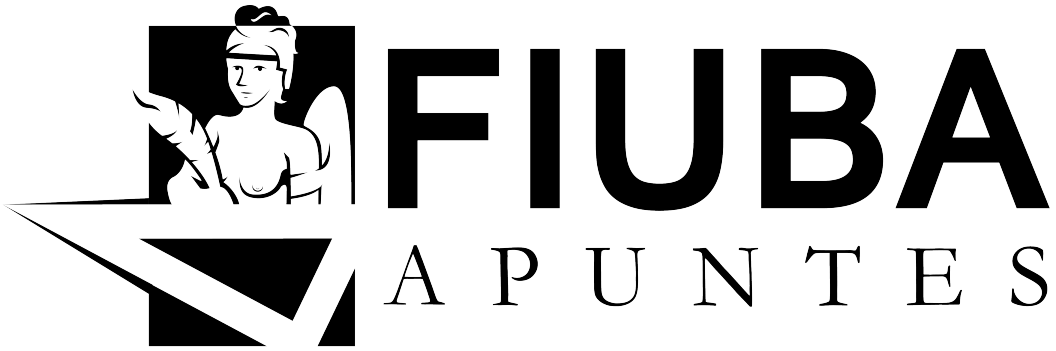

Taller de Programación I (75.42 - 95.08)

Resumen

fiuba-apuntes.github.io

Última actualización: 08/03/2015

LICENCIA

Este es un resumen (y no un sustituto) de la licencia. Este resumen destaca sólo algunas de las características clave y los términos de la licencia real. No es una licencia y no tiene valor legal. Usted debe revisar cuidadosamente todos los términos y condiciones de la licencia actual antes de usar el material licenciado.

Usted es libre para:

Compartir — copiar y redistribuir el material en cualquier medio o formato

Adaptar — remezclar, transformar y crear a partir del material

El licenciante no puede revocar estas libertades en tanto usted siga los términos de la licencia

Bajo los siguientes términos:

Atribución — Debe reconocer adecuadamente la autoría, proporcionar un enlace a la licencia e indicar si se han realizado cambios. Puede hacerlo de cualquier manera razonable, pero no de una manera que sugiera que tiene el apoyo del licenciador o lo recibe por el uso que hace.

NoComercial — Usted no puede hacer uso del material con fines comerciales.

CompartirIgual — Si usted mezcla, transforma o crea nuevo material a partir de esta obra, usted podrá distribuir su contribución siempre que utilice la misma licencia que la obra original.

No hay restricciones adicionales — Usted no puede aplicar términos legales ni medidas tecnológicas que restrinjan legalmente a otros hacer cualquier uso permitido por la licencia.

Aviso:

Usted no tiene que cumplir con la licencia para los materiales en el dominio público o cuando su uso esté permitido por una excepción o limitación aplicable.

No se entregan garantías. La licencia podría no entregarle todos los permisos que necesita para el uso que tenga previsto. Por ejemplo, otros derechos como relativos a publicidad, privacidad, o derechos morales pueden limitar la forma en que utilice el material.

Índice

Acerca del proyecto

FIUBA Apuntes nació con el objetivo de ofrecer en formato digital los apuntes de las materias que andan rondando por los pasillos de FIUBA y que los mismos sean fácilmente corregidos y actualizados.

Cualquier persona es libre de usarlos, corregirlos y mejorarlos.

Encontrarás más información acerca del proyecto o más apuntes en fiuba-apuntes.github.io.

¿Por qué usamos LaTeX?

LaTeX es un sistema de composición de textos que genera documentos con alta calidad tipográfica, posibilidad de representación de ecuaciones y fórmulas matemáticas. Su enfoque es centrarse exclusivamente en el contenido sin tener que preocuparse demasiado en el formato.

LaTeX es libre, por lo que existen multitud de utilidades y herramientas para su uso, se dispone de mucha documentación que ayuda al enriquecimiento del estilo final del documento sin demasiado esfuerzo.

