Indice

Antes de empezar

Antes de comenzar vamos a repasar algunos consejos que creemos que te serán útiles para completar la práctica con éxito y para adquirir habilidades como buen desarrollador.

Aprovecha las facilidades de C++ todo lo posible

En C++ se utilizan conceptos como clases, objetos, flujos o sobrecarga de operadores. Además implementa una extensa librería estándar que es de gran ayuda para los programadores. Utiliza las abstracciones de C++ todo lo posible para desarrollar la lógica del programa.

Obviamente nuestro objetivo es aprender cómo funciona el sistema operativo. Así que utilizaremos necesariamente la interfaz de programación del sistema operativo, que está escrita en C. Pero no tienes que hacer toda la práctica en C. Simplemente tendrás que crear clases y métodos que envuelvan esa interfaz de bajo nivel para utilizarla más fácilmente con las abstracciones de C++.

Por ejemplo, la función de la librería del sistema que devuelve el valor de una variable de entorno tiene este prototipo:

```
char* getenv(const char* name);
```

Pero no hace falta que te líes a trabajar con cadenas `char*` de C. Simplemente crea tu propia versión que utilice las mucho más cómodas std::string de C++:

```
std::string getenv(const std::string& name)
{
	char* value = getenv(name.c_str());
	if (value) {
		return std::string(value);
	}
	else {
		return std::string();
	}
}
```

Y utilízala donde lo necesites en tu programa.

Programa de forma defensiva

La mayor parte de las funciones de la librería del sistema pueden fallar por diversos motivos. Es decir, comprueba siempre estos errores y trátalos adecuadamente, por ejemplo, mostrando un error y haciendo que termine el programa correctamente.

Por ejemplo, la función getenv() anterior devuelve NULL si la variable no existe. Si usas esa función, pero olvidas comprobar si ha devuelto NULL, confiando ciegamente en que la variable a la que quieres acceder siempre existe, te arriesgas a que tu programa termine con una violación de segmento sin motivo aparente. Por eso en la versión adaptada a C++ comprobamos si vale NULL y si es así

devolvemos una cadena vacía. Obviamente eso es mucho más seguro y profesional que no hacer ninguna comprobación.

Deja todo ordenado al salir

Básicamente hay dos motivos para terminar un programa:

1. Ya hizo su trabajo, todo se ha ejecutado correctamente y ya no hace falta que se ejecute más. Recuerda que en ese caso el programa debe terminar con el estado 0 o EXIT_SUCCESS.
2. U ocurrió algún error irrecuperable.

Cuando ha ocurrido algún error la norma **es mostrar el mensaje correspondiente por la salida de error y terminar con un estado diferente de 0 o EXIT_SUCCESS**. El estándar ofrece EXIT_FAILURE como valor a retornar en main() para indicar un error:

```
return EXIT_FAILURE;
```

Pero **es conveniente es que el valor devuelto sea distinto de 0 y diferente para cada tipo de error**. Por ejemplo, 4 cuando no se pudo abrir un archivo, 5 cuando no se pudo ejecutar un programa, 10 cuando si no hay suficiente memoria, etc.

```
#define EXIT_OPEN_INPUT_ERROR 10

// ...

return EXIT_OPEN_INPUT_ERROR;
```

Eso permite manejar los errores desde scripts de la shell u otros programas, que así pueden saber la causa por la que el programa que han ejecutado ha fallado.

En C++ un programa termina con una sentencia return en la función main() o mediante la función de la librería estándar std::exit(). No es conveniente usar directamente la llamada al sistema exit(). Esa llamada sabe terminar cualquier proceso en ejecución, pero no sabe nada de las particularidades de finalizar un programa en C++. De hecho **en C++ ni siquiera es buena idea llamar directamente a std::exit()** —ni a otras funciones similares como std::abort()— **especialmente en funciones y métodos del programa diferentes a main()**, porque no se preocupa de llamar a todos los destructores de todos objetos para liberar los recursos de forma segura antes de terminar el programa.

La forma correcta de terminar un programa es dejar que la ejecución vuelva hasta main() y termine allí saliendo de la función con la sentencia 'return'. ¿Cómo en caso de error se salta rápidamente hasta el main() en un lenguaje como C++? Usando excepciones, como veremos más adelante.

Evita el mal uso de los espacios de nombres

Un mal hábito, según la comunidad de desarrolladores de C++, de programación en C++ que lamentablemente sigue siendo muy común es programar usando así el espacio de nombres "std":

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char* argv[])
{
      cout << "¡Hola, mundo!\n";
      return EXIT_SUCCESS;
}
```

Los espacios de nombre están para evitar la colisión de nombres entre clases y funciones. Cuanto más complejo es nuestro programa, más probable es que estas colisiones ocurran, así que lo mejor es usarlos donde sea necesario:

```
#include <cstdlib>
#include <iostream>

int main(int argc, char* argv[])
{
      std::cout << "¡Hola, mundo!\n";
      return EXIT_SUCCESS;
}
```

Y solo si al hacerlo la legibilidad del código se ve afectada, puedes evitarlo en casos concretos:

```
#include <cstdlib>
#include <iostream>

int main(int argc, char* argv[])
{
      using std::cout;

