



UNIVERSIDAD DE BUENOS AIRES  
FACULTAD DE INGENIERÍA  
Año 2015 - 1<sup>er</sup> Cuatrimestre

TALLER DE PROGRAMACIÓN I (75.42)

## Resumen de Taller de programación I

AUTOR:  
Lafroce Matías  
<mlafroce@gmail.com>

- 91378

# Índice

<b>1. Archivos</b>	<b>2</b>
1.1. Primitivas	2
1.1.1. FILE *fopen(const char *path, const char *mode)	2
1.1.2. size_t fread(const void *ptr, size_t size, size_t nmemb, FILE *stream)	2
1.1.3. size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)	2
1.1.4. int fseek(FILE *stream, long offset, int whence)	3
1.1.5. long ftell(FILE *stream)	3
<b>2. Sockets</b>	<b>4</b>
2.1. Primitivas	4
2.1.1. int socket(int domain, int type, int protocol)	4
2.1.2. int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)	4
2.1.3. int listen(int sockfd, int backlog)	4
2.1.4. int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)	5
2.1.5. int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)	5
2.1.6. int shutdown(int sockfd, int how)	5
2.1.7. int close(int sockfd)	5
2.2. Protocolos TCP/UDP	6
2.2.1. ssize_t recv(int sockfd, void *buf, size_t len, int flags)	6
2.2.2. ssize_t send(int sockfd, void *buf, size_t len, int flags)	6
2.3. Ejemplo	7
2.3.1. Cliente	7
2.3.2. Servidor	8
<b>3. Threads POSIX</b>	<b>9</b>
3.1. Primitivas	9
3.1.1. pthread_create	9
3.1.2. pthread_join	9
3.1.3. pthread_exit	10
3.1.4. pthread_cancel	10
3.2. Mutex	10
3.2.1. pthread_mutex_init	10
3.2.2. pthread_mutex_lock	11
3.2.3. pthread_mutex_unlock	11
3.2.4. pthread_mutex_destroy	11
<b>4. Programación en C++</b>	<b>12</b>
4.1. Objetos	12
4.1.1. Encapsulamiento (accesibilidad)	12
4.1.2. Herencia	12
4.1.3. Polimorfismo	12
4.2. Vida de un objeto	12
4.3. Patrón RAII	12
4.4. Templates	12
4.5. Biblioteca STL	12
4.5.1. Ejemplo	13

# 1. Archivos

Un archivo representa un flujo de información. En Unix, todo es representado como un archivo: ficheros regulares, directorios, procesos, dispositivos, etc. Los archivos abiertos por el sistema son almacenados en una tabla del kernel cuyo índice se conoce como *file descriptor*.

Un file descriptor es básicamente una variable del tipo *int*. Existen 3 file descriptors standard para los 3 flujos de datos standard POSIX: *stdin* (*Standard input*), *stdout* (*Standard output*) y *stderr* (*Standard error*), con los valores 0, 1 y 2 respectivamente.

Para la lectura y escritura sobre los archivos, se utilizan las siguientes primitivas, que utilizan los file descriptors para realizar las correspondientes llamadas al sistema.

## 1.1. Primitivas

### 1.1.1. FILE \*fopen(const char \*path, const char \*mode)

Abre un fichero del *file system* con la ruta indicada por *path*, y devuelve un puntero a su file descriptor.

#### Parámetros

- **path**: Ruta al archivo que se desea abrir.
- **mode**: Modo de apertura: lectura, escritura, o agregar al final. Los modos disponibles son:
  - **r**: Solo lectura, desde el principio del archivo.
  - **r+**: Lectura y escritura, desde el principio del archivo.
  - **w**: Solo escritura, si existe, vacía todo el archivo, sino, lo crea.
  - **w+**: Lectura y escritura, vacía todo el archivo, sino, lo crea.
  - **a**: Solo escritura, si existe, se posiciona al final del archivo, sino, lo crea.
  - **a+**: Lectura y escritura, desde el final del archivo, sino existe, lo crea.

