Taller de Programación I
Preparación para el examen integrador

Última modificación: Septiembre 2020

Colaboradores
Mauro Parafati - mparafati@fi.uba.ar
Nicolas Aguerre - naguerre@fi.uba.ar
Santiago Klein - sklein@fi.uba.ar
Taiel Colavecchia - tcolavecchia@fi.uba.ar
Yuhong Huang - yhuang@fi.uba.ar

Índice

1. Preguntas teóricas

Recopilación de preguntas teóricas únicas evaluadas en los finales históricos de la materia.

1.1. Compilación

1. **Describa el proceso de transformación de código fuente a un ejecutable. Precise las etapas, las tareas desarrolladas y los tipos de error generados en cada una de ellas.**

   Las etapas por las que pasa el código fuente son:

   a) *Pre-compilacion*: reemplazos de directivas del compilador, expansión de macros, defines, includes de headers, etc.
   Tipos de error: errores de uso de directivas del preprocesador (sintácticos, macros no definidas, includes de archivos inexistentes, dependencias circulares).

   b) *Compilación*: parseo, verificación de sintaxis, traducción y ensamblado del código preprocesado, resultando en un archivo '.o'. Tipos de error: errores de sintaxis, semánticos, de tipado, etc.

   c) *Linkedición*: se realiza el armado del ejecutable partiendo de los archivos objeto que componen al programa. Tipos de error: falta de definiciones o definiciones múltiples de funciones y variables declaradas en algún módulo como externas (undefined references).

2. **Explique la diferencia entre las etapas de compilacion y enlazado (linking). Escriba un breve ejemplo de codigo con errores para cada una de ellas indicandolos de forma clara.**

   La principal diferencia está en que durante el proceso de compilación se toma como entrada código de lenguaje de alto nivel y como salida lenguaje de máquina (archivo objeto). El enlazado consiste en tomar uno (o varios) archivos objeto y convertirlos en un único ejecutable.

   Error de compilación:

   ```
   int main(argc, argv[]) {}
   ```

   Error de enlazado:

   ```
   #include <stdio.h>

   int sumar(int a, int b);

   int main(int argc, const char* argv[]) {
       printf("%i\n", sumar(1, 3));
       return 0;
   }
   ```

3. **¿En qué consiste el proceso de enlazado (o linking)?**

   El proceso de enlazado consiste en tomar modulos de programa ya compilados de forma independiente, y generar a partir de estos un unico archivo ejecutable. El enlazado se produce al reemplazar las referencias a simbolos indefinidos por sus direcciones correctas dentro del programa (que pueden pertenecer a cualquiera de los modulos que se estan enlazando).

4. **Describa el uso de las siguientes instrucciones de precompilación: `#define`, `#undef`, `#include`, `#ifdef`, `#ifndef`, `#if`, `#elif` y `#else`.**

   - La directiva `#define`: se le asigna un nombre identificable por el preprocesador a un fragmento de codigo. Dicho identificador puede ser utilizado mas adelante en el código para insertar copias del fragmento de código que identifica.

- La directiva `#undef`: se remueve la definición del identificador con el que se llama. En consecuencia, los subsiguientes llamados a dicho identificador serán ignorados por el preprocesador. Asimismo, puede aplicarse para identificadores que no estén definidos.
- La directiva `#include`: se sigue la ruta indicada para agregar código de un archivo a otro.
- La directiva `#if`: se evalua una expresión lógica formada por símbolos que el precompilador entiende, y continúa procesando el bloque de código que corresponda según la condición se cumpla o no. El bloque generado puede ser seguido de `#elif`, `#else`, y debe terminar con `#endif`.
- La directiva `#elif`: se evalua una subsecuente expresión lógica luego de un `#if` (o `#elif`) previo, y funciona de igual manera.
- La directiva `#else`: el precompilador agrega la sección de codigo a continuación de esta directiva si la/s condicion/es anterior/es es/fueron falsas.
- La directiva `#ifdef`: se evalúa si la definición del identificador está definida.
- La directiva `#ifndef`: se chequea la condición opuesta de `#ifndef`, cuando el identificador no está definido o la definición se remueve por `#undef`, la condición es verdadera, si no es falsa.
- La directiva `#endif`: su uso es obligatorio para cerrar el bloque demarcado por alguna directiva condicional (`#ifdef`, `#ifndef`, `#if`).

Ejemplo integrador:

```
#ifndef __ARCHIVO_H__
#define __ARCHIVO_H__

#ifdef DEBUG
#define ASSERT(cond) if (!(cond)) {fprintf(stderr, "assert failed in \
        %s:%d\n", __FILE__, __LINE__); exit(1)}
#else
#define ASSERT(cond)
#endif // DEBUG

#define PI 3.14159
#define CIRCLE_AREA(radio) (PI*(radio)*(radio))

#if TP0
int foo(int a, char* s) {
    printf("Se definió la constante TP0\n");
}
#elif TP1
int foo(double b, double c) {
    printf("Se definió la constante TP1\n");
}
#else
int foo(void) {
    printf("No hay más TPs\n");
}
#endif // TPs

#undef NO_QUEREMOS_ESTA_MACRO

#endif // __ARCHIVO_H__
```

5. **¿Qué es una macro de C? Detalle las buenas prácticas para su definición. Ejemplifique.**

   Una macro es un nombre que se le pone a un fragmento de código, y que luego se traduce en tiempo de pre-procesamiento. El programador es el responsable de asignar dicho nombre al código mediante la directiva #**define**, y luego el pre-compilador se encarga de reemplazar las ocurrencias de dicho nombre por el código que corresponde. Por ejemplo:

```
#define MIN(a, b) (((a) < (b)) ? (a) : (b))

    int main(int argc, char* argv[]) {
        int a = 2;
        int b = 5;
        int c = MIN(a, b); // (((a) < (b)) ? (a) : (b))
    return 0;
}
```

   Para su buena utilización, es necesario utilizar paréntesis para encapsular los operandos, y las operaciones que se realicen. De otra forma, como el compilador literalmente reemplaza la expresión con los valores recibidos, pueden ocurrir errores de consistencia con resultados no deseados: en la próxima pregunta puede encontrarse un ejemplo de este problema.

   Otra buena práctica es definir los nombres en mayúsculas, para diferenciar las mismas de las funciones.

6. **¿Por qué se recomienda encerrar los parámetros de una MACRO de C entre paréntesis (ej.: #define SUMA(a,b) (a)+(b) ? Ejemplifique.**

   Se recomienda el uso de parentesis entre los parametros para asegurar el correcto funcionamiento de la macro, independientemente del input que se le pase. Por ejemplo, si se tuviera la siguiente macro:

```
#define MULTIPLICAR(a,b) a * b
```

   y se la llamara de la forma

```
int b = MULTIPLICAR(1 + 2, 3 + 4);
```

   El resultado deseado es el de *MULTIPLICAR(3,7)* (teniendo por resultado 21). Sin embargo, al no poner parentesis, el fragmento de codigo al expandir la macro termina siendo:

```
int b = 1 + 2 * 3 + 4;
```

   Que da por resultado 11. En cambio, si se definiera la macro como

```
#define MULTIPLICAR (a,b) (a) * (b)
```

   el resultado seria

```
int b = (1 + 2) * (3 + 4);
```

   Que da por resultado 21, que era lo que se esperaba.

7. **¿Qué sucede si se escribe una macro recursiva? ¿cómo se expande?**

   Si se escribe una macro "recursiva", el preprocesador reemplazara una única vez la macro (es decir, no se expandira de forma recursiva). Por ejemplo, si se tiene la macro:

```
#define FACTORIAL(n) (n == 1) ? (1) : (n) * (FACTORIAL(n-1))
```

   Y se la utiliza en el código:

```c
int main(void){
    int a = 5;
    printf("El factorial de 5 es %d", FACTORIAL(5));
}
```

Esto se expande de la forma:

```c
int main(void){
    int a = 5;
    printf("El factorial de 5 es %d", (a == 1) ? (1) : (a) * (FACTORIAL(a-1)));

}
```

Lo cual dará error de compilación, debido a que `FACTORIAL` no existe en la tabla de símbolos.

8. **Explique qué se entiende por "compilación condicional". Ejemplifique mediante código. ¿En qué parte del proceso de transformación de código se resuelve?**

   Se trata de una facilidad que nos provee el preprocesador para incluir o excluir determinadas porciones de código en función de constantes conocidas al momento de compilar. Por ejemplo:

   ```c
   #ifdef DEBUG
   #define ASSERT(cond) if (!(cond)) {fprintf(stderr, "assert failed \
           in %s:%d\n", __FILE__, __LINE__); exit(1);}
   #else
   #define ASSERT(cond)
   #endif

   int main() {
       ASSERT(1 == 0);
       return 0;
   }
   ```

   Si compilamos este código con la constante 'DEBUG' definida, se realizará el chequeo en 'assert'. Caso contrario, no se incluirá el chequeo ya que 'assert(1 == 0)' se reemplazará por '' (vacío).

   Es resuelta durante la etapa de pre-compilación.

9. **¿Cómo se evita que un .h sea compilado más de una vez dentro de la misma unidad? ¿Qué instrucciones de precompilación suelen utilizarse? Ejemplifique.**

   Es convención, al comienzo de cada archivo "header" (.h) definir una constante con el nombre del archivo y utilizar la directiva del precompilador `#ifdef` para que se incluya una única vez. A esto se le llaman "wards". Un ejemplo:

   ```c
   // Archivo sumar.h
   #ifndef __SUMAR_H__
   #define __SUMAR_H__

   int sumar(const int a, const int b);

   #endif //__SUMAR_H__
   ```

10. **¿Qué características debe tener un compilador C para se considerado "portable"?**

    Para ser considerado portable, tiene que respetar 3 elementos:

    - Sintaxis: tiene que compilar, entender, parsear y validar la sintaxis C.

- Conjunto de librerías estándar: tiene que proveer librerías estándar.
- El proceso de transformación de código C a ejecutable debe seguir el protocolo estandarizado.

11. ¿Qué valor arroja `sizeof(int)`? Justifique.

Depende de la arquitectura y del compilador. Pueden ser 2, 4, u 8 bytes. En una arquitectura de 16bits, por lo general será 2. En una de 32, 4. Y en una de 64, puede ser 8 o muchas veces 4 ya que se decidió por razones de compatibilidad mantener el valor.

12. Describa los pasos realizados por la instrucción de preprocesador `#include`. ¿Por qué se encuentra desrecomendado utilizar `#include` con archivos .c? Justifique.

Los pasos realizados por la instrucción include son, en primer lugar, determinar la cantidad de espacio necesaria para insertar el contenido del archivo que esta siendo incluido en el archivo que esta invocando tal directiva. Una razon para evitar incluir el contenido directo de los archivos .c, es evitar la aparicion de simbolos repetidos. Si se incluyen de forma directa los archivos .c, es probable que multiples modulos utilicen simbolos con nombres iguales para propositos diferentes, llevando a un error de linkeo. Por otra parte, el uso de los archivos .h tiene el proposito de exponer unicamente la interfaz publica de un modulo, lo cual permite cambiar la implementacion de los mismos sin necesidad de recompilar todos los modulos que hagan uso de ellos.

1.2. Concurrencia

1. ¿Qué es un thread? Describa sus características en términos de area de memoria y otras características que le son propias y que comparte con el resto de los threads de un programa.

Un thread es literalmente un hilo de ejecución de un programa, que tiene su propio stack de ejecucion (y conjunto de registros, lo cual permite la ejecucion concurrente), y que comparte con los demás threads del programa: el heap, el code segment, el data segment, y los file descriptors.

2. Explique las diferencias entre un hilo y un proceso. ¿En qué casos es conveniente utilizar dichos elementos para programación paralela?

Un proceso es la ejecución de una instancia de un programa dado, distintos procesos no comparten memoria entre sí. Cada instancia de un mismo programa pueden tener varios hilos en ejecución, estos son unidades de procesamiento independientes que comparten al menos el data-segment, code-segment y heap. Dependiendo de la implementación los hilos de un programa podrían verse al sistema operativo como un proceso en sí (este es el caso de Linux).

Cada uno de estos presenta sus ventajas y desventajas. Se podrían usar procesos diferentes para alcanzar una mayor seguridad y aislamiento entre las partes de una aplicación. Los hilos de un mismo programa son normalmente utilizados cuando se requiere que las diferentes lineas de procesamiento compartan la memoria y recursos.

3. ¿Qué es un programa multi-hilo (Multi-thread)? ¿Cuales son sus ventajas?