      cout << "¡Hola, mundo!\n";
      return EXIT_SUCCESS;
}
```

Netcp básico

El objetivo de la práctica es desarrollar una herramienta que nos permite manipular archivos entre distintos ordenadores, permitiendo copiar y cambiar el nombre a ficheros entre distintos ordenadores. Es un sistema básico de transferencia de archivos en red.

Comunicaciones en red

Para hacer un sistema de transferencia de ficheros necesitamos que los procesos de Netcp lanzados por los distintos usuarios se puedan comunicar:

- Como en la actualidad la mayor parte de los ordenadores están conectados a través de redes TCP/IP, el mecanismo de paso de mensajes que escojamos debe soportar dicha tecnología para realizar el envío de datos entre los equipos. **En la mayor parte de los sistemas operativos modernos eso significa que tenemos que usar sockets**.
- Todo sistema operativo soporta al menos dos tipos de sockets. Los primeros son los **stream sockets**: orientados a conexión, con garantías de que los datos llegan, llegan en el orden en el que se envían y basados en el protocolo TCP. Los segundos son los **datagram sockets**: no orientados a conexión, sin garantías de que los datos lleguen o lo hagan en el orden en el que se enviaron y basados en el protocolo UDP. Debido a sus características los stream sockets son usados ampliamente para desarrollar todo tipo de servicios —Web (HTTP), correo electrónico (SMTP), transferencia de archivos (FTP) o acceso remoto (SSH)— pero **utilizaremos datagram sockets basados en UDP porque son un poco más sencillos de utilizar y porque en una red de área local (LAN) es prácticamente imposible que se pierdan datos**.

Sockets

Como vimos en las clases de teoría y en los ejemplos de los apuntes, **un socket se crea con la llamada al sistema socket()** y, como ocurre con casi cualquier recurso que le podemos pedir al sistema operativo que nos cree, **devuelve un descriptor de archivo con el que identificar al socket en cuestión en las futuras operaciones** que queramos hacer con él.

```
#include <sys/types.h>
#include <sys/socket.h>

...

int fd = socket(domain, type, protocol)
```

Esto significa que los sockets abiertos en un proceso padre son heredados por los procesos hijos al hacer fork(), como ocurre con los archivos abiertos.

Tal y como indica el manual de socket, el significado de los parámetros de la llamada al sistema es el siguiente:

- **domain**: dominio o familia de la dirección. Indica el tipo de tecnología de comunicación que queremos que utilice nuestro socket. Valores posibles:
    - AF_INET, si queremos usar TCP/IP
    - AF_INET6, si queremos usar TCP/IPv6
    - AF_UNIX, si queremos usar un tipo de socket local a la máquina, similar a las tuberías.

  Nosotros usaremos AF_INET porque estamos interesados en la tecnología TCP/IP utilizada en Internet.
- **type**. Indica el tipo de socket que nos interesa según los requisitos de la aplicación. Valores posibles:
    - SOCK_STREAM. Si domain es AF_INET, indica que el socket es para comunicaciones orientadas a conexión con TCP
    - SOCK_DGRAM. Si domain es AF_INET, indica que queremos UDP.
    - SOCK_RAW. Indica acceso directo a los paquetes IP.

  Tal y como hemos comentado, nosotros usaremos SOCK_DGRAM.
- **protocol**. Indica el protocolo específico que queremos que sea utilizado internamente por el socket para el envío de los datos. Esto es útil con algunas familias (domain) que soportan varios protocolos para cada tipo (type) de socket. En nuestro caso la familia IP (AF_INET) con sockets tipo datagram (SOCK_DGRAM) sólo admite UDP como protocolo, por lo que este argumento debe ser 0.

Con todo esto, crear nuestro primer socket sería algo así

```
#include <sys/socket.h>

...

int fd = socket(AF_INET, SOCK_DGRAM, 0);
if (fd < 0) {
      std::cerr << "no se pudo crear el socket: " <<
            std::strerror(errno) << '\n';
      return 3;  // Error. Terminar con un valor diferente y > 0
}
```

Observa que comprobamos si hemos tenido éxito antes de continua. Existen muchos motivos por los que puede fallar la llamada al sistema socket().

Asignar una dirección al socket

Un socket es una abstracción que representa un extremo de un canal de comunicación entre dos procesos en una red de ordenadores. Cada proceso puede utilizar su socket para recibir mensajes de otros procesos o para enviarlos a través de dicho canal de comunicación, sin tener que entrar en los detalles de cómo se usa el protocolo de red escogido ni cómo se gestionan los dispositivos de red involucrados en la comunicación. Estos aspectos son responsabilidad del sistema operativo y del hardware de la red pero no del programador de las aplicaciones.

Para poder mandar un mensaje al socket de otro proceso este debe tener un nombre, es decir, una dirección única en la red. Y lo mismo ocurre para que nuestro socket pueda recibir mensajes. En

---

la tecnología TCP/IP (AF_INET) la dirección de un socket se indica por la dirección IP de la máquina y un número de puerto entre 1 y 65535.

Una dirección se asigna al socket que acabamos de crear mediante la llamada al sistema bind().

```
#include <sys/socket.h>

...

int bind(int fd, const sockaddr* addr, socklen_t addrlen);
```

Obviamente el primer argumento 'fd' de la función es el descriptor de archivo del socket previamente creado con la llamada al sistema socket() en el paso anterior.

La interfaz de socket fue diseñada con una interfaz genérica que debe dar cabida a todas las tecnologías de comunicación existentes y futuras. Por eso bind() no acepta una dirección IP y un número de puerto como parámetros. En su lugar recibe un puntero a la estructura sockaddr genérica que pueda dar cabida a cualquier tipo de dirección de cualquier tecnología existente.

Según la familia de protocolos con la que queramos trabajar, se usa una versión de sockaddr adecuada para las direcciones de red de esa familia. En nuestro caso, como queremos trabajar con la familia AF_INET (TCP/IP) tendremos que utilizar la estructura sockaddr_in, declarada en <netinet/ip.h>, para crear las direcciones:

```
struct sockaddr_in {
    sa_family_t    sin_family;  // siempre debe valer AF_INET
    in_port_t      sin_port;    // número de puerto en
                                // orden de bytes de la red.
    struct in_addr sin_addr;    // dirección de IP
};

