

# El libro de la vida

Carlos Malanche & Víctor Milchorena

20 de noviembre de 2018

Aquí está todo el material que vimos en el curso. Espero les sea una útil guía de estudio, y en su caso hasta una buena referencia en el futuro para las clases que tomarán después.

*Carlos & Víctor.*

# Índice

<b>1. Historia de la computación</b>	<b>8</b>
1.1. Algoritmos	8
1.2. La primera máquina programable	8
1.2.1. Máquina analítica	8
1.3. Máquina de Turing	9
1.3.1. Un poco de Turing	9
1.3.2. Ahora sí, la <i>máquina</i>	9
1.4. Arquitectura de Von Neumann	10
1.5. La historia de Unix	10
1.5.1. La estructura de <i>Unix</i>	11
1.5.2. El movimiento del software libre	11
1.5.3. La vieja confiable: <i>Linux</i>	11
1.5.4. Mención curiosa	11
1.6. ¿Y MS-DOS?	12
<b>2. Sistemas Unix</b>	<b>13</b>
2.1. Introducción	13
2.1.1. WSL ( <i>Windows Subsystem for Linux</i> ) en Windows	13
2.1.2. Cmder en Windows	14
2.2. La estructura de una computadora, según Unix	14
2.3. Navegando entre archivos	14
2.3.1. Cambio de directorio	15
2.3.2. Antes de seguir	16
2.3.3. Creación de archivos y directorios	16
2.3.4. Tipos de archivo y permisos	17
2.3.5. Tuberías (y rellenando archivos)	18
2.3.6. Por último, mover y copiar	20
<b>3. Más comandos útiles</b>	<b>21</b>

3.1.	El Usuario	21
3.2.	El Superusuario	21
3.3.	Búsqueda de archivos	22
3.4.	Edición de archivos	22
3.4.1.	Truco: ejecución de comandos desde <b>vim</b>	23
3.4.2.	Otras funciones y comandos	23
3.4.3.	Ayuda	24
3.5.	Uso avanzado de <b>bash</b>	24
3.5.1.	Monitor de recursos	24
3.5.2.	Matando un proceso	24
3.5.3.	Procesos de fondo	25
3.5.4.	Una calculadora	26
3.5.5.	El mejor comando del mundo, <b>ssh</b>	26
3.5.6.	<i>Wildcards</i>	27
4.	<b><i>Shell scripting y awk</i></b>	<b>28</b>
4.1.	<i>Shell scripting</i>	28
4.1.1.	Variables en <i>shell scripting</i>	28
4.1.2.	Argumentos del comando	29
4.1.3.	Instrucciones básicas dentro de un <i>script</i>	29
4.2.	<b>awk</b>	31
4.2.1.	Columnas	32
4.2.2.	<b>BEGIN</b> y <b>END</b>	33
4.2.3.	Condiciones de línea	34
4.2.4.	<i>Pattern Matching</i>	35
4.3.	Conclusión	36
4.4.	Anexo: Escape de caracteres	36
5.	<b>C, la máquina</b>	<b>37</b>
5.1.	Lenguajes de programación	37
5.1.1.	Definición	37

5.1.2.	Compuertas lógicas	37
5.1.3.	Una suma	39
5.1.4.	Ensamblador, <b>Assembly</b>	40
5.1.5.	Lenguajes de alto nivel	40
5.2.	Un lenguaje de alto nivel: <b>C</b>	41
5.2.1.	Los 4 pasos de la compilación	42
5.2.2.	La estructura del lenguaje	43
5.2.3.	Instrucciones	43
5.2.4.	Variables	44
<b>6.</b>	<b>Más cosas de C</b>	<b>46</b>
6.1.	Funciones	46
6.1.1.	La función <b>printf</b>	46
6.1.2.	Operaciones comunes	47
6.1.3.	Múltiples condiciones	48
6.2.	Instrucciones básicas	48
6.2.1.	<b>if ...else</b>	48
6.2.2.	<b>for</b>	48
6.2.3.	<b>while</b>	49
6.3.	Más variables, <i>dirección de memoria</i>	49
6.3.1.	Apuntadores, <b>pesadilla inminente</b>	49
6.4.	<i>Casting</i> (no encontré traducción alguna)	51
6.5.	Variable global y local	51
6.6.	¿Y los archivos <b>.h</b> ?	52
6.7.	Recibiendo información	52
6.7.1.	Lectura por <b>stdin</b>	52
6.7.2.	Lectura y escritura de archivos	53
6.8.	<i>Arrays, malloc, free y sizeof</i>	54
6.9.	Funciones matemáticas	55
6.9.1.	$\pi$	55

<b>7. C++</b>	<b>56</b>
7.1. Programación orientada a objetos	56
7.2. Las pequeñas diferencias	56
7.3. Clases en C++	56
7.3.1. La palabra <code>this</code>	57
7.3.2. Constructor	57
7.4. Variables estáticas y <i>namespaces</i>	58
7.4.1. ¿Y las variables estáticas?	58
7.5. Cadenas de caracteres	59
7.5.1. La función <code>to_string</code>	59
7.5.2. Conversión inversa	59
7.6. Escritura y lectura	60
7.6.1. Lectura de <code>stdin</code>	60
7.6.2. Lectura y escritura de archivos	60
7.7. Estructuras de datos	61
7.7.1. <code>std::vector</code>	61
7.7.2. <code>std::set</code>	62
7.7.3. No son los únicos	62
<b>8. Graficando con C++</b>	<b>63</b>
8.1. <code>matplotlib.cpp</code>	63
8.1.1. Dependencias	63
8.1.2. Un código de ejemplo	63
<b>9. Números Pseudoaleatorios</b>	<b>65</b>
9.1. Para qué	65
9.2. Cómo	65
9.2.1. Método de Lehmer	65
9.2.2. Números reales	66
9.2.3. Cambiando el rango	66
9.3. Generación de números pseudoaleatorios en C/C++	67

9.3.1. C . . . . .	67
9.3.2. C++ . . . . .	67
9.3.3. Números no deterministas . . . . .	67
9.4. Un experimento chistoso . . . . .	67
<b>10.Aplicación I</b>	<b>71</b>
10.1. Integración estocástica ( <i>Monte Carlo</i> ) . . . . .	71
10.2. Anexo teórico . . . . .	72
<b>11.Aplicación II</b>	<b>75</b>
11.1. Regresión Lineal . . . . .	75
11.1.1. Modelos lineales . . . . .	75
<b>12.Aplicación III</b>	<b>78</b>
12.1. Búsqueda de raíces en funciones . . . . .	78
12.1.1. Método de la bisección . . . . .	78
<b>13.Aplicación IV</b>	<b>80</b>
13.1. Integración numérica de ecuaciones diferenciales ordinales . . . . .	80
13.1.1. Derivación numérica . . . . .	80
13.1.2. Una ecuación diferencial . . . . .	81

# 1. Historia de la computación

## 1.1. Algoritmos

Un algoritmo es una serie de pasos a seguir para encontrar la solución de un problema. El concepto no sólo se usa en computación, pero es probablemente en computación en donde más estudiado se tiene el tema.

Dado que las computadoras tienen recursos limitados, los algoritmos han sido un tema de investigación fuerte pues comprar un procesador rápido no es la única manera de acelerar un proceso. Se pueden hacer las cosas con inteligencia. Supongamos el siguiente problema

**Entrada:** Un número natural  $n$   
**Salida:** El resultado de la operación  $1 + 2 + \dots + n$

Pues bien, existen al menos dos maneras de hacer esto. La primera, la fuerza bruta. Sumamos los números del 1 a  $n$ . Esto se refleja en  $n$ -operaciones. Pero ¿No habrá una manera de hacer esta operación más rápidamente? La respuesta es sí.

Podemos simplemente utilizar la fórmula

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Esto se puede concretar en 3 operaciones, una suma, una multiplicación y una división, **independientemente** del valor de  $n$ . Esto es glorioso, un problema cuya complejidad de solución no depende de la entrada.

Más adelante en el curso veremos un poco del estudio de algoritmos y su eficiencia (espacial y temporal), lo importante ahora es que les quede clara la importancia del estudio de los algoritmos. En especial para el cómputo científico, hay muchos algoritmos que necesitan correr en *clusters* de miles de procesadores para poder obtener una respuesta en un tiempo razonable (a veces semanas). Si bien el desarrollo de estos algoritmos no es tarea de los *físicos* necesariamente, el entendimiento del tema permite que no caigamos en errores básicos a la hora de programar nuestras simulaciones o hacer análisis de datos.

## 1.2. La primera máquina programable

Charles Babagge, que era un ingeniero inglés, es al que se le considera el padre de la computación. Fue por allá de 1819 que comenzó a diseñar una máquina capaz de hacer cálculos. Esta máquina estuvo completada, anunciada y publicada en 1822. Lamentablemente, Babagge no vivió para ver su idea funcionando como él la había diseñado, fue hasta 2002 que se hizo una implementación real de su máquina de diferencias.

La mecánica de fondo es algo conocido como el método de diferencias, atribuido a Newton. Este método permite la reconstrucción y evaluación de polinómios en diversos puntos partiendo del conocimiento de algunos puntos. La máquina funcionaba mecánicamente, sin electricidad, por lo que se tenía que dar vueltas a una manivela para realizar cálculos. EL gobierno británico le financió el proyecto pero no estimaron la ingeniería necesaria para concretar el proyecto, y eventualmente se dejó morir (el proyecto). Esto también por el desinterés mismo que mostró Babagge ante el proyecto después de darse cuenta que un concepto mucho más general era posible, la *máquina analítica*.

### 1.2.1. Máquina analítica

Fue hasta 1837 que Babagge diseñó la máquina analítica (que en realidad comenzó en 1833), una máquina que contaría con un *CPU abstracto*, en el sentido de que no era localizable dentro de la máquina y más bien era el resultado de la mecánica de su dispositivo, memoria (1000 números con 40 dígitos decimales, masomenos 16.2kB) y *control de flujo*, lo que significa que se podían tomar decisiones y hacer ciclos, según se le pidiera a la máquina. Con estos elementos, la máquina es *Turing-completa* (cosa que veremos un poco más adelante).



## 1.3. Máquina de Turing

### 1.3.1. Un poco de Turing

Alan Turing fue un matemático inglés que nació en 1912. Para los que hayan visto *The Imitation Game*, seguramente ya conocerán una de sus más grandes atribuciones: haber *roto* el código enigma que utilizaban los Nazis para comunicarse.

**Nota:** Debo admitir que yo no he visto tal película, pero sólo de ver el *trailer* me doy cuenta que no debe ser un buen espejo de la realidad: *Turing* parece el estereotípico genio que no tiene sentido del humor y es un tanto introvertido, cuando en las biografías escritas describen a Turing como alguien muy alegre. En fin, sólo es para que tengan en cuenta que ver la película no es saber su historia.

Este logro fue conseguido con ayuda de un equipo polaco conocido como Biuro Szyfrów (el servicio secreto polaco) quienes ya habían logrado con anterioridad descifrar varios mensajes encriptados.

En 1952 Turing fue encontrado muerto en su casa. Dicen que fue un suicidio pues fue envenenado por cianuro, pero al no haber cartas de despedida y al saber que él desarrollaba experimentos que involucraban cianuro, la conclusión es aún menos certera.

### 1.3.2. Ahora sí, la máquina

Uno de los logros teóricos más grandes de Turing fue describir la conocida como *máquina de Turing*, que da pauta a todo lo que conocemos hoy.

**Nota:** Las computadoras que utilizamos y las máquinas de Turing son equivalentes pero tienen diferencias. El funcionamiento de una máquina de Turing **no** es el funcionamiento de su computadora, a eso iremos con el modelo de Von Neumann.

La máquina de Turing consiste, físicamente, de 2 elementos. Un cabezal, que a su vez contiene la máquina, y una cinta con posiciones. El cabezal puede leer y escribir en la cinta, y puede moverse a la derecha o a la izquierda de la misma para seguir leyendo caracteres.

Una máquina de turing, de manera formal, se puede describir con 4 elementos. Estos elementos son

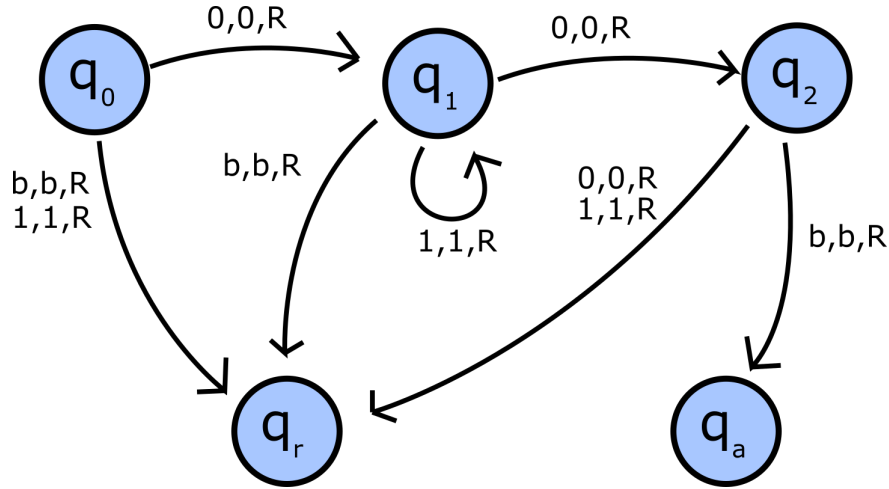
- $\Gamma$ : Un alfabeto. Esto quiere decir, un conjunto finito de símbolos. Estos sirven para la comunicación entre la máquina y quien la utiliza.
- $\Sigma$ : El conjunto de símbolos de entrada, especificado como una cadena de caracteres que contiene únicamente símbolos del alfabeto.
- $Q$ : Un conjunto  $Q$  que contiene *estados*  $q_0, \dots, q_{n-1}$ . Estos estados son propios de la máquina y determinan el comportamiento de la misma.
- $\delta$ : Una función de transición  $\delta$  que podríamos decir está definida como  $\delta : \Gamma \times Q \rightarrow \Gamma \times \{R, L\} \times Q$ , esto quiere decir, la máquina toma un par de argumentos que vienen del alfabeto y del conjunto de estados, y regresa un símbolo del alfabeto, una *dirección* ( $R$  para *right*,  $L$  para *left*) y un nuevo estado.

Si nos ponemos muy estrictos, hay otras 3 *definiciones* para la máquina de Turing

- $b$ : conocido como elemento blanco, que está en  $\Gamma$ , es el único elemento que se puede repetir un número infinito de veces en la cinta
- $F$ : Conjunto de estados *finales*
- $q_0$ : Estado inicial de la máquina de Turing (obviamente contenido en  $Q$ )

Este concepto abstracto de máquina es en lo que están fundamentadas todas las computadoras de hoy en día.

Vamos a ver un ejemplo de una máquina de Turing. En este caso, vamos a diseñar una que no tiene necesidad de escribir en la cinta, y que detecta las expresiones regulares de la forma  $01^*0$  (esto quiere decir, detecta si una cadena dada es de la forma 010, 0110, 01110, etc. o no).