Un programa multi-hilo contiene dos o más partes que pueden correr concurrentemente, y cada parte puede handlear diferentes tareas al mismo tiempo y realizar los usos óptimos sobre los recursos disponibles. Particularmente, si se trabaja con operaciones bloqueantes, es muy útil el uso de threads para aprovechar al máximo los recursos y mantener al proceso "vivo".

4. ¿Qué significa que una función es blocante? ¿Cómo subsanaría esa limitación en término de mantener el programa 'vivo'?

Una función es bloqueante si no retorna inmediatamente tras su ejecución, por lo que el hilo donde se ejecuta esta función puede quedar a la espera de algún evento para poder terminar la función y continuar. Si el programa es single-threaded, esto implica que el programa no avanza hasta que dicha función retorne. Una opción es utilizar múltiples hilos, ya que si un hilo se bloquea el programa puede seguir funcionando en otro.

5. **Explique en qué situaciones es recomendable utilizar programación multi-hilo (multithreading) para realizar cierto procesamiento. ¿Existe algún caso donde utilizar multithreading sea perjudicial?**

   Es útil utilizar programacion multi-hilo en programas que necesiten realizar operaciones de entrada/salida al mismo tiempo que operaciones de procesamiento intensivas para el procesador, en terminos de mantener el programa ”vivo” a ojos del usuario. Por cada thread que se utiliza en un programa, se agrega un cierto overhead (debido a que los context switches no son instantaneos y el uso de recursos de sincronización). En un servidor que maneja una cantidad de clientes del orden de los miles, mantener un thread por cliente puede causar que la frecuencia con la cual ocurren los context switches afecte de forma severa la performance del servidor.

6. **¿Qué recursos conoce para que 2 o más threads controlen el acceso concurrente a estructuras de memoria comunes?**

   Para que dos o más threads controlen el acceso concurrente a estructuras de memoria comunes y no se generen problemas de consistencia, se pueden utilizar:

   - *exclusión mutua:* permite que únicamente un hilo ejecute un segmento de código determinado al mismo tiempo. Para implementar la exclusión mutua, se puede hacer uso de mutex, condition variables, monitores, colas bloqueantes, semáforos, entre otras primitivas utilizadas.
   - *atomicidad:* permite efectuar operaciones de manera inmediata ("de un tirón"), sin desglosarse en varias instrucciones que den lugar a race conditions.

7. **¿Cómo se logra que 2 threads accedan (lectura/escritura) a un mismo recurso compartido sin que se generen problemas de consistencia? Ejemplifique.**

   Esto se logra utilizando el mecanismo de exclusión mutua (MUTEX). Los mutex proveen de dos funciones: ‘void lock();‘ y ‘void unlock();‘. Utilizando la primera, se intenta tomar control sobre el recurso compartido, y utilizando la segunda, se cede el control.

   Para proteger un recurso compartido, los hilos que quieran acceder primero deberán realizar la toma del mutex mediante ‘mutex.lock()‘. Sólo un hilo podrá tener el mutex a la vez, por lo que si varios hilos quieren hacer ‘mutex.lock()‘ al "mismo tiempo", sólo uno efectivamente lo logrará y podrá continuar con la ejecución. Para el resto de los hilos, cuando quieran hacer ‘mutex.lock()‘, esta será una operación bloqueante hasta que el hilo que actualmente tenga el mutex lo libere, momento en el cual todos los mutex a la espera intentarán nuevamente tomarlo. El scheduler definirá qué hilo logra tomarlo (escapa al control del programador de alto nivel). El hilo con el mutex puede modificar el recurso, y luego liberar el mutex.

   Ejemplo: (implementación conocida como Monitor)

```
class ProtectedNumber {
private:
    int number;
    std::mutex m;

public:
    ProtectedNumber() : number(0) {}
```

```
    void set_number(int new_number) {
        m.lock();
        number = new_number;
        m.unlock();
    }

    int get_number() const {
        m.lock();
        int aux = number;
        m.unlock();

        return aux;
    }
}
```

8. **Explique el concepto de mutex. Escriba un ejemplo simple de código donde su uso resulte fundamental para garantizar el correcto empleo de recursos en un contexto concurrente. Describa los métodos disponibles y su uso.**

   El mutex es un objeto con dos estados posibles, tomado y liberado, que puede ser manipulado desde varios hilos simultáneamente. Cuando un hilo solicite el mutex lo recibe de inmediato si está liberado. Cualquier otro hilo que lo solicite quedará suspendido hasta que otro hilo lo libera, cuando un hilo lo libera, uno de los hilos que está en espera, lo recibirá. Este proceso se repite hasta que no haya más hilos y el mutex quede nuevamente liberado.

   Por ejemplo: cuando el hilo 1 toma el mutex, el mutex está liberado, y accede al recurso compartido que este en la linea siguiente, y en el mismo tiempo, el hilo 2 intenta tomar el mutex, se quedará suspendido hasta que se libre el mutex.

```
#include <mutex>
#include <thread>

std::mutex m;
int number = 0;

void setNumber(){
    m.lock();
    number++;
    m.unlock();
}

int main(){
    std::mutex t1, t2;
    t1 = std::thread(setNumber);
    t2 = std::thread(setNumber);
    t1.join();
    t2.join();
    return 0;
}
```

9. **¿Qué es un Deadlock? Ejemplifique.**

   Un deadlock es el bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente que compiten por recursos del sistema o bien se comunican entre ellos.

   Por ejemplo:

```cpp
int main(int argc, char* argv[]) {
    std::mutex m;
    m.lock();
    std::thread t1([](std::mutex& m){
        m.lock(); // se bloquea esperando que el mutex m se libere
        m.unlock();
    }, std::ref(m));

    t1.join(); // se bloquea esperando a que t1 termine
    m.unlock();

    return 0;
}
```

Otro ejemplo:

```cpp
int main(int argc, char* argv[]) {
    std::thread t([](){
        std::mutex m;
        m.lock();
        m.lock(); // se bloquea ad infinitum
        m.unlock();
    });
    t.join();

    return 0;
}
```

10. ¿Qué función conoce para lanzar un nuevo hilo de trabajo (Thread)? ¿Qué parámetros requiere? Ejemplifique mediante código.

En *C++* se puede lanzar un thread instanciando un objeto de la clase **std::thread**, el cual, en su constructor, recibe un puntero a la función que ejecutará el mismo, seguido de los argumentos de la misma.

Verbigracia:

```cpp
void foo() {
    printf("Esto se ejecuta en otro thread secundario\n");
}

int main(const int argc, char const* argv[]) {
    printf("Comienza el hilo ppal \n");
    std::thread t(foo);
    t.join();
    printf("Termina el hilo ppal \n");
    return 0;
}
```

11. Describa las formas en las cuales un thread puede finalizar su ejecución. Ejemplifique.

Los threads, como cualquier proceso, pueden finalizar de forma exitosa o ante un error. En el caso exitoso, el thread logrará llegar al final de su ejecución y retornará: se convertirá entonces en un hilo *joineable*, que permitirá a su invocador realizar el `join` sin bloquearlo. Ante una excepción durante la ejecución, la misma se propagará hacia arriba hasta llegar a la función principal del hilo, donde en caso de no ser manejada, el mismo terminará debido a

este error (como cualquier otro proceso). Es importante destacar que en C++ la excepción debe ser manejada en el mismo hilo, ya que no puede propagarse al hilo principal de ejecución (de forma *natural*).

12. **¿Qué función utiliza para esperar la terminación de un thread? Ejemplifique mediante código.**

    Se utiliza la función 'join' sobre un hilo en ejecución, que se encargará de unir al hilo con el principal y liberar los recursos utilizados.

    ```
    ...
    std::thread t1([]() {fprintf(stderr, "hola!\n");});
    t1.join();
    ...
    ```

13. **¿Qué ventaja ofrece un lock RAII frente al tradicional lock/unlock?**

    El lock RAII genera más robustez en el código, ya que en el constructor se adquiere el mutex y en el destructor se lo libera, minimizando los errores y posibles deadlocks generados por el programador.

14. **¿Cómo puede saberse si un thread se encuentra activo o si ya terminó su ejecución?**

    Existe un metodo de thread llamado *joinable*, que devuelve booleano indicando si el thread ya termino su ejecucion (y por ende puede llamarsele join sin que el thread llamante se bloquee).

15. **Indique las diferencias entre hilos verdes (o de usuario) e hilos "heavy" (o de kernel).**

    Los hilos *heavy* (o de kernel) son vistos por el scheduler del sistema operativo como procesos independientes, alternando su ejecución de la misma forma que alterna la ejecución del resto de los procesos del sistema. En cambio, los hilos *verdes* (o de usuario), incorporan un nuevo scheduler en *user-space* que switchea entre los distintos hilos (notar que nunca habrá paralelismo real en este caso), para luego finalmente ir al scheduler del sistema operativo. Es decir que el scheduler del sistema operativo ve a todos los threads como un sólo proceso al que darle lugar, y luego dentro de dicho proceso se define qué thread se ejecuta.

1.3. Sockets

1. **¿Qué diferencias existen entre una comunicación UDP y una TCP? Describa un uso adecuado para cada uno de ellos.**

    TCP es un protocolo orientado a la conexión, y como tal, las partes deben establecer la conexión y luego cerrarla. En cambio, UDP es un protocolo orientado a los datagramas, en el cual no hay necesidad de establecer, mantener y cerrar una conexión, por lo que tiene menos overhead y es más rápido. A su vez, TCP garantiza la comunicación de los datos y su orden, mientras que UDP no lo hace (tiene que ser manejado por la aplicación).

2. **Describa la forma de enviar datos mediante UDP y TCP haciendo referencia a las funciones necesarias en cada proceso.**

    Para TCP, se utiliza `send` ya que una vez establecida la conexión sólamente se deben enviar los datos (con un loop apropiado).

    Para UDP, al no tener una conexión definida, se debe utilizar la función `sendto` a la cual se le debe proveer el destinatario para cada mensaje a enviar.

3. **Explique el uso de la función RECEIVE en comunicaciones orientadas a la conexión. Describa sus parámetros y su comportamiento en relación a la cantidad de bytes que "lee".**

   La función **recv**, cuyo prototipo es:

   ```
   ssize_t recv(int sockfd, void *buf, size_t len, int flags);
   ```

   Se utiliza para recibir bytes de un socket determinado. `sockfd` representa el file descriptor del socket a través del cual recibiremos los datos, `buf` representa el buffer donde se almacenarán los bytes leídos, `len` es la cantidad de bytes máxima a leer, y `flags` permite agregar indicadores adicionales, como por ejemplo **MSG_DONTWAIT**, o **MSG_WAITALL**. Si no fuera por este último parámetro, su utilización sería equivalente a la de la syscall `read(2)`.

   Es importante aclarar que en el contexto de las comunicaciones orientadas a la conexión (orientadas a bytes) nada nos garantiza que se reciban efectivamente la cantidad de bytes que solicitamos con el parámetro `len`, ya que si bien TCP nos garantiza que no habrá perdida de paquetes y que los mismos llegarán ordenados, nada nos asegura que una sóla llamada a `recv` lea todos los bytes deseados. Es por esto que si queremos recibir una cantidad determinada de bytes es necesario implementar una función que loopee hasta recibirlos, como la siguiente:

   ```
   /* Implementación de recv_all para bloquearse hasta recibir una determinada
    * cantidad de bytes. */
   ssize_t recv_all(int skt_fd, char* buf, size_t len, int flags) {
       ssize_t received = 0;
       ssize_t last_received = 0;
       while (received < len) {
           last_received = recv(skt_fd, &buf[received],
                                len - received, flags);
           if (last_received < 0) {
               return -1;
           } else if (last_received == 0) {
               return received;
           } else {
               received += last_received;
           }
       }

       return received;
   }
   ```

4. **Explique la función listen haciendo referencia a su prototipo/firma y propósito. ¿Qué parametros tiene y para que sirven?**

   La función `listen` se utiliza para esperar conexiones nuevas en un servidor que utiliza el protocolo TCP. La firma es:

   ```
   int listen(int skfd, int backlog);
   ```

   Toma como parámetro un file-descriptor (de un socket que debe estar *"bindeado"*) y un `int` con la máxima cantidad de clientes para la cola "en espera".

5. **¿Qué propósito tiene la función socket?**

   ```
   int socket(int domain, int type, int protocol);
   ```

La funcion `socket` tiene como proposito crear un extremo de comunicacion, y en el caso exitoso, devuelve un file descriptor que hace referencia a dicho extremo. Sus argumentos son `int domain, int type` e *int protocol*. El primer argumento `int domain`, hace referencia a la familia de protocolos de comunicacion que utilizara el socket (por ejemplo `AF_INET` para el protoclo IPv4). El argumento `int type` hace referencia al tipo de socket (por ejemplo `SOCK_STREAM` para socket TCP), y finalmente el argumento `int type` sirve para enviar flags que modifican el comportamiento del socket (por ejemplo `SOCK_NONBLOCK` para hacerlo no bloqueante).