// Dirección IP
struct in_addr {
    uint32_t       s_addr;      // dirección en orden de bytes
                                // de la red.
};
```

Algunos detalles que debemos tener en cuenta para rellenar estas estructuras:

- **s_addr puede tener alguno de los siguientes valores:**
  - Cualquier dirección IP, en formato entero de 32 bits. **Se puede utilizar la función inet_aton() para convertir cadenas con direcciones de la forma "192.168.200.5" en estos enteros de 32 bits**.
  - INADDR_LOOPBACK, si queremos hacer referencia a la dirección 127.0.0.1 de la interfaz de red loopback de la máquina local
  - o INADDR_ANY (0.0.0.0), si queremos indicar algo así como que da igual, usa las direcciones que tiene actualmente mi ordenador.
- **sin_port es el número de puerto del socket entre 1 y 65525:**

    - **Si indicamos 0, es que queremos que el sistema operativo asigne uno cualquiera** que esté disponible.
- Si tenemos 's_addr' o 'sin_port' en formato entero —ojo que con el resultado de inet_aton() no hace falta porque ya lo hace bien por nosotros— tenemos que asegurarnos que están guardados en el orden de bytes de la red —que en Internet es big-endian— antes de copiarlos al campo correspondiente en la estructura sockaddr_in. Las funciones **htonl()** y **htons()** pueden usarse para convertir enteros de 32 bits **(l)** —como las direcciones IP que van en 's_addr'— o enteros de 16 bits **(s)** —como los números de puerto— del orden de bytes usado por nuestra CPU **(h)** al que espera la interfaz de comunicaciones **(n)**. De ahí el nombre de **h**to**n***().

Veamos un ejemplo para asignar dirección a nuestro socket de forma que escuche en todas las direcciones IP de nuestra máquina en un puerto escogido por el sistema operativo:

```
#include <sys/socket.h>
#include <netinet/in.h>

...

// Dirección del socket local
sockaddr_in local_address{};    // Así se inicializa a 0, como se
recomienda
local_address.sin_family = AF_INET;   // Pues el socket es de dominio
AF_INET
local_address.sin_addr.s_addr = htonl(INADDR_ANY);
local_address.sin_port = htons(0);

// Asignar la dirección al socket local
int result = bind(fd, reinterpret_cast<const sockaddr*>(&local_address),
                  sizeof(local_address))
if (result < 0) {
     std::cerr << "falló bind: " << z << '\n';
     return 5;  // Error. Terminar con un valor diferente y > 0
}
```

Algunas cosas destacables sobre el siguiente ejemplo son:

- Nuevamente se comprueban los posibles errores de la llamada al sistema. bind() puede fallar por múltiples motivos fuera de nuestro control.
- `reinterpret_cast<const sockaddr*>()` es necesario porque bind() espera un puntero al formato genérico de direcciones sockaddr*, pero 'local_address' es 'sockaddr_in' porque estamos interesados en trabajar con la familia TCP/IP de protocolos de Internet.

Respecto a como crear correctamente las direcciones:

- Si estamos creando una dirección sockaddr_in para recibir datos:
    - En 'sin_port' debemos indicar el puerto donde queremos escuchar.
- Si por el contrario construimos la dirección sockaddr_in para asignarla al socket en nuestro extremo de la conexión y enviar datos:

- En 'sin_port' podemos indicar 0, si no hay motivo por el que tengamos que usar un puerto concreto. El sistema operativo nos asignará uno libre.

[TAREA] Implementa make_ip_address()

Vas a tener que crear unas cuantas direcciones sockaddr_in para desarrollar este programa, así que te recomendamos que implementes la función:

```
sockaddr_in make_ip_address(int port, const std::string& ip_address =
                            std::string());
```

que reciba una dirección IP como cadena y un puerto como entero y devuelva una estructura sockaddr_in perfectamente inicializada con los valores correspondientes. Si la dirección IP en ip_address es una cadena vacía `std::string()`, la dirección IP almacenada en sockaddr_in deberá ser INADDR_ANY.

[TAREA] Implementa la clase Socket

C++ es un lenguaje orientado a objetos y los descriptores de archivo de los sockets junto con las llamadas al sistema que los manejan se prestan a ser encapsuladas en una clase. Por eso te recomendamos que implementes la clase:

```
class Socket
{
  public:
     Socket(const sockaddr_in& address);
     ~Socket();

  private:
     int fd_;
};
```

El constructor se encargaría de:

1. Crear un socket, mediante la llamada al sistema socket(), cuyo descriptor se almacena en el atributo privado "fd_" para que lo puedan utilizar otros métodos de la clase.
2. Vincular dicho socket a la dirección de Internet especificada mediante el argumento 'address' del constructor, utilizando la llamada al sistema bind().

Obviamente, en el destructor se debe cerrar el descriptor de archivo del socket para liberar sus recursos. Para eso se utiliza la llamada al sistema close().

Este diseño de la clase Socket donde los recursos necesarios se reservan en el constructor y se liberan el destructor responde a un patrón de diseño característico de C++ denominado RAII. Es vital usarlo siempre que creemos clases que reservan algún recurso para garantizar que sea liberado de forma segura. De hecho, el motivo por el que se recomienda terminar una aplicación de C++ desde main() —retornando de la función con la sentencia 'return'—es porque así estamos

seguros de que se han destruido correctamente todos los objetos creados durante la vida de la aplicación. No tenemos esa misma garantía si invocamos std::exit() desde cualquier otra función del programa, pues así no serían llamados ni los destructores de los objetos creados dinámicamente con 'new' ni los destructores de los objetos creados localmente por las funciones invocadas para llegar a aquella desde la que se llama a std::exit().

Enviar un mensaje

Enviar un mensaje al socket de otro proceso desde el nuestro se hace con la llamada al sistema sendto():

```
#include <sys/socket.h>

...

int sendto(int fd, const void* buffer, size_t length, int flags,
           const sockaddr* dest_addr, socklen_t dest_len)
```

que obviamente espera en 'dest_addr' la dirección del destinatario de nuestro mensaje. Recuerda que en nuestro caso debemos indicar ahí un puntero a una estructura sockaddr_in, debido a que estamos usando la familia AF_INET (TCP/IP). Igualmente, 'dest_len' es el tamaño de la estructura a la que apunta el argumento 'dest_addr'. Es decir, debe ser 'sizeof(sockaddr_in)'.

Como en el caso de bind(), el parámetro 'fd' es el descriptor de archivo del socket en nuestro extremo de la conexión.

Respecto al resto de los argumentos:

- **flags**, debe valer 0 puesto que no es útil para sockets UDP.
- **buffer**, es la dirección de la memoria donde comienza el mensaje que queremos enviar.
- **length**, es la longitud en bytes de dicho mensaje.

Por lo tanto, así es como podríamos mandar un mensaje a otro proceso:

```
#include <array>
#include <sys/socket.h>
#include <netinet/in.h>

...