Esta herramienta es muy utilizado en el ámbito científico, para la publicación de papers, tesis u otros documentos. Incluso, en FIUBA, es utilizado para crear los enunciados de exámenes y apuntes oficiales de algunos cursos.

¿Por qué usamos Git?

Git es un software de control de versiones de archivos de código fuente desde el cual cualquiera puede obtener una copia de un repositorio, poder realizar aportes tanto realizando *commits* o como realizando *forks* para ser unidos al repositorio principal.

Su uso es relativamente sencillo y su filosofía colaborativa permite que se sumen colaboradores a un proyecto fácilmente.

GitHub es una plataforma que, además de ofrecer los repositorios git, ofrece funcionalidades adicionales muy interesantes como gestor de reporte de errores.

1. Archivos

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

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

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

1.1. Primitivas

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

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

Parámetros

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

1.1.2. size_t fread(const void *ptr, size_t size, size_t nmemb, FILE *stream)

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

Parámetros

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

1.1.3. size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)

Escribe desde un buffer de datos *nmemb* elementos de tamaño *size*, hacia el flujo de datos correspondiente.
Para más información, leer la documentación de fread

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

Cambia el indicador de posición asociado al archivo.

Parámetros

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

1.1.5. long ftell(FILE *stream)

Devuelve la posición actual sobre el archivo.

2. Sockets

2.1. Primitivas

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

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

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

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

Parámetros

- **domain**: Define si la familia de protocolos de la conexión. Algunos valores usados son: PF_LOCAL (comunicación local), PF_INET (IPv4), PF_INET6 (IPv6).
- **type**: Define el tipo de conexión. Algunos de los valores valores más usados son SOCK_STREAM y SOCK_DGRAM para protocolos TCP y UDP respectivamente.
- **protocol**: Define el protocolo a utilizar, se lo puede dejar en 0 para que se elija el apropiado según el tipo de conexión.

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

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

Parámetros

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

  La estructura que se utiliza generalmente es la siguiente:

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

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

2.1.3. int listen(int sockfd, int backlog)

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

Parámetros

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

2.1.4. int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen)

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

Parámetros

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

2.1.5. int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen)

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

Parámetros

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

2.1.6. int shutdown(int sockfd, int how)

Desactiva toda o parte de la conexión.

Parámetros

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

2.1.7. int close(int sockfd)

Cierra el socket, y libera los recursos correspondientes.

Parámetros

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

2.2. Protocolos TCP/UDP

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

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

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

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

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

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

Parámetros

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

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

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

Parámetros

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

2.3. Ejemplo

2.3.1. Cliente

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring> //Necesario para el memset
#include <cstdio>

#define BACKLOG 20
#define MSG_SIZE 30

int main(int, char**){
        printf("Iniciando el cliente en la direccion 127.0.0.1:8080\n");
        int socketFd = socket(PF_INET, SOCK_STREAM, 0); //Creo el socket

        char serverAddress[] = "127.0.0.1";

        struct sockaddr_in address; //Armo los datos para conectarse
        address.sin_family = AF_INET;
        address.sin_port = htons(8080); //Seteo el puerto, en formato de red
        address.sin_addr.s_addr = inet_addr(serverAddress);
        memset(address.sin_zero, 0, sizeof(address.sin_zero));

        int connected = connect(socketFd, (struct sockaddr *) &address,
                sizeof(struct sockaddr_in)); //Me conecto a la direccion.
        if (connected != 0){
                printf("Falla al conectar\n");
                return connected;
        }

        char message[MSG_SIZE];
        int bytesRecv = 0;

        printf ("Recibiendo el mensaje...\n");
        //Le envio 30 bytes al cliente (un numero arbitrario
        while (bytesRecv < MSG_SIZE && bytesRecv != -1){
            // Agrego offsets si es que no se envia todo el mensaje
            bytesRecv += recv(socketFd, message + bytesRecv, MSG_SIZE - bytesRecv, 0);
            printf("Recibido %d bytes\n", bytesRecv);
        }
        message[29] = 0; //Cierro string

        printf ("Recibo el mensaje %s\n", message);

        shutdown(socketFd, 0); //Dejo de transmitir datos

        close(socketFd); //Cierro file descriptor

        printf("Adios, vuelvas prontos\n");

        return 0;
}
```

2.3.2. Servidor

```
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <cstring> //Necesario para el memset
#include <cstdio>