6. **Explique el propósito del llamado a bind y accept en una aplicación servidor TCP/IP. Indique que parámetros poseen, y el retorno esperado.**

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

Cuando un socket es creado con la syscall *socket*, este existe en un namespace pero no tiene una direccion (address) asignada a el. Bind le asigna la direccion especificada por address al socket referido en el file descriptor socketfd. Tradicionalmente, a esta operacion se le llama "asignarle un nombre a un socket". Devuelve 0 en caso de ser exitoso, y -1 en caso de error.

```
int accept(int socketfd, struct sockaddr *adadr, socklen_t *addr);
```

Es utilizada en sockets orientados a la conexion (como es el caso de los sockets TCP). Extrae la primera conexion de la cola de conexiones entrantes y le asigna a dicha conexion un file descriptor, que es devuelto como valor de retorno de la funcion. Si se especifica un valor en sockaddr y socklen, estos valores son llenados con la informacion de la conexion entrante (el *peer*).

7. **Explique el propósito y uso de la función BIND en una comunicación UDP.**

La función `bind` en el protocolo UDP se utiliza para vincular un cierto puerto para recibir mensajes, si no se hiciera esto, el programa no tendría una puerta de entrada y salida para los mismos.

8. **Comunicaciones: ¿Qué formas conoce para determinar cuando se recibió un paquete de datos completo?**

Es necesario definir como parte del protocolo de comunicación la estructura de los paquetes: se puede optar por diseñarlos de tamaño fijo, para así poder recibirlos utilizando la función `recv_all` descripta anteriormente (que loopea sobre recv hasta recibir len bytes), o se puede utilizar un caracter centinela como delimitador de los mismos, en cuyo caso hay que tener distintas precauciones, ya que podría suceder que se quiera que dicho caracter forme parte de un paquete en sí.

Todas estas cuestiones forman parte del diseño del protocolo, parte fundamental de cualquier comunicación ya que sin este sería imposible lograrla.

9. **Suponga que dos aplicaciones (A y B) se encuentran comunicadas por un socket TCP. La aplicación A envía a la B un paquete de 123 bytes. ¿Como puede recibir la aplicación B esos datos? En ún solo paquete?...Partidos en 2?.. en 3?....**

La conexión TCP nos garantiza que no habrá pérdida de paquetes, y que los bytes llegarán ordenados. Sin embargo, nada nos garantiza que en una llamada a `send` o `recv` se envíe o se reciba la cantidad de bytes especificada en el parámetro **len**, por lo que si queremos enviar o recibir una cantidad determinada de bytes, es necesario loopear realizando el **send** o el `recv` según corresponda hasta que se hayan enviado o recibido la totalidad de los bytes.

En el caso del ejemplo, los datos enviados por A podrían llegar en cualquier cantidad de paquetes menor a 124, ya que siempre se envía/recibe al menos un byte (caso contrario, significa que la conexión fue cerrada).

1.4. C/C++

1. **¿Qué significa `this` en C++? ¿Dónde se usa explícita o implícitamente? ¿Dónde no es necesario?**

   `this` es un puntero a la instancia que está ejecutando un método de la clase. Este se encuentra de forma implícita en los llamados a los métodos de las clases y dentro de ellos, al acceder a los miembros (de datos o funciones). Explícitamente, debe ser utilizado cuando hay un conflicto de nombres en un método, por ejemplo:

```
class MiClase {
    private:
        int a;
    public:
        void foo(int a) {
            this->a; // Atributo de instancia
            a; // La variable recibida por parámetro
        }
}
```

2. **Explique qué es cada uno de los siguientes, haciendo referencia a su inicialización, su comportamiento y el area de memoria donde residen:**

   ***a*) Una variable global static.**

   ***b*) Una variable local static**

   ***c*) Un atributo de clase static.**

   a) Variable **global static**: variable que reside en el data segment, sólo pueden ser accedidas desde el archivo donde se definen (tienen file-scope). Se inicializa cuando comienza el programa y se libera cuando termina.

   b) Variable **local static**: variable que reside en el data segment, sólo puede ser accedida en el scope donde fue definida (scope de función). Su valor se preserva entre sucesivos llamados a la misma función. Se inicializan siempre con un valor. Si no se especifica uno, la inicializacion default asignada por el compilador es 0 (o NULL de tratarse de un puntero).

   c) Atributo de **clase static**: nuevamente, es una variable que reside en el data segment, común a todas las instancias de dicha clase. Al igual que en el caso anterior, de no especificarse un valor de inicializacion, se les asigna en tiempo de compilacion una inicializacion a un valor de 0 o NULL, dependiendo del tipo del atributo.

3. **Explique qué usos tiene la palabra reservada static en C++. De un breve ejemplo de uno de ellos.**

   Los usos de esta palabra son los siguientes:

   - Variable global static: descripta previamente. Ejemplo:

```
static int a = 3;
```

   - Función global static: será una función definida únicamente en el archivo donde se encuentra declarada. Ejemplo:

```
static int sumar(int a, int b) {
    return a+b;
}
```

   - Variable local static: descripta previamente. Ejemplo:

```
void foo() {
    static size_t run_times = 1;
    ...
}
```

- Atributo de clase static: descripto previamente. Ejemplo:

```
class AlumnoFiuba {
    static bool cursada_virtual;
};

bool AlumnoFiuba cursada_virtual = true;
```

- Método de clase static: será una función global que puede acceder únicamente a los miembros estáticos de la clase. Ejemplo:

```
class AlumnoFiuba {
    static bool cursa_virtualmente() {
        return AlumnoFiuba::cursada_virtual;
    }
};
```

4. **Indique 2 posibles usos del modificador const. Utilice en cada caso una breve sentencia de código que lo emplee junto con la explicación del significado/consecuencia de su uso.**

- Para recibir en una función argumentos constantes que por contrato no podrán ser modificados dentro de la función. Por ejemplo:

```
int sumar(const int& num1, const int& num2) {
    return num1 + num2;
}
```

  En este caso, no se podría cambiar los valores de num1 y num2. Es decir, fallaría una asignación del tipo `num1 = 10`.

- Para evitar que cambie el estado de un objeto en un método de clase. Por ejemplo, si tenemos:

```
class Fraccion {
    private:
        int numerador, denominador;

    public:
        ...
        int getNumerador() const;
        ...
}
```

  En este caso, dentro del método getNumerador no podrán modificarse los atributos de la clase Fracción (numerador/denominador).

5. **¿Por qué se recomienda utilizar los modificadores `const &` en la firma de una función cuando se requiere pasaje de parámetros std::string en C++? Escriba un breve ejemplo con una función que recibe un std::string sin utilizar `const &` junto con su invocación.**

Principalmente, el `&` para que no se copie la memoria al llamar una función. Además, permite llamados a la función con strings literales como se muestra en el ejemplo:

```
void foo(const std::string& str);

int main(int argc, const char* argv[]) {
    foo("Hola!");
    return 0;
}
```

6. ¿Qué significado tiene la palabra reservada const cuando es antepuesta a un parámetro de una función/método? ¿Para qué sirve?. Ejemplifique.

Agregar la palabra reservada const antepuesta a un parámetro de una función implica que, dentro de la función, el parámetro no puede ser modificado por contrato.

Esto es muy útil para poder tener seguridad de que al llamar a una función que toma como parámetros ciertos recursos como podrían ser punteros, o valores pasados por referencia, los mismos no van a ser modificados (como podría ser el caso de un `string` de C++ pasado por referencia).

Por ejemplo:

```
void imprimir_string(const std::string& buffer){
    std::cout << buffer << std::endl;
}
```

7. ¿Qué significado tiene la palabra reservada const cuando es colocada como terminación de una declaración de un método? Ejemplifique con una clase y una invocación donde sea necesario su uso.

Cuando se agrega la palabra reservada const como terminacion de una declaracion de un metodo, esto significa que una llamada a dicho metodo no modificara de ninguna forma el estado de la instancia para la cual se llamo el metodo. Esto es necesario a la hora de invocar metodos sobre instancias de clases que fueron pasadas antepuestas por la palabra const en la firma de una funcion. Por ejemplo:

```
void imprimir_punto(const Punto& punto){
    std::cout << punto.get_x() << " - " << punto.get_y() << std::endl;
}
```

Para que esto funcione, los metodos `get_x` y `get_y` deben tener const al final de su declaracion:

```
class Punto{
    public:
        int get_x() const;
        int get_y() const;
};
```

8. Describa qué tipos de variables y atributos de clases se guardan en los diferentes areas de memoria (stack, data segment, heap, code segment).

Se guardan en el stack:

- Variables locales de las funciones (no declarados como static).
- Atributos de clase no estaticos (pues todos los atributos de una instancia no estatica se guardan en el stack frame de la funcion donde se creo esa instancia).

Se guardan en el data segment:

- Todas las variables globales (tanto static como no static).
- Las variables locales declaradas como static.
- Los atributos de clase declarados como static.

Se guardan en el heap:

- Todas las instancias y variables dinamicas (creados mediante new, o asignadas en memoria que fue reservada con malloc).

9. Explique cómo funciona la sobrecarga de operadores en C++.

La sobrecarga de operadores permite utilizar los objetos diseñados con una sintaxis más simple y de integración mucho más sencilla y legible al código. Se puede lograr definiendo los operadores como métodos de clase:

```
class NumeroComplejo {
    public:
        bool operator<=>(const NumeroComplejo& z1, const NumeroComplejo& z2);

        NumeroComplejo operator+(const NumeroComplejo& z1,
                                 const NumeroComplejo& z2);
        NumeroComplejo operator-(const NumeroComplejo& z1,
                                 const NumeroComplejo& z2);

        friend std::ostream& operator<<(std::ostream& ostream,
                                        const NumeroComplejo& z);
        friend std::istream& operator>>(std::istream& istream,
                                        const NumeroComplejo& z);
};
```

10. ¿Qué son las excepciones en C++? Dé un ejemplo de uso que incluya las clausulas try/catch.

Las excepciones son proveen un mecanismo de reacción frente a circunstancias excepcionales (como errores en ejecución), en las cuales se transfiere el control a handlers especiales.

```
int main(const int argc, char* const argv[]) {
    ...
    try {
        // código que puede llegar a lanzar excepciones
    } catch(const std::exception& e) {
        // handlear excepción
    }
}
```

11. ¿Qué es un functor? ¿Qué ventaja ofrece frente a una función convencional? Ejemplifique.

Un functor es un objeto de C++ que sobrecarga el operador () (‘`operator()`‘) para que las instancias de dicho objeto sean .ejecutables". La ventaja que poseen frente a una función convencional es que son objetos, y por lo tanto además del comportamiento, tienen un estado asociado. Además, permiten descoplar el momento en el que se pasan los parámetros a una función del momento en el cual se ejecuta.

Por ejemplo, un sumador:

```
class Sumador {
private:
    int current_value;

public:
    Sumador() : current_value(0) {}

    void operator()(int value) {
        current_value += value;
    }
```

```
    int get_value() const {
        return current_value;
    }
};
```

12. ¿En qué consiste el patrón de diseño RAII? Ejemplifique.

El patrón de diseño RAII permite controlar el uso de recursos limitados (como la memoria, sockets, file descriptors, etc) mediante el uso de los scopes. La vida útil de un objeto está asociada al scope donde el mismo fue instanciado, por lo que cuando la variable se vaya del scope, la misma será destruída y su destructor será ejecutado (incluso ante excepciones, lo que hace que este patrón sea muy poderoso y permita el diseño de objetos exception-safety).

Consiste en inicializar los recursos a utilizar en la inicialización de los objetos, y liberarlos en el destructor. De esta forma, evitaremos en gran medida tener leaks de recursos, y tener que preocuparnos por la liberación de los mismos en sitios no triviales.

13. Explique el concepto de referencias en C++. ¿Qué diferencias posee respecto de los punteros?

- Las referencias deben ser definidas en el momento de su declaracion, y su valor no puede ser modificado luego. Los punteros pueden ser definidos y cambiados en cualquier momento, como cualquier variable.
- Las referencias son un mecanismo para crear un alias (es decir, un nombre nuevo) para una instancia de una clase, mientras que los punteros son simplemente la direccion de memoria de dicha instancia. En este sentido, las referencias son una abstracción de mas alto nivel que los punteros.
- Los punteros pueden apuntar a su vez a otros punteros, ofreciendo multiples niveles de indireccion. Las referencias solo ofrecen un unico nivel de indireccion.