// Dirección del socket remoto
sockaddr_in remote_address{};   // Porque se recomienda inicializar a 0
remote_address.sin_family = AF_INET;
remote_address.sin_port = htons(puerto);
inet_aton("X.X.X.X", &remote_address.sin_addr);

// Estructura de los mensajes
```

```
struct Message {

     ...                                // Otros campos del mensaje

     std::array<char, 1024> text;       // Igual que "char text[1024]"
                                        // pero recomendado así en C++

     ...                                // Otros campos del mensaje

};

// Enviar un mensaje "¡Hola, mundo!" al socket remoto
std::string message_text("¡Hola, mundo!");

Message message;
message_text.copy(message.text.data(), message.text.size() - 1, 0)

...                                 // Inicializar otros campos de Message


int result = sendto(fd, &message, sizeof(message), 0,
     reinterpret_cast<const sockaddr*>(&remote_address),
     sizeof(remote_address));
if (result < 0) {
     std::cerr << "falló sendto: " << std::strerror(errno) << '\n';
     return 6;  // Error. Terminar con un valor diferente y > 0
}
```

A la hora de escoger la estructura del mensaje hay un par de cosas que debes tener en cuenta:

- **La estructura debe ser un POD** —o Plain Old Data— en C++. Que una estructura sea POD significa que es compatible con los tipos usados en C y que puede ser manipulada por funciones de C —recuerda que las llamadas al sistema están diseñadas para ser usadas en C, no C++—. **Un POD es un tipo escalar —int, float, char, punteros, etc.— un array de un tipo POD o una clase, estructura o unión que sólo campos de tipo POD, no tiene destructor, constructor, operador de asignación, campos privados o protegidos, clases base ni funciones virtuales**. Por lo tanto podemos usar campos de tipo ‘char[]’ —tal y com hemos hecho— pero no std::string.
- **No debe contener punteros ni campos con estructuras, clases o uniones que contiene punteros**. El motivo de esto es obvio. Un puntero es una dirección en la memoria y al enviarlo mandamos dicha dirección no el contenido al que apunta. En un proceso diferente la dirección de memoria a la que apunta el puntero enviado puede contener cualquier cosa.

[TAREA] Implementa Socket::send_to()

Ahora que sabes enviar datos a otro proceso te recomendamos que añadas a la clase Socket un método para enviar mensajes:

```
class Socket
{
  public:
     Socket(const sockaddr_in& address);
     ~Socket();

     void send_to(const Message& message, const sockaddr_in& address);

  private:
     int fd_;
};
```

Recibir un mensaje

Recibir un mensaje enviado desde otro proceso se hace con la llamada al sistema recvfrom():

```
#include <sys/socket.h>

...

int recvfrom(int fd, void* buffer, size_t length, int flags,
             sockaddr* src_addr, socklen_t* src_len)
```

Como en el caso de sendto(), el parámetro 'fd' es el descriptor de archivo del socket en nuestro extremo de la conexión y 'flags' debe valer 0 porque no es útil con sockets UDP.

Respecto al resto de los argumentos:

- **buffer**, es la dirección de la memoria donde queremos que el sistema operativo deposite el mensaje.
- **length**, es la longitud en bytes que hemos reservado para guardar el mensaje. La llamada al sistema recvfrom() nunca copiará más de 'length' bytes en el bloque de memoria indicado por 'buffer'.
    - En un socket UDP, **si el mensaje es más grande que el tamaño indicado en length, los bytes que no quepan serán descartados**.
    - La llamada **recvfrom() devuelve como valor de retorno el número de bytes recibidos y copiados en 'buffer'**.
- **src_addr**, a la vuelta de la llamada al sistema contendrá la dirección del remitente del mensaje. Si esa información no es relevante, puede indicarse "nullptr" en el puntero al invocar la función.
- **src_len**, debe valer "nulltptr" si 'src_addr' es "nullptr". En caso contrario debe ser un puntero a una variable que a la vuelta contendrá el tamaño de la estructura apuntada por 'src_addr'. Es

---

decir, que en nuestro caso debería valer siempre sizeof(sockaddr_in) a la vuelta de la llamada al sistema.

Por lo tanto, así es como podríamos recibir un mensaje de otro proceso:

```
#include <array>
#include <sys/socket.h>
#include <netinet/in.h>

...

// Dirección del socket remoto
sockaddr_in remote_address{};   // Porque se recomienda inicializar a 0
socklen_t src_len = sizeof(remote_address);

// Estructura de los mensajes
struct Message {

     ...                              // Otros campos del mensaje

     std::array<char, 1024> text;     // Igual que "char text[1024]"
                                      // pero recomendado así en C++

     ...                              // Otros campos del mensaje
};

// Recibir un mensaje del socket remoto
Message message;

int result = recvfrom(fd, &message, sizeof(message), 0,
     reinterpret_cast<sockaddr*>(&remote_address), &src_len);
if (result < 0) {
     std::cerr << "falló recvfrom: " << std::strerror(errno) << '\n';
     return 8;
}

// --
// Vamos a mostrar el mensaje recibido en la terminal

// Primero convertimos la dirección IP como entero de 32 bits
// en una cadena de texto.
char* remote_ip = inet_ntoa(remote_address.sin_addr);

// Recuperamos el puerto del remitente en el orden adecuado para nuestra
CPU
int remote_port = ntohs(remote_address.sin_port);

// Imprimimos el mensaje y la dirección del remitente
```

```
std::cout << "El sistema " << remote_ip << ":" << remote_port <<
     " envió el mensaje '" << message.text.data() << "'\n";
```

Antes de continuar mira la última línea del programa:

```
std::cout << "El sistema " << remote_ip << ":" << remote_port <<
     " envió el mensaje '" << message.text.data() << "'\n";
```

Al escribirla estamos presuponiendo que '`message.text`' es un array que almacena una cadena de caracteres terminada en \0 para marcar dónde termina la cadena ¿Pero qué ocurriría si no fuera así? ¿qué pasaría si alguien, en lugar de usar el programa que hemos escrito para mandar mensajes, creara otro para mandar un mensaje sin un sólo \0 en su 'contenido'?

Como imaginarás, seguramente nuestro programa volcaría todo '`message.text`' y parte de la memoria del programa, quizás hasta provocar una violación de segmento, porque std::cout no dejaría de imprimir bytes en la memoria hasta encontrar algún \0 en algún sitio. Por eso **es importante no suponer nada ni al comunicarnos con otros programas ni al leer de archivos.** Nada nos garantiza que la otra parte está haciendo bien su parte del trabajo.

En el ejemplo que nos ocupa podemos mejorar la seguridad de nuestro programa de forma muy sencilla asegurándonos que al menos hay un 0 a final del buffer