N

### 1.1.2. size\_t fread(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)

Escribe en un buffer de datos *nmemb* elementos de tamaño *size*, provenientes del flujo de datos correspondiente. Devuelve la cantidad de items transferidos. La función es bloqueante.

#### Parámetros

- **ptr**: Puntero al buffer sobre el que se quiere escribir los contenidos del flujo de datos.
- **size**: Tamaño en bytes de los datos que se quieren leer.
- **nmemb**: Cantidad de elementos que se quieren leer.
- **stream**: Puntero al file descriptor del que se quieren leer los datos.

N

### 1.1.3. size\_t fwrite(const void \*ptr, size\_t size, size\_t nmemb, FILE \*stream)

Escribe desde un buffer de datos *nmemb* elementos de tamaño *size*, hacia el flujo de datos correspondiente.

*Para más información, leer la documentación de fread*

**1.1.4. int fseek(FILE \*stream, long offset, int whence)**

Cambia el indicador de posición asociado al archivo.

**Parámetros**

- **stream**: Puntero al file descriptor del archivo.
- **offset**: Distancia respecto al punto de referencia elegido.
- **whence**: Punto de referencia desde el cuál desplazarse una distancia *offset*. Los valores posibles son SEEK\_SET, SEEK\_CUR, y SEEK\_END para marcarlo como relativo al principio, fin, o posición actual del archivo.

**1.1.5. long ftell(FILE \*stream)**

Devuelve la posición actual sobre el archivo.

## 2. Sockets

### 2.1. Primitivas

Un socket es un flujo de datos que se utilizar para comunicar procesos entre si. Los sockets tienen un dominio sobre el cual se comunican, como por ejemplo, internet IPv4 e IPv6, o Unix para comunicar procesos dentro del mismo sistema. Además, poseen un tipo de conexión, por ejemplo, si es punto a punto o no, si los paquetes son de longitud fija o variable, etc, y poseen un protocolo sobre el que se realiza la transmisión de datos.

A continuación se describen las primitivas más utilizadas para el manejo de sockets POSIX en C. Nótese que los sockets se manejan igual que los archivos, y que como tales, las acciones sobre ellos se realizan mediante su *file descriptor*.

#### 2.1.1. `int socket(int domain, int type, int protocol)`

Crea un nuevo socket y devuelve su número de *socket descriptor* (o -1 si hay un error).

##### Parámetros

- **domain:** Define si la familia de protocolos de la conexión. Algunos valores usados son: PF\_LOCAL (comunicación local), PF\_INET (IPv4), PF\_INET6 (IPv6).
- **type:** Define el tipo de conexión. Algunos de los valores valores más usados son SOCK\_STREAM y SOCK\_DGRAM para protocolos TCP y UDP respectivamente.
- **protocol:** Define el protocolo a utilizar, se lo puede dejar en 0 para que se elija el apropiado según el tipo de conexión.



#### 2.1.2. `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)`

Asocia al socket a una dirección. Devuelve 0 en éxito o -1 en caso de error.

##### Parámetros

- **sockfd:** Número del socket descriptor al que se le quiere asociar la dirección.
- **addr:** Dirección a la que se quiere asociar el socket.

La estructura que se utiliza generalmente es la siguiente:

```
1 struct sockaddr_in {
2     sa_family_t sin_family; /* address family: AF_INET */
3     in_port_t sin_port;     /* puerto en formato de red (pasarle htons(port))
4     /*
5     in_addr sin_addr;       /*IP a la que se quiere asociar */
6 };
```

- **addrlen:** Tamaño de la estructura: sizeof(struct sockaddr\_in); Si el socket se utilizará como cliente, no es necesario bindearlo antes de hacer un connect.



#### 2.1.3. `int listen(int sockfd, int backlog)`

Marca el socket como pasivo, es decir, que escuche conexiones entrantes. El socket debe ser del tipo SOCK\_STREAM o SOCK\_SEQPACKET. Devuelve 0 en éxito o -1 en caso de error.