14. ¿Qué es un parámetro opcional en C++? ¿Cómo se utiliza? ¿Dónde puede usarse? Ejemplifique.

Un parámetro opcional es un parámetro que posee un valor por defecto en la firma del método. Al invocar a dicho método, se puede optar por darle un valor explícito o no. Si no lo pasamos de forma explícita, el parámetro toma entonces este valor por defecto.

Los parámetros por defecto deben ser definidos al final de la firma de la función, es decir que los parámetros que le sigan a uno opcional deben tener también un valor por defecto. No se puede dejar un hueco (parámetro opcional) en medio de la lista, ya que se generaría ambigüedad al llamar a dicho método.

Ejemplo:

```
// declarada en .h
int foo(int x, int y = 0);

// definida en .cpp
int foo(int x, int y){
    return x + y ;
}

int main(){
    printf( "el numero es %d ", foo(2));
    return 0;
}
```

15. ¿Qué diferencia existe entre el pasaje de parámetros por valor (ej.: void MiFuncion (MiClase);) y el pasaje por referencia constante (ej.: void MiFuncion(const &MiClase);)?

La diferencia radica en que al pasar los parámetros por valor se efectúa una copia del objeto que es pasado (ocupando mayor espacio en el stack), mientras que si se pasa por referencia constante no se realiza la copia, sino que se accede al objeto mediante el alias o referencia, no pudiéndose modificar su estado interno debido al modificador const que la antepone.

16. ¿Qué son las librerías STL? ¿Qué recursos ofrece? Explique y ejemplifique alguna de sus clases contenedoras (a su elección).

La Standard Template Library (STL) es una colección de estructuras de datos genéricas y algoritmos escritos en C++. Ofrece distintos tipos de recursos: algoritmos, iteradores, contenedores, funciones objeto, y adaptadores.

Entre los contenedores, tenemos varios tipos:

- Contenedores lineales: listas, vectores, y doble colas.
- Contenedores asociativos: conjuntos, multiconjuntos, maps, y multimaps.
- Contenedores adaptados: pilas, colas, y colas de prioridad.

Por ejemplo, dos contenedores muy utilizados son:

- **std::list<T>**: contenedor lineal que soporta inserción y eliminación de elementos en tiempo constante (utilizando iteradores). Se suele implementar como una lista doblemente enlazada.

```
std::list<int> lista = {1, 2};
lista.push_back(3);
lista.push_back(7);
lista.push_back(3);

for (auto it : lista) {
    std::cout << *it << std::endl;
}
```

- **std::unordered_map<T, U>**: contenedor asociativo que almacena elementos formados por la combinación de una clave y un valor mapeado. Permite el acceso constante a sus elementos mediante la clave.

```
typedef int AlumnoId;
class Alumno {...};

std::unordered_map<AlumnoId, Alumno> alumnos;
alumnos.emplace(std::piecewise_construct_t, std::forward_as_tuple(3),
                std::forward_as_tuple(...));
alumnos.emplace(std::piecewise_construct_t, std::forward_as_tuple(4),
                std::forward_as_tuple(...));
Alumno juan = alumnos.at(4); // acceso constante
```

17. ¿Qué ventajas y desventajas ofrecen las librerías STL frente a una implementación con punteros del correspondiente TDA?

Las librerías STL permiten hacer uso del paradigma RAII. Así, al hacer uso de los distintos objetos que ofrece la STL, uno puede abstraer la reserva y liberación de los recursos limitados que emplean, brindando un código más robusto, seguro e incluso legible. La desventaja que suge de hacer uso de las STL es que uno no conoce a priori la implementación a bajo nivel de las estructuras que ofrece.

18. ¿Por qué motivo las librerías STL son distribuidas mediante su Código Fuente y no compiladas en una .lib/.obj/.o?

La STL está implementada como templates. Por este motivo, no se podría compilar pues el compilador necesita saber el espacio que ocupan las estructuras con las cuales son utilizadas. Entonces, a la hora de hacer uso de los recursos que ofrece y especializarlos en clases determinadas es que se reemplaza el código fuente por las dichas clases correspondientes, pudiendo recién ahí ser compiladas.

19. Para utilizar una clase X con las librerías STD se piden algunas características particulares (Contructor default, operador ==, etc.) ¿por qué cree que estas son requeridas? ¿que sucedería si no se proveen las mismas? Justifique.

Dichas características son requeridas para garantizar la compatibilidad y plena funcionalidad de la clase que se está implementando con las estructuras y métodos provistas por la STL. De no proveerse las mismas, se comportarían por default, lo cual puede provocar comportamientos no deseados y, por ende, bugs.

20. ¿Qué objeto se provee en forma standard en C++ para el manejo de la salida standard?. Ejemplifique su uso.

Para la salida standard, se ofrece bajo el header `<iostream>` el objeto `std::ostream cout` el cual apunta a la salida estándar.

21. ¿Qué objeto se provee en ISO C para manejar la consola de entrada? Ejemplifique.

El objeto que se provee en ISO C para manejar la consola de entrada es `FILE *stdin`, y puede manipularse como cualquier otro archivo. Sin embargo, no debe alterarse su valor.

Ejemplo:

```c
#include <stdio.h>
int main(){
    char string[256];
    printf("ingrese su nombre: ");
    fgets(string, 256, stdin);
}
```

22. El segundo parámetro de la función fopen indica el modo de apertura del archivo. ¿Cuales son los modos posibles?

Los modos posibles son:

- "r" Sólo lectura, lanza error si el archivo no existe.
- "r+" Lectura y escritura, también lanza error si el archivo no existe.
- "w" Sólo escritura crea el archivo y (si ya existe lo trunca, es decir, elimina su contenido).
- "w+" Lectura y escritura, creando el archivo o vaciandolo si ya existiera.
- "a" Sólo lectura, crea el archivo si no existe y comienza posicionado al final del archivo.
- "a+" Lectura y escritura, crea el archivo si no existe y comienza posicionado al final del archivo.

Estos modos se pueden combinar con los modificadores `t` o `b`, que indican que el archivo abierto es de texto o binario, respectivamente.

23. ¿Qué significa _LINE_ y _FILE_ en un fuente C? ¿Qué usos tienen? ¿En qué parte del proceso de transformación de código se resuelven?

- __LINE__ es reemplazado por el número de línea en el que se escribió.

- __FILE__ es reemplazado por el nombre del archivo en el que se escribió.

Las resuelve el preprocesador, y son empleadas más que nada para debugging.

24. El símbolo de precompilación __LINE__ se expande como el número de línea donde se encuentra. Demuestre mediante un ejemplo su utilidad.

```
#define ERROR(msg) \
    fprintf(stderr, "error in %s, file %s, line %i \n", msg, __FILE__, __LINE__); \
    exit(1);
```

25. ¿Qué es un iterador de la librería estándar de C++? Ejemplifique su uso.

Un iterador es un objeto que permite al programador recorrer una colección sin que sea necesario que conozca la implementación de la misma. Usualmente lo utiliza para búsquedas, consultas de elementos, etc. Los contenedores STL se pueden recorrer utilizando iteradores.

```
std::vector<int> elementos;
for (auto i = elementos.begin(); i != elementos.end(); i++){
    std::cout << *i << std::endl;
}
```

26. Explique qué son los métodos virtuales puros y para qué sirven. De un breve ejemplo donde su uso sea imprescindible.

Los métodos virtuales puros son aquellos que están obligados a implementar las clases hijas. Son imprescindibles para declarar interfaces y clases abstractas.

```
class Dibujable {
    private:
        ...

    public:
        ...
        virtual void dibujar() const = 0;
        ...
};

class Triangulo : public Dibujable {
    private:
        ...

    public:
        ...
        void dibujar() const;
}
```

27. ¿Qué significa la palabra virtual antepuesta a un método de una clase? ¿Qué cambios genera a nivel compilación y al momento de ejecución?

La palabra virtual antepuesta a un metodo de una clase indica que, de existir una implementacion para dicho metodo en una clase derivada, al hacerse un llamado polimorfico a una instancia de dicha clase derivada a traves de un puntero a la clase base, el metodo llamado sera el de la clase derivada y no el de la clase base. Este mecanismo se llama *dynamic binding* (o late binding), y se implementa mediante la llamada tabla de metodos virtuales, la cual debe ser consultada por el programa en ejecucion cada vez que se haga una llamada a uno de estos metodos.

Esta tabla de métodos virtuales se construye en la etapa de compilación. Cuando se compila un método con la palabra virtual, se lo agrega a la tabla al igual que el override de estos.

28. Explique el concepto de object slicing (objeto recortado). Escriba un breve ejemplo sobre cómo esto afecta a una función que pretende aplicar polimorfismo sobre uno de sus parámetros.

El object slicing sucede cuando un objeto de una clase derivada es asignado a una variable de su clase base, y entonces los atributos adicionales de la clase derivada se pierden.

```
class Base {
    protected:
        int i;
    public:
        Base(int a) { i = a; }

        virtual void display() {
            cout << "I am Base class object, i = " << i << endl;
        }
};

class Derived : public Base {
    protected:
        int j;
    public:
        Derived(int a, int b) : Base(a) { j = b; }

        virtual void display() {
            cout << "I am Derived class object, i = " << i << ", j = " << j << endl;
        }
};

// Global method, Base class object is passed by value
void somefunc (Base obj) {
    obj.display();
}

int main() {
    Base b(33);
    Derived d(45, 54);
    somefunc(b);
    somefunc(d);     // Object Slicing, the member j of d is sliced off,
                     // since the argument is of Base class.
    return 0;
}
```

29. ¿Qué recomendación especial tendría Ud. al momento de escribir una clase que usa memoria dinámica y que es usada con polimorfismo? Ejemplifique.

Se debe declarar el destructor como virtual, ya que de esta manera las clases hijas pueden implementar su propio destructor para liberar la memoria alocada.

```
class Base {
public:
    Base();
    virtual ~Base();
```

```
};

class Derivada {
private:
    int* arreglo; // memoria dinámica;
    size_t largo;
public:
    Derivada();
    virtual ~Derivada(); // acá se debería liberar
                         // el arreglo dinámico
};
```

30. Explique los distintos tipos de herencia que admite C++. De un breve ejemplo de al menos una de ellas.

- **Herencia pública**: aquella en la que todos los miembros públicos y protegidos de la clase base conservan esos mismos niveles de acceso respectivamente en la clase derivada.
- **Herencia protegida**: aquella en la que todos los miembros públicos de la clase base adquieren el nivel de acceso protegido en la clases derivadas, mientras que los miembros protegidos conservan su nivel de acceso. Esto indica que una clase derivada puede luego heredar a otra clase los mienbros protegidos que heredó de su clase base.
- **Herencia privada**: aquella en la que todos los miembros públicos y protegidos de la clase base adquieren el nivel de acceso privado en las clases derivadas. De ahí se desprende que una clase derivada que haya heredado mediante herencia privada no puede heredar a otras clases los miembros que ha heredado de otras clases.

Ejemplo de herencia pública:

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

31. ¿Qué es la herencia múltiple? Escriba declaraciones de métodos/clases que ejemplifiquen sus conceptos.

Es un mecanismo mediante el cuál se puede, como su nombre lo indica, heredar de varias clases al mismo tiempo. La clase derivada heredará los métodos y los atributos de todas las clases base. Puede ser útil para diseñar una clase que implemente múltiples interfaces.

Por ejemplo:

```
class A {
    public:
        void foo();
}

class B {
    public:
        void bar();
}

class C : public A, public B {...}

int main() {
    C c;
    c.foo(); // método de la clase A
    c.bar(); // método de la clase B
```

```
    return 0;
}
```

Sin embargo, hay que tener cuidado ya que esto puede generar ambigüedad y errores de compilación por colisión de nombres. Si dos clases de las que se heredan utilizan el mismo nombre se generará un error de compilación ya que no se sabe a qué método debe llamar la clase hija al invocarlo, por ejemplo.

32. Resuma las accesibilidad de 3 tipos de componentes de una clase (píblico, protegido y privado).

Una clase tiene componentes (atributos y métodos) públicos, protegidos, y privados. Los componentes **públicos** pueden ser accedidos por cualquier función dentro del programa. Los componentes **protegidos**, en cambio, sólo pueden ser accedidos dentro de la clase y en las clases que hereden de esta (clases hijas). Finalmente, los componentes **privados** sólo pueden ser accedidos dentro de la clase.

33. Supongamos que la clase Hijo hereda de Padre; que Padre posee varios métodos públicos de utilidad en Hijo, a excepción de 1 que deseamos "ocultar". ¿Cómo podemos realizar esto? Ejemplifique.