```
message.text[1023] = '\0';
std::cout << "El sistema " << remote_ip << ":" << remote_port <<
     " envió el mensaje '" << message.text.data() << "'\n";
```

[TAREA] Implementa Socket::receive_from()

Obviamente, ahora que sabes recibir datos de otro proceso, te recomendamos que añadas a la clase Socket un método para recibir mensajes:

```
class Socket
{
  public:
     Socket(const sockaddr_in& address);
     ~Socket();

     void send_to(const Message& message, const sockaddr_in& address);
     void receive_from(Message& message, sockaddr_in& address);

  private:
     int fd_;
};
```

Leer datos de un archivo

El primer paso para enviar el contenido de un archivo es abrirlo. Para ello se utilizan la familia de llamadas al sistema de manipulación de archivos consistentes en:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open( const char* pathname, int flags )
int open( const char* pathname, int flags, mode_t mode )
int close( int fd )
```

Algunos de los **flags** disponibles para la apertura de ficheros:

O_RDONLY	0000	El fichero se abre sólo para lectura.
O_WRONLY	0001	El fichero se abre sólo para escritura.
O_RDWR	0002	El fichero se abre para lectura y escritura.
O_TMPFILE	020000 000	Se crea un archivo temporal sin nombre que solo va a ser usado por el proceso actual
O_CREAT	0100	El fichero deberá ser creado si no existía previamente.
O_EXCL	0200	Provoca que la llamada a open falle si se especifica la opción O_CREAT y el fichero ya existía.
O_TRUNC	1000	Fija el tamaño del fichero a cero bytes.
O_APPEND	2000	El apuntador de escritura se situa al final del fichero, se escribirán al final los nuevos datos.

Y el argumento **mode** está relacionado con los permisos del archivo en el sistema de archivos al crear un archivo nuevo:

S_IROTH	0000	Activar el bit de lectura para todo los usuarios.
S_IWOTH	0001	Activar el bit de escritura para todo los usuarios.
S_IXOTH	0002	Activar el bit de ejecución para todo los usuarios.
S_IRGRP	0010	Activar el bit de lectura para todo los usuarios pertenecientes al grupo.
S_IRGRP	0020	Activar el bit de escritura para todo los usuarios pertenecientes al grupo.

S_IRGRP	0040	Activar el bit de ejecución para todo los usuarios pertenecientes al grupo.
S_IRUSR	0100	Activar el bit de lectura para el propietario.
S_IWUSR	0200	Activar el bit de escritura para el propietario.
S_IXUSR	0400	Activar el bit de ejecución para el propietario.
S_ISVTX	1000	Activa el “sticky bit” en el fichero.
S_ISGID	2000	Activa el bit de SUID en el fichero.
S_ISUID	4000	Activa el bit de SGID en el fichero.
S_IRWXU	S_IRUSR + S_IWUSR + S_IXUSR	Activar el bit de lectura, escritura y ejecución para el propietario.
S_IRWXG	S_IRGRP + S_IWGRP + S_IXGRP	Activar el bit de lectura, escritura y ejecución para todo los usuarios pertenecientes al grupo.
S_IRWXO	S_IROTH + S_IWOTH + S_IXOTH	Activar el bit de lectura, escritura y ejecución para todo los usuarios.

Para leer y escribir el contenido de un archivo abierto se utilizan las siguientes funciones

```
ssize_t read(int fd, void* buf, size_t count)
ssize_t write(int fd, void* buf, size_t count)
```

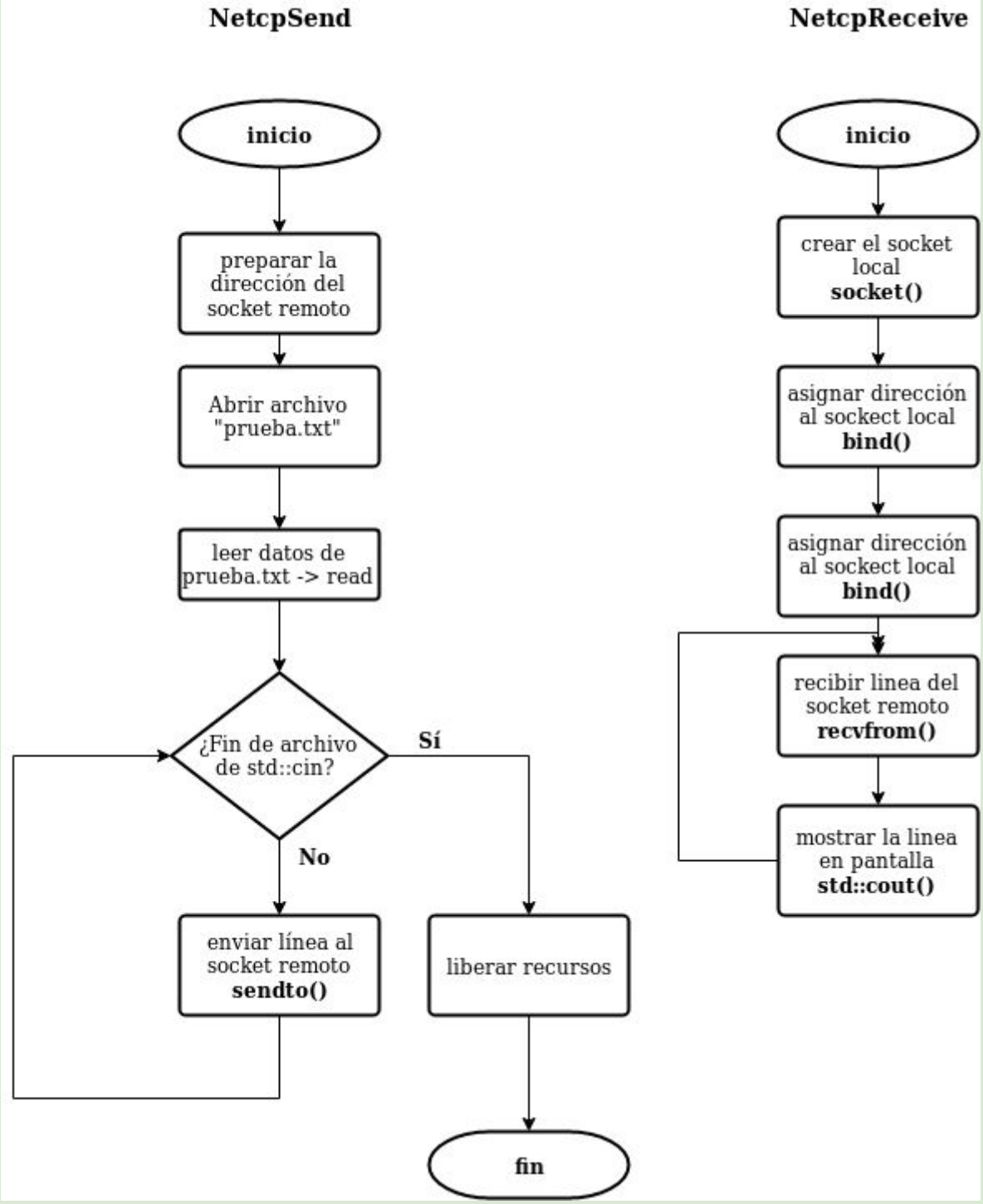
Para leer el contenido de un archivo hay que especificar una zona de memoria válida —a la que apunta ‘buf’— la cantidad de bytes a leer (count) y el descriptor de un archivo abierto (fd).
Para escribir el contenido de un archivo, la zona de memoria apuntada por ‘buf’ debe contener los datos que vamos a escribir y ‘count’ indica cuántos bytes de esa región queremos escribir.