En el caso de este sencillo ejemplo, tenemos  $Q = \{q_0, q_1, q_2, q_a, q_r\}$ ,  $F = \{q_a, q_r\}$ ,  $\Gamma = \{0, 1, b\}$ ,  $\Sigma = \{0, 1\}$  y la más compleja función de transición

$$\delta(0, q_0) = (0, R, q_1)$$

$$\delta(1, q_0) = (1, R, q_r)$$

$$\delta(b, q_0) = (b, R, q_r)$$

$$\delta(0, q_1) = (0, R, q_2)$$

$$\delta(1, q_1) = (1, R, q_1)$$

$$\delta(b, q_1) = (b, R, q_r)$$

$$\delta(0, q_2) = (0, R, q_r)$$

$$\delta(1, q_2) = (1, R, q_r)$$

$$\delta(b, q_2) = (b, R, q_a)$$

$q_r$  es el estado que indica que la cadena fue rechazada, es decir que no tiene el formato  $01^*0$ , y  $q_a$  indica que la cadena es aceptada.

## 1.4. Arquitectura de Von Neumann

Descrita en 1945 por el matemático y físico John Von Neumann, la también conocida como arquitectura de *Princeton* describe los componentes necesarios para obtener una computadora. De manera abstracta, estos son los 4 componentes

- Una unidad de proceso: Esta contiene un ALU (*Arithmetic Logic Unit*) y los registros del procesador (memoria de rápido acceso, generalmente poca capacidad).
- Una unidad de control: Este elemento le dice al procesador (osea al ALU y a la memoria) como actuar ante las instrucciones mandadas por los programas. Provee sincronía y precisión de ejecución.
- Una memoria: en esta se almacena información e instrucciones. Es importante que ambas cosas estén en la misma memoria.
- Almacenamiento externo: Una unidad en que se pueden guardar datos en grandes cantidades para poder moverse a la memoria de la computadora cuando sea necesario.
- Dispositivos de entrada y de salida: Comúnmente algún monitor y un teclado. Le permiten al usuario comunicarse con la computadora y darle instrucciones a la unidad de control.

## 1.5. La historia de Unix

A finales de los años 60, el MIT, AT & T Bell Labs y General Electric tenían un proyecto en común conocido como *Multics*. Este proyecto tenía como meta crear un sistema que permitiera el uso de muchos usuarios de manera concurrente. Dado que el

progreso del proyecto no fue lo suficientemente rápido, Bell Labs decidió dejar de participar en él.

Envueltos en el proyecto original estaba Ken Thompson, quien decidió empezar una especie de proyecto personal *derivado* de *Multics*. Este proyecto eventualmente se convirtió en proyecto de Bell Labs, a cargo de Ken Thompson y Dennis Ritchie, y junto con un equipo de empleados de Bell Labs desarrollaron el principio de *Unix*.

En 1970 este sistema operativo existía bajo el nombre *Unics* (*Uniplexed Information and Computing Service*), pero el sistema no era multitarea aún. Esto se suponía era una burla al nombre *Multics*, sugerida *aparentemente* por *Brian Kernighan*. No hay autoría de la transición del nombre *Unics* a lo que tenemos hoy, *Unix*.

*Unix* se fue convirtiendo en una herramienta importante de la compañía, y los elementos que faltaban se fueron agregando poco a poco (herramientas de procesamiento de palabras, manual de ayuda, etc.). Todo esto fueron desarrollos internos de la compañía y *Unix* no vio la luz pública hasta 1973. El acceso al sistema operativo fue limitado de todos modos pues involucraba un costo muy elevado.

#### 1.5.1. La estructura de *Unix*

Unix está formado por 3 elementos básicos:

- **Kernel:** Esto es básicamente el sistema operativo, es el control de la máquina, es la unidad de control y quien coordina las tareas del sistema.
- **Shell:** Es el intérprete de comandos. Permite la interacción entre el usuario y la computadora.
- **Utilidades:** Es todo el conjunto de programas que no son parte del Kernel pero que ayudan a realizar muchas tareas, como editar archivos u ordenarlos o graficar o cualquier cosa por el estilo.

#### 1.5.2. El movimiento del software libre

En 1983, Richard Stallman comienza el proyecto GNU que tenía como principal objetivo el hacer un sistema como *Unix* pero disponible de manera libre a todos. Hubo esfuerzos muy grandes y digamos que se estaba armando el sistema operativo de los sueños de GNU pero el kernel nunca obtuvo la suficiente popularidad. Sin embargo, las piezas del sistema operativo estaban ya disponibles siguiendo la *licencia pública general de GNU* (*GNU GPL, general public license*).

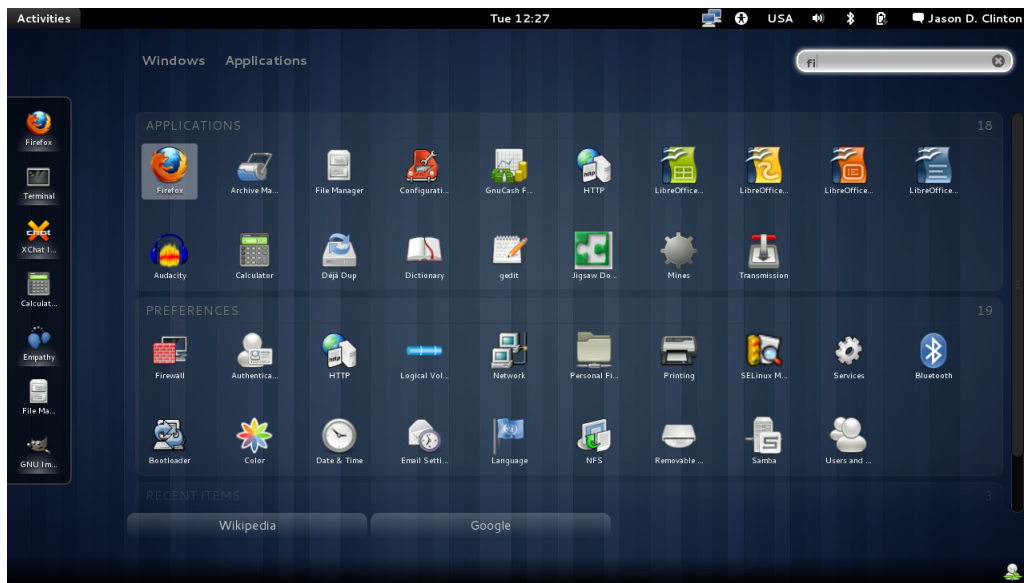
#### 1.5.3. La vieja confiable: *Linux*

Ahora, yendo a 1991, comienza la historia de *Linux*, un sistema operativo libre que comenzó como un proyecto personal de *Linus Trovalds* (estudiante finés de la universidad de Helsinki). Curiosamente, era un proyecto personal con el que Linus no esperaba obtener fama ni nada, pero dos años más tarde con la adición de una interfaz gráfica, el proyecto se popularizó.

**Dato curioso:** El nombre Linux obviamente tiene que ver con Linus y la x final que *Unix* tenía, así como la fonética de *Multics*. Linus había originalmente descartado ese nombre pues lo consideró demasiado egocéntrico, y originalmente el quería que se llamara *Freax*, alguna especie de combinación entre *free*, *freak* y la distintiva *x* de *Unix*.

#### 1.5.4. Mención curiosa

Un exalumno de esta facultad, Miguel de Icaza, diseñó en conjunto con Federico Mena un entorno de escritorio llamado GNOME.



Si bien la distribución más conocida hasta ahora de *Linux*, *Ubuntu*, viene con *Unity*, hay muchos elementos de esta interfaz que son compartidos con GNOME (*Nautilus* como explorador de archivos por ejemplo).

## 1.6. ¿Y MS-DOS?

A pesar de que el temario pide conocimiento de MS-DOS, lo vamos a ignorar, porque hoy en día *nadie* usa MS-DOS. MS-DOS es una alternativa a Unix, creada por IBM y Microsoft, pero que se queda corta. No es multitarea, no es multiusuario, sólo hay una versión libre (*FreeDOS*), es oficialmente obsoleto, sólo existe en 32 bits y la lista sigue y sigue. Sólo hacemos una mención para que sepan qué es. Si quieren ver algo *parecido* a MS-DOS, abran una aplicación de windows llamada *command prompt*. **No es MS-DOS**, lo fue hasta antes de Windows XP, una versión de Windows que salió cuando ustedes nacieron, en el 2000. Ahora es otra aplicación pero que aún guarda muchas similitudes.

*Carlos & Víctor*

## 2. Sistemas Unix

### 2.1. Introducción

Sacado directamente de Wikipedia, Unix es una familia de sistemas operativos multiusuario y multitarea. Son caracterizados por poseer una *estructura modular* conocida como *la filosofía de Unix*. Esta filosofía se resume en los siguientes tres puntos (Peter H. Salus en *A Quarter-Century of Unix (1994)*):

- Escriba programas que hagan una cosa y que la hagan bien.
- Escriba programas que trabajen bien juntos
- Escriba programas que manejen cadenas de caracteres, porque ésa es una interfaz universal.

*MacOS* y las distribuciones de *Linux* (como Debian, Ubuntu, Fedora, etc.) no han dejado esta mentalidad y permiten el uso de algo conocido como la *terminal*, que pareciera un remanente de los principios del cómputo, cuando las interfaces gráficas eran realmente inexistentes.

En el caso de MacOS basta con abrir *Spotlight* (`cmd + barra espaciadora`) y escribir *terminal*. Para *Linux*, en general basta con buscar la palabra *terminal* en la herramienta de búsqueda del sistema (o presionar `ctrl+alt+t` en Debian ó Ubuntu).

En el lamentable caso de Windows, no existe tal cosa, pues ellos tomaron otra ruta y desarrollaron otro sistema conocido como MS-DOS. Para decirlo de manera sencilla, es una lástima pues no podemos ejecutar los mismos comandos en MS-DOS que en la terminal de UNIX. Sin embargo hay maneras de darle la vuelta a este problema.

Si bien podríamos pasar las primeras semanas hablando de Unix y MS-DOS, sería una verdadera pérdida de tiempo, pues al menos las 500 supercomputadoras más rápidas del planeta corren alguna distribución de *Linux*. La gran mayoría de los servidores del planeta corren *Linux*. El éxito de Windows (hasta ahora) ha sido mayormente en ambientes de oficina y hogares.

Es por esta razón que vamos a irnos directo a Unix.

**Nota:** En caso de sólo tener windows, lo que recomendamos en el curso es que hagan una instalación conocida como *Dual-Boot* con la que pueden tener Windows y Linux (preferentemente Ubuntu en su versión 18.04LTS) sin que uno interfiera con el otro. Lo único que sacrificarían sería espacio en su unidad de almacenamiento (sea SSD o HDD), y ganan la experiencia de saber utilizar otro sistema operativo.

Aún así, sabemos que hay casos en que les es imposible hacer esto (protecciones de la tarjeta madre, complicaciones por las configuraciones del BIOS, o simplemente porque la computadora no es suya y tienen miedo de estropearlo todo), y para ello están las siguientes instrucciones, con lo que pueden tener el poder ~~completo~~ casi completo de Linux en Windows.

En todo caso estamos para ayudarles con estos problemas técnicos porque son requisitos fundamentales para lo que queda del curso. Osea, todo el curso.

#### 2.1.1. WSL (*Windows Subsystem for Linux*) en Windows

Desde masomenos mayo del presente año, salió de su fase *beta* y se puede instalar fácilmente una herramienta conocida como *subsistema de Windows para Linux*. Las instrucciones las pueden encontrar en [esta página](#). Basta con ejecutar un comando en *Powershell* (algo así como MS-DOS recargado) e instalar directamente de la tienda de aplicaciones de Windows su subsistema de preferencia. Recomendamos que sea Ubuntu.

Este subsistema es algo así como un intérprete de alto rendimiento. Se encarga de convertir las instrucciones de Unix a instrucciones que Windows sabe como manejar, el rendimiento comienza a ser competitivo con una instalación normal de *Ubuntu*, pero no es un soporte 100 % nativo de todos modos.

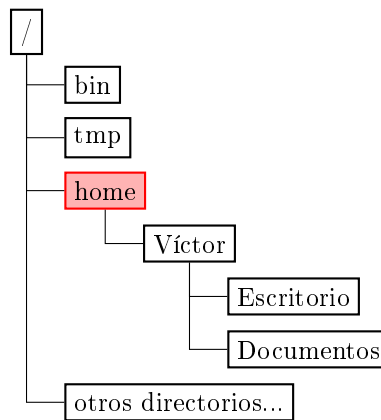
### 2.1.2. Cmder en Windows

Si ya de plano ni eso quieren hacer, o quieren el **soporte nativo en Windows** de lo que usaremos en el curso, entonces ésta es su mejor opción.

Cmder es un programa que se puede descargar [aquí](#), cuya meta es *emular* la funcionalidad de una terminal en Unix. Por ello mismo, tiene sus propias versiones de los comandos más utilizados en Unix (los veremos en unos momentos). Si planean irse por esta opción, descarguen la versión *mini*. En todo caso, **esto es lo que menos recomendamos** porque va a dificultarles entender la estructura de Linux cuando inicien sesiones remotas.

## 2.2. La estructura de una computadora, según Unix

Para Unix, todo en su computadora son archivos (todo, sus memorias USB son un archivo que a su vez contiene archivos, sus bocinas son un archivo, todo es un archivo), y cada archivo es el nodo de un gran árbol. Esto se sigue de la mentalidad de que la interfaz universal son las cadenas de caracteres.



Una buena explicación de los elementos principales se encuentra en [wikipedia también](#). Pero vamos a discutir los principales:

Nombre	Descripción
/	Directorio raíz
/bin	Contiene los archivos binarios, es decir, los ejecutables principales de Unix
/home	Es el directorio que contiene a los usuarios y sus archivos personales
/usr	Contiene más ejecutables que no son críticos para el sistema, además de bibliotecas.
/media	Los medios extraíbles aparecen dentro de este directorio.
/tmp	Archivos que no van a sobrevivir un reinicio.

Un archivo en particular que puede ser de uso es `/dev/null`, `dev` se refiere a *devices*. Este archivo se puede usar como basurero (cuando la salida de un programa no la queremos ver, podemos dirigirla a `/dev/null`).

Esta estructura tiene un símil en Windows (masomenos), donde el directorio raíz comúnmente es conocido como `C:\`

**Nota:** si están utilizando `cmder`, entonces van a poder usar la diagonal misma de Unix, `/`. `Cmder` entiende que se refieren a un *backslash*.

## 2.3. Navegando entre archivos

Bien, ahora si pasamos a movernos a través del árbol de archivos. Al abrir una terminal, se colocará en un nodo del árbol de archivos. Por lo general este directorio será *home*, con lo que nos referimos al directorio que les es asignado como usuarios.