Se puede realizar de varias formas. La más sencilla, sería declarar el método como privado. Si se lo quiere mantener público, lo que se puede hacer es marcar el método como deleted en la clase hija. Ejemplo:

```
class Padre {
    public:
        ...
        void foo();
        void bar(); // queremos ocultarlo
        ...
};

class Hijo : public Padre {
    public:
        void bar() = delete;
};
```

De esta forma, no podrá llamarse a `bar()` desde una instancia de Hijo.

34. ¿Qué ambigüedades pueden producirse en C++ al momento de decidir que versión de un método se debe invocar? Ejemplifique.

Se pueden producir ambigüedades si se utiliza polimorfismo mediante punteros con herencia, sin utilizar el modificador virtual en los métodos de la clase Base que se llamen polimórficamente. Como se explicó anteriormente, si no se definen como virtual, ante un llamado polimórfico mediante un puntero a la clase Base se llamará al método de la clase Base.

35. Si un método no es virtual, ¿puede redefinirse para especializar su comportamiento en clases derivadas? ¿Tiene alguna limitación esta especialización?

Sí y no. Si la pregunta es si se puede overridear un método de la clase base que no es virtual, la respuesta es no: sólo pueden overridearse los métodos virtuales de la clase base. Sin embargo, se puede definir **otro método** con **el mismo nombre** en la clase hija, y en este caso este método "esconderá" al de la clase base. Sin embargo, esto puede traer ambigüedades y bloquear comportamientos esperados en algunos escenarios, como lo es el del polimorfismo mediante punteros:

```
class Base {
    ...
    void foo() {
        std::cout << "Base::foo()" << std::endl;
    }
    ...
}

class Derivada : public Base {
    ...
    void foo() {
        std::cout << "Derivada::foo()" << std::endl;
    }
    ...
}

int main() {
    Derivada d;
    Base* b = &d;

    d.foo();
    b->foo();
    return 0;
}
```

La salida será:

```
Derivada::foo()
Base::foo()
```

Y esto se debe a que como el método no es virtual en la clase padre, no se decide en run-time qué método se llama. Para los métodos no virtuales, siempre se llama a su asociado según el tipo de dato desde donde se invoque al mismo.

36. ¿En qué casos recomendaría Ud. el uso de un destructor virtual? ¿Cuando son necesarios? Ejemplifique.

Indicar el destructor como virtual tiene las mismas implicancias que con cualquier otro método: si se lo define como virtual puro, las clases hijas **deberán** overridearlo. Si se lo define como virtual, las clases hijas **pueden** overridearlo, y si se lo define como no virtual, se lo llamará **siempre** que se destruya una clase hija.

Sin embargo y en este caso particular, es importante notar que si se utiliza polimorfismo, puede ser peligroso que el destructor de la clase Base no sea virtual, ya que ante una destrucción polimórfica de un objeto Derivada que reside en el heap, se llamará solo al destructor de Base, ignorando el destructor de la clase Derivada, por lo que se podrían estar generando leaks de recursos.

Marcando el destructor como virtual, le damos la oportunidad a las clases que hereden de que implementen su propio destructor para liberar los distintos recursos dinámicos que utilicen, si es que es necesario. De esta forma, se puede destruir un objeto Derivada mediante un puntero a Base, llamando a ambos destructores.

Ejemplo:

```
class Base {
        public:
                Base(){
```

```
                        std::cout << "Base();\n";
                }

                ~Base(){
                        std::cout << "~Base();\n";
                }
};

class Derivada : public Base {
        public:
                Derivada(){
                        std::cout << "Derivada();\n";
                }

                ~Derivada(){
                        std::cout << "~Derivada();\n";
                }

};


int main() {
        Base* b = new Derivada();
        delete b;
        return 0;
}
```

En este caso, la salida será:

```
Base();
Derivada();
~Base();
```

En cambio, si agregamos el virtual al destructor de Base, la salida será:

```
Base();
Derivada();
~Derivada();
~Base();
```

37. Explique qué es y para qué sirve un constructor de copia en C++.

a) Indique cómo se comporta el sistema si éste no es definido por el desarrollador.

b) Explique al menos una estrategia para evitar que una clase particular sea copiable.

c) Indique qué diferencia existe entre un constructor de copia y uno move.

Los constructores por copia permiten inicializar un objeto a partir de otro, copiando sus atributos. Si no es definido por el desarrollador, se genera automáticamente y por defecto un constructor por copia naive que copia bit a bit el objeto. Para evitar que sea copiable, podemos eliminar el constructor y su operador asignación por copia:

```
Clase(const Clase&) = delete;
Clase& operator=(const Clase&) = delete;
```

E implementarle un constructor de movimiento (o no implementarlo y no permitir entonces que el mismo pueda inicializarse a partir de otros objetos).

El constructor por movimiento a diferencia del constructor por copia, se *roba* los atributos del objeto origen, invalidando los recursos que no tengan sentido ser copiados en el mismo, pero dejando al objeto en un estado valido ya que posteriormente se llamará a su destructor. Sirve para no copiar file descriptors, punteros, etc., así como para evitar copias innecesarias.

38. Explique qué es y para qué sirve un constructor MOVE en C++. Indique cómo se comporta el sistema si éste no es definido por el desarrollador.

El constructor por movimiento permite, como su nombre lo indica, generar un nuevo objeto partiendo de otro, pero en vez de copiar los contenidos, se los "mueve". El objetivo de su implementación es evitar copias de recursos, ya sea porque resulta innecesario o incorrecto, como sería el caso de la copia de un puntero a un objeto en el heap que luego será liberado en el destructor. Si esto se realizara de la forma *naive* copiando el puntero, luego tendríamos un *double-free problem*.

En este contexto surge el constructor por movimiento: se mueven los atributos del objeto viejo al nuevo, para posteriormente **invalidarlos** en el objeto viejo (en el caso de un puntero, por ejemplo, settearlo en `NULL`). De esta forma el objeto viejo seguirá siendo válido (debe serlo ya que se llamará a su destructor), pero no nos generará problemas de consistencia.

Si el constructor por movimiento no es definido por el desarrollador, se incluye uno por default que realiza el movimiento con `std::move` de los atributos, pero **no los invalida** en el viejo objeto, por lo que si se desea mover atributos que no deben ser duplicados, será necesario implementar uno propio.

39. ¿Qué cuidado especial tendría Ud. al momento de escribir un operador= para una clase que usa memoria dinámica? Ejemplifique.

Tendria el cuidado de identificar si lo que se desea hacer es una shallow copy o una deep copy. En particular, si se trata de un asignador por copia, el comportamiento que asumiria es el de deep copy (es decir, reservar memoria y realizar una copia de los contenidos de memoria dinamica de la clase). Si se tratase de un asignador por movimiento, haria una shallow copy del puntero, y luego invalidaria el puntero del objeto que fue movido.

40. Describa un caso en el cual resulte imprescindible incluir un Constructor de Copia en una clase. Justifique mediante ejemplos.

```
class VectorDinamico {
private:
    void* array; // Mantiene un puntero al array
                 // dinámico allocado en el Heap
    size_t len;
    size_t max_size;
public:
    VectorDinamico();

    // Constructor por copia, allocará en el heap una copia del array
    VectorDinamico(const VectorDinamico& Other);
};
```

41. Describa el concepto de templates en C++. De un breve ejemplo de una función template.

El concepto de templates (tambien llamado programacion generica) permite escribir secciones de codigo de forma generica, sin necesidad de especificar tipos con los que se esta trabajando. Es particularmente util en casos donde se deberia escribir codigo muy similar entre dos

implementaciones de una funcion o de una clase que solo difieran en el tipado de alguna de las variables involucradas.

Por ejemplo:

```
template<class C>
std::list<C> make_copy(const std::list<C>& other){
    std::list<C> result;
    for(auto element:other){
        result.push_back(element);
    }
    return result;
}
```

42. ¿Por qué las librerías que usan Templates se publican con todo el código fuente y no como un .h y .o/.obj?

Porque las especializaciones de las templates (clases generadas a partir de las mismas) se resuelven en compilación, por lo que el .o variará según las clases que sea necesario generar. Las templates proveen de reglas al compilador, como si se tratase de una *receta* para generar clases, tomando ciertos parámetros.

43. Explique el uso de valores por defecto en templates de C++. Escriba un ejemplo.

Los valores por defecto permiten especializar una template a un tipo de dato específico, así se podrá definir las funciones con modificaciones para ese tipo de dato, de ser necesario.

44. ¿Describa brevemente el contenedor map de la librería estándar de C++? Ejemplifique su uso.

```
std::map<
    class Key,
    class T,
    class Compare = std::less<Key>,
    class Allocator = std::allocator<std::pair<const Key, T>
    > mapa_ejemplo;
```

Es un diccionario ordenado que utiliza como clave el tipo de dato `Key` para allocar el tipo de dato `T`. Opcionalmente puede recibir una clase particular para comparación de claves (que debe cumplir con los requisitos de tipo *Compare*) y una clase que cumpla con los requisitos de *Allocator* (así como muchos otros contenedores de la STL). Este diccionario tiene tiempo logarítmico para todas las operaciones (inserción, acceso, búsqueda -por clave- y eliminación).

45. ¿Qué elementos debo exigir a un equipo de desarrollo externo para poder utilizar una función de la librería que ellos están desarrollando? En qué parte del proceso de compilación se resuelven las llamadas a dicha función.

...

46. ¿Por qué se dice que los sistemas escritos en lenguaje C/C++ son portables? ¿Qué precausiones son necesarias para escribir un programa C/C++ que cumpla con esa cualidad?

La mayor parte de las aplicaciones y sistemas escritos en C/C++ funcionarán en cualquier otro sistema que permita la compilación con el mismo standard. Para lograr esto, se debe evitar la utilización de librerías no-estándar, así como en todos los lugares que sea posible el mismo estándar para la construcción de los sistemas.

47. ¿Qué características debe tener una clase para poder ser utilizada en un `std:list`? Justifique.

Para el estándar C11 (utilizado actualmente -2020- en la cátedra) los elementos que se quieran utilizar en una `std::list` deben cumplir con los requerimientos denominados *CopyAssignable* (debe tener definido el operador asignación `operator=(const T& other)`) y *CopyConstructible* (debe permitir la construcción por copia);

48. Para implementar una "Lista Genérica" se dicuten 2 enfoques: Uso de Templates; o Implementación de una lista de void *. ¿Qué ventajas ofrece cada una de ellas?

La implementación con `void*` suele ser complicada por el manejo de punteros, pero el código será compilado una sola vez sin importar los tipos de datos que se utilicen en la lista (ya que el *casteo* de los punteros queda como responsabilidad del usuario que se hagan correctamente). Se debe tener en cuenta además, que para este caso todo lo que se almacenan son punteros, por lo que las variables deben encontrarse siempre allocadas en el stack de trabajo, o utilizar el Heap.
Las Templates son una facilidad que ofrece C++ para escribir una clase (o función) genérica una única vez, además no se deben hacer comprobaciones de punteros nulos (lo que lleva a menos errores de tipo *segmentation fault*). La desventaja resulta en que por cada especialización de cada template, se genera todo el código para una clase diferente. Esto se puede resolver especializando la template en `void*` y luego escribiendo la misma para un tipo genérico de punteros, logrando la versión de la lista con punteros, pero para cualquier tipo.

49. ¿Qué es necesario que tenga la clase `MiClase` para que sea válido el siguiente código? Justifique.

```
MiClase C[2];
C[1] = ( C[0] > C[1] ) ? MiClase(7.3) : MiClase("7.2");
```

Analicemos parte por parte estas dos líneas:

- Por el `MiClase C[2]` es necesario un **constructor por defecto** (sin parámetros): `MiClase();`.
- Por el `=`, es necesario que posea el **operador asignación**, ya sea por copia o movimiento. Por ejemplo: `MiClase& operator=(const MiClase& copiable);`
- Por el $>$ es necesario que tenga definido el **operador** $>$ para poder comparar ambos objetos.
- Finalmente, se deben definir dos constructores más para satisfacer la segunda línea: `MiClase(float x);` y `MiClase(const char* x);`

1.5. Programación orientada a eventos, GUIs

1. ¿Qué características posee la programación orientada a eventos? ¿Cómo se programa en dicho paradigma?

Se basa en un paradigma en el que el funcionamiento de un programa es *reaccionar* a sucesos. En este, las acciones que ejecuta el sistema son en respuesta a eventos (sucesos) que surgen de los *actores* en el sistema. Los eventos pueden ser de todo tipo, provocados por interacción del usuario, paso del tiempo, o incluso otros eventos y el mismo código. La implementación del paradigma se basa en tener actores que disparan eventos que son agregados a una cola (*cola de eventos*) de la cual se extraen los mismos (por ejemplo, cada un cierto período de tiempo, o simplemente tan rápido como se pueda) por lo que llamamos *Event Loop*. Este se encargará de despachar los eventos a sus respectivos *handlers* (manipuladores). Es muy normal que se utilicen varios *Threads* para la implementación de un sistema orientado a eventos, especialmente en librerías gráficas que interactúan con el usuario. De esta forma,

los recursos compartidos por los actores o los diferentes eventos podrían requerir la utilización correcta de estructuras que permitan la concurrencia en el sistema.