Ambas funciones nos devolverán la cantidad de bytes leídos o escritos —que nunca será mayor que ‘count’—. En el caso de las lecturas, es complicado que podamos leer todo el archivo de una sola vez, por lo que es necesario llamar repetidamente a read(), generalmente en un bucle. Cuando se llega al final del archivo, read() nos lo indicará devolviendo que ha leído 0 bytes, por lo que esa puede ser la condición de salida del bucle.

[TAREA] Hora de empezar con Netcp

Con todo lo que sabes ahora ha llegado el momento de que desarrolles tu primer prototipo de programa Netcp. En este caso estará dividido en 2 códigos independientes. NetcpSend consistirá en

abrir un fichero de prueba “prueba.txt” almacenado en el mismo directorio que Netcp, leer su contenido y enviarlo al receptor. Por otro lado NetcpReceive esperará a recibir el fichero y mostrará su contenido por pantalla. El diagrama de flujo que describe su comportamiento esperado es aproximadamente el siguiente:



Cuando te pongas manos a la obra, hay unas cuantas cosas que debes tener en cuenta:

- **Para probar necesitarás dos versiones de tu programa**. El programa NetcpRecevie que escucha en un puerto de tu máquina local y el programa NetcpSend que manda el mensaje con el contenido del archivo al puerto donde escucha la otra.

- **Valora crear una clase File, similar que la clase Socket pero para gestionar el descriptor de archivo**. ¿Por qué? Para facilitar la gestión del recurso "archivo abierto" en caso de error o cuando ya no lo necesitamos más, igual que Socket gestiona el socket abierto. Por eso puede ser buena idea tener una clase File con un constructor que abra el archivo con open(), un destructor que cierra con close() y métodos para leer y escribir el contenido.

Manejo de errores

Supongo que mirando los ejemplos anteriores te habrás dado cuenta de cómo hemos comprobado el valor devuelto por socket(), bind(), sendto() y recvfrom() para detectar condiciones de error.

```
#include <iostream>
#include <cerrno>           // para errno
#include <cstring>          // para std::strerror()
#include <sys/socket.h>     // para socket()

...

int fd = socket(domain, type, protocol);
if (fd < 0) {
      std::cerr << "no se pudo crear el socket: " <<
            std::strerror(errno) << '\n';
      return 3;  // Error. Termina siempre con un valor > 0
}
```

Y lo mismo hay que hacer con open(), read() y write.

Detección de errores

Al programar es conveniente que siempre comprobemos las condiciones de error de cualquier función o llamada al sistema para, si fuera necesario, terminar nuestro programa de forma ordenada. Para saber cómo nos informa una función de cualquier error es aconsejable echarle un vistazo a su documentación.

Si lo hacemos descubriremos que la mayor parte de las llamadas al sistema lo hacen siempre de la misma manera:

- **Las funciones o llamadas al sistema devuelven un valor menor de 0 en caso de error.**
- **El código que nos permite conocer la causa del error lo dejan almacenado en la variable global 'errno'**.
    - Esta es una variable entera y se declara en el archivo de cabecera <cerrno>.
    - En el mismo archivo se declara una macro para cada posible valor de 'errno'. Sin embargo no todos los errores son posibles en todas las funciones. En la documentación de cada llamada se indican qué errores son posibles. Por ejemplo, mira la página de socket().

- **Con std::strerror() se puede obtener una cadena descriptiva para cualquier código de error**. Esta función se declara en <cstring>.

Mensajes de error

En realidad en nuestros ejemplos no estamos generando mensajes de error siguiendo la convención más común. Para comprobarlo, hagamos una prueba:

```
$ ls /nodir
ls: no se puede acceder a /nodir: No existe el archivo o el directorio
```

o dos:

```
$ /bin/ls /nodir
/bin/ls: no se puede acceder a /nodir: No existe el archivo o el directorio
```

A ahora veamos de dónde sale cada parte del error de 'ls':

- **"No existe el archivo o el directorio" es lo que devuelve std::strerror() cuando errno es EEXIST**. Es decir, cuando una llamada al sistema descubre que el archivo o directorio indicado no existe.
- **"no se puede acceder a /nodir" es la parte del mensaje con la que contribuye el propio comando 'ls'**. A fin de cuentas con lo que devuelve std::strerror() estamos seguros que el problema está en un archivo o directorio que no existe pero ¿qué archivo o directorio?. El comando debe aportar esta información para que sepamos lo que estaba intentando hacer.
- **'ls' es el nombre del programa**. A bote pronto parece una tontería indicar que el error es de "ls" cuando está claro que el usuario sabe qué comando ha ejecutado. Pero cuando un el error lo imprime un programa invocado desde un script de BASH o desde otro programa, ayuda mucho saber quién se está quejando y por qué.