**Parámetros**

- **sockfd**: Número del socket descriptor a pasivar.
- **backlog**: Cantidad máxima de conexiones a encolar

**2.1.4. int accept(int sockfd, struct sockaddr \*addr, socklen\_t \*addrlen)**

Genera un socket a partir de la primera de las conexiones encoladas en el socket correspondiente a sockfd. Devuelve el número del file descriptor de la nueva conexión o -1 en caso de error.

**Parámetros**

- **sockfd**: Socket escuchador. Tiene que estar previamente pasivado con *listen* (Por lo que también es prerequisite haber llamado a *bind*)
- **addr**: Puntero a la estructura en la que se escribe la dirección del cliente que se conecta. Se le puede pasar 0 para ignorar esta información.
- **addrlen**: Tamaño de la estructura sockaddr. Si addr es 0, se recomienda que addrlen sea 0 también.

**2.1.5. int connect(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen)**

Conecta el socket a la dirección pasada por addr. Devuelve 0 en éxito o -1 en caso de error.

**Parámetros**

- **sockfd**: Número del socket descriptor que se quiere conectar como “cliente”.
- **addr**: Dirección al que se quiere conectar el socket.
- **addrlen**: Tamaño de la estructura sockaddr.

**2.1.6. int shutdown(int sockfd, int how)**

Desactiva toda o parte de la conexión.

**Parámetros**

- **sockfd**: Número del socket descriptor a desactivar.
- **how**: Especifica si se quiere desactivar la escritura, lectura o ambas

**2.1.7. int close(int sockfd)**

Cierra el socket, y libera los recursos correspondientes.

**Parámetros**

- **sockfd**: Número del socket descriptor a cerrar.



## 2.2. Protocolos TCP/UDP

A continuación se describen los protocolos más utilizados para comunicaciones sobre redes: *TCP* y *UDP*, y algunas de sus primitivas.

El protocolo *TCP* se utiliza en conexiones de streaming punto a punto, donde se utiliza un socket para cada punta de cada conexión. Esto implica que si un servidor está conectado a varios clientes, debe tener un socket distinto para cada uno de ellos. Este protocolo además tiene como característico que los paquetes siempre llegan a destino y en el orden en que fueron mandados.

El protocolo *UDP*, por otro lado, se utiliza en conexiones basadas en datagramas, donde las conexiones no son punto a puntos, sino que el servidor realiza un *broadcast* de un mensaje y los clientes lo reciben como pueden. Este protocolo no asegura que todos los paquetes lleguen del servidor al cliente, ni asegura que se mantenga el orden al recibirlos. Es un protocolo más liviano que *TCP* y se utiliza cuando lo importante no es la fidelidad de los datos, sino la rápida transmisión de los mismos.

Dado que el socket es un archivo con un file descriptor asociado, podemos realizar lectura y escritura como lo haríamos con cualquier otro archivo. Sin embargo, para aprovechar los protocolos ya existentes, se utilizarán las funciones *recv* y *send*. Cabe destacar que las funciones *read* y *write* del standard de Unix funcionan sobre el file descriptor del socket, pero estarían traspasando el protocolo de la conexión, ya que realizan escritura y lectura a bajo nivel.

### 2.2.1. `ssize_t recv(int sockfd, void *buf, size_t len, int flags)`

Lee una cadena de bytes del socket y devuelve la cantidad de bytes que leyó, o -1 en caso de error.

#### Parámetros

- **sockfd**: Número del socket descriptor a leer.
- **buf**: Buffer sobre el que se va a escribir lo recibido.
- **len**: La cantidad de bytes que se espera leer (no necesariamente se llegan a leer todos).



### 2.2.2. `ssize_t send(int sockfd, void *buf, size_t len, int flags)`

Escribe una cadena de bytes del socket y devuelve la cantidad de bytes que envió, o -1 en caso de error.