2. **Describa el concepto de loop de eventos (events loop) utilizado en programación orientada a eventos y, en particular, en entornos de interfaz gráfica (GUIs).**

   Un loop de eventos es un patrón de diseño muy utilizado para los video juegos, así como para las interfaces gráficas. Consiste en tener un hilo que "loopee" constantemente y generalmente a período controlado realizando distintas acciones en cada iteración, y luego pasando a la siguiente.

   En programación orientada a eventos, en cada iteración se suelen procesar todo tipo de eventos entrantes al programa, ya sean eventos generados por dispositivos de I/O, eventos externos, eventos internos del programa, etc.

   En las interfaces gráficas, en cada iteración se limpia la pantalla (por ejemplo, en C++ y con SDL utilizando `SDL_RenderClear`), se dibujan los componentes de la vista con su estado actualizado según la iteración (utilizando en SDL `SDL_RenderCopy`), y se presenta la pantalla al usuario (utilizando `SDL_RenderPresent`). El loop, por lo general, será muy similar siguiendo esta estructura:

```cpp
// flag de ejecución
bool quit = false;

// función que se ejecutará una vez por iteración
void func();

int main(int argc, char* argv[]) {
    // loop de eventos

    while (!quit) {
        func();

        // opcionalmente, antes de pasar a la próxima iteración, podemos
        // dormir el thread para mantener un frame-rate constante
    }

    return 0;
}

void func() {
    // handlear eventos internos y externos
    // (con SDL, por ejemplo, sacando eventos de la cola con SDL_PollEvent)
    // settear el flag quit en caso de querer salir

    // actualizar estado interno, realizar acciones, etc.
}
```

1.6. Programación orientada a objetos

1. **¿Qué es la POO (Programación Orientada a Objetos)? ¿Qué ventajas ofrece su uso?**

   La programación orientada a objetos es un paradigma cuyo fundamento es el modelado de todos los componentes del sistema como "Objetos", que tienen un estado y un comportamiento. El flujo de los sistemas programados bajo este paradigma se desarrolla de acuerdo al envío de mensajes entre los objetos que lo componen.

Ventajas:

- **Descomponer un problema** Permite la división del problema de parte pequeñas. Y esto supone una gran ventaja a la hora de encarar rompecabezas complejos que con la programación funcional serian una tarea titánica y enrevesada.
- **Programas mas fácil de mantener** Favorece el mantenimiento de un programa, y resulta sencilla la tarea modificar o eliminar código.
- **Orden y legibilidad** Los código de la POO por lo general resulta más ordenado y legible. Las clases y objetos son fáciles de identificar. Y un programa resulta más comprensible en comparación a las intrincadas lineas de código que suelen predominar en la programación estructurada.

2. ¿Qué es el polimorfismo? Ejemplifique mediante código.

En programación orientada a objetos, el polimorfismo se refiere a la propiedad por la que es posible enviar mensajes sintácticamente iguales a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.

Ejemplo:

```
class Animal {
    public:
        Animal() {}
        virtual void saludar() const = 0;
};

class Perro : public Animal {
    public:
        void saludar() const {
            std::cout << "Guau! Soy un perro." << std::endl;
        }
};

class Gato : public Animal {
    public:
        void saludar() const {
            std::cout << "Miau! Soy un gato." << std::endl;
        }
};

int main() {
    Perro perro;
    Gato gato;
    std::list<Animal*> animales = {&perro, &gato};
    for (auto it : animales) {
        it->saludar();
    }

    return 0;
}
```

La salida del programa sera:

```
Guau! Soy un perro.
Miau! Soy un gato.
```

1.7. Otros

1. **¿Qué es una función callback? ¿Por qué es importante en entornos gráficos de programación? De un breve ejemplo.**

   Es una función (normalmente pasada como argumento a otra parte del código) que será ejecutada más adelante, de esta manera podría ejecutarse una porción de código en un momento determinado, cuando otra parte del programa lo requiera.

   En entornos gráficos, es muy utilizada para el manejo de eventos. Las diferentes acciones posibles están definidas por funciones que se ejecutan ante los diferentes eventos (ej, click del mouse) y se puede utilizar una *callback* para que se ejecute una acción (que es modificable únicamente cambiando qué función de callback se utiliza) como manejo del evento.

```
class Boton;
class EventQueue;
class Caller;
calss Event; // : public Caller;

EventQueue event_queue;

void boton_enivar_mensaje(Event& ev, Caller& caller) {
    Message& m = caller.get_message();
    event_queue.push(EventSendMessage(m));
}

void boton_cerrar_aplicacion(Event& ev, Caller& caller) {
    MainWindow& window = caller.get_main_window();
    window.close(0);
}
```

Se procesó desde 1C 2010 hasta los finales del 2C 2019 inclusive.

2. Ejercicios de red

Esqueletos genéricos y pasos a seguir para cualquier ejercicio de red, ya sea de un cliente o un servidor.

2.1. Aplicativo cliente

Para generar un aplicativo cliente que se conecte mediante un socket TCP a un servidor, es necesario seguir los siguientes pasos:

1. Definir a qué **host** (ip) y **puerto** (o servicio) nos queremos conectar.
2. Obtener direcciones del servidor para conectarnos utilizando **getaddrinfo**.
3. Realizar un ciclo iterando sobre las direcciones obtenidas, creando un socket con **socket** para cada una de ellas y, en caso de éxito, intentando establecer conexión con el servidor mediante **connect**.
4. Una vez conectados, realizar el ejercicio particular.
5. Liberar recursos, apagando el socket con **shutdown**, cerrándolo con **close**, y liberando la memoria de las direcciones (obtenidas previamente con *getaddrinfo*) utilizando **freeaddrinfo**.

A continuación se incluye un breve esqueleto de código C que cumple con los pasos recién explicados, considerando la mayoría de los errores posibles en el proceso:

```c
// Includes y defines necesarios para utilizar los sockets POSIX
#define _POSIX_C_SOURCE 200112L
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

// Includes propios de C
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>

// Definición de códigos de retorno
enum ret { SUCCESS = 0, BAD_USAGE, GETADDRINFO_ERR, OUT_OF_ADDRESSES,
           SEND_ERR, RECV_ERR };

int main(int argc, char* argv[]) {
    // Definir dirección host y puerto (servicio) objetivo
    if (argc != 3) {
        fprintf(stderr, "usage: %s <host> <port>\n", argv[0]);
        return BAD_USAGE;
    }

    const char *host = argv[1];
    const char *port = argv[2];

    // Obtener direcciones del servidor con getaddrinfo
    struct addrinfo hints;
```

```
    struct addrinfo *result, *ptr;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET; // AF_INET == IPv4, AF_INET6 == IPv6
    hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM == TCP, SOCK_DGRAM == UDP
    hints.ai_flags = 0; // AI_PASSIVE == Server, 0 == Client
    if (getaddrinfo(host, port, &hints, &result)) {
        return GETADDRINFO_ERR;
    }

    // Iterar sobre las direcciones obtenidas para conectarnos con connect
    int skt;
    bool connected = false;
    for (ptr = result; (ptr != NULL) && (!connected); ptr = ptr->ai_next) {
        skt = socket(ptr->ai_family, ptr->ai_socktype, ptr->ai_protocol);
        if (skt < 0) {
            continue;
        }

        if (connect(skt, ptr->ai_addr, ptr->ai_addrlen)) {
            close(skt);
        } else {
            connected = true;
        }
    }
    freeaddrinfo(result); // liberamos recursos

    // Chequear conexión, caso contrario retornar con error
    if (!connected) {
        return OUT_OF_ADDRESSES;
    }

    // Realizar el ejercicio en particular
    // ...

    // Liberar recursos con shutdown y close
    shutdown(skt, SHUT_RDWR);
    close(skt);

    return SUCCESS;
}
```

2.2. Aplicativo servidor

Para generar un aplicativo servidor que escuche conexiones entrantes en un puerto determinado mediante un socket TCP, es necesario seguir los siguientes pasos:

1. Definir en que **puerto** queremos escuchar conexiones.
2. Obtener la ínformación de la dirección en la que queremos escuchar utilizando **getaddrinfo**.
3. Crear el socket aceptador con **socket**, opcionalmente reutilizar la dirección con **setsockopt**, y realizar el **bind** (al puerto deseado) y el **listen** sobre el mismo, definiendo un número de conexiones máximas en espera de ser aceptadas.

4. Ahora podemos aceptar conexiones con **accept** para realizar el ejercicio particular que nos pidan.
5. Liberar recursos, apagando (con **shutdown**) y cerrando (con **close**) todos los sockets utilizados (el *accepter* y los *sockets peer*), y liberando la memoria de las direcciones (obtenidas previamente con *getaddrinfo*) utilizando **freeaddrinfo**.

A continuación se incluye un el esqueleto de código C que cumple con estos pasos, considerando errores:

```c
// Includes y defines necesarios para utilizar los sockets POSIX
#define _POSIX_C_SOURCE 200112L
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

// Includes propios de C
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
#include <stdlib.h>

// Definición de cantidad máxima de conexiones en espera
#define MAX_IN_QUEUE 10

// Definición de códigos de retorno
enum ret { SUCCESS = 0, BAD_USAGE, GETADDRINFO_ERR, SOCKET_ERR, SETSOCKOPT_ERR,
           BIND_ERR, LISTEN_ERR, ACCEPT_ERR, SEND_ERR, RECV_ERR };

int main(int argc, char* argv[]) {
    // Definir puerto en el que escucharemos conexiones
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        return BAD_USAGE;
    }

    const char *port = argv[1];

    // Obtener direccion con getaddrinfo
    struct addrinfo hints;
    struct addrinfo *result;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET; // AF_INET == IPv4, AF_INET6 == IPv6
    hints.ai_socktype = SOCK_STREAM; // SOCK_STREAM == TCP, SOCK_DGRAM == UDP
    hints.ai_flags = AI_PASSIVE; // AI_PASSIVE == Server, 0 == Client
    if (getaddrinfo(0, port, &hints, &result)) {
        return GETADDRINFO_ERR;
    }

    // Crear el socket acceptador con socket
    int skt = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
    if (skt < 0) {
        freeaddrinfo(result);
```

```
        return SOCKET_ERR;
    }

    // (opcional) Reutilizar dirección con setsockopt
    int val = 1;
    if (setsockopt(skt, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val))) {
        close(skt);
        freeaddrinfo(result);
        return SETSOCKOPT_ERR;
    }

    // Realizar bind al puerto determinado
    if (bind(skt, result->ai_addr, result->ai_addrlen)) {
        close(skt);
        freeaddrinfo(result);
        return BIND_ERR;
    }

    // Realizar listen especificando máximo número de conexiones en espera
    if (listen(skt, MAX_IN_QUEUE)) {
        close(skt);
        freeaddrinfo(result);
        return LISTEN_ERR;
    }

    // A modo de ejemplo, aceptamos una conexión con accept
    int peer = accept(skt, NULL, NULL);
    if (peer < 0) {
        close(skt);
        freeaddrinfo(result);
        return ACCEPT_ERR;
    }

    // Realizar el ejercicio en particular
    // ...

    // Liberar recursos con shutdown, close, y freeaddrinfo
    shutdown(peer, SHUT_RDWR);
    close(peer);
    shutdown(skt, SHUT_RDWR);
    close(skt);
    freeaddrinfo(result);

    return SUCCESS;
}
```

2.3. Comunicación con send/recv

Una vez establecida una conexión TCP, ambas partes pueden comunicarse entre sí utilizando las syscalls **send** y **recv**, cuyas firmas son:

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

En el caso del *send*, se suele utilizar el flag **MSG_NOSIGNAL**. Para el *recv*, normalmente no se utilizan flags.