Pueden verificar el directorio en el que se encuentran escribiendo `pwd` (que significa *path of working directory*) y presionando **enter** después.

```
carlos@peregrine: ~  
carlos@peregrine:~$ pwd  
/home/carlos
```

Así de simple. Pueden ver que yo me encontraba en el directorio `/home/carlos`. De paso, explico lo que significa lo que está escrito en verde: **carlos** es el nombre de usuario con el que me encuentro en el equipo (la computadora vaya) bajo el nombre **peregrine**. Después de los puntos viene una tilde `~` que significa que estoy en el directorio `/home/carlos` (es como un *sinónimo*, si así lo quieren ver. Un *alias*)

`pwd` es un comando, la manera en que un comando se ejecuta en Unix es la siguiente

$$\underbrace{\text{ls}}_{\text{comando}} \underbrace{-l}_{\text{opciones}} \underbrace{./\text{Documentos}}_{\text{argumentos}}$$

El comando `ls` enlista los archivos dentro de un directorio. En este caso, la opción `-l` le dice que lo haga en forma de lista (todas las opciones son anteceditas por un guión), y el argumento `./Documentos` dice que enliste los archivos dentro del folder **Documentos**, que debiera localizarse en el nodo actual del árbol de archivos (en un momento vamos a eso). Para el caso de `pwd`, no hay ni argumentos ni opciones, es sólo el comando.

```
carlos@peregrine: ~  
carlos@peregrine:~$ ls -l  
total 60  
drwxrwxr-x 3 carlos carlos 4096 Dez 10 2017 Android  
drwxrwxr-x 4 carlos carlos 4096 Dez 16 2017 AndroidStudioProjects  
drwxr-xr-x 5 carlos carlos 4096 Dez 16 2017 Desktop  
drwxr-xr-x 3 carlos carlos 4096 Dez 16 2017 Documents  
drwxr-xr-x 4 carlos carlos 4096 Dez 13 2017 Downloads  
-rw-r--r-- 1 carlos carlos 8980 Sep 30 2017 examples.desktop  
drwxrwxr-x 23 carlos carlos 4096 Dez 10 2017 Matlab  
drwxr-xr-x 2 carlos carlos 4096 Sep 30 2017 Music  
drwxr-xr-x 2 carlos carlos 4096 Jul 12 17:29 Pictures  
drwxr-xr-x 2 carlos carlos 4096 Sep 30 2017 Public  
drwxrwxr-x 2 carlos carlos 4096 Sep 5 2017 Telegram  
drwxr-xr-x 2 carlos carlos 4096 Sep 30 2017 Templates  
drwxr-xr-x 2 carlos carlos 4096 Sep 30 2017 Videos
```

Podemos ver que al ejecutar `ls -l` se enlistan, en forma de lista, los archivos del nodo en el que nos encontramos. Hay muchas columnas de información, por el momento vamos a concentrarnos únicamente en la última columna, que son los nombres de los archivos. Más adelante veremos lo que son las otras columnas.

### 2.3.1. Cambio de directorio

Para cambiar de directorio se utiliza el comando *change directory*, `cd`, cuyo único argumento es el directorio destino.

```
carlos@peregrine: ~/Documents  
carlos@peregrine:~$ cd Documents/  
carlos@peregrine:~/Documents$ ls -la  
total 12  
drwxr-xr-x 3 carlos carlos 4096 Dez 16 2017 .  
drwxr-xr-x 36 carlos carlos 4096 Jul 12 17:27 ..  
drwxrwxr-x 2 carlos carlos 4096 Dez 10 2017 MATLAB
```

En esa imagen me cambié al directorio **Documents** (la diagonal del final nada cambia), y después enliste los archivos dentro del directorio (lo cual hace la opción `-l`) pero le pedí al comando que también liste los archivos ocultos (para eso es la opción `-a`, las puedo combinar escribiendo `-la`). Los archivos ocultos en *Linux* se identifican porque comienzan por un punto.

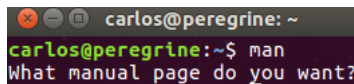
¿Qué son esos dos directorios, `.` y `..`, que invaden el directorio **Documents**?

Es una excelente pregunta con una cálida respuesta. Para facilitar la navegación entre nodos, en **todo** nodo hay mínimo dos archivos, que da la coincidencia son directorios. El primero de ellos es `.`, el cual simboliza el directorio actual. Es decir, el comando `cd .` ó `cd ./` nos lleva al **mismo directorio** en el que estamos parados. Suena inútil pero tiene su propósito, se explica un poco más adelante.

El otro directorio, llamado `..`, es la representación del directorio superior, del que el nodo actual es un *hijo*. Como ejemplo, estando dentro de `/home/carlos/Documents` si yo ejecuto `cd ..` me moveré al directorio `/home/carlos`. Pueden hacer experimentos en su computadora con los comandos `pwd`, `cd` y `ls`, no hay manera en que puedan destruir su computadora con ellos.

### 2.3.2. Antes de seguir

Antes de continuar, les diré un comando mágico que les puede salvar la vida si no tienen internet (en ausencia de Google este comando tuvo su mayor importancia). El comando se llama `man`, y despliega el *manual de ayuda* asociado al comando que le pasen como argumento

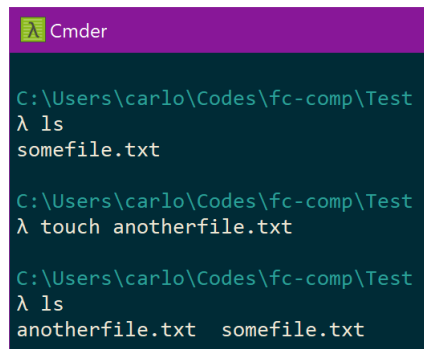
A terminal window titled 'carlos@peregrine: ~' showing the command 'man' being entered. The prompt is 'What manual page do you want?'.

```
carlos@peregrine: ~$ man
What manual page do you want?
```

Pueden intentar ejecutar en su computadora `man ls` para que les describa todas las opciones del comando `ls`, así como su funcionalidad.

### 2.3.3. Creación de archivos y directorios

Temporalmente voy a utilizar `cmdr` sólo para darle variedad a mis capturas de pantalla. Para generar un archivo vacío, en el explorador de Windows simplemente haríamos click derecho y veríamos la opción para realizar dicha tarea. En *Nautilus* (el explorador por defecto de Ubuntu) haríamos lo mismo. Pero ¿Cómo hacemos esto desde la terminal? Fácil, utilizamos el comando `touch`.

A terminal window titled 'Cmdr' showing the execution of the 'touch' command. The prompt is 'C:\Users\carlo\Codes\fc-comp\Test'. The command 'ls' shows 'somefile.txt'. Then 'touch anotherfile.txt' is entered. Finally, 'ls' shows 'anotherfile.txt somefile.txt'.

```
C:\Users\carlo\Codes\fc-comp\Test
λ ls
somefile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ touch anotherfile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ ls
anotherfile.txt somefile.txt
```

Y *voilà*, el archivo apareció. Este comando también sirve para cambiar la fecha de creación de los archivos.

Para eliminar el archivo que acabamos de generar, podemos utilizar el comando `rm` (*remove*)



```

Cmder

C:\Users\carlo\Codes\fc-comp\Test
λ ls
anotherfile.txt  somefile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ rm anotherfile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ ls
somefile.txt

```

Por último, si queremos generar nuevos nodos en el árbol (es decir, nuevos *directorios*) existe el comando `mkdir` para la creación, y `rmdir` para la eliminación. Los directorios y los archivos comunes no son tratados de igual manera por el sistema.

```

Cmder

C:\Users\carlo\Codes\fc-comp\Test
λ mkdir nicedir

C:\Users\carlo\Codes\fc-comp\Test
λ ls
nicedir/  somefile.txt

```

**Nota:** En el caso de `cmdr` es fácil distinguir entre archivos normales y directorios, pues hay un código de color para ellos. No siempre será el caso en todas las terminales. En todo caso, existe una solución.

### 2.3.4. Tipos de archivo y permisos

Al ejecutar `ls -l` se nos despliegan varias columnas de información, la primera es de *gran* importancia. Su formato es el siguiente:

`drwxr-xr-x` →       $\underbrace{\text{d}}$        $\underbrace{\text{rwx}}$        $\underbrace{\text{r-x}}$        $\underbrace{\text{r-x}}$   
 Tipo de archivo    Permiso del dueño    Permiso de grupo    Permiso de usuarios

Explicado de manera sencilla, el primer carácter sería algo así como *permisos avanzados del sistema*, lo cual le dice a Unix qué trato puede darle al archivo. Nosotros podemos interpretarlo como tipo de archivo, pues puede ser un directorio (d), un enlace simbólico (l) o un archivo común (-). Existen otros pero no son de relevancia para nosotros.

Los siguientes 9 caracteres vienen en grupos de 3, y son los permisos de *lectura* (r), *escritura* (w) y *ejecución* (x) para el creador del archivo, para los que están en su mismo grupo, y para cualquier usuario del sistema, respectivamente.

**Nota:** En Unix, cada usuario tiene acceso a una cuenta en el sistema, y estas cuentas tienen permisos. Un administrador tiene acceso a todo el sistema y tiene el poder de generar otras cuentas. Después, existen algo que se llaman *grupos*, una de sus funcionalidades es dar autorización de acceso a ciertos archivos. Por ejemplo, yo podría crear un grupo llamado *computacion* y dar acceso de lectura a los archivos en donde yo coloque las tareas. Puedo generar una cuenta para cada uno de ustedes, incluirlos en el grupo *computacion* y así ustedes tendrán acceso a dichos archivos.

El grupo al que *pertenece* cada archivo está indicado en la 4ta columna del resultado del comando `ls -l`.

Para cambiar los permisos de un archivo, lo más sencillo es ver estos 9 últimos caracteres como una cadena binaria, que se transforma en 3 dígitos que van (cada uno) del 0 al 7

$$\text{rwxr-xr-x} \rightarrow 111101101 \rightarrow 755$$

Y ahora basta con utilizar el comando `chmod` (*change mode*) pasando como primer argumento el nuevo número correspondiente a los permisos que queremos, y como segundo argumento la ruta del archivo que queremos modificar.

```
carlos@peregrine: ~/Documents/nicedir
carlos@peregrine:~/Documents/nicedir$ ls -l
total 0
-rw-rw-r-- 1 carlos carlos 0 Jul 13 12:54 somefile.txt
carlos@peregrine:~/Documents/nicedir$ chmod 765 somefile.txt
carlos@peregrine:~/Documents/nicedir$ ls -l
total 0
-rwxrw-r-x 1 carlos carlos 0 Jul 13 12:54 somefile.txt
```

**Nota:** El cambio de regreso a Unix es porque estos comandos sólo funcionan en Unix. El sistema de grupos y de permisos es diferente en Unix y en Windows. Sin embargo, es necesario que lo sepan usar porque el día que hagan inicio de sesión remota en un sistema Unix, lo van a necesitar. Así que recuérdelo.

### 2.3.5. Tuberías (y rellenando archivos)

Recordemos la filosofía de Unix: *Escriba programas que trabajen bien juntos y El lenguaje universal son las cadenas de caracteres*. Vamos a mostrar esta mentalidad *concatenando* comandos y mandando la salida de los comandos a archivos en lugar de la pantalla.

Cuando ejecutamos un comando como `ls`, la salida es una cadena de caracteres, que es desplegada en la terminal. Podemos dirigir la salida del comando hacia un archivo, usando el operador `>`.

```
Cmder
C:\Users\carlo\Codes\fc-comp\Test
λ ls -l
total 0
drwxr-xr-x 1 carlo 197609 0 Jul 13 05:22 nicedir/
-rw-r--r-- 1 carlo 197609 0 Jul 13 04:21 somefile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ ls -l > somefile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ ls -l
total 1
drwxr-xr-x 1 carlo 197609 0 Jul 13 05:22 nicedir/
-rw-r--r-- 1 carlo 197609 128 Jul 13 11:18 somefile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ cat somefile.txt
total 0
drwxr-xr-x 1 carlo 197609 0 Jul 13 05:22 nicedir/
-rw-r--r-- 1 carlo 197609 0 Jul 13 11:18 somefile.txt
```

Un poco más a detalle: redirigí la salida del comando `ls -l` al archivo `somefile.txt`. Después, utilicé el comando `cat` (que viene de *concatenate*) para desplegar el contenido del archivo.

`cat` sirve también para escribir directamente desde la terminal hacia un archivo. Para esto, ejecutamos `cat` sin argumentos, y lo dirigimos a un archivo.

```
Cmder
C:\Users\carlo\Codes\fc-comp\Test
λ cat > somefile.txt
No me metan en el archivo por favor :(

C:\Users\carlo\Codes\fc-comp\Test
λ cat somefile.txt
No me metan en el archivo por favor :(
```

La primer parte de esta figura la escribí yo. Para avisarle a la computadora que ya no quería seguir insertando texto, interrumpí el comando. Para ello, existe la combinación de teclas sagrada:

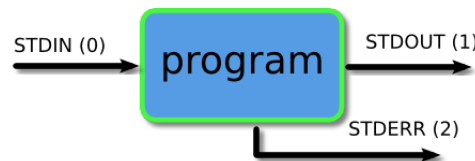
Terminar comando → `ctrl+c`

Bueno, pero claramente hemos sobre-escrito lo que el archivo contenía antes. ... ¿Habrà alguna manera de añadir texto al final de un archivo? Por supuesto que sí, se usa entonces el operador »

```
Cmder
C:\Users\carlo\Codes\fc-comp\Test
λ cat >> somefile.txt
Ni modo, es por el bien de la clase

C:\Users\carlo\Codes\fc-comp\Test
λ cat somefile.txt
No me metan en el archivo por favor :(
Ni modo, es por el bien de la clase
```

Ahora, lo que hicimos fue desviar la salida del comando hacia un archivo, pero en realidad hay dos posibles salidas de texto de un comando, como se ve en la siguiente imagen



Cada salida está simbolizada por un número, siendo 1 la salida normal y 2 cuando hay un error. Ejecutemos un comando que de una salida correcta y un error para ilustrar esto mejor.

```
Cmder
C:\Users\carlo\Codes\fc-comp\Test
λ cat somefile.txt other.txt
No me metan en el archivo por favor :(
Ni modo, es por el bien de la clase
cat: other.txt: No such file or directory

C:\Users\carlo\Codes\fc-comp\Test
λ cat somefile.txt other.txt 1> NUL
cat: other.txt: No such file or directory

C:\Users\carlo\Codes\fc-comp\Test
λ cat somefile.txt other.txt 2> NUL
No me metan en el archivo por favor :(
Ni modo, es por el bien de la clase
```

Ya que el archivo `other.txt` no existe, `cat` arroja un error. En la primer ejecución se imprimen ambas salidas a la pantalla. En la segunda, mandamos la salida normal al basurero, y en la tercera mandamos los errores al basurero.

**Nota:** Como ya se había mencionado antes, el basurero en Unix es `/dev/null`, cuyo *equivalente* en Windows es `NUL`. Es decir: la versión *Unix* de la última instrucción en la imagen anterior sería `cat somefile.txt other.txt 2>/dev/null`.

Por último, ¿Qué pasa si queremos que la salida de un comando sea la entrada de otro? Para eso existe la *tubería*, denotada por el caracter `|`. E siguiente ejemplo ilustra (de una manera un poco redundante) el uso de la tubería.

```
λ Cmder
C:\Users\carlo\Codes\fc-comp\Test
λ cat somefile.txt | awk '{gsub("o", "e"); print}'
Ne me metan en el archive per faver :(
Ni mede, es per el bien de la clase
```

No se preocupen por el segundo comando; Ese lo vamos a ver hasta la segunda semana. Lo que hice, fue leer lo que hay en el archivo `somefile.txt` para después pasarlo al comando `awk` (digo que es redundante porque el comando `awk` podría sin problemas leer un archivo).

Pueden investigar, como *tarea moral*, lo que los comandos `head`, `tail` y `wc` (*word count*) hacen.

