera la repetición constante de código. Podemos observar el uso de templates en el siguiente extacto de código C++:

```
template<class T>
class Array {
    T data[64];

public:
    void set(int p, T v) {
        data[p] = v;
     }

    T get(int p) {
        return data[p];
```

```
12      }
13  };
14
15  int main(){
16      Array<int> intArray;
17      Array<long> longArray;
18
19      return 0;
20  }
```

De esta forma pudimos generalizar el codigo para un array, y en cualquier momento que necesitemos utilizar una instancia de Array le especificamos el tipo de datos que necesitamos, reemplazando el template T por el tipo de dato especificado.

# 6. Smart Pointers

Un smart pointer es un objeto que encapsula un puntero, liberando el recurso al que apunta en su destructor, evitando asi tener que liberar la memoria manualmente. Al ser un objeto que contiene otras cosas ademas del puntero encapsulado, agrega un pequeño overhead de espacio en memoria. C++11 nos provee una serie de distintos smart pointers, en los que se incluyen:

- **Unique_ptr:** Encapsula a un puntero a un objeto en el heap. No es copiable, unicamente movible. Cuando unique_ptr termina su ciclo de vida, destruye el objeto al que apunta.
- **Shared_ptr:** Encapsula un puntero a un objeto en el heap. Posee un contador de referencias, por lo que puede ser copiado, aumentando en uno el contador. Cuando el puntero termina su ciclo de vida o se le asigna otro valor, el contador decrementa. Se libera el objeto apuntado cuando el contador llega a 0.
- **Weak_ptr:** Similar al shared_ptr. Apunta a una direccion gestionada por un shared_ptr, y asignarle un valor a un weak_ptr no aumenta el contador del shared_ptr.Para acceder al contenido, debe ser promocionado a un shared_ptr de forma temporal.

# 7. Modelo cliente-servidor

## 7.1. Servidor simple

Los servidores mas simples son aquellos que unicamente pueden atender a un cliente a la vez. Estos servidores consisten en un simple bucle en el cual se espera una conexión remota, se la procesa cuando llega y vuelve a esperar por la próxima conexión. Es un servidor extremadamente simple que no permite multiples clientes ni hacer cosas en paralelo mientras se espera por una conexión. Se utiliza en servidores RPC o servidores web simples.

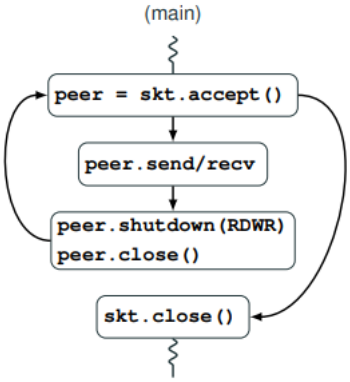



*Figura 0.4.* Gráfico de un servidor simple.

## 7.2. Servidor multi-cliente

Un servidor multi-cliente nos permite atender a varios clientes en simultaneo, haciendo uso de los threads para crear hilos de ejecución para cada cliente que se acepta, mientras que el hilo principal unicamente queda a la espera de una conexión y la procesa creando un thread que se comunique con dicho cliente.

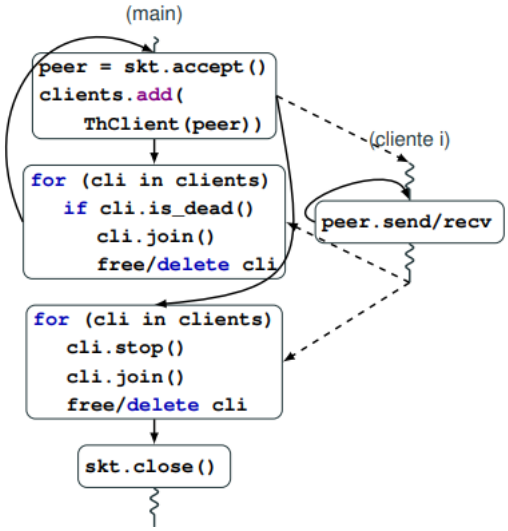



*Figura 0.5.* Gráfico de un servidor multi-cliente.

Hay que ser cuidadosos a la hora de indicarle a un hilo que termine su ejecución, dado que puede estar en el medio de una comunicación con el cliente o puede dejar recursos sin liberar si el hilo es matado abruptamente mediante un llamado a stop/kill.