#### Parámetros

- **sockfd**: Número del socket descriptor a escribir.
- **buf**: Buffer sobre el que se va a leer el mensaje a enviar.
- **len**: La cantidad de bytes que se espera escribir (no necesariamente se llegan a enviar todos).
- **flags**: Es recomendable usar `MSG_NOSIGNAL` para evitar que se emitan señales `SIGPIPE` cuando la conexión está rota.



## 2.3. Ejemplo

### 2.3.1. Cliente

```

1  #include <sys/socket.h>
2  #include <arpa/inet.h>
3  #include <unistd.h>
4  #include <cstring> //Necesario para el memset
5  #include <stdio.h>
6
7  #define BACKLOG 20
8  #define MSG_SIZE 30
9
10 int main(int, char**){
11     printf("Iniciando el cliente en la direccion 127.0.0.1:8080\n");
12     int socketFd = socket(PF_INET, SOCK_STREAM, 0); //Creo el socket
13
14     char serverAddress[] = "127.0.0.1";
15
16     struct sockaddr_in address; //Armo los datos para conectarse
17     address.sin_family = AF_INET;
18     address.sin_port = htons(8080); //Seteo el puerto, en formato de red
19     address.sin_addr.s_addr = inet_addr(serverAddress);
20     memset(address.sin_zero, 0, sizeof(address.sin_zero));
21
22     int connected = connect(socketFd, (struct sockaddr *) &address,
23                             sizeof(struct sockaddr_in)); //Me conecto a la direccion.
24     if (connected != 0){
25         printf("Falla al conectar\n");
26         return connected;
27     }
28
29     char message[MSG_SIZE];
30     int bytesRecv = 0;
31
32     printf ("Recibiendo el mensaje...\n");
33     //Le envio 30 bytes al cliente (un numero arbitrario
34     while (bytesRecv < MSG_SIZE && bytesRecv != -1){
35         // Agrego offsets si es que no se envia todo el mensaje
36         bytesRecv += recv(socketFd, message + bytesRecv, MSG_SIZE - bytesRecv,
37                             0);
38         printf("Recibido %d bytes\n", bytesRecv);
39     }
40     message[29] = 0; //Cierro string
41
42     printf ("Recibo el mensaje %s\n", message);
43
44     shutdown(socketFd, 0); //Dejo de transmitir datos
45
46     close(socketFd); //Cierro file descriptor
47
48     printf("Adios, vuelvas pronto\n");
49
50     return 0;
51 }
```



### 2.3.2. Servidor

```
1 #include <sys/socket.h>
2 #include <arpa/inet.h>
3 #include <unistd.h>
4 #include <cstring> //Necesario para el memset
5 #include <stdio.h>
6
7 #define BACKLOG 20
8
9 int main(int, char**){
10     printf("Iniciando el servidor\n");
11     int socketFd = socket(PF_INET, SOCK_STREAM, 0); //Creo el socket
12
13     struct sockaddr_in address; //Armo los datos para bindearse
14     address.sin_family = AF_INET;
15     address.sin_port = htons(8080); //Seteo el puerto, en formato de red
16     address.sin_addr.s_addr = INADDR_ANY;
17     memset(address.sin_zero, 0, sizeof(address.sin_zero));
18     //Bindeo al puerto 8080
19     bind(socketFd, (struct sockaddr*)&address, sizeof(struct sockaddr_in));
20
21     listen(socketFd, BACKLOG); //Pasivo el socket
22
23     printf("Esperando conexión...\n");
24     //Acepto una conexión e ignoro la información de la misma
25     int clientFd = accept(socketFd, 0, 0);
26     printf("Conexión aceptada\n");
27
28     int bytesSent = 0;
29
30     printf("Enviando datos\n");
31     //Le envío 30 bytes al cliente (un número arbitrario)
32     while (bytesSent < 30 && bytesSent != -1){
33         char message[30] = "Este es un mensaje prueba :D\n";
34         // Agrego offsets si es que no se envía todo el mensaje
35         bytesSent = send(clientFd, message + bytesSent, 30 - bytesSent,
36             MSG_NOSIGNAL);
37         printf("Recibido %d bytes\n", bytesSent);
38     }
39     printf("Datos enviados\n");
40
41     shutdown(socketFd, 0); //Dejo de transmitir datos
42     shutdown(clientFd, 0);
43
44     close(socketFd);
45     close(clientFd);
46
47     printf("Adios, vuelvas pronto\n");
48
49     return 0;
50 }
```