En resumen, un mensaje de error adecuado podría ser así:

```
std::cerr << "netcp: no se pudo crear el socket: " <<
      std::strerror(errno) << '\n';
```

Propagación de errores

Por norma los errores deben manejarse de forma lo más local posible. Es decir, lo más cerca posible al punto donde son detectados. Por ejemplo, si el constructor de nuestra clase Socket recibe un error al usar la llamada al sistema socket(), lo mejor es que el mismo evite llamar a bind() y que muestre un mensaje de error.

Pero incluso así, lo suyo es que quien intentó crear un Socket() sea informado de que la operación no pudo realizarse, por sí debe hacer algo como consecuencia. Es decir, los errores debe propagarse hacia la función invocadora.

Recordemos que algunos errores deben hacer que el programa termine pero los programas en C++ deben ser terminados volviendo de main(). Luego **no parece buena idea ejecutar exit() allí donde ocurre un error**, si no notificar de lo que ha pasado hacía arriba —propagando el error de funciones invocadas a invocadoras— hasta que la ejecución vuelva a main() y se pueda terminar el programa.

Estilo C

En C la solución tradicional es la misma que hemos visto usar a las llamadas al sistema. Es decir, **devolver un valor de retorno concreto para indicar el error —por ejemplo un false, 0 o un valor negativo— y dejar que se use "errno" allí donde sea necesario** si se quiere conocer el motivo del error. Si una función falla, porque alguna de las que ella llama devolvió un error, lo que debe hacer es propagarlo hacia arriba devolviendo a su vez el valor que indica que hubo un error.

Si necesitamos crear nuestro propios tipos de errores porque con los soportados con "errno" no es suficiente, la mejor solución es crear nuestra propia variable estilo-errno donde guardar el último código de error ocurrido, de entre la serie de errores que nos inventemos.

Excepciones de C++

Por fortuna C++ nos proporciona excepciones, que es un mecanismo muchísimo más potente que el tradicional de C.

Una excepción no es más que un objeto que contiene información sobre un error ocurrido durante la ejecución de nuestro programa. Cuando ocurre un error, se crea el objeto excepción adecuado con la información necesaria sobre el error y se "lanza" para notificar al resto del programa lo que ha ocurrido. Algunos de estas excepciones son lanzadas por las clases y funciones de las librerías que utilizamos, mientras que otros los podemos lanzar nosotros desde nuestro propio código.

Lanzar un objeto excepción desde una función es muy similar a devolverlo con "return", **solo que para lanzar excepciones se utiliza la palabra clave "throw"**. Veamos un ejemplo:

```
#include <stdexcept>
#include <iostream>

class MyClass
{
public:

	void MyFunc(char c)
	{
		if(c > 128) {
			throw std::invalid_argument("Argumento demasiado grande.");
		}
	}
};

int main()
{
```

```
	MyClass object;
	try
	{
		object.MyFunc(256);   // Causa una excepción
		std::cout << "En caso de excepción esto no se verá nunca.\n";
	}
	catch(std::invalid_argument& e)
	{
		std::cerr << e.what() << '\n';
		return 1;  // Error. Terminar con un valor diferente y > 0
	}

	return 0;  // Exito.
}
```

La palabra clave "throw" se comporta como un "return". La diferencia entre ambas es que:

- **return <valor>**, hace que la función termine y la ejecución vuelva a la función invocadora, retornando <valor>.
- **throw <valor>**, también retorna a la función invocadora llevando <valor> pero de ahí salta directamente —sin ejecutar ninguna sentencia más— a la función invocadora de esa y de ahí a la invocadora de esa y así sucesivamente hasta salir de main(); si nadie para el proceso, como veremos más adelante. **Si la excepción sale de main() sin haber sido detenida antes, el programa termina inesperadamente llamando a std::terminate()**.

Como ya sabemos, al salir de una función con “return” **todas las variables y objetos locales son destruidos y sus recursos son liberados**. Lo mismo ocurre con “throw” para cada una de las funciones de las que retorna en su recorrido hacia main(). Eso asegura que al lanzarse una excepción los recursos de los objetos que no se van a usar más son liberados correctamente, si nos hemos preocupado de implementar destructores correspondientes.

Si una parte del código necesita hacer algo especial cuando una función invocada falla con una excepción, lo que debe hacer es interceptarla. Ese “algo especial” puede ser cualquier cosa que se nos ocurra. Incluido mostrar al usuario un mensaje de error adecuado. Una vez manejado el error se puede continuar con la ejecución del programa, a partir del punto de intercepción, o relanzar la excepción para notificar el problema a funciones superiores.

Las excepciones se interceptan con un manejador de excepciones:

1. **Consisten en un bloque “try” donde se incluye el código que puede generar las excepciones que se quiere interceptar**. Tengamos en cuenta que el bloque interpretará tanto las excepciones generadas directamente por el código incluido en el bloque, como las generadas por las funciones invocadas por este o funciones invocadas por funciones invocadas por el código y así sucesivamente. En resumen, **el recorrido ascendente de una excepción lanzada con “throw” se detendrá al encontrar un bloque “try”**.
2. Después del bloque **try** se incluyen varios manejadores mediante la palabra clave **catch** sequida de una referencia o una variable entre paréntesis. Los bloques **catch** están asociados al bloque **try** anterior, de tal forma que si se lanza una excepción dentro de **try** la ejecución saltará al

código del bloque **catch** cuyo objeto tenga un tipo que encaje con el del objeto usado en el **throw** para desencadenar la excepción.