### 2.3.6. Por último, mover y copiar

Primero está el comando para copiar archivos, `cp` (*copy*).

```
cp ruta_origen ruta_destino
```

y para mover archivos, existe el comando `mv` (*move*).

```
mv ruta_origen ruta_destino
```

No es muy difícil ver que este comando también se puede usar para renombrar archivos. Aquí hay un ejemplo de cómo copiar (a la izquierda) y uno de cómo mover (a la derecha.)

```
λ Cmder
C:\Users\carlo\Codes\fc-comp\Test
λ ls -l
total 1
drwxr-xr-x 1 carlo 197609  0 Jul 13 05:22 nicedir/
-rw-r--r-- 1 carlo 197609 75 Jul 13 12:59 somefile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ cp somefile.txt otherfile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ ls -l
total 2
drwxr-xr-x 1 carlo 197609  0 Jul 13 05:22 nicedir/
-rw-r--r-- 1 carlo 197609 75 Jul 13 16:29 otherfile.txt
-rw-r--r-- 1 carlo 197609 75 Jul 13 12:59 somefile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ cat otherfile.txt
No me metan en el archivo por favor :(
Ni modo, es por el bien de la clase
```

```
λ Cmder
C:\Users\carlo\Codes\fc-comp\Test
λ ls -l
total 2
drwxr-xr-x 1 carlo 197609  0 Jul 13 05:22 nicedir/
-rw-r--r-- 1 carlo 197609 75 Jul 13 16:29 otherfile.txt
-rw-r--r-- 1 carlo 197609 75 Jul 13 12:59 somefile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ mv otherfile.txt nicedir\

C:\Users\carlo\Codes\fc-comp\Test
λ ls
nicedir/  somefile.txt

C:\Users\carlo\Codes\fc-comp\Test
λ cd nicedir\

C:\Users\carlo\Codes\fc-comp\Test\nicedir
λ ls -l
total 1
-rw-r--r-- 1 carlo 197609 75 Jul 13 16:29 otherfile.txt
```

Esto es todo por la primer semana. En la siguiente vamos a ver comandos más avanzados, y vamos a ver lo que es *shell scripting*.

*Carlos & Víctor.*

### 3. Más comandos útiles

Bueno, pues continuaremos con los comandos que les van a ser de utilidad y con otros conceptos de importancia en Unix.

#### 3.1. El Usuario

Ustedes cuentan con un usuario en su computadora (así como les daremos un usuario en los servidores de la facultad). Pueden ver su nombre de usuario si ejecutan el comando `whoami` (*who am I*).

Al instalar Unix en sus computadoras habrán tenido que crear un nombre de usuario y una contraseña. Cuando les hagamos la cuenta en el servidor les vamos a proporcionar una contraseña alfanumérica que deben cambiar a una que puedan recordar. Para ello está el comando `passwd`.

**Nota:** Todo comando en Unix que involucre una contraseña, no va a desplegar el texto que ustedes escriban con el teclado. Esto es meramente por seguridad (siempre hay mirones que pueden robar su información). Windows usa *asteriscos*, pero esos tienen la debilidad de mostrar la longitud de la contraseña.

#### 3.2. El Superusuario

Vamos a hablar de una categoría de usuario en particular de Unix, conocido como el *superusuario*. Este usuario es básicamente el administrador del sistema, al que no se le puede negar ningún permiso, puede modificar cualquier archivo y puede hacer lo que el quiera con las cuentas de los otros usuarios, únicamente detenido por barreras éticas y morales.

**Nota:** En Windows también hay ciertas barreras, los usuarios también tienen jerarquía. Sólo que esta no se controla por la terminal. Lo más que se puede hacer para emular esta característica es iniciar la terminal en modo administrador. Ya saben, esa ventanita molesta que interrumpe todo para preguntarnos si estamos seguros de lo que estamos haciendo.

Hay comandos que no se ejecutan por diversas razones: porque pueden provocar una inestabilidad en el sistema, porque la sintaxis se puede confundir con la de otro comando de menor relevancia, o simplemente porque no tenemos el privilegio de ejecutarlos como usuarios comunes. Para ello, llega el comando mágico:

```
sudo
```

**Nota:** Este comando puede ser BASTANTE peligroso. Si lo van a usar, deben estar muy seguros de la instrucción que ejecutan después.

El comando `sudo` básicamente nos convierte en superusuarios, lo que significa que los permisos de escritura y lectura dejan de tener sentido. El comando nos va a pedir la contraseña de nuestra cuenta. Se vuelven irrelevantes, y ustedes podrán modificar cualquier archivo de sistema. Los argumentos de este comando son el comando a ejecutar y sus argumentos. Por ejemplo, adivinen, **sin ejecutar**, qué es lo que haría el comando siguiente (usen `man` para conocer las opciones de `rm`):

```
sudo rm -rf /
```

**Nota:** NO EJECUTEN EL COMANDO.

Hay ocasiones en que es *seguro* ejecutar el comando: se necesita ser administrador para poder actualizar el sistema, por ejemplo. Incluso para simplemente leer algunos archivos (lo cual no puede dañar el sistema) se necesitan permisos de superusuario. Sólo sepan que si encuentran instrucciones de cómo hacer algo en internet que involucra el comando `sudo`, hay que leer con calma y no sólo copiar y pegar las instrucciones.

### 3.3. Búsqueda de archivos

Por si se lo han preguntado, sí, se puede hacer una búsqueda de archivos desde la terminal. Supongamos que queremos buscar un archivo que se llama `cpuinfo` en el sistema, y no tenemos ni idea de dónde se podría localizar así que queremos buscar en todo el sistema.

```
find / -name cpuinfo
```

Desmembrando esa instrucción, el primer argumento es el directorio en el que se va a comenzar la búsqueda. En este caso realizamos una búsqueda desde la raíz. Después, con `-name` le decimos a `find` que vamos a darle el nombre del archivo a buscar, el cual es `cpuproc`.

Si ejecutan ese comando, se darán cuenta que hay errores saliendo en pantalla.

```
find: '/var/lib/bluetooth/E0:94:67:F9:36:6F': Permission denied
find: '/var/lib/udisks2': Permission denied
find: '/var/lib/color/.cache': Permission denied
find: '/var/lib/lightdm': Permission denied
find: '/var/lib/polkit-1': Permission denied
find: '/var/lib/update-notifier/package-data-downloads/partial': Permission denied
find: '/var/spool/rsyslog': Permission denied
find: '/var/spool/cron/crontabs': Permission denied
find: '/var/spool/cups': Permission denied
find: '/var/log/speech-dispatcher': Permission denied
find: '/var/cache/ldconfig': Permission denied
find: '/var/cache/apt/archives/partial': Permission denied
find: '/var/cache/cups': Permission denied
find: '/var/cache/lightdm/dmrc': Permission denied
carlos@peregrine:~/Documents/fc-comp$
```

Todos estos son directorios a los que no tenemos acceso como usuarios normales. ¿Se les ocurre cómo podríamos tener acceso a ellos?

```
carlos@peregrine:~/Documents/fc-comp$ ls -l /var/cache/lightdm
total 4
drwx----- 2 root root 4096 Sep 30 2017 dmrc
carlos@peregrine:~/Documents/fc-comp$
```

En todo caso, el archivo que yo quería encontrar está oculto entre los errores. Como ejercicio, ¿Cómo hacen para poder distinguir entre errores y salida normal para no tener que estar leyendo línea por línea?

### 3.4. Edición de archivos

Para editar un archivo tenemos dos herramientas, `vi` y `vim`. Son editores de texto (como el bloc de notas) cuya interfaz corre sobre la terminal. De preferencia utilicen `vim`, es muy probable que se encuentre instalado en cualquier sistema al que hagan inicio de sesión remota. Si no está en Ubuntu, lo pueden instalar con:

```
sudo apt install vim
```

Si quieren ir directo al grano y aprenderlo de una manera divertida, aquí está un [jueguito](#) que les enseña lo que hace cada tecla en `vim`. Pero hay un resumen (no muy informativo) aquí:

`vim` es complicado, pero vamos a hacerlo paso por paso:

- Al entrar estaremos en el modo *normal*. En este modo no podemos escribir nada.

- Para movernos al modo de edición (uno de ellos), presionamos la tecla `i`. A partir de ahora podemos escribir como en cualquier editor.
- Para salir del modo de edición, presionamos `esc`
- En el modo normal podemos ejecutar comandos, los cuales comienzan con `:`. Prueben lo siguiente, abran `vim` ejecutando el comando sin argumentos, entren en modo de edición para escribir *Hola mundo*, salgan del modo de edición y escriban `:w hola.txt`, después presionen `enter`. `vim` lee esto como

$\underbrace{\quad\quad\quad}_{:}$        $\underbrace{\quad\quad\quad}_{w}$        $\underbrace{\quad\quad\quad}_{hola.txt}$   
 Ahí viene un comando    Ese comando es guardar    El nombre será hola.txt

- Para salir de `vim`, ejecuten el comando `:q`

Con eso ya tienen el funcionamiento más básico de `vim`. Lo lindo del cómputo es que pueden aprender haciendo experimentos, y aprenderán por la práctica.

- En el modo normal se pueden copiar líneas. Para esto, nos ubicamos encima de la línea que queremos copiar (navegando con las flechas, o con las teclas `h,j,k,l`) y presionamos dos veces la letra `y`. Tal vez nos aparecerá un anuncio que dirá *1 line yanked*. Ahora la línea está en el *portapapeles*.
- Para pegar, nos vamos a la línea en donde queremos pegar y presionamos dos veces la tecla `d`.
- Para copiar varias líneas, podemos anteceder el `yy` de un número, por ejemplo si tecleamos `6 yy` se copiarán las 6 líneas a partir de donde está el cursor.

#### 3.4.1. Truco: ejecución de comandos desde vim

Podemos ejecutar comandos de `bash` desde `vim`. Para ello está el comando `:!` , todo lo que venga después se ejecutará en `bash` y podremos ver el resultado. Prueben a ejecutar `:! ls -l`

#### 3.4.2. Otras funciones y comandos

*Con la práctica se hace al maestro.* Así que no desesperen si hay muchas cosas que no saben usar. Se necesitaría un curso entero para entender `vim` en su totalidad, y ese no es el objetivo. Les daremos las herramientas para que puedan completar tareas sencillas con `vim` y el resto de las herramientas que veremos en el curso, y si necesitan algo, pueden utilizar *Google* o *Stackoverflow* para encontrar solución a sus problemas de informática.

##### 3.4.2.1 Volcar todo un archivo al archivo actual.

Si hay un archivo que contiene cosas que quieren *copiar y pegar* dentro del archivo que tienen abierto con `vim`, pueden utilizar el comando `:r nombre_del_archivo`.

##### 3.4.2.2 Forzar un comando

A veces `vim` se va a negar a ejecutar un comando por razones de seguridad (por ejemplo, no les va a permitir salir de la edición de un archivo si no han guardado los cambios). Cuando quieran forzar un comando, pongan al final un signo de admiración. Por ejemplo, `:q!` los saca de `vim` aunque hayan cambios sin guardar en el archivo.

### 3.4.3. Ayuda

Recuerden que esto es de tiempos en que el internet no era nuestro pan de cada día. Así que los manuales de ayuda aún están integrados en los programas. Para desplegar ayuda respecto a algún comando, pueden ejecutar el comando `:help` seguido del comando del que quieran obtener información. Pueden probar a ejecutar `:help 42`.

## 3.5. Uso avanzado de bash

`bash` es el intérprete de comandos que hemos estado utilizando desde la semana anterior. `bash` es un *Unix Shell*, pues hay varias opciones aunque la más usada es `bash`. Hasta ahora, hemos realizado operaciones básicas: búsqueda de archivos, creación de archivos y directorios, navegación, cambio de permisos, etc. Ahora nos movemos a cosas un poco más *especializadas*.

### 3.5.1. Monitor de recursos

En Unix, tenemos también herramientas como el administrador de tareas en Windows. Para desplegarlo, bastará con ejecutar el comando `top`

```
carlos@peregrine: ~/Documents/fc-comp
top - 18:50:28 up 3:17, 1 user, load average: 0.93, 0.65, 0.68
Tasks: 252 total, 1 running, 250 sleeping, 0 stopped, 1 zombie
%Cpu(s): 3.5 us, 1.9 sy, 0.0 ni, 94.2 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st
KiB Mem : 8031740 total, 1248372 free, 2958856 used, 3824512 buff/cache
KiB Swap: 8190972 total, 8190972 free, 0 used, 4080824 avail Mem

      PID USER      PR  NI    VIRT    RES    SHR  S  %CPU  %MEM     TIME+ COMMAND
 1110 root        20   0   478904  141648  91596 S   11.3   1.8   7:36.11 Xorg
 1912 carlos     20   0  1582772  226944  68256 S   11.3   2.8  12:19.64 compiz
 5786 carlos     20   0  3450820  1.454g 144992 S    7.3  19.0  12:03.31 Web Content
 1969 carlos     9  -11  501480   16700  13212 S    3.6   0.2   6:47.55 pulseaudio
 2030 carlos     20   0   513892   24420  19940 S    3.3   0.3   6:30.99 indicator-multi
 7064 carlos     20   0   428304   22152  18752 S    2.3   0.3   0:00.17 gnome-screensho
 5450 carlos     20   0  2535744  304416  151344 S    2.0   3.8   2:37.58 firefox
 1690 carlos     20   0   643352   39212  26320 S    1.3   0.5   2:57.83 unity-panel-ser
 1481 carlos     20   0   43968    4380   2808 S    1.0   0.1   1:36.37 dbus-daemon
 1850 carlos     20   0   476880   12924  11436 S    1.0   0.2   2:11.67 indicator-appli
 7053 carlos     20   0   42260    3860   3192 R    0.7   0.0   0:00.11 top
 1520 carlos     20   0   33176    1644   1248 S    0.3   0.0   0:33.83 upstart-dbus-br
 2151 carlos     20   0   664704   38840  28872 S    0.3   0.5   0:15.55 gnome-terminal-
 2187 carlos     20   0  1551484  120200  82044 S    0.3   1.5   3:40.92 code
 5518 carlos     20   0  2052544  198924 104680 S    0.3   2.5   6:01.29 Web Content
 5825 root        20   0         0         0         0 S    0.3   0.0   0:00.16 kworker/3:2
 5914 root        20   0         0         0         0 S    0.3   0.0   0:00.23 kworker/7:2
 6881 root        20   0         0         0         0 S    0.3   0.0   0:00.41 kworker/u16:1
 6898 root        20   0         0         0         0 S    0.3   0.0   0:00.13 kworker/6:0
 6971 root        20   0         0         0         0 S    0.3   0.0   0:00.02 kworker/5:2
    1 root        20   0  185660    6304   4016 S    0.0   0.1   0:01.43 systemd
    2 root        20   0         0         0         0 S    0.0   0.0   0:00.02 kthreadd
    4 root        0 -20         0         0         0 S    0.0   0.0   0:00.00 kworker/0:0H
    6 root        0 -20         0         0         0 S    0.0   0.0   0:00.00 mm_percpu_wq
```

La columna de la izquierda, PID, es el *identificador de proceso*, un número único que identifica una comando en ejecución. No tiene caso que nos detengamos mucho a discutir toda la información que aparece en su pantalla, pero es bueno que noten que aparecen los porcentajes de uso del cpu, así como de memoria, el tiempo que llevan existiendo los procesos, el número de usuarios y el usuario dueño de cada proceso.

### 3.5.2. Matando un proceso

Para detener un proceso de la manera violenta, utilizamos el comando `kill`. Este puede venir en uso cuando un proceso lleva atascado un rato y nosotros no tenemos acceso a él, ya sea porque lo generó otro usuario o porque es un proceso de fondo.