### 3. Threads POSIX

Una forma de trabajar con concurrencia es mediante el uso de hilos de ejecución, o también conocidos como *Threads*. Un thread, también llamado "proceso de peso liviano", se diferencia de un proceso en el nivel de autonomía que poseen.

Cada thread posee su propio:

- **Stack pointer:** Cada thread tiene sus propias variables.
- **Registros**
- **Propiedades de scheduling:** Cada thread tiene su prioridad o política de ejecución.
- **Datos específicos del thread**

Pero los threads provenientes de un mismo proceso comparten:

- **Espacio de memoria:** Esto implica que comparten el *code segment*, *data segment* y *heap*.
- **Recursos del proceso padre:** Algunos recursos como los file descriptors se comparten a través de todos los procesos.

Dado que varios threads pueden leer y escribir una misma dirección de memoria, debe existir una sincronización explícita en el código a ejecutar.

La creación de threads es bastante sencilla. En esencia, se utiliza una primitiva que recibe, entre otros parámetros, un puntero a la función a ejecutar en paralelo, y un puntero que se utilizará como parámetro en la función del thread. A continuación se detallan las primitivas utilizadas para la creación y destrucción correcta de threads.

#### 3.1. Primitivas

##### 3.1.1. pthread\_create

**Firma:** `int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void*(*start_routine)(void *), void *arg)`

Crea un nuevo hilo, almacena el id en *\*thread* y devuelve 0 en caso de éxito. Este nuevo hilo ejecuta una función pasada por parámetro, y una vez finalizado, debe hacerse el join para liberar sus recursos.

###### Parámetros

- **thread:** estructura sobre la que se almacenaran los datos del hilo creado (por ejemplo, el id).
- **attr:** atributos para la creación del hilo, como pueden ser el tamaño del stack, si se le puede hacer join, etc. Setear este atributo en NULL carga los valores default
- **start\_routine:** puntero a una función que tenga la firma *void\* miFuncion(void\* arg)*, para ser ejecutada en el nuevo hilo.
- **arg:** parámetro a pasar a la función puesta en el parámetro start\_routine.



##### 3.1.2. pthread\_join

**Firma:** `int pthread_join(pthread_t thread, void **retval)`

Espera a que el hilo finalice y libera sus recursos.

###### Parámetros

- **thread:** El hilo a finalizar. Si el hilo ya terminó de operar, join termina automáticamente.
- **retval:** Puntero al lugar donde se copia el contenido devuelto por start\_routine. Si es NULL, no se copia nada.



### 3.1.3. pthread\_exit

**Firma:** void pthread\_exit(void \*retval)

Finaliza el hilo que lo llamó. Esta función se llama implícitamente cuando termina la *start\_routine*.

En C++ pthread\_exit no garantiza la llamada a destructores, a diferencia de llamar a *return*, que garantiza la restauración del stack y destrucción de variables. Además, *return* en la función *main* implica una llamada a *exit()*, terminando la aplicación. *pthread\_exit*, por otro lado, solo terminaría la ejecución del hilo main, y la aplicación correría hasta que todos los hilos terminen o se llame a *exit()*, *abort()*, etc.

### 3.1.4. pthread\_cancel

**Firma:** void pthread\_cancel(pthread\_t thread)

Mata el el thread pasado por parámetro. ¡Pum! Le pega un tiro en la cabeza, lo tira al río y corre. Esta función debe evitarse ya que puede tener comportamientos extraños al combinarse con manejo de excepciones en c++, entre otros problemas.