#define BACKLOG 20

int main(int, char**){
    printf("Iniciando el servidor\n");
    int socketFd = socket(PF_INET, SOCK_STREAM, 0); //Creo el socket

    struct sockaddr_in address; //Armo los datos para bindearse
    address.sin_family = AF_INET;
    address.sin_port = htons(8080); //Seteo el puerto, en formato de red
    address.sin_addr.s_addr = INADDR_ANY;
    memset(address.sin_zero, 0, sizeof(address.sin_zero));
    //Bindeo al puerto 8080
    bind(socketFd, (struct sockaddr*) &address, sizeof(struct sockaddr_in));

    listen(socketFd, BACKLOG); //Pasivo el socket

    printf("Esperando conexión...\n");
    //Acepto una conexión e ignoro la información de la misma
    int clientFd = accept(socketFd, 0, 0);
    printf("Conexión aceptada\n");

    int bytesSent = 0;

    printf("Enviando datos\n");
    //Le envío 30 bytes al cliente (un número arbitrario)
    while (bytesSent < 30 && bytesSent != -1){
        char message[30] = "Este es un mensaje prueba :D\n";
        // Agrego offsets si es que no se envía todo el mensaje
        bytesSent = send(clientFd, message + bytesSent, 30 - bytesSent, MSG_NOSIGNAL);
        printf("Recibido %d bytes\n", bytesSent);
    }
    printf("Datos enviados\n");

    shutdown(socketFd, 0); //Dejo de transmitir datos
    shutdown(clientFd, 0);

    close(socketFd);
    close(clientFd);

    printf("Adios, vuelvas prontos\n");

    return 0;
}
```

3. Threads POSIX

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

Cada thread posee su propio:

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

Pero los threads provenientes de un mismo proceso comparten:

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

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

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

3.1. Primitivas

3.1.1. pthread_create

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

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

Parámetros

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

3.1.2. pthread_join

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

Espera a que el hilo finalice y libera sus recursos.

Parámetros

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

3.1.3. pthread_exit

Firma: void pthread_exit(void *retval)

Finaliza el hilo que lo llamó. Esta función se llama implicitamente cuando termina la *start_routine*.

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

3.1.4. pthread_cancel

Firma: void pthread_cancel(pthread_t thread)

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

3.2. Mutex

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

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

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

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

O más explicitamente:

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

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

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

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

3.2.1. pthread_mutex_init

Firma: int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
Inicializa el objeto con la dirección *mutex*, y los atributos en la dirección *mutexattr*.

Parámetros

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

3.2.2. pthread_mutex_lock

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

3.2.3. pthread_mutex_unlock

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

3.2.4. pthread_mutex_destroy

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

4. Programación en C++

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

4.1. Objetos

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

4.1.1. Encapsulamiento (accesibilidad)

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

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

Ejemplo:

```
class Perro{
public:
    /**
     * Cualquier clase o función puede llamar a este método
     */
    void ladrar(); //
protected:
    /**
     * Solo instancias de Perro o de clases derivadas
     * a él pueden acceder a este atributo.
     */
    std::string raza;
private:
    /**
     * Solo instancias de Perro pueden acceder a este atributo.
     */
    int edad;
};
```

4.1.2. Herencia

C++ permite armar una jerarquía de clases, a través de la *herencia*. Una clase que hereda de otra clase, es decir, una clase *derivada*, recibe y extiende los atributos y comportamiento de la clase *base*. El tipo de herencia determina el nivel de acceso que tendrán los atributos y métodos heredados por la clase base.

Tipo de herencia	Atrib. base público	Atrib. base protegido	Atrib. base privado
Pública	Público en clase derivada	Protegido en derivada	No posee acceso
Protegida	Protegido en clase derivada	Protegido en derivada	No posee acceso
Privada	Privado en clase derivada	Privado en derivada	No posee acceso

4.1.3. Polimorfismo

Polimorfismo es la propiedad por la que dos objetos de tipo distinto respondan a un mismo mensaje de distinta forma

Métodos virtuales:
Una de las formas de utilizar polimorfismo en C++ es mediante la herencia. Una clase base posee un método público o protegido, y una clase derivada puede "sobreescribirlo", cambiando el funcionamiento del método.

En el siguiente ejemplo, la clase base "Perro"posee 3 métodos, de los cuales uno es sobreescrito por la clase derivada Doge. De esta manera, la clase derivada conserva las otras funcionalidades, pero tiene su propia versión del método ladrar.

```
class Perro{
public:
    /**
     * Método base, todos los perros ladran por defecto de la misma forma
     */
    std::string ladrar() const{
        return "guau";
    }

    /**
     * Otros métodos base
     */
    int getEdad() const{
        return this->edad;
    }

    void setEdad(int edad){
        this->edad = edad;
    }

private:
    int edad;
};

class Doge : public Perro{
public:
    /**
     * Se sobreescribe la forma de ladrar
     */
    std::string ladrar() const{
        return "wow, such code, many meme";
    }
};
```

Sin embargo, un problema que presenta este código es que si yo realizo lo siguiente:

```
Perro *perro = new Perro;
perro->ladrar(); // Devuelve "guau"
Doge *doge = new Doge();
doge->ladrar(); // Devuelve "wow, such code"
Perro *perroDoge = new Doge();
perroDoge->ladrar(); // Error! Devuelve "wow, such code" en vez de "guau"
```

La llamada a ladrar no funcionará como uno esperaría, y en vez de llamar al método en su versión de la clase derivada, llamará a la versión en la clase padre. Para forzar la llamada de la versión en la clase derivada, el método debe ser declarado como *virtual*. Cuando un método se declara virtual, se crea una tabla de métodos virtuales para la clase, la *vtable*, de manera que cada clase derivada vaya agregando entradas a la tabla con los métodos virtuales que redefinió. Así, cuando se llama a un método polimorfico en tiempo de ejecución, se accede a la tabla de métodos virtuales, se busca el método correspondiente a la instancia que lo llamó y se ejecuta.

La declaración del método quedaría de la siguiente forma:

```
class Perro{
    [...]
    virtual std::string ladrar() const;
};
```

Así, al llamar al método ladrar, las salidas serán como uno lo esperaba

```
Perro *perro = new Perro;
perro->ladrar(); // Devuelve "guau"
Doge *doge = new Doge();
doge->ladrar(); // Devuelve "wow, such code"
Perro *perroDoge = new Doge();
perroDoge->ladrar(); // Devuelve "wow, such code"
```

4.1.4. Clases abstractas e interfaces

:

Un método *virtual puro* es aquel que no tiene ninguna implementación en la clase que lo declaró. Por ejemplo, es posible querer tener una clase Figura que englobe las figuras 2d más conocidas, y es posible querer que estas clases tengan, por ejemplo, un método Area() que calcule el área de las figuras. El polimorfismo nos permitiría crear varios tipos de Figura, y que cada una tenga su forma de calcular su área, como por ejemplo, un rectángulo con base por altura, o un círculo con pi por radio al cuadrado. Sin embargo, la clase Figura, una Figura abstracta que no existe, no tiene una implementación del método Area. Frente a esta situación, una de las opciones que tenemos es que Area sea un método virtual puro, haciendo que Figura sea una clase *abstracta*.

Cuando una clase posee una clase posee uno o más métodos virtuales puros, se la llama clase abstracta. Una clase abstracta no puede instanciarse, y sus clases derivadas deben implementar los métodos virtuales puros que faltan, o también serán clases abstractas. Cuando una clase posee todos sus métodos virtuales puros, se la suele llamar *interfaz*.

4.2. Vida de un objeto

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

4.3. Patrón RAII

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

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

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

4.3.1. Ejemplo

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