Por ejemplo, si yo quisiera cerrar *firefox* (el mejor navegador del mundo mundial) por medio de `kill`, ejecutaría



```
kill 5450
```

El `pid` se puede fácilmente encontrar con `top`, o con el comando `ps`.

### 3.5.3. Procesos de fondo

Supongamos que tenemos un proceso que se quedará de fondo. Tomemos de ejemplo la siguiente instrucción.

```
(sleep 10; echo "Hola mundo!") &
```

Ejecútenla sin el *ampersand* del final. La terminal estará en pausa por 10 segundos, y después verán la cadena de caracteres *Hola mundo!*. Ahora ejecútenla con el *ampersand*. Unix va a darles el número de proceso de esa instrucción y liberará la terminal para que puedan utilizarla. Lo que sea que estén haciendo se verá interrumpido a los 10 segundos por la cadena de caracteres anterior.

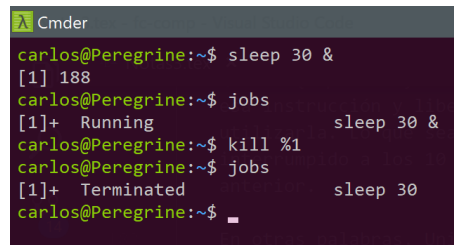
En otras palabras, Unix permite mandar procesos de fondo (por ejemplo, si van a copiar recursivamente muchos archivos, pueden pasar esto al fondo y hacer otras cosas mientras se realiza la copia).

**Nota:** El paréntesis está ahí para avisarle a Unix que es un único comando el que estamos mandando.

El *operador* `;` le indica a Unix que ahí se termina un comando. Esto nos permite ejecutar comandos de manera secuencial.

#### 3.5.3.1 Manejo de procesos de fondo

Un comando que está tomando mucho tiempo se puede mandar al fondo también. Para ello hay que interrumpirlo pero no con `ctrl+c`, se debe hacer con `ctrl+z` que manda una señal de pausa al proceso. Tanto usando `ctrl+z` como usando `&` (que **no es lo mismo**, pues la última no pone en pausa al proceso), nos va a aparecer en pantalla un número entre corchetes



```
Cmder
carlos@Peregrine:~$ sleep 30 &
[1] 188
carlos@Peregrine:~$ jobs
[1]+  Running                  sleep 30 &
carlos@Peregrine:~$ kill %1
carlos@Peregrine:~$ jobs
[1]+  Terminated              sleep 30
carlos@Peregrine:~$
```

ese número es el *número de la tarea*. Es de utilidad porque lo podemos utilizar para manejarlo. Para ver todas las tareas actuales se utiliza el comando `jobs`, el cual nos dice el estado de las tareas y sus correspondientes identificadores. Para poner en pausa al proceso número 1 se puede utilizar el comando `kill` con la opción `-STOP`

```
kill -STOP%1
```

El símbolo de porcentaje le avisa a `kill` que el número 1 no es un identificador de proceso, corresponde al identificador que vemos en el comando `jobs`.

Los comandos `bg` y `fg` son de mucha utilidad, representan las palabras *background* y *foreground* respectivamente. Con `bg %n` pasamos el proceso número `n` al fondo y lo reanudamos, mientras que `fg %n` hace lo equivalente pero no en el fondo (es decir, continua el proceso ocupando la terminal).

Casi por inducción, pueden matar los procesos de fondo basándose en el número que aparece en `jobs` usando el símbolo de porcentaje y el comando `kill` pero sin la opción `-STOP`.

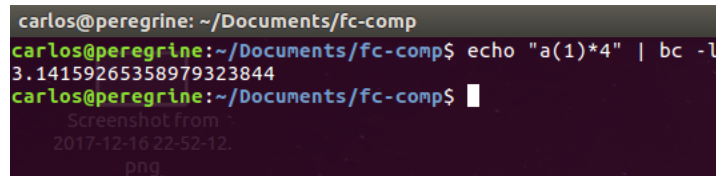
**Nota:** Ejecutar `bg` o `fg` sin argumentos hará que el último proceso añadido a esta *fila* de procesos se mueva al fondo o se reanude en la terminal respectivamente.

### 3.5.4. Una calculadora

Si necesitan hacer cálculos sencillos, Unix tiene una calculadora integrada llamada `bc`. Si se ejecuta el comando sin argumentos, se iniciará el modo *interactivo* (por llamarlo de algún modo), para salir basta con escribir *quit* y presionar **enter**. Aquí pueden escribir las operaciones que quieran y al presionar **enter** verán el resultado. Si intentan hacer una división cuyo resultado no sea un número entero, verán que el resultado va a ser de todos modos un entero (osea, un resultado equivocado).

Esto es el modo por defecto de trabajar de `bc`, pero si usamos el argumento `-l` la calculadora iniciará en modo avanzado. Con esto no sólo podemos hacer operaciones con *pseudorreales*, pero también vienen funciones trigonométricas cargadas (pueden ver el manual para ver cómo se llaman).

Si queremos realizar un cómputo a partir de una cadena de caracteres, podemos usar una *tubería* para que `bc` evalúe la expresión correspondiente.



```
carlos@peregrine: ~/Documents/fc-comp
carlos@peregrine:~/Documents/fc-comp$ echo "a(1)*4" | bc -l
3.14159265358979323844
carlos@peregrine:~/Documents/fc-comp$
```

La imagen es una captura de pantalla de una terminal. Muestra el prompt de usuario 'carlos@peregrine' y la ruta de trabajo '~/Documents/fc-comp'. Se ejecuta el comando 'echo "a(1)\*4" | bc -l', donde 'a(1)' es una función de la biblioteca de la calculadora 'bc' que devuelve el valor de pi. El resultado mostrado es '3.14159265358979323844'. El cursor parpadea en la línea siguiente.

La imagen anterior es un simple ejemplo de cómo se puede calcular *pi* con esta calculadora (la *arcotangente* de 1 es  $\pi/4$ ).

### 3.5.5. El mejor comando del mundo, `ssh`

Ahora llegamos a la sección de inicio de sesión remota. Para el inicio de sesión remota sólo necesitamos dos cosas: un nombre de usuario en algún servidor, y la dirección del mismo servidor. La última parte ya nos la patrocina Unix, se llama `ssh` (*secure shell*).

Si nuestro nombre de usuario es `usuario` y el servidor está bajo la dirección `192.168.0.1`, o bajo el dominio `dominioepico.com`, podemos iniciar sesión con

```
ssh usuario@dominioepico.com
```

o con

```
ssh usuario@192.168.0.1
```

Probablemente se nos va a solicitar la contraseña de la cuenta con la que estamos intentando acceder al servidor. Una vez que la entreguemos, habremos iniciado sesión en el servidor. Todos los comandos anteriores se pueden ejecutar desde la terminal y en realidad serán ejecutados en el servidor.

#### 3.5.5.1 Transferencia de archivos

Claro, hacer *log in* en el servidor no es suficiente para completar las tareas y trabajos de este curso. También necesitamos saber cómo transferir archivos entre ambos equipos. Para ello existe el comando `scp` (lo adivinaron, *secure copy*).

Este comando se ejecuta siempre desde la su computadora (la razón se las explicaré/se las habré explicado en clase).

Supongamos que el archivo que queremos copiar se llama `hola.txt` y se encuentra en el subdirectorio `archivos` del directorio `home` en el servidor. Y ese archivo lo queremos copiar a donde estamos parados actualmente en nuestro ordenador. Para ello ejecutaríamos

```
scp usuario@dominioepico.com:~/archivos/hola.txt ./
```

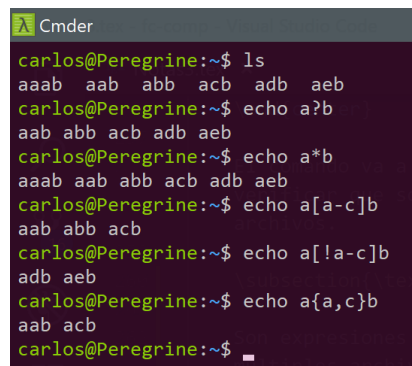
El comando va a solicitar la contraseña, y después de verificar que somos nosotros se realizará la transferencia de archivos.

### 3.5.6. *Wildcards*

Son expresiones usadas por varios comandos para poder manejar múltiples archivos, particularmente permitirle al usuario especificar un rango de archivos. Un ejemplo que ya hemos usado es el asterisco, que básicamente significa lo que sea. Los caracteres especiales que podemos utilizar son los siguientes

- `?` El signo de interrogación, que representa un único carácter, específicamente a-z (incluyendo mayúsculas) ó 0-9.
- `*` El asterisco, que representa cualquier número de caracteres (incluyendo **cero**).
- `[ ]` Los corchetes son para indicar un rango válido de caracteres o números. Por ejemplo, `m[a-c]l` representa las opciones `mal`, `mb1` y `mcl`.
- `{,}` Las llaves con valores separados por comas sirven para indicar opciones exclusivas, por ejemplo, `m{a,c}l` representa las opciones `mal` y `mcl`
- `!` El signo de admiración es la negación. Es decir, `!a` significa cualquier caracter que no sea `a`. A veces se usa la *cuña* en lugar del signo de admiración (^).

Para que vean lo que hacen estos *wildcards*, la siguiente captura de pantalla los usa todos



```
Cmder
carlos@Peregrine:~$ ls
aaab aab abb acb adb aeb
carlos@Peregrine:~$ echo a?b
aab abb acb adb aeb
carlos@Peregrine:~$ echo a*b
aaab aab abb acb adb aeb
carlos@Peregrine:~$ echo a[a-c]b
aab abb acb
carlos@Peregrine:~$ echo a[!a-c]b
adb aeb
carlos@Peregrine:~$ echo a{a,c}b
aab acb
carlos@Peregrine:~$
```

**Nota:** El carácter de escape `\` les permite representar uno de estos caracteres especiales de manera específica. Es decir, si uno de sus archivos se llama `a*.txt` y lo quieren borrar, tendrían que ejecutar `rm a\*.txt`. En realidad, el carácter de escape es mucho más que eso, lo veremos después.

*Carlos & Víctor.*

## 4. *Shell scripting* y `awk`

Llegamos a la parte final de nuestro curso exprés de `bash`, después de esto nos moveremos a los lenguajes de programación principales para el resto del curso.

### 4.1. *Shell scripting*

Ok, ya se está comenzando a poner interesante esto. ¿Qué tal si hay una combinación de comandos que ejecutamos muy, muy seguido y no queremos estarlos repitiendo? Pues, sería muy lindo escribir las instrucciones en algún archivo para poder ejecutarlas cada vez que queramos, ¿No? A esto se le conoce como *shell scripting*. Vamos a hacer un pequeño ejemplo. Tomen la instrucción:

```
date +%T | awk -F ':' '{print ($1+11) %24":"$2":"$3}'
```

La cual nos da la hora en Pekín, suponiendo que el comando se ejecuta en una máquina en la zona GMT-5. Ahora, no queremos estar escribiendo el comando entero todo el tiempo. Coloquémoslo en un archivo de texto, usando `vim`, y guardemos el archivo bajo el nombre `util.sh`.

Pues bien, ahora para ejecutarlo nos va a bastar con cambiar los permisos tal que el creador pueda ejecutar el comando (por ejemplo, `chmod 755 util.sh`), y para ejecutarlo simplemente escribiremos lo siguiente en la terminal

```
./util.sh
```

El punto diagonal se puede ver muy innecesario pero es una protección para Unix. De esta manera, no basta con escribir el nombre de un *script* para que se pueda ejecutar.

Ahora ya podemos ver el resultado de la ejecución

```
carlos@peregrine: ~/Documents/fc-comp/Test
carlos@peregrine:~/Documents/fc-comp/Test$ ./util.sh
3:57:11
carlos@peregrine:~/Documents/fc-comp/Test$
```

¿Pero qué tal si queremos hacer algo más complejo? Por ejemplo, guardar dicha cadena de caracteres en una variable para desplegar un mensaje más amigable

#### 4.1.1. Variables en *shell scripting*

Podemos declarar variables en shell. Para ello, en realidad basta con darles nombre y darles valor. Como ejemplo, creemos la variable `nombre` que contenga su nombre. Después, vamos a usar `echo` para imprimirlo en pantalla.

```
carlos@peregrine: ~/Documents/fc-comp/Test
carlos@peregrine:~/Documents/fc-comp/Test$ cat ejemplo.sh
nombre="Carlos"
echo $nombre
carlos@peregrine:~/Documents/fc-comp/Test$ ./ejemplo.sh
Carlos
carlos@peregrine:~/Documents/fc-comp/Test$
```

**Nota:** Es muy importante que respeten no poner espacios entre el nombre de la variable, el signo de igual, y el valor de la variable. *Shell* es un poco *diva*.

Notemos que para poder acceder al valor de la variable hay que poner un signo de dinero antes de su nombre, de otro modo el comando imprimiría *nombre*. Y qué pasa si lo que yo quiero es darle como valor el resultado de un comando? Si ustedes escriben lo siguiente

```
nombre=echo "Carlos"
```

van a obtener un error. Esto es porque a *shell* hay que decirle que tiene que ejecutar lo que está enfrente del igual y el resultado es lo que va a guardar en la variable *nombre*. Para ello hay que rodear **todas las instrucciones que querramos** en paréntesis, y anteponer un signo de dinero

```
nombre=$(echo "Carlos")
```

Vamos a usar de ejemplo la instrucción de la hora en china.

```
carlos@peregrine: ~/Documents/fc-comp/Test
carlos@peregrine:~/Documents/fc-comp/Test$ cat util.sh
china=$(date +%T | awk -F ":" '{print ($1+11)%24":"$2":"$3}')
echo "La hora en china es $china"
carlos@peregrine:~/Documents/fc-comp/Test$ ./util.sh
La hora en china es 4:15:02
carlos@peregrine:~/Documents/fc-comp/Test$
```

#### 4.1.2. Argumentos del comando

¿Y qué pasa si este *shell script* lo queremos usar como un comando? Pues nos hará falta poder recibir argumentos. Pero esto ya está contemplado. Al ejecutar un *script* con argumentos, *shell* nos hará de favor de *colocar* el primer argumento en una variable de nombre 1 (por lo que podemos acceder a su valor con *\$1*), el segundo argumento en 2, y así respectivamente para cada argumento (igualito que en *awk* para las columnas).

Pónganse de ejercicio hacer un comando simplísimo que les repita lo que sea que pongan de primer argumento.

**Nota:** Por si llega a serles de utilidad, pueden obtener todos los argumentos separados por espacio en una sola variable. La variable es *@*, así que pueden acceder a sus valores con *\$@*.

#### 4.1.3. Instrucciones básicas dentro de un *script*

Vamos a tratar de no indagar demasiado en esto porque se va a comenzar a poner complicado, pero vamos a enseñar tres conceptos que son de mucha utilidad en *shell scripting*.

##### 4.1.3.1 for

Vamos a plantearnos que queremos que un *script* nos diga *hola* 7 veces. Pues la primer idea es hacer un script que se vea como

```
echo "hola"
echo "hola"
echo "hola"
echo "hola"
```

```
echo "hola"
echo "hola"
echo "hola"
```

Lo cual funciona, pero es bastante redundante. *shell* nos permite escribir lo siguiente:

```
for VAR in 1 2 3 4 5 6 7
do
    echo "hola"
done
```

Lo que *shell* hará es ejecutar lo que está entre **do** y **done** por cada elemento frente a la palabra **in**, y almacenará el elemento en **VAR** (que en realidad puede ser cualquier palabra que queramos, es sólo el nombre de la variable en el ciclo) por si necesitamos acceder a él.