Por ejemplo, si intentamos reservar una cadena de caracteres std::string de 10GB:

```
#include <iostream>
#include <string>

int main(int argc, char* argv[])
{
	try {
		std::string text(10000000000, 'x');   // Causa excepción
		std::cout << "Esta línea no se va a ejecutar nunca\n";
	}
	catch(std::bad_alloc& e) {
		std::cout << "Memoria insuficiente\n";
		return 1;  // Error. Terminar con un valor diferente y > 0
	}
	catch (...) {
		std::cout << "Error desconocido\n";
		return 99; // Error. Terminar con un valor diferente y > 0
	}

	return 0;
}
```

y no tenemos memoria suficiente, **el constructor de std::string fallará con una excepción std::bad_alloc**, que será **capturada por el primer "catch" para mostrar el mensaje "Memoria insuficiente"**:

- Se escogerá el primer "catch" porque tiene el tipo que mejor encaja con el objeto de la excepción, que también es de tipo std::bad_alloc.
- Antes de ejecutar el código en el bloque "catch" se le asigna a "e" el objeto de la excepción —especificado en el "throw" que lanzó la excepción, dentro del constructor de std::string— que contiene información sobre el error.
  En un bloque "catch" se puede usar tanto "std::bad_alloc e" como "std::bad_alloc& e" pero hemos preferido usar una referencia para evitar que se genere una copia del objeto al asignarlo a "e"; de forma similar a como ocurre al pasar objetos a funciones.
- Una vez ejecutado el bloque "catch" la ejecución continúa en "return 0", sin ejecutar ningún otro bloque "catch", como ocurre en el caso de los bloques "if-else".

Finalmente **el último bloque "catch (...)" es opcional y se usa como bloque por defecto para capturar cualquier excepción** que no sea capturada por bloques anteriores.

```
	catch (...) {
		std::cout << "Error desconocido\n";
	}
```

```
      return 0;
}
```

Si no existe ningún bloque "catch" adecuado para la excepción, esta continuará su recorrido hacia arriba hasta encontrar una función con un manejador donde encaje o hasta salir de main() y terminar el programa llamando a std::terminate().

Errores del sistema y excepciones

Como hemos comentado, podemos crear nuestras propias clases de excepción si tenemos necesidades particulares. Sin embargo, el estándar de C++ predefine unas pocas excepciones que podemos utilizar según nuestras necesidades y que se declaran en el archivo de cabecera <exception>. Una de ellas es std::system_error, que se utilizan para notificar errores del sistema.

Como mínimo necesita que se le indique un código de error del sistema —el valor de 'errno'— y opcionalmente una cadena descriptiva del error:

```
throw std::system_error(errno, std::system_category());
throw std::system_error(errno, std::system_category(),
      "no se pudo crear el socket");
```

Es decir, que el código para crear un socket podría quedar tal que así:

```
#include <sys/socket.h>
#include <system_error>
#include <cerrno>

...

int fd = socket(AF_INET, SOCK_DGRAM,0);
if (fd < 0)
      throw std::system_error(errno, std::system_category(),
            "no se pudo crear el socket");
```

No poder crear un socket no tiene porque ser un error crítico para una aplicación cualquiera. Pero si eso pasa en una herramienta como nuestro talk ¿qué otra cosa podemos hacer? Por lo tanto podemos interceptar la excepción en main(), mostrar un mensaje de error descriptivo y terminar el programa.

```
#include <iostream>
#include <system_error>

...

int main(int argc, char* argv[])
{
      try {

            ...
```

```
            Socket socket("127.0.0.1", 0)

            ...
      }
      catch(std::bad_alloc& e) {
            std::cerr << "mytalk" << ": memoria insuficiente\n";
            return 1;
      }
      catch(std::system_error& e) {
            std::cerr << "mitalk" << ": " << e.what() << '\n';
            return 2;
      }
      catch (...) {
            std::cout << "Error desconocido\n";
            return 99;
      }

      return 0;
}
```

std::system_error tiene dos métodos interesantes con información sobre el error:

- **code(), que devuelve el código de error especificado al crear el objeto**. En nuestro ejemplo el valor de 'errno'.
- **what(), que devuelve un mensaje descriptivo del error**. std::system_error interamente compone este mensaje con la cadena "no se pudo crear el socket", especificada al crear el objeto, seguida del texto descriptivo devuelto por std::strerror(errno).

Patrón protected_main()

Un patrón muy común para hacer el código más legible es separar el manejo de excepciones del resto del código de la función main(). Básicamente consiste en extraer el código del bloque "try" para ponerlo en una función llamada protected_main().

```
int protected_main(int argc, char* argv[])
{
      ...

      Socket socket("127.0.0.1", 0)

      ...

      return 0;
}
```

Y luego llamar a esa función desde el bloque "try" de main():

```
int main(int argc, char* argv[])
{
	try {

		return protected_main(argc, argv)
	}
	catch(std::bad_alloc& e) {
		std::cerr << "mytalk" << ": memoria insuficiente\n";
		return 1;
	}
	catch(std::system_error& e) {
		std::cerr << "mitalk" << ": " << e.what() << '\n';
		return 2;
	}
	catch (...) {
		std::cout << "Error desconocido\n";
		return 99;
	}
}
```

Esto nos permite programar libremente en protected_main() las tareas y funcionalidades de nuestra aplicación, sabiendo que cualquier excepción será interceptada y tratada adecuadamente en main() antes de salir del programa.

Maneja bien los errores

Revisa tu programa.

- Asegúrate de comprobar todas condiciones de error posibles al solicitar servicios al sistema —en la clase Socket y en código que gestiona el archivo abierto— y lanza la excepción correspondiente. Si has optado por crear la clase File, una de las ventajas es que la comprobación de errores la podrás hacer en sus métodos, simplificando el resto del programa.
- Intercepta las excepciones en main() e implementa protected_main().
- En caso de error, recuerda mostrar un mensaje informativo al usuario por la salida de error y termina correctamente el programa retornado por main().

Es importante que tengas presente esto en las futuras mejoras que hagas al programa a lo largo de la práctica. Gran parte del código de cualquier aplicación real es para el manejo de los errores del programa.