```
#include <iostream>

/**
 * Una clase que representa a un buffer, mal implementado
 */
class WrongBuffer{
public:
    WrongBuffer(): buffer(0){}
    /*
     * Crea un buffer de tamaño size
     */
    void startBuffer(int size){
        if (this->buffer){
            delete this->buffer;
        }
        this->buffer = new char[size];
    }
    /*
     * Limpia el buffer
     */
    void deleteBuffer(){
        delete[] this->buffer;
    }
    /**
     * Devuelve el buffer almacenado
     */
    char* getBuffer(){
        return this->buffer;
    }
private:
    char* buffer;
};

/**
 * Implementacion mas prolija del buffer
 */
class Buffer{
public:
    /*
     * Constructor
     */
    Buffer(int size){
        this->buffer = new char[size];
    }
    /*
     * Destructor, limpia el buffer
     */
    ~Buffer(){
        delete[] this->buffer;
    }
    /**
     * Devuelve el buffer almacenado
     */
    char* getBuffer(){
        return this->buffer;
    }
private:
    char* buffer;
};

/**
 * Ejemplo de codigo sin patron RAII
 */
void badExample1(){
    WrongBuffer buff;
    buff.startBuffer(42);
    /**
```

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

4.4. Templates

Los templates son una herramienta que permite técnicas de *programación genérica*, utilizando tipos de datos como parámetros. También pueden ser utilizados para *metaprogramación*, una técnica que permite evaluar código en la etapa de compilación en vez de tiempo de ejecución. Esto genera código temporal que puede ser optimizado por el compilador, mejorando el performance de la aplicación.

4.4.1. Funciones y clases template

Las funciones template se declaran anteponiendo la palabra clave **template** y declarando el nombre de los tipos de datos variables. En el siguiente ejemplo se implementa la función *max*, para obtener el máximo entre 2 variables de un mismo tipo, y se muestra como se utiliza.

```
1  template <typename T>
2  T max(T x, T y) {
3      return x > y ? x : y;
4  }
5
6  int a = 1, b = 2;
7  float c = 5, pi = 3.14;
8
9  int result = max<int>(a, b);
10 float result2 = max<float>(c, pi);
```

La firma template ¡typename T¿declara a la "variable"T. Una vez declarada, puede ser utilizado como si fuera una clase o un tipo de dato. Luego, cuando se quiere usar una ïnstancia"de la función, se la llama poniendo el tipo de dato entre los símbolos mayor y menor, como por ejemplo max ¡int¿(a, b).

Es notable ver que el código temporal generado al utilizar un template solo se genera si es necesario. Es decir, en el ejemplo, solo se generará codigo para una función *max* para los tipos de datos *int* y *float*. Por este motivo, las funciones template **no pueden ser precompiladas**, y generalmente se definen en el mismo header con el que se distribuyen.

De manera similar, se pueden declarar clases template, que pueden tener tipos de datos de atributos variables, u otras funciones template. Por ejemplo

```
1  template <typename T>
2  class Buffer{
3  public:
4      T popData();
5      void pushData(T data);
6  private:
7      T bufferData[10];
```

```
};

Buffer<Perro> bufferPerros;
```

4.4.2. Otros parámetros

Los templates pueden recibir otros parámetros constantes que no sean tipos de datos. Por ejemplo, puedo parametrizar el largo de la clase anterior Buffer.