Cabe destacar que esa serie de números se puede reemplazar con el comando **seq**.

Para un uso más interesante, podemos hacer un comando que salude a varios de nuestros amiguitos.

```
for nombre in Carlos Víctor Jesús
do
    # Ponemos el signo de dinero $ para sacar el valor de nombre
    echo "Hola $nombre"
done
```

**Nota:** Así como ya lo veremos más adelante, se puede (**y debe**) poner comentarios en los códigos. En el caso de *shell scripting* cualquier línea que comience por **#** es ignorada.

La ejecución de este *script* tendrá como resultado

```
Hola Carlos
Hola Víctor
Hola Jesús
```

#### 4.1.3.2 while

La segunda instrucción básica con la que vamos a lidiar es conocida como **while**. La estructura es muy parecida a la del **for**, sólo que la primer instrucción es **while [ condition ]**, como se ve en el siguiente ejemplo

```
s=4
while [ $s -ge 0 ]
do
    echo "Hello!"
    s=$((echo "$s-1" | bc))
done
```

**Nota:** Es clave respetar los espacios que rodean la condición.

La condición es **\$s -ge 0**, que significa *\$s greater or equal to 0*, y tendrá como resultado la escritura de la cadena **Hello!** 5 veces. La siguiente tabla contiene todos los operadores de comparación que *shell* entiende

Instrucción	Significado
<code>-eq</code>	igual ( <i>equals</i> )
<code>-ne</code>	diferente ( <i>not equal to</i> )
<code>-lt</code>	menos que ( <i>less than</i> )
<code>-le</code>	menos o igual que ( <i>less than or equal to</i> )
<code>-gt</code>	mayor que ( <i>greater than</i> )
<code>-ge</code>	mayor o igual que ( <i>greater than or equal to</i> )

#### 4.1.3.3 if...else

Por último, tenemos la condición más importante, `if`. La estructura es como la de `while` pues tiene una condición.

```
if [ $1 != "stop" ]
then
    echo "First argument is not \"stop\""
else
    echo "First argument is \"stop\""
fi
```

En este caso, la comparación es de cadenas de caracteres, para lo que están los siguientes dos operadores

Instrucción	Significado
<code>=</code>	igual
<code>!=</code>	diferente

**Nota:** `else` permite ejecutar código alternativo en caso de que la condición no se cumpla. Se puede omitir e ir directo al `fi`

**Nota:** Si tenemos más de 9 argumentos (nótese que el nombre del script corresponde a `$0`), podemos usar las *llaves* para englobar el número del argumento, por ejemplo el argumento doce sería `${12}`

## 4.2. awk

*awk is love. awk is life.* A pesar de que no lo vamos a presentar como lo que es (*un poderosísimo lenguaje de programación para procesamiento de textos de alto rendimiento*), vamos a ver el funcionamiento esencial del comando `awk`.

Este comando procesa texto, y puede hacer reconocimiento de patrones, sustituciones, recortes y muchas otras cosas por el estilo que ustedes desearían hacer a algún archivo de datos. Vamos a tomar por ejemplo la siguiente cadena de caracteres

Y cuando despertó, el dinosaurio todavía estaba allí.

Ahora, vamos a ver cómo `awk` toma esta cadena, y simplemente la imprime.

```
carlos@peregrine: ~/Documents/fc-comp/Test
carlos@peregrine:~/Documents/fc-comp/Test$ cat ElDinosaurio.txt | awk '{print}'
Y cuando despertó, el dinosaurio todavía estaba allí
carlos@peregrine:~/Documents/fc-comp/Test$
```

Pero podemos hacer mucho más. Vamos de una vez a ver el uso avanzado de `awk`. Lo que se encuentra dentro de las llaves (que a su vez están dentro de las comillas) son las instrucciones que `awk` va a seguir por cada línea del archivo. Pueden ser varias siempre y cuando estén separadas por punto y coma, por ejemplo, podemos poner `print 2` veces.

```
carlos@peregrine: ~/Documents/fc-comp/Test
carlos@peregrine:~/Documents/fc-comp/Test$ cat ElDinosaurio.txt | awk '{print; print}'
Y cuando despertó, el dinosaurio todavía estaba allí
Y cuando despertó, el dinosaurio todavía estaba allí
carlos@peregrine:~/Documents/fc-comp/Test$
```

¿Qué tal si queremos hacer algo más avanzado? Por ejemplo, queremos quitar todos los espacios de la oración.

```
carlos@peregrine: ~/Documents/fc-comp/Test
carlos@peregrine:~/Documents/fc-comp/Test$ cat ElDinosaurio.txt | awk '{gsub(" ", ""); print}'
Ycuandodespertó,eldinosauriotodavíaestabaallí
carlos@peregrine:~/Documents/fc-comp/Test$
```

La palabra `gsub` es una función. De estas vamos a ver muchísimas en lo que resta del curso. Momentáneamente digamos que una función tiene argumentos y regresa algo, en este caso `gsub` es una función que toma como primer argumento el carácter (o cadena de caracteres) a buscar y como segundo el carácter (o cadena) con la que se va a reemplazar cada incidencia.

**Nota:** Varias instrucciones se pueden separar con punto y coma, así como en la terminal. En este caso las instrucciones son `gsub` y `print`

Bien pude reemplazar las letras *o* con *e* para hacer el texto un poco más *inclusivo*.

```
carlos@peregrine: ~/Documents/fc-comp/Test
carlos@peregrine:~/Documents/fc-comp/Test$ cat ElDinosaurio.txt | awk '{gsub("o", "e"); print}'
Y cuande despertó, el dinesaurie tedavía estaba allí
carlos@peregrine:~/Documents/fc-comp/Test$
```

No sólo eso, pero `awk` nos permite dividir una entrada de texto en columnas para trabajar con cada una de ellas de manera independiente.

#### 4.2.1. Columnas

Por defecto, el *delimitador* de las columnas es cualquier número de espacios o tabuladores. Ya habíamos discutido que `ls -l` arrojaba varias columnas como resultado. Bien, pues si queremos ver únicamente los permisos de ejecución de todos los archivos en una carpeta, podemos hacer lo siguiente

```
carlos@peregrine: ~/Documents/fc-comp/Test
carlos@peregrine:~/Documents/fc-comp/Test$ ls -l | awk '{print $1}'
total
-rw-rw-r--
-rw-rw-r--
-rwxr-xr-x
-rw-rw-r--
-rwxr-xr-x
-rwxr-xr-x
-rwxr-xr-x
carlos@peregrine:~/Documents/fc-comp/Test$
```

Así es, `$n` se refiere a la *n*-ésima columna del renglón que `awk` procesa.



Además de esto, **awk** puede ser aplicado a renglones en donde se encuentre algún patrón de caracteres. Tomemos como ejemplo lo siguiente: el archivo `/proc/cpuinfo` contiene información sobre la frecuencia del procesador, pero está un poco *oculto*, o más bien, hay mucha información que no nos interesa de momento. Las líneas que contienen las frecuencias de cada núcleo del procesador contienen las palabras `cpu MHz`

Una búsqueda de patrones se le indica a **awk** al escribirlo antes de las instrucciones (pero dentro de las comillas) rodeado por dos diagonales

```
carlos@peregrine: ~/Documents/fc-comp/Test
carlos@peregrine:~/Documents/fc-comp/Test$ cat /proc/cpuinfo | awk '/cpu MHz/ {print}'
cpu MHz      : 935.028
cpu MHz      : 1267.304
cpu MHz      : 867.474
cpu MHz      : 1071.034
cpu MHz      : 912.067
cpu MHz      : 1036.757
cpu MHz      : 914.030
cpu MHz      : 845.572
carlos@peregrine:~/Documents/fc-comp/Test$
```

El patrón, escrito como `/cpu MHz/` es una *expresión regular*, algo así como (*wildcards*)<sup>2</sup>, mucho más poderoso para hacer *pattern matching* o encajar patrones. Eso lo vemos un poquito más a detalle adelante en este mismo pdf.

Como ejercicio para ustedes, piensen cómo le hacen para obtener únicamente las frecuencias (el número) sin todas las demás cosas. No hay que modificar mucho.

**awk** permite la definición de funciones y variables, en este caso vamos a usar sólo la definición de variables, así como otras utilidades de **awk**

#### 4.2.2. BEGIN y END

**awk** permite ejecutar código antes de comenzar la lectura de un archivo, y también al finalizar la lectura de un archivo, para eso están las palabras reservadas **BEGIN** y **END** respectivamente.

Supongamos la siguiente listita de calificaciones, guardada en un archivo llamado `notas.txt`

```
Carlos García,10
Felipe Ortega,6
Andrés Moreno,5
Jesús Hitler,9
Michael Jordan,8
Michael Jackson,8
E. Peña,1
```

El malvado profesor que otorgó las notas dijo que si el grupo no obtenía colectivamente al menos 50 puntos, los iba a reprobar a todos. Sin embargo, les regaló también 5 puntos. ¿Cómo calculamos esto con **awk**? Primero, el carácter de separación es una coma, lo cual hay que notificarle a **awk**

```
cat notas.txt | awk -F ',' '{print $2}'
```

Esta instrucción va a imprimir todas las calificaciones en pantalla.

```
Cmder
carlos@Peregrine:/mnt/c/Users/Carlos/Codes/fc-comp/Clases/Clase04$ cat notas.txt | awk -F ',' '{print $2}'
10
6
5
9
8
8
1
carlos@Peregrine:/mnt/c/Users/Carlos/Codes/fc-comp/Clases/Clase04$
```

Siguiente punto a recordar, es que para declarar una variable en `awk` basta con nombrarla. El valor por defecto de las variables es cero. Hagamos pues una variable llamada `sum`

```
cat notas.txt | awk -F ',' '{sum=sum+$2; print sum}'
```

Recuerden que esa instrucción se va a ejecutar una vez por cada línea.

```
Cmder
carlos@Peregrine:/mnt/c/Users/Carlos/Codes/fc-comp/Clases/Clase04$ cat notas.txt | awk -F ',' '{sum=sum+$2;print sum}'
10
16
21
30
38
46
47
carlos@Peregrine:/mnt/c/Users/Carlos/Codes/fc-comp/Clases/Clase04$
```

Parece un poco innecesario ejecutar `print sum` por cada línea, ¿No? Por ello vamos a utilizar la palabra `END`, con lo cual le especificamos al código qué hacer al terminar de leer todas las líneas.

```
cat notas.txt | awk -F ',' '{sum+= $2} END {print sum}'
```

El resultado de ese hermoso código es

```
Cmder
carlos@Peregrine:/mnt/c/Users/Carlos/Codes/fc-comp/Clases/Clase04$ cat notas.txt | awk -F ',' '{sum+= $2} END {print sum}'
47
carlos@Peregrine:/mnt/c/Users/Carlos/Codes/fc-comp/Clases/Clase04$
```

La variable `sum` tiene persistencia desde que se llama por primera vez hasta que finaliza `awk`, así que no hay problema con usarla en `END`. Ya casi lo tenemos todo. Ahora falta algo, el profesor dijo que regalaría 5 puntos, sería bueno **inicializar** la variable `sum` con el valor 5. Para eso usamos `BEGIN`

```
cat notas.txt | awk -F ',' 'BEGIN {sum=5} {sum+= $2} END {print sum}'
```

Esto imprime 52.

### 4.2.3. Condiciones de línea

Ahora el profesor, muy mala onda, añadió la regla que en la suma sólo se cuentan estudiantes que no reprobaron. Para esto, podemos condicionar el segundo bloque de instrucciones para que no se ejecute en todas las líneas.

```
cat notas.txt | awk -F ',' 'BEGIN {sum=5} $2>=6 {sum+= $2} END {print sum}'
```

Con lo que el grupo sólo llega a 46 puntos. Todos reprobados.

#### 4.2.4. Pattern Matching

Un poco atrás usamos la expresión regular `/cpu MHz/` para sacar información del archivo `/proc/cpuinfo`. Esto se puede utilizar de maneras mucho más avanzadas. Vamos a ver rápidamente lo que son expresiones regulares, mejor conocidas como *regex*.

##### 4.2.4.1 Expresiones regulares (*Regex*)

La siguiente lista son los caracteres regulares.

- `.` : equivale al *wildcard* `?`, es decir, un carácter único
- `*` : el carácter anterior aparece 0 o más veces
- `?` : el carácter anterior aparece 0 o 1 vez
- `+` : el carácter anterior aparece 1 o más veces
- `{n}` : el carácter anterior aparece **exactamente** *n* veces
- `{n, m}` : el carácter anterior aparece **al menos** *n* veces y **no más** de *m* veces
- `[abc]` : el carácter es uno de los incluidos entre los corchetes.
- `[^abc]` : el carácter no es uno de los que están entre los corchetes.
- `[a-h]` : el carácter está en el rango **a** hasta **h** (incluyendo los límites)
- `()` : operador de agrupamiento, para tratar una serie de caracteres como uno solo
- `|` : la operación lógica OR
- `^` : este operador implica que queremos que la secuencia esté únicamente al principio
- `$` : este operador implica que queremos que la secuencia esté únicamente al final

Esto les va a ser de muchísima, muchísima utilidad para la tarea. Siguiendo el ejemplo de las notas y el profesor mala onda, vamos a imprimir el nombre de todos los estudiantes que contengan una vocal y luego la consonante r, en algún lado de su nombre. Comparando el ejemplo de `cpu MHz`, es claro que el *regex* se una como condición **en toda la línea**. Nosotros podemos hacer *pattern matching* con una columna en particular, para eso se usa la expresión `$n ~ /aquivaelregex/`

```
Cmder
carlos@Peregrine:/mnt/c/Users/Carlos/Codes/fc-comp/Clases/Clase04$ cat notas.txt | awk -F ',' '$1 ~ /[aeiou]r/ {print $1}'
Carlos García
Andrés Moreno
Jesús Hitler
Michael Jordan
carlos@Peregrine:/mnt/c/Users/Carlos/Codes/fc-comp/Clases/Clase04$ _
```

**Nota:** El carácter `/` no es parte del *regex*, simplemente le dice a `awk` que lo que está contenido entre esos dos símbolos es un *regex*.

##### 4.2.4.2 Combinando condiciones

Yo quería verlo bien en `C`, pero vamos a verlo de una vez. Podemos hacer múltiples condiciones, por ejemplo que la primer columna contenga una vocal seguida de la consonante r, ó que el estudiante haya reprobado. Los dos tipos de relaciones entre condiciones disponibles a nosotros en `awk` son AND representado por `&&` y OR representado por `||`

El ejemplo dado se ejecutaría como

```
cat notas.txt | awk -F ',' '$1 ~ /[aeiou]r/ || $2<6 {print $1}'
```

### 4.3. Conclusión

Ya con esto tienen suficientes herramientas para entretenerse en Unix. Esto debiera, además, hacerles saber que nadie tiene la capacidad ni la necesidad de memorizar absolutamente todo lo que la computadora puede hacer. Lo que importa es que sepan cómo buscar ayuda y en dónde buscarla.

También debe ser suficiente para la primera y segunda tarea. La idea del curso es que estas habilidades las vamos a utilizar **todo el tiempo**, así que no olviden nada de lo que aprendieron en las primeras clases.

*Carlos & Víctor.*

### 4.4. Anexo: Escape de caracteres

Nunca supimos dónde poner este tema, no es muy largo así que este será el lugar.