Es importante mencionar que ambas syscalls devuelven -1 en caso de errores, 0 en caso de que el socket haya sido cerrado, y un número positivo en caso de que se hayan enviado/recibido bytes. Sin embargo, este número puede ser distinto del parámetro solicitado **len**, puesto que nada nos asegura que en una sóla llamada a send o a recv se envíen o se reciban los bytes pedidos. Es por esto que si queremos enviar o recibir una determinada cantidad de bytes, es necesario implementar las siguientes funciones:

```c
/* Implementación de recv_all para bloquearse hasta recibir una determinada
 * cantidad de bytes. */
ssize_t recv_all(int skt_fd, char* buf, size_t len, int flags) {
    ssize_t received = 0;
    ssize_t last_received = 0;
    while (received < len) {
        last_received = recv(skt_fd, &buf[received], len - received, flags);
        if (last_received < 0) {
            return -1;
        } else if (last_received == 0) {
            return received;
        } else {
            received += last_received;
        }
    }

    return received;
}
```

```c
/* Implementación de send_all para bloquearse hasta enviar una determinada
 * cantidad de bytes. */
ssize_t send_all(int skt_fd, const char* buf, size_t len, int flags) {
    ssize_t sent = 0;
    ssize_t last_sent = 0;
    while (sent < len) {
        last_sent = send(skt_fd, &buf[sent], len - sent, flags);
        if (last_sent < 0) {
            return -1;
        } else if (last_sent == 0) {
            return sent;
        } else {
            sent += last_sent;
        }
    }

    return sent;
}
```

2.4. Syscalls necesarias

Se detallan en esta sección los prototipos de las syscalls que deben ser conocidas para los ejercicios de red:

```c
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
```

```
                struct addrinfo **res);

// Retorna 0 en caso exitoso, -1 en caso de error.
```

```
void freeaddrinfo(struct addrinfo *res);
```

```
int socket(int domain, int type, int protocol);

// Retorna un fd válido en caso exitoso, -1 en caso de error.
```

```
int setsockopt(int sockfd, int level, int optname,
                       const void *optval, socklen_t optlen);

// Retorna 0 en caso exitoso, -1 en caso de error.
```

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

// Retorna 0 en caso exitoso, -1 en caso de error.
```

```
int listen(int sockfd, int backlog);

// Retorna 0 en caso exitoso, -1 en caso de error.
```

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);

// Retorna un fd válido en caso exitoso, -1 en caso de error.
```

```
int connect(int sockfd, const struct sockaddr *addr,
                   socklen_t addrlen);

// Retorna 0 en caso exitoso, -1 en caso de error.
```

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);

// Retorna la cantidad de bytes enviados, o -1 en caso de error.
```

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);

// Retorna la cantidad de bytes recibidos, o -1 en caso de error.
// Si el otro extremo de la conexión se cerró, retorna 0 (eof).
```

```
int shutdown(int sockfd, int how);

// how = SHUT_RD, SHUT_WR, o SHUT_RDWR
// Retorna 0 en caso exitoso, -1 en caso de error.
```

```
int close(int fd);

// Retorna 0 en caso exitoso, -1 en caso de error.
```

3. Ejercicios de archivos

Categorización de los distintos tipos de ejercicios de archivos que fueron evaluados, indicando estrategias de resolución y funciones que facilitan la implementación de las mismas.

3.1. Estrategia según tipo de ejercicio

Se suelen evaluar dos tipos de ejercicios de archivos diferentes: aquellos en los que el archivo final tenga un tamaño **mayor al original**, y aquellos en los que el tamaño sea **menor o igual**. Si bien existen múltiples formas de resolver ambos tipos de ejercicios, se propone una estrategia de implementación para cada uno de ellos:

3.1.1. El archivo se achica

Este caso es el más sencillo por dos importantes razones:

1. El archivo original ya **tiene suficientes bytes** para lo que tenemos que escribir sobre él desde un principio;
2. El puntero de escritura **siempre va por detrás** del puntero de lectura, por lo que no nos debemos preocupar por pisar datos que aun no leímos.

La estrategia entonces es muy sencilla: se llevan dos punteros, uno de lectura, y uno de escritura, y un contador de bytes (para saber el tamaño final del archivo). Se comienza a recorrer el archivo desde el principio, leyendo en el puntero de lectura un dato. Una vez que leemos el dato, decidimos si tenemos que reemplazarlo por otro dato (en este caso más pequeño) o si el dato queda como está en el archivo. En ambos casos, simplemente debemos escribir en el archivo (previamente posicionándolo en el puntero de escritura) lo que corresponda, pisando datos existentes (que ya leímos), y se incrementa el contador de bytes en los bytes escritos. Al terminar, se llama a **ftruncate** para cortar el archivo en el valor del contador de bytes finales.

3.1.2. El archivo se agranda

Este caso es más complicado, justamente por una razón análoga a las previamente explicadas: si siguieramos la misma estrategia, el puntero de escritura **se adelantaría** al puntero de lectura, puesto que cuando corresponda modificar un dato, se reemplazará por uno **de mayor tamaño** (escribiendo más bytes).

Frente a este problema, se sigue la siguiente estrategia:

1. Primero se calcula el tamaño en bytes del archivo final. Para esto, dependiendo del ejercicio puede bastar con simplemente hacer algún cálculo con el valor que devuelva **ftell** (en aquellos casos en que los datos de origen tengan tamaño fijo), o puede ser necesario recorrer el archivo de principio a fin para calcularlo.
2. Una vez que se tiene el tamaño final, se settean los punteros de escritura y lectura:

   ```
   long read_seek = original_size;      // (en bytes)
   long write_seek = final_size;        // (en bytes)
   ```

   y se llama a **ftruncate** con el tamaño final para agregar los bytes necesarios.
3. Se procesa el archivo desde el final hasta el principio, leyendo y escribiendo datos *en reversa* (es decir, el puntero de lectura/escritura se retrocede **antes** de leer/escribir, y no después).

3.2. Funciones útiles

Resulta útil para resolver estos ejercicios, definirse funciones para leer y escribir datos. Estas pueden variar entre ejercicio y ejercicio, dependiendo de si se trata de un archivo de texto o binario, y del tamaño de los archivos a leer. Sin embargo, el pseudo-código general no cambia:

```
// Pseudo-código para función de lectura

leer(archivo, puntero, origen):
    posicionar el archivo en puntero
    leer del archivo a origen y guardar retorno en aux
    actualizar puntero con ftell
    retornar aux
```

```
// Pseudo-código para función de escritura

escribir(archivo, puntero, destino):
    posicionar el archivo en puntero
    escribir destino en el archivo y guardar retorno en aux
    actualizar puntero con ftell
    retornar aux
```

```
// Pseudo-código para función de lectura en reversa

// PRE: puntero apunta al final del dato a leer
leer_reversa(archivo, puntero, origen):
    retroceder puntero en el tamanio a leer
    si el puntero es negativo, retornar (eof)
    posicionar el archivo en puntero
    leer del archivo a origen y retornar el retorno
```

```
// Pseudo-código para función de escritura en reversa

// PRE: puntero apunta al final del dato a leer
escribir_reversa(archivo, puntero, origen):
    retroceder puntero en el tamanio a escribir
    si el puntero es negativo, retornar (eof)
    posicionar el archivo en puntero
    escribir destino en el archivo y retornar el retorno
```

3.3. Syscalls necesarias

Se detallan en esta sección los prototipos de las syscalls que deben ser conocidas para los ejercicios de manipulación de archivos:

```
FILE *fopen(const char *pathname, const char *mode);

// Retorna un FILE* en caso exitoso, NULL en caso de error.
```

```
int fclose(FILE *stream);

// Retorna 0 en caso exitoso, EOF en caso de error.
```

```c
int fgetc(FILE *stream);

// Retorna un unsigned char casteado a int, o EOF ante error/eof.
```

```c
int fputc(int c, FILE *stream);

// Retorna el unsigned char escrito casteado a int, o EOF ante error/eof.
```

```c
char *fgets(char *s, int size, FILE *stream);

// Reads in at most one less than size characters from stream and stores
// them into the buffer pointed to by s. Reading stops after an EOF or a
// newline. If a newline is read, it is stored into the buffer.
// A terminating null byte ('\0') is stored after the last character.

// Retorna s en caso exitoso, o NULL en caso de error.
```

```c
int fputs(const char *s, FILE *stream);

// Writes the string s to stream, without its terminating null byte ('\0').

// Retorna un número positivo en caso exitoso, o EOF ante error.
```

```c
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

// Reads nmemb items of data, each size bytes long, from the stream
// pointed to by stream, storing them at the location given by ptr.

// Retorna el número de bytes leidos en caso exitoso, o ante error,
// un "short item count" (o el valor 0).
```

```c
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

// Writes nmemb items of data, each size bytes long, to the stream
// pointed to by stream, obtaining them from the location given by ptr.

// Retorna el número de bytes escritos en caso exitoso, o ante error,
// un "short item count" (o el valor 0).
```

```c
int fseek(FILE *stream, long offset, int whence);

// whence = SEEK_SET, SEEK_CUR, o SEEK_END.
// Retorna 0 en caso exitoso, -1 en caso de error.
```

```c
long ftell(FILE *stream);

// Retorna el offset actual en el archivo, -1 en caso de error.
```

```c
void rewind(FILE *stream);

// Equivalente a fseek(stream, 0L, SEEK_SET);
```

```c
int feof(FILE *stream);

// Retorna 0 si el flag eof no está prendido, >0 si lo está.
```

3.3.1. Syscalls no estándar

Las siguientes syscalls no forman parte del estándar de c99, es necesario definir cierto valor de las constantes **_XOPEN_SOURCE** y **_POSIX_C_SOURCE**. Sin embargo, se utilizan en prácticamente todos los ejercicios de archivos:

```c
int fileno(FILE *stream);

// Retorna el file descriptor del stream.
```

```c
int ftruncate(int fd, off_t length);

// Cause the regular file referenced by fd to be truncated to a size of
// precisely length bytes.

// If the file previously was larger than this size, the extra data is  lost.
// If  the  file previously was shorter, it is extended, and the extended
// part reads as null bytes ('\0').

// The file offset is not changed.

// Retorna 0 en caso exitoso, -1 en caso de error.
```

4. Ejercicios gráficos

Esqueletos para realizar los ejercicios gráficos de GUIs evaluados hasta el momento, ya sean de dibujar o de agregar widgets con comportamiento, utilizando Qt.

4.1. Esqueleto aplicativo genérico

A continuación se incluye un breve código C++ que permite levantar una ventana genérica con un widget personalizado utilizando Qt. Esto será suficiente para los ejercicios que nos pidan, ya que en este widget personalizado podemos dibujar, o agregar widgets pre-definidos de Qt para cumplir con los ejercicios.

```
#include <QtWidgets>

class MyWindow : public QWidget {
   protected:
    void paintEvent(QPaintEvent* event) {
        // Función que se llamará cada vez que se dibuje nuestra ventana.
        // Si queremos dibujar formas, hay que hacerlo aquí para que
        // en cada loop se vuelva a dibujar.
    }

   public:
    MyWindow(QWidget* parent = 0) : QWidget(parent) {
        // Constructor. Aquí podemos agregar distintos widgets, layouts, etc.
    }
};

int main(int argc, char* argv[]) {
    QApplication app(argc, argv);
    MyWindow w;
    w.show();
    return app.exec();
}
```

4.2. Dibujando formas

En resumen, para dibujar utilizamos el objeto **QPainter**. Le modificaremos dos atributos importantes según lo que queramos hacer:

- **El trazo**, llamando a **QPainter::setPen(QPen pen)**. Esto definirá con qué color y qué grosor se dibuja el trazo.

```
QPainter painter;
painter.setPen(QPen(Qt::red, 2));    // trazo rojo de grosor 2px
painter.setPen(Qt::NoPen);           // sin trazo
```

- **El relleno**, llamando a **QPainter::setBrush(QBrush brush)**. Esto definirá con qué color se pinta el interior de un trazo.

```
QPainter painter;
painter.setBrush(QBrush(Qt::red));    // relleno rojo
painter.setBrush(Qt::NoBrush);        // sin relleno
```

Para dibujar lineas utilizaremos **QPainter::drawLine**, para trazos más complejos utilizaremos un objeto **QPainterPath**. Para elipses, utilizaremos **QPainter::drawEllipse**.

Se incluyen a continuación distintas definiciones de la función **paintEvent** para nuestra clase **MyWindow**, con el objetivo de lograr dibujar distintas formas geométricas.