```
template <typename T, int Size>
class Buffer{
public:
    T popData();
    void pushData(T data);
private:
    T bufferData[Size];
};

Buffer<Perro, 20> bufferPerros;
```

Los parámetros también pueden ser punteros a función, métodos o incluso otras clases templates.

Además, los parámetros pueden ser declarados con un valor por defecto, de manera que la declaración de la clase anterior puede quedar así:

```
template <typename T, int Size = 10>
class Buffer;

Buffer<Perro> bufferChicoPerros; // Posee longitud 10
Buffer<Perro,50> bufferGrandePerros; // Posee longitud 50
```

4.4.3. Especialización

La *especialización*, como ligeramente sugiere el nombre, consiste en especializar (o redefinir) una función template para un tipo de dato particular. En el siguiente ejemplo se muestra como redefinir la función *max* para el tipo de dato bool, de forma que devuelva si alguno de los 2 es true.

```
template <>
bool max<bool>(bool x, bool y) {
    return x || y;
}
```

Nota no es necesario agregar ¡**bool**¿a la función *max*, debido a que el compilador puede deducir los argumentos del template.

4.5. Biblioteca STL

C++ nos trae muchas clases y funciones utiles en su biblioteca standard. Esta biblioteca, con templates con contenedores, algoritmos de ordenamiento, etc. está programada con templates, y se la conoce como *STL* o *Standard Template Library*. Algunos de los contenedores más usados en la STL son

- **Vector**: Representa un array de "longitud variable".
- **List**: Representa una lista enlazada.
- **Set**: Representa un arbol binario (de la variante rojo-negro).
- **Map**: Representa un arbol binario, con nodos que almacenan un par clave valor.

Cada uno de estos contenedores presenta un *iterador*, que permite recorrer los elementos de la lista secuencialmente.

Una de las ventajas de utilizar la biblioteca STL es que no es necesario volver a implementar código ya existente, evitandose también posibilidad de errores y pudiendo utilizar código ya optimizado para ciertas estructuras (por ejemplo, la clase vector tiene una versión optimizada para el tipo bool, en el que utiliza un bit para cada elemento del vector). Otra ventaja es que las clases de la STL es que respetan el patron RAII, lo que nos ayuda a mantener control de la memoria dinámica utilizada. Además, son a prueba de excepciones, por lo que podemos utilizarlas en un entorno donde se utilizan excepciones.

4.5.1. Ejemplo

```
#include <vector>
#include <iostream>

int main(){
    // Declaro un vector dinámico para almacenar ints
    std::vector<int> vectNumeros;
    // El vector se inicia vacío
    std::cout << "El tamaño de mi vector es de " \
         << vectNumeros.size() << std::endl;
    // Agrego 3 números, uno a uno.
    vectNumeros.push_back(0);
    vectNumeros.push_back(3);
    vectNumeros.push_back(5);
    // El tamaño final del vector es 3
    std::cout << "El tamaño de mi vector es de " \
         << vectNumeros.size() << std::endl;
    // Declaro un iterador para recorrer el vector secuencialmente.
    std::vector<int>::iterator it;
    for (it = vectNumeros.begin(); it != vectNumeros.end(); ++it){
        //Muestro en pantalla el elemento sobre el que estoy iterando.
        std::cout << "Iterando sobre " << *it << std::endl;
    }
}
```

Bibliografía

[1] Bruce Eckel, *Thinking in C++, Volume 1*, 2nd Edition, January 13, 2000.

[2] Brian "Beej Jorgensen" Hall, *Beej's Guide to Network Programming*, Version 3.0.15, July 3, 2012.

[3] Pablo Roca, Gonzalo Merayo *Apuntes oficiales de la cátedra.*

Colaboradores

Quienes se mencionan a continuación han colaborado y aportado tanto al proyecto FIUBA Apuntes como en este apunte, redactándolo, corrigiéndolo, agregando gráficos, etc.

- Matías Lafroce (mlafroce@gmail.com)

¿Querés colaborar en el proyecto? Conocé más sobre el proyecto en fiuba-apuntes.github.io.

Historial de cambios

03/01/2015 Resumen de sockets completo.

04/01/2015 Resúmenes de archivos y threads completos.

07/01/2015 Patrón RAII e introducción a programación en C++.

08/03/2015 Polimorfismo, templates y STL