Hay caracteres que la computadora entiende pero ameritan un trato distinto. Uno de ellos es lo que ocurre cuando presionamos **enter** en el teclado al editar un archivo en **vim**, algo conocido como *salto de línea*. Dado que cualquier archivo no es más que una secuencia de caracteres binarios, ¿Cómo identifica la computadora esto? En particular, el salto de línea está denotado como `\n`, es decir, lo que nosotros vemos como

```
Hola mundo!  
Hola amigos!
```

La computadora lo ve como

```
Hola mundo!\nHola amigos!\0
```

El último carácter es conocido como **EOF** y sirve para indicar el final de un archivo. A la diagonal hacia atrás (`\`) se le conoce como carácter de escape, el cual permite un uso alterno de los caracteres (sólo de ciertos).

Otros conocidos son el tabulador `\t`, y *backspace* `\b`. Este caracter sirve además para hacer que caracteres que en sí tienen una función especial puedan ser utilizados simplemente como caracteres.

Como rápido ejemplo, el asterisco `*` casi en todos lados significa *todo*. Por ejemplo, el comando `cat *` va a aplicar `cat` a todos los elementos en la carpeta en la que se ejecuta el comando. Si tuviéramos un archivo llamado `*` y quisieran aplicar `cat` sólo a este archivo, escribirían `cat \*`

Las comillas también sirven para indicar dónde comienza una cadena de caracteres y dónde termina. Para evitar este comportamiento especial y tratar a las comillas `"` como un caracter cualquiera, lo *escapamos* usando `\`.

## 5. C, la máquina

Aún nos falta hablar de muchas cosas sobre la computadora y Unix. Pero para poder explicarlas, es mejor ver de una vez lo que es un lenguaje de programación, y en particular, el lenguaje C.

### 5.1. Lenguajes de programación

¿Cómo funcionan los comandos que hemos estado ejecutando? No pueden ser simplemente *scripts*, esto es evadir la pregunta, como la *panspermia* al querer saber el origen de la vida. ¿Qué es lo que específicamente le dice a la computadora cómo hacer las cosas? No podremos explicar con total precisión lo que el procesador de la computadora hace al recibir instrucciones ni la estructura mínima del conjunto de instrucciones básicas que el procesador puede ejecutar, pero vamos a acercarnos lo más posible.








#### 5.1.1. Definición

Un lenguaje de programación es un vocabulario con un conjunto de reglas para explicarle a una computadora qué tarea queremos que se realice. Es un lenguaje común que tenemos los humanos con las computadoras. Hay *palabras* básicas, *axiomáticas* que piden a la computadora realizar una tarea.

#### 5.1.2. Compuertas lógicas

En su nivel más básico, son las llamadas *compuertas lógicas*. Éstas trabajan sobre elementos que sólo puede ocupar dos posibles estados, cosas binarias, 0 y 1. Las operaciones más básicas son AND y OR. Además de estas existe XOR, NAND, XNOR y NOR. Por último, la compuerta más sencilla es la negación, NOT.

### Logic Gates

Name	NOT	AND	NAND	OR	NOR	XOR	XNOR																																																																																																
Alg. Expr.	$\overline{A}$	$AB$	$\overline{AB}$	$A + B$	$\overline{A + B}$	$A \oplus B$	$\overline{A \oplus B}$																																																																																																
Symbol																																																																																																							
Truth Table	<table><tr><th>A</th><th>X</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	A	X	0	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	0	0	1	0	1	0	0	1	1	1	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	1	0	1	1	1	0	1	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	1	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	1	0	1	0	1	0	0	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	B	A	X	0	0	0	0	1	1	1	0	1	1	1	0	<table><tr><th>B</th><th>A</th><th>X</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	B	A	X	0	0	1	0	1	0	1	0	0	1	1	1
A	X																																																																																																						
0	1																																																																																																						
1	0																																																																																																						
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	1																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	0																																																																																																					
0	1	1																																																																																																					
1	0	1																																																																																																					
1	1	0																																																																																																					
B	A	X																																																																																																					
0	0	1																																																																																																					
0	1	0																																																																																																					
1	0	0																																																																																																					
1	1	1																																																																																																					

Esta imagen recién robada contiene las llamadas tablas de verdad. Como ven, en todas las compuertas excepto la primera hay dos entradas y una salida. Ahora, esto es lo más básico que va a hacer la computadora. ¿Cómo le hacemos para sumar dos números, 6 y 7, por ejemplo? Sacado de [aquí](#), vamos a construir un sumador que pueda, como mínimo, sumar estos dos números.

##### 5.1.2.1 Representación binaria

Ya que la computadora sólo entiende 0 y 1, entonces vamos a escribir los números 6 y 7 en binario. Un número cualquiera se puede cambiar de base.

Sea  $s$  una cadena de símbolos, tal que el elemento  $s_i$  cuya posición se cuenta de derecha a izquierda comenzando por 0 es tal que  $s_i \in S_r$ , siendo  $S_r$  un conjunto de cardinalidad  $r$  (es decir,  $|S_r| = r$ ). Al numerito  $r$  le llamaremos *Radix* y es la base del sistema a usar. En el caso particular del sistema decimal,  $r = 10$  y  $S_r = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ . En el sistema decimal, podríamos tener una cadena de símbolos, por ejemplo  $s = 134$  (es decir,  $s_0 = 4$ ,  $s_1 = 3$  y  $s_2 = 1$ ). Sea  $x$  la cantidad representada por la cadena  $s$ . ¿Cómo sabemos cuánto vale  $x$  dada la cadena y la base?

**Nota:** La manera común de denotar la base para una cadena de símbolos, es colocar el *radix* como subíndice de un paréntesis que rodee el número. Por ejemplo,  $(134)_{10}$

Cuando escribamos un número en binario, es decir con *radix* 2, lo vamos a anteceder por **0b** (por el momento no se pregunta, sólo se acepta).

La fórmula para conocer el valor de  $x$  es la siguiente:

$$x = \sum_{i=0}^{|s|-1} s_i r^i$$

Es un poco redundante, pero entonces el número 134 se descompone (usando la fórmula anterior) como

$$134 = 1 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$$

Ahora, dado un número  $x$  y un *radix*  $r$ , ¿Cómo encontramos la cadena  $s$ ? Podemos intentar despejar la  $s_i$  en que estamos interesados

$$\begin{aligned} \sum_{j=0}^{i-1} s_j r^j + s_i r^i + \sum_{j=i+1}^{|s|-1} s_j r^j &= x \\ s_i r^i + \sum_{j=i+1}^{|s|-1} s_j r^j &= x - \sum_{j=0}^{i-1} s_j r^j \\ s_i + \sum_{j=i+1}^{|s|-1} s_j r^{j-i} &= \frac{x - \sum_{j=0}^{i-1} s_j r^j}{r^i} \end{aligned}$$

En esta ecuación aún es imposible encontrar cualquier  $s_i$ , pero se puede hacer un truco sencillo. Vamos a cambiar los límites de la suma para que comience por cero, con lo que podremos factorizar una  $r$

$$s_i + r \sum_{j=0}^{|s|-1-i} s_j r^j = \frac{x - \sum_{j=0}^{i-1} s_j r^j}{r^i}$$

No debiera ser muy difícil ver que el residuo de dividir el lado izquierdo por  $r$  será simplemente  $s_i$ , así que procedemos

$$s_i = \left( \frac{x - \sum_{j=0}^{i-1} s_j r^j}{r^i} \right) \% r$$

Ecuación que funciona para cualquier  $i$ , menos  $i = 0$ , la cual simplemente es

$$s_0 = x \% r$$

**Nota:** En las últimas ecuaciones estamos utilizando el símbolo de porcentaje % para denotar el residuo de la división entre los elementos que conecta. Es decir,  $C = A \% B$  corresponde a la  $C \geq 0$  más pequeña que cumpla  $C \equiv A \pmod{B}$ .

Con este procedimiento, los números 6 y 7 tienen la siguiente representación binaria

$$6 = 0b110$$

$$7 = 0b111$$

### 5.1.3. Una suma

Una computadora realiza sumas por medio de las compuertas lógicas de la manera en que ustedes sumaban en la primaria. O como aún suman si es un número grande. ¿Recuerdan cómo?

```

  6
+7
--
13

```

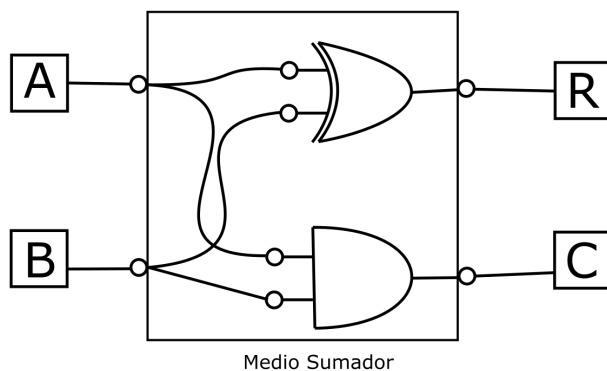
Es decir, *6 y 7 hacen 13, que son 3 y llevas 1. Después, 1 y 0 hacen 1. Listo, es 13.* Vamos a repetir el ejercicio en binario.

```

  11   (lo que llevan)
-----
 0b110
+0b111
-----
 0b1101

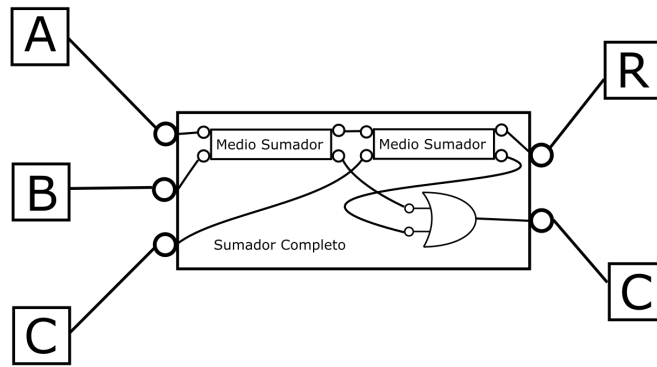
```

Ahora queremos un circuito de compuertas lógicas que pueda hacer esto. *Uf.* No está tan fácil. Primero vamos a hacer lo que se le conoce como un *medio-sumador*.

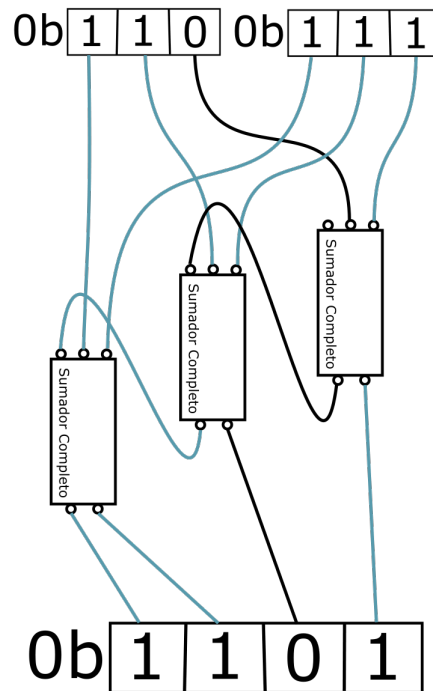


El medio sumador puede realizar la operación de sumar dos números de 1 bit, la unidad elemental de los binarios, y devuelve el resultado de la suma (incluyendo el *carry*, como le llamaremos ahorita).

Lo que sigue, es hacer un elemento (conocido como *sumador completo*) que realice la misma suma pero pueda tomar en cuenta el *carry*.



Ahora sí, para poder hacer la suma de dos números de 3 bits, se necesitan 3 sumadores completos en cadena, como en el siguiente diagrama.



Los cables azules son los que transportan un 1. Como tarea mental, piensen cómo se multiplica con compuertas lógicas.

#### 5.1.4. Ensamblador, Assembly

El lenguaje que le dice a la computadora que hacer en su nivel más elemental es el lenguaje máquina. Es decir, se le dicta a la computadora la operación lógica a seguir entre registros en el sistema. A cualquier lenguaje que es casi al nivel de lo que la máquina entiende se le conoce como Ensamblador. Todo se puede hacer en ensamblador pero es un lenguaje muy complejo. Vamos a ver pedazos de este lenguaje más adelante.

#### 5.1.5. Lenguajes de alto nivel

Los lenguajes como ensamblador son conocidos como lenguajes de bajo nivel. Los programadores rápidamente notaron que hablar el mismo lenguaje que la máquina no era productivo, y requería demasiado esfuerzo. Así que inventaron los lenguajes de alto



nivel. Un lenguaje de alto nivel es un lenguaje que se parece más a los que nosotros hablamos, y menos al que la máquina entiende. En lugar de estar manejando registros de memoria e instrucciones lógicas, a la computadora se le puede decir, en C++ por ejemplo, que haga un nuevo vector con la instrucción `std::vector x = new std::vector()` (lo cual no debe ser nada claro para ustedes en este momento, pero al menos son capaces de reconocer las palabras **new** y **vector**).

Entre menos tengamos que entender cómo hace la computadora las cosas, más alto el nivel de programación.

Claro, la máquina no entiende este lenguaje, por lo que es necesario algún tipo de *traductor*. Los hay de dos tipos

- Intérpretes
- Compiladores

En el caso de los lenguajes interpretados, al momento de la ejecución de un código se va haciendo la *traducción* en tiempo real a lenguaje máquina.

Para los lenguajes compilados, se tiene que generar un archivo conocido como *ejecutable* a partir de un *código fuente*. El *ejecutable* contiene únicamente instrucciones que la computadora ya puede entender.

## 5.2. Un lenguaje de alto nivel: C

Vamos a ver nuestro primer lenguaje de alto nivel, llamado C. Es un lenguaje compilado, y por lo mismo necesitamos conseguir un compilador para poder generar ejecutables. No voy a escribir todas las instrucciones para la instalación del compilador de C. Quienes quieran tener estas herramientas disponibles en Windows será mejor que me pregunten.

Para los afortunados que ya estén usando *Linux*, basta con que ejecuten los siguientes comandos en la terminal

```
sudo apt-get update
sudo apt-get install build-essential
```

Para verificar que su instalación fue correcta, ejecuten el comando `gcc` con la opción `-v`, lo cual les dirá la versión de compilador que tienen instalado.

Como es costumbre, vamos primero a construir el código fuente de un programa introductorio conocido como *Hello World*. El equivalente en *shell scripting* sería poner la instrucción `echo "Hola Mundo"` en un *script* para después ejecutarlo.

A partir de ahora también son libres de decidir qué editor de texto usar. Si quieren usar el *bloc de notas*, adelante. Sólo **recuerden** que el uso básico de `vim` es parte de la evaluación.