4.2.1. Dibujar líneas rectas

```
void paintEvent(QPaintEvent* event) {
    size_t w = size().width();
    size_t h = size().height();

    QPainter painter(this);
    painter.setPen(QPen(Qt::red, 1));  // trazo (color, grosor)
    painter.drawLine(0, 0, w, h);      // origen (x, y), destino (x, y)
    painter.drawLine(0, h, w, 0);

    painter.setPen(QPen(Qt::blue, 2));
    painter.drawLine(w/3, 0, w/3, h);
}
```

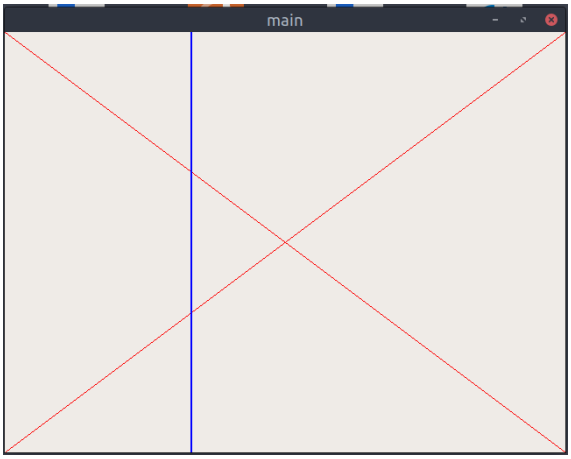

Figura 1: Ventana generada

4.2.2. Dibujar trazo (sin relleno)

```
void paintEvent(QPaintEvent* event) {
    size_t w = size().width();
    size_t h = size().height();

    QPainter painter(this);
    QPainterPath path;
    path.moveTo(0, h);
    path.lineTo(w / 2, 0);
    path.lineTo(w, h);
    path.lineTo(0, h);

    painter.setPen(QPen(Qt::blue, 4));
    painter.drawPath(path);
}
```

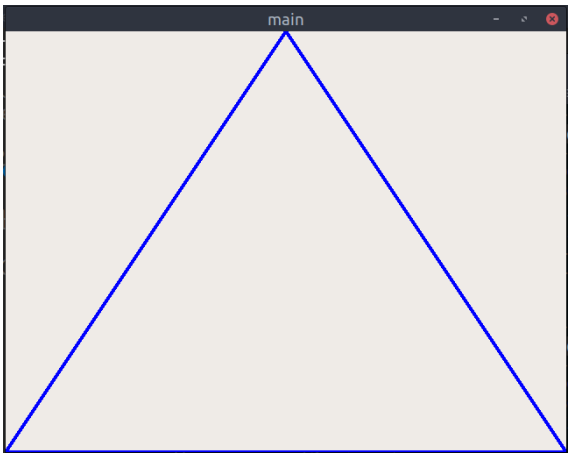

Figura 2: Ventana generada

4.2.3. Dibujar y rellenar trazo

```
void paintEvent(QPaintEvent* event) {
    size_t w = size().width();
    size_t h = size().height();

    QPainter painter(this);
    QPainterPath path;
    path.moveTo(0, h);
    path.lineTo(w / 2, 0);
    path.lineTo(w, h);
    path.lineTo(0, h);

    painter.setPen(Qt::NoPen);
    painter.setBrush(QBrush(Qt::blue));
    painter.drawPath(path);
}
```

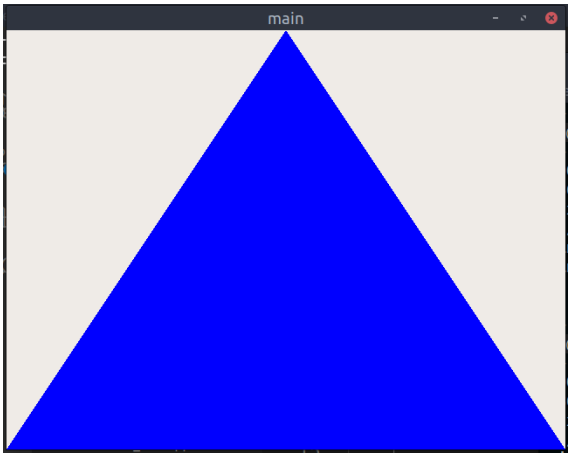

Figura 3: Ventana generada

4.2.4. Dibujar elipse (sólo trazo)

```
void paintEvent(QPaintEvent* event) {
    size_t w = size().width();
    size_t h = size().height();

    QPainter painter(this);
    painter.setPen(QPen(Qt::red, 2));
    painter.setBrush(Qt::NoBrush);
    // Podemos dibujar las elipses de dos formas: o definimos un rectángulo
    // y se dibujará una elipse dentro de este, o definimos el centro y los
    // radios de sus ejes.
    painter.drawEllipse(0, 0, w / 4,
                        h / 4);  // rectángulo desde(x, y), hasta(x, y)
    painter.setPen(QPen(Qt::blue, 2));
    painter.drawEllipse(QPoint(w / 2, h / 2), w / 3,
                        h / 3);  // centro(x, y), radio_ax, radio_ay
}
```

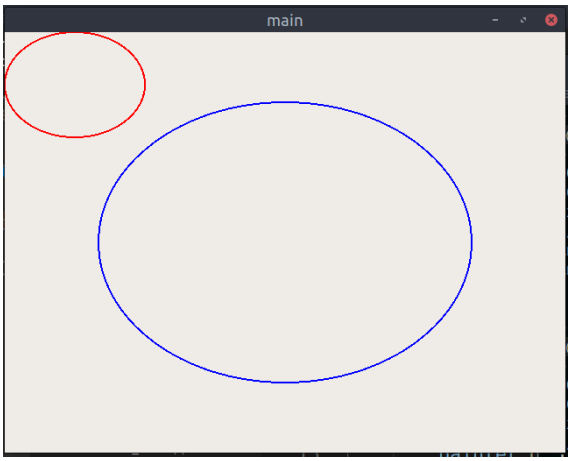

Figura 4: Ventana generada

4.2.5. Dibujar y rellenar elipse

```
void paintEvent(QPaintEvent* event) {
    size_t w = size().width();
    size_t h = size().height();

    QPainter painter(this);
    painter.setPen(Qt::NoPen);
    painter.setBrush(QBrush(Qt::red));
    // Podemos dibujar las elipses de dos formas: o definimos un rectángulo
    // y se dibujará una elipse dentro de este, o definimos el centro y los
    // radios de sus ejes.
    painter.drawEllipse(0, 0, w / 4,
                        h / 4);  // rectángulo desde(x, y), hasta(x, y)
    painter.setBrush(QBrush(Qt::blue));
    painter.drawEllipse(QPoint(w / 2, h / 2), w / 3,
```

```
                    h / 3);  // centro(x, y), radio_ax, radio_ay
}
```

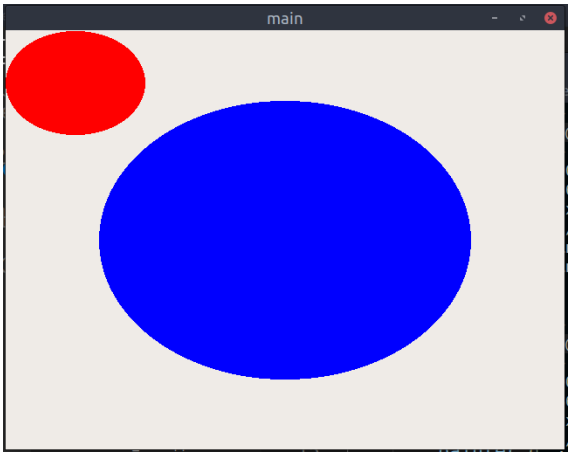

Figura 5: Ventana generada

4.3. Agregando widgets

Para agregar widgets a nuestra clase **MyWindow**, utilizaremos el constructor de la misma. Los widgets en Qt se crean dinámicamente con **new**, y al indicar a **MyWindow** como padre, esta toma el *ownership* sobre los mismos y se encarga de destruirlos en su propio destructor. Por lo tanto, para agregar un botón, basta con realizar: `QPushButton* btn = new QPushButton("Botón", this);` en el constructor. Luego, podemos modificar estos widgets con sus métodos para posicionarlos donde queramos, asignarles handlers de eventos, cambiar su geometría, agregarlos a una grilla, etc.

Los widgets más utilizados son: **QPushButton**, **QGridLayout**, **QLineEdit**, **QComboBox**, **QCheckBox** y **QMessageBox**. Se ejemplifica su uso a continuación.

4.3.1. QPushButton

Podemos indicarle un texto a contener en el constructor: **QPushButton *btn = new QPushButton("Botón", this);**, asignarle una función de callback para cuando se lo clickea mediante **connect**: **connect(btn, &QPushButton::clicked, this, &MyWindow::onButton);** (donde **onButton** debe ser un método de nuestra clase MyWindow). Finalmente podemos modificar su geometría (tamaño y posición) con: **setGeomtry(int x, int y, int w, int h);**.

Ejemplo:

```
MyWindow(QWidget *parent = 0) : QWidget(parent) {
    QPushButton *btn = new QPushButton("Botón", this);
    connect(btn, &QPushButton::clicked, this, &MyWindow::onButton);
    btn->setGeometry(10, 10, 100, 50);
}
```

4.3.2. QGridLayout

Grilla que nos permite agregar distintos widgets sin preocuparnos por su posición en la ventana. Se crea con **QGridLayout *grid = new QGridLayout(this);**, se le agregan widgets con **addWidget(QWidget* widget, int row, int column);** y se settea como layout de la ventana con **setLayout(grid)**.

Ejemplo de utilización:

```
MyWindow(QWidget *parent = 0) : QWidget(parent) {
    QGridLayout *grid = new QGridLayout(this);

    QPushButton *btn1 = new QPushButton("Botón 1", this);
    QPushButton *btn2 = new QPushButton("Botón 2", this);
    QPushButton *btn3 = new QPushButton("Botón 3", this);

    grid->addWidget(btn1, 0, 0);
    grid->addWidget(btn2, 1, 0);
    grid->addWidget(btn3, 2, 0);

    setLayout(grid);
}
```

4.3.3. QLineEdit

Los **QLineEdit** son los conocidos **TextBox**s. En ellos, el usuario puede ingresar texto. Se pueden construir de la siguiente forma: **QLineEdit *edit = new QLineEdit(this);**, y podemos acceder a su texto utilizando su método **text**, así como settearle un texto determinado con **setText**, o limpiar su contenido con **clear**.

Ejemplo de uso combinándolo con una grilla y un botón, y utilizando **findChild** para obtener el edit en el callback del botón:

```
MyWindow(QWidget *parent = 0) : QWidget(parent) {
    QGridLayout *grid = new QGridLayout(this);
    QLineEdit *edit = new QLineEdit(this);
    QPushButton *btn = new QPushButton("Clear", this);
    connect(btn, &QPushButton::clicked, this, &MyWindow::onButton);

    grid->addWidget(edit, 0, 0);
    grid->addWidget(btn, 1, 0);

    setLayout(grid);
}

void onButton() {
    QLineEdit *edit = this->findChild<QLineEdit *>();
    if (edit) {
        edit->clear();
    }
}
```

4.3.4. QComboBox

Se crea con **QComboBox *cb = new QComboBox(this);** se obtiene su tamaño con **count()**, se agregan opciones con **addItem(const char *opc)**, se eliminan con **removeItem(size_t idx)**, y se accede a una opción con **itemText(size_t idx)**.

Ejemplo de uso:

```
MyWindow(QWidget *parent = 0) : QWidget(parent) {
    QComboBox *cb = new QComboBox(this);
    cb->addItem("Opción 1");
```

```
    cb->addItem("Opción 2");
    cb->addItem("Opción 3");
}
```

4.3.5. QCheckBox

Se puede crear indicando el texto del mismo: **QCheckBox *chb = new QCheckBox("Texto", this);**, y se puede conocer su estado con el método **isChecked()**.

Ejemplo de uso:

```
MyWindow(QWidget *parent = 0) : QWidget(parent) {
    QGridLayout *grid = new QGridLayout(this);
    QCheckBox *chb = new QCheckBox("Saludar?", this);
    QPushButton *btn = new QPushButton("Botón", this);
    connect(btn, &QPushButton::clicked, this, &MyWindow::onButton);

    grid->addWidget(chb, 0, 0);
    grid->addWidget(btn, 1, 0);

    setLayout(grid);
}

void onButton() {
    QCheckBox *chb = this->findChild<QCheckBox *>();
    if (chb && chb->isChecked()) {
        std::cout << "Hola!" << std::endl;
    }
}
```

4.3.6. QMessageBox

Con **QMessageBox** podemos lanzar una ventana de diálogo. Ejemplo de uso agregándole un checkbox:

```
MyWindow(QWidget* parent = 0) : QWidget(parent) {
    QGridLayout* grid = new QGridLayout(this);
    QPushButton* btn = new QPushButton("Dialog!", this);
    connect(btn, &QPushButton::clicked, this, &MyWindow::onButton);

    grid->addWidget(btn, 0, 0);
    setLayout(grid);
}

void onButton() {
    QMessageBox msgBox;
    QCheckBox* cb = new QCheckBox("Un check box!");
    msgBox.setCheckBox(cb);
    msgBox.setText("Una ventana de diálogo.");
    msgBox.exec();
}
```