## 3.2. Mutex

Uno de los principales problemas que trae la concurrencia es la consistencia de datos. Consideremos la siguiente función:

```
1 void Foo::increment(){
2     this->value++;
3 }
```

Uno pensaría que no tiene nada de malo, y que la función es segura de usar de forma concurrente. Sin embargo, si la reescribimos de la siguiente forma:

```
1 void Foo::increment(){
2     this->value = this->value + 1;
3 }
```

O más explícitamente:

```
1 void Foo::increment(){
2     int aux = this->value;
3     this->value = aux + 1;
4 }
```

Se puede ver un poco más claro los problemas de correr esa función en múltiples hilos a la vez.

Imaginemos que la variable entera *value* se inicializa con el valor 0. 2 hilos ejecutan la función *increment*, en simultáneo. El resultado esperado es que, al finalizar ambos hilos, *value* termine con el valor 2. Sin embargo, dado que el orden en que se ejecutan los hilos es incierto, podría ocurrir que la variable *aux* tome el mismo valor en ambos hilos, y que en la siguiente línea, ambos hilos almacenen el mismo valor en la variable compartida *value*. Estos problemas se deben a que no podemos saber si las instrucciones se ejecutan en un único paso o en varios, es decir, si son atómicas o no.

Para evitar estos problemas, asumimos que todas las instrucciones No son atómicas y utilizamos restricciones de concurrencia, como la exclusión mutua, o *mutex*. Los mutexes se utilizan para controlar el acceso a una porción de código de la aplicación, permitiendo el acceso de un hilo por vez.

### 3.2.1. pthread\_mutex\_init

**Firma:** int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*mutexattr)

Inicializa el objeto con la dirección *mutex*, y los atributos en la dirección *mutexattr*.

**Parámetros**

- **mutex**: El mutex a inicializar.
- **mutexattr**: Atributos del mutex, como por ejemplo, si es un mutex normal, recursivo, o posee chequeo de errores. Si se coloca NULL, se toman los parámetros default.

**3.2.2. pthread\_mutex\_lock**

**Firma** `int pthread_mutex_lock(pthread_mutex_t *mutex)` Traba un mutex. Cualquier otro thread que ejecute un *lock* sobre un mutex ya trabado queda esperando a que se destrabe el mutex para continuar con la ejecución.

**3.2.3. pthread\_mutex\_unlock**

**Firma** `int pthread_mutex_unlock(pthread_mutex_t *mutex)` Destraba un mutex. Si un thread trata de destrabar un thread que él mismo no trabó, el comportamiento es indefinido.

**3.2.4. pthread\_mutex\_destroy**

**Firma** `int pthread_mutex_destroy(pthread_mutex_t *mutex)` Destruye el mutex. Si se quiere volver a utilizar, se debe llamar a `pthread_mutex_init`. Se puede destruir un mutex destrabado de forma segura, pero destruir uno que está trabado trae comportamiento indefinido.

## 4. Programación en C++

C++ no solo es compatible con las bibliotecas de C, sino que ofrece el paradigma de programación orientada a objetos, y herramientas de metaprogramación como templates.

### 4.1. Objetos

La programación orientada a objetos llega a C++ de la mano de las *clases*. Las clases definen como se crean los objetos, que son estructuras que poseen propiedades atributos y comportamiento (métodos).

#### 4.1.1. Encapsulamiento (accesibilidad)

C++ refuerza el encapsulamiento mediante niveles de acceso a sus atributos y métodos. Los 3 niveles que existen son:

- **Public:** El atributo/método es visible para todos, es decir, puedo acceder al mismo tanto desde adentro como fuera de la clase.
- **Protected:** El atributo/método es visible para la clase y sus derivadas. Solo puedo acceder al atributo/método desde la clase que lo posee o desde las clases que derivan de ella (utilizando la herencia correspondiente).
- **Private:** El atributo/método es únicamente visible para la clase que lo declara.