### 7.3. Modelo cliente-servidor final

Podemos incluso mejorar el modelo anterior, moviendo el aceptador de clientes a un hilo aparte que se encargue de esa tarea, mientras el hilo principal hace otras cosas o bien espera un input del usuario para cerrar ordenadamente el servidor.

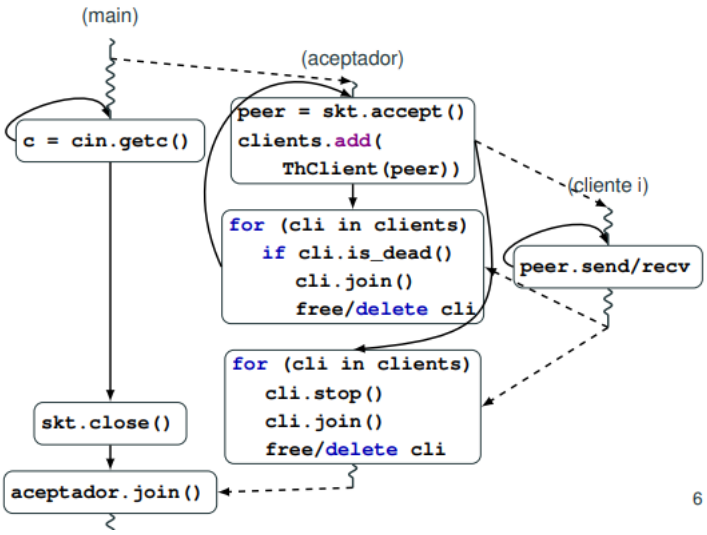


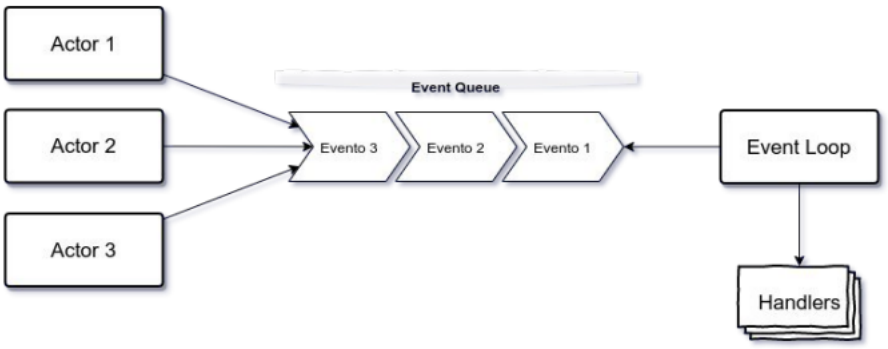

*Figura 0.6.* Gráfico de un servidor multi-cliente completo.

Podemos observar que en este gráfico, el aceptador seguira funcionando mientras el usuario no ingrese por consola cierto input, y que una vez que ese input es ingresado, se cierra el socket aceptador y se espera a que finalize el thread aceptador.

## 8. Programación orientada a Eventos

La programación orientada a eventos se define como un paradigma de programación en el cual toda acción que ejecute el sistema sera en respuesta a los sucesos que acontezcan, denominados eventos. Las acciones a ejecutar son lo que debemos programar y se llaman manejadores, mientras que a la fuente de los eventos se los llama actores. Los eventos pueden venir de varios lugares: Acciones del usuario (click del mouse, botones, etc), Time-based (pasaron 5 segundos desde otro evento X), Generados por otro evento (eventos disparados desde nuestro código) o eventos del entorno (hotplug). Todos estos actores van a querer acceder a recursos compartidos, por lo tanto vamos a tener problemas de concurrencia (siempre y cuando estemos utilizando programación multihilo). Una forma efectiva de solucionar este problema es modelando cada evento como una estructura de dato. Estos eventos van a ser generados por actores e insertados en una cola de eventos, denominada **Event Queue**. Finalmente, hay un thread que se encarga de atender a los eventos que se encuentran en la cola. El loop que se encarga de remover los

eventos de la event queue y atenderlos se llama **Event Loop**. Podemos visualizar el funcionamiento con el siguiente grafico:




Si un handler demanda mucho tiempo, deberia generar un hilo que se encargue de manejarlo, caso contrario el event loop quedaria bloqueado a la espera de que finalize el handler. De esta manera, el handler puede correr sobre otro thread y cuando termina su ejecución, generar un evento para que el thread principal pueda unir el thread del handler.