#### 4.1.2. Herencia

TODO: Tabla de accesibilidad según tipo de herencia

#### 4.1.3. Polimorfismo

TODO: Ejemplo de polimorfismo, explicar métodos virtuales, interfaces.

## 4.2. Vida de un objeto

TODO: Explicar como funcionan los constructores, destructor y stack.

## 4.3. Patrón RAII

El acrónimo *RAII* proviene del inglés *Resource Acquisition Is Initialization*, y hace referencia a un diseño de código en el que los recursos utilizados por un objeto son reservados en el momento de su creación (por el constructor), y son liberados en el momento de su destrucción (por el destructor).

El uso del patrón RAII ayuda a escribir un código más sólido y conciso, debido a que se suele hacer uso de variables cuya vida depende del scope en el que fue declarado. Esto trae algunas ventajas:

- **Encapsulamiento:** La adquisición de recursos (memoria dinámica, archivos, etc), se define una sola vez en el constructor de la clase y la liberación de los mismos en el destructor.
- **Exception-safety:** Los recursos que se encuentran abiertos en el momento que se lanza una excepción son liberados a medida que se llaman los destructores del *scope* en el que se lanzó la misma. De esta manera, si una excepción no es atrapada, se van llamando los destructores y liberando recursos a medida que se va restaurando el *stack*.

### 4.3.1. Ejemplo

En el siguiente ejemplo se muestra como usar un constructor y destructor para mantener control de un recurso (En este caso, memoria dinámica).

```

1  #include <iostream>
2
3  /**
4   * Una clase que representa a un buffer, mal implementado
5   */
6  class WrongBuffer{
7  public:
8      WrongBuffer(): buffer(0){}
9      /*
10       * Crea un buffer de tamaño size
11       */
12     void startBuffer(int size){
13         if (this->buffer){
14             delete this->buffer;
15         }
16         this->buffer = new char[size];
17     }
18     /*
19     * Limpia el buffer
20     */
21     void deleteBuffer(){
22         delete[] this->buffer;
23     }
24     /**
25     * Devuelve el buffer almacenado
26     */
27     char* getBuffer(){
28         return this->buffer;
29     }
30 private:
31     char* buffer;
32 };
33
34 /**
35 * Implementacion mas prolija del buffer
36 */
37 class Buffer{
38 public:
39     /*
40     * Constructor
41     */
42     Buffer(int size){
43         this->buffer = new char[size];
44     }
45     /*
46     * Destructor, limpia el buffer
47     */
48     ~Buffer(){
49         delete[] this->buffer;
50     }
51     /**
52     * Devuelve el buffer almacenado
53     */
54     char* getBuffer(){
55         return this->buffer;
56     }
57 private:
58     char* buffer;
59 };
60
61 /**
62 * Ejemplo de codigo sin patron RAI
63 */
64 void badExample1(){
65     WrongBuffer buff;
66     buff.startBuffer(42);
67     /**
68     * ERROR! El programador debe recordar liberar la memoria, de lo contrario
69     * la aplicacion tendra leaks.
70     * En esta seccion de codigo no pueden lanzarse excepciones, ya que

```

```
71     * se perderia el puntero al buffer sin haberlo liberado.
72     */
73     buff.deleteBuffer();
74 }
75
76 /**
77  * Ejemplo de codigo que sigue el patron RAII
78  */
79 void example(){
80     Buffer buffer(42);
81     /**
82      * Notese que hacer Buffer *buffer = new Buffer(42) tambien estaria
83      * rompiendo el patron RAII.
84      */
85     char* buffContent = buffer.getBuffer();
86     /**
87      * Al finalizar la funcion, se llama al destructor de buffer. El destructor
88      * se llama automaticamente, no es necesario que el programador lo haga
89      * explicitamente, por lo que la memoria reservada en la instancia buffer
90      * es liberada siempre.
91      */
92 }
```

## 4.4. Templates

TODO

## 4.5. Biblioteca STL

TODO