Bien, el código que tienen que escribir es el siguiente (guárdenlo en un archivo llamado `main.c`, para este punto ya debieran saber que las terminaciones de los archivos no tienen sentido alguno, sólo sirven de identificadores para la computadora y para nosotros)

```
#include <stdio.h>

// Función principal
int main(){
    printf("Hola, mundo!\n");
    return 0;
}
```

No se ve muy complicado, ¿O sí? Primero vamos a compilarlo, después vamos a analizar lo que hace el código. Para compilarlo ejecutamos el comando `gcc main.c`. Debería generarse un archivo llamado `a.out` (`a.exe` si usan Windows) en el mismo directorio. Para ejecutarlo basta con correr el comando `./a.out`

```
Cmder
carlos@DESKTOP-2KTINDL:/mnt/c/Users/Carlos/Codes/fc-comp/Test$ ls
main.c
carlos@DESKTOP-2KTINDL:/mnt/c/Users/Carlos/Codes/fc-comp/Test$ gcc main.c
carlos@DESKTOP-2KTINDL:/mnt/c/Users/Carlos/Codes/fc-comp/Test$ ls
a.out  main.c
carlos@DESKTOP-2KTINDL:/mnt/c/Users/Carlos/Codes/fc-comp/Test$ ./a.out
Hola, mundo!
carlos@DESKTOP-2KTINDL:/mnt/c/Users/Carlos/Codes/fc-comp/Test$ |
```

Vamos a ver a detalle qué es lo que está haciendo `gcc`

### 5.2.1. Los 4 pasos de la compilación

Al ejecutar `gcc main.c` han ocurrido 4 pasos que vamos a estudiar brevemente.

#### 5.2.1.1 Preproceso

En el preproceso, el compilador abre el archivo `main.c` y busca cualquier línea que comience por `#`, pues estas son instrucciones específicas para este paso. Del mismo modo, el preprocesamiento se encarga de eliminar comentarios que nos ayudan a nosotros a entender el código. Estos comentarios los reconoce el compilador pues comienzan con `//`. En el caso del `main.c` que les di, el comentario es `Función principal`.

Para ver el resultado de la precompilación, podemos ejecutar el comando `gcc` con la opción `-E`, y redirigimos la salida a un archivo de texto, por ejemplo `pre.txt`.

En el caso de nuestro `main.c`, la única instrucción con `#` (`#include`) le dice al compilador que reemplace dicha línea con el contenido del archivo `stdio.h`. Aquí hay definiciones de funciones y cosas que veremos adelante.

```
extern char *ctermid (char *__s) __attribute__ ((__nothrow__ , __leaf__));
# 840 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));

extern int ftrylockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));

extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 868 "/usr/include/stdio.h" 3 4

# 2 "main.c" 2

# 3 "main.c"
int main(){
    printf("Hola, mundo!\n");
    return 0;
}
carlos@DESKTOP-2KTINDL:/mnt/c/Users/Carlos/Codes/fc-comp/Test$ |
```

La salida es bastante texto así que sólo por ser ilustrativo pongo la imagen de las últimas líneas del archivo.

#### 5.2.1.2 Compilación

En este paso se toma el resultado del preproceso y se convierte a lenguaje ensamblador. Esto es la traducción. Para poder ver el resultado en lenguaje ensamblador de nuestro código, pasamos la instrucción `-S` al compilador. Se va a generar un archivo llamado `main.s`

```

1 | .file "main.c"
2 | .text
3 | .section .rodata
4 | .LC0:
5 |     .string "Hola, mundo!"
6 |     .text
7 |     .globl main
8 |     .type main, @function
9 | main:
10 | .LFB0:
11 |     .cfi_startproc
12 |     pushq %rbp
13 |     .cfi_def_cfa_offset 16
14 |     .cfi_offset 6, -16
15 |     movq %rsp, %rbp
16 |     .cfi_def_cfa_register 6
17 |     leaq .LC0(%rip), %rdi
18 |     call puts@PLT
19 |     movl $0, %eax
20 |     popq %rbp
21 |     .cfi_def_cfa 7, 8
22 |     ret
23 |     .cfi_endproc
24 | .LFE0:
25 |     .size main, .-main
26 |     .ident "GCC: (Ubuntu 7.3.0-16ubuntu3) 7.3.0"
27 |     .section .note.GNU-stack,"",@progbits
~

```

Yo sé que el código de C no es claro, pero esto es mucho peor.

#### 5.2.1.3 Ensamblado, *Assembly*

En este paso, se toman las instrucciones de ensamblador y se genera algo conocido como *código objeto*. Esto ya son instrucciones binarias, y ya no está a nuestro alcance entender lo que hay en el archivo. Son las instrucciones exactas y precisas que el procesador debe ejecutar para realizar la tarea que el código fuente solicitó.

De todos modos, si quieren verlo, ejecuten el comando `hexdump` con `main.o` como argumento.

#### 5.2.1.4 Enlazado, *Linking*

En el paso anterior se ha producido el código objeto únicamente del archivo fuente. Pronto lo veremos, pero para ser más eficientes, hay código guardado en *bibliotecas* para que no tengamos que estarlo compilando o reescribiendo cada vez. Este código no está incluido en el paso anterior, este paso se llama *enlazado* porque toma estas piezas faltantes del código, y las coloca (además de reordenar como sea necesario para la ejecución).

El resultado de este último paso es el archivo `a.out`

### 5.2.2. Las estructura del lenguaje

Ahora sí. El lenguaje C es un lenguaje imperativo secuencial. Esto quiere decir que el lenguaje consta de instrucciones que cambian el estado del sistema, y que estas instrucciones se ejecutan de manera secuencial, léase, una después de la otra.

#### 5.2.3. Instrucciones

Las instrucciones simples son aquellas que con `;` terminan. No tiene por qué haber una por cada línea, pueden haber múltiples en una línea.

## 5.2.4. Variables

El lenguaje permite la declaración de variables, para hacer distintas operaciones con ellas. Una variable no es más que una dirección de memoria (también lo veremos...) y un valor. En el caso de `main.c` no hemos utilizado variable alguna.

Las variables en C se declaran de la siguiente forma:

`int x;    →    int                      x                      ;`  
Tipo de variable    Nombre de la variable    Fin de la instrucción

Hay muchos tipos de variable, y vamos a ver después que incluso nosotros podemos definir nuestros tipos de variable. En particular, los tipos de variable estándar son los de la siguiente tabla

Identificador	Tamaño (bits)	Descripción
<code>int</code>	32	Entero
<code>unsigned int</code>	32	Entero (positivo)
<code>long</code>	64	Entero
<code>unsigned long</code>	64	Entero (positivo)
<code>short</code>	16	Entero
<code>float</code>	32	Real de precisión finita
<code>double</code>	64	Real de precisión finita
<code>char</code>	8	Caracter
<code>unsigned char</code>	8	Caracter (positivo)
<code>long double</code>	80	Real de precisión finita

Hay un tipo extra, llamado `void` pero su uso como variable es un poco avanzado, vamos a verlo primero en una función.

### 5.2.4.1 Enteros

Los enteros son las variables más fáciles de entender. Se almacenan en la memoria del sistema ocupando entre 16 bits y 64 bits. Ahora, hay un pequeño problema. ¿Cuál es el valor máximo que una cadena de 32 bits puede representar de manera binaria? Si regresamos a la fórmula para cambio de base, podemos ver que una cadena  $s$  con *radix*  $r$  de longitud  $|s|$  puede representar como máximo el número  $\text{máx} = r^{|s|} - 1$ . En este caso, ese número es  $2^{32} - 1 = 4,294,967,295$  (esto es 2 ordenes de magnitud abajo de la deuda de México, en dólares). Es un buen número pero para ciertos cálculos no es suficiente. Por otro lado, tenemos que  $2^{64} - 1 = 18,446,744,073,709,551,615$ . Bastante buen número. Estos valores pueden ser alcanzados por `unsigned int` y `unsigned long`, respectivamente, pues son enteros positivos.

¿Qué pasa si queremos representar el -1? Las variables `short`, `int` y `long` lo pueden hacer sin problemas. El primer modo de solucionar esto fue quitarle 1 bit a los números y utilizarlo como un signo.

Supongamos un tipo de variable llamado `miniint` de 4 bits. Un `unsigned miniint` tendría de rango el intervalo  $[0, 15]$ , osea 16 posibles números representados. Quitándole el primer bit para usarlo para el signo, tomando casi siempre el 0 como + y el 1 como -. Ahora podríamos representar el rango  $[-7, 7]$ , que son 15 números representados...

### ¿Qué pasó?

El problema es que ahora hay 2 ceros, uno positivo y uno negativo, pero no hay diferencia entre ellos cuando se habla de enteros. Lo que es peor, nuestro método para hacer sumas con compuertas lógicas es un absoluto desastre usando esta gramática.

Para solucionar este último problema, se propone otro mapeo. Usando el último bit como el signo, el mapeo es  $(s) \rightarrow [0] + (s) = (0, s_2, s_1, s_0)_2$  (esta operación es concatenar el dígito 0 a la izquierda para pasar a una cadena de 4 caracteres), y para los negativos el mapeo es  $-(s) \rightarrow [1] + (s) = (1, s_2, s_1, s_0)$ , que se traduce en  $-0 \rightarrow 8, -1 \rightarrow 9, -2 \rightarrow 10$  y así. Alguien inteligente se preguntó si había otro mapeo inteligente que se pudiera hacer que respetara la suma de números binarios.

La solución es la siguiente: Las concatenaciones se respetan pero para los negativos se hace una negación de toda la cadena antes de la concatenación, es decir  $-(s) \rightarrow [1] + (!s) = (1, !s_2, !s_1, !s_0)$ , que se traduce en  $-0 \rightarrow 15, -1 \rightarrow 14, -2 \rightarrow 13$

**Nota:** El símbolo ! se usa para denotar negación binaria, NOT.

Aunque ustedes no lo crean, esto arregla el problema de la suma si se considera que el *carry* generado por el último bit, el del *signo*, vale 1 y se suma en caso de ser generado. A esto se le conoce como *One's Complement*. Hagan pruebas sumando 3 y -5 por ejemplo.

La solución final para representar negativos y positivos se conoce como *Two's Complement*, y el mapeo es muy similar. El mapeo para negativos es  $-(s) \rightarrow [1] + (!s) + (0, 0, 0, 1)$ , en caso de que se genere un *carry* simplemente lo tiramos a la basura, lo ignoramos. El mapeo se traduce en  $-0 \rightarrow 0, -1 \rightarrow 15, -2 \rightarrow 14$ . Recorrimos el mapeo, pero ganamos algo: el doble cero se fue, y en su lugar apareció un -8.

#### 5.2.4.2 Flotantes

Los números flotantes son el modo en que la computadora puede representar los *reales* (en realidad, son puros *racionales*). Para poder entender cómo funciona esto, va a ser mucho mucho más fácil pensar primero en notación científica. Piensen en el producto entre 270,000,000 y 0.000,000,3. Hacer este producto a mano, con fuerza bruta, es un tanto complicado. Pero si somos un tanto más inteligentes, podemos reescribir ambos números como  $2.7 \times 10^8$  y  $3 \times 10^{-7}$ . Esto es considerablemente más fácil por pura asociatividad del producto, pues se puede reordenar como  $(2.7)(3)(10^8)(10^{-7}) = 8.1 \times 10^1 = 81$

Pues la computadora hace exactamente lo mismo, pero en base 2. Adivinen por ejemplo, cuál es el siguiente número

$$10110001 \times 2^{1111}$$

Así es, esto es  $177 \times 2^{15} = 5,799,936$ . Noten que con los mismos bits sólo podríamos llegar hasta el 4096 usando enteros. Hay 3 factores formando este número, el exponente, el signo y la *mantissa* (el número antes del *radix* con la potencia). Como un acuerdo, los flotantes tienen 1 bit para el signo, 8 para el exponente, y 23 para la *mantissa*

$$\underbrace{0}_{\text{Signo}} \underbrace{000\ 0000}_{\text{Exponente}} \underbrace{0000\ 0000\ 0000\ 0000\ 0000\ 0000}_{\text{Mantissa}}$$

En el caso de máquinas de 64 bits, los números son 1 bit para el signo, 11 para el exponente y 53 para la *mantissa*.

Pregunta: ¿Cuál es el ínfimo y el supremo de los números que se pueden representar con esta representación?

## 6. Más cosas de C

### 6.1. Funciones

El otro elemento que nos queda ver para casi entender por completo el programa que hemos hecho la semana pasada son las funciones.

Una función en C es un conjunto de instrucciones que quedan agrupadas y se pueden llamar bajo un *nombre*. Estas funciones pueden o no regresar valores, y pueden o no depender de variables. las funciones en C tienen la siguiente estructura:

`int suma(int a, int b){return a+b;} →`

int  
Tipo de retorno

suma  
nombre

(int a, int b)  
argumentos

{return a+b;}  
instrucciones

En el caso de tener una función que tiene algún valor de retorno, la última instrucción debe ser la instrucción **return**, la cual le dice a la función qué exactamente es lo que debe regresar (en este caso, **a+b**).

**Nota:** Cuando queremos una función que no regrese valor alguno, se utiliza el tipo **void** y se omite la instrucción de retorno.

Ahora, **todo programa** necesita una función llamada **main** y es el punto de comienzo de ejecución de cualquier programa en C. La estructura de esta función es peculiar, su tipo de valor de retorno es un entero, generalmente se usa para indicar si la ejecución del programa fue correcta o no. Como argumentos nosotros no le pusimos argumento alguno pero en el caso general se le debe poner (**int argc, char \*\*argv**). Esa doble estrella la explicaremos más adelante.

Ya casi lo tenemos decifrado.

```
// Primero incluimos el archivo stdio.h que contiene la definición de printf
#include<stdio.h>

// Función principal
int main(){
    //Llamamos a printf, que toma como primer argumento la cadena a imprimir
    printf("Hola, mundo!");
    return 0; //Regresamos 0 para indicar que todo salió bien
}
```

#### 6.1.1. La función printf

Esta función es *tricky*. Ya vimos cómo usarla para imprimir una cadena de caracteres. Pero eso no es lo único que sabe hacer, también nos permite imprimir el valor de cualquiera de las variables estándar que C contiene, pero lo hace de una manera chistosa: Nosotros le damos una cadena de caracteres con *identificadores* en donde queremos que coloque el contenido de alguna variable, además de darle las variables mismas como argumento, y C se encarga de reemplazar los valores de las variables en los lugares indicados en la cadena.

Estos identificadores van anteceditos por un símbolo de porcentaje (así es como C sabe que debe colocar una variable ahí). Para ello está la siguiente tabla de *identificadores*, pues hay uno para cada tipo de variable.

Lo siguiente es pasarle como argumentos los valores de las variables que queremos que imprima. Por ejemplo, vamos el siguiente programa (que de paso, incluye el uso de la función **suma**, definida antes de **main**).

```
#include<stdio.h>

// Función completamente innecesaria para sumar
```

Tipo	Identificador
int	d
unsigned int	u
long	ld
unsigned long	lu
short	hi
float	f
double	lf
char	c
unsigned char	hhu
long double	Lf
<i>apuntador</i>	p

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

int main(){
    //Declaramos dos variables, con un valor inicial
    int a = 20;
    int b = 11;
    // Llamamos a la función suma, y almacenamos su resultado en una variable
    int resultado = suma(a,b);
    // Imprimimos todo
    printf("El resultado de %d + %d es %d",a, b, resultado);
    return 0;
}
```

Ya se empezó a poner divertido esto.

### 6.1.2. Operaciones comunes

Las operaciones primarias entre variables enteras o entre variables flotantes son la suma ( $a+b$ ), la resta ( $a-b$ ), la multiplicación ( $a*b$ ), la división ( $a/b$ ) y el módulo (sólo variables enteras,  $a\%b$ ).

A esto se añade *azúcar sintáctica* como la siguiente:

- La operación  $a = a + b$  se puede reescribir como  $a += b$
- La operación  $a = a - b$  se puede reescribir como  $a -= b$
- La operación  $a = a * b$  se puede reescribir como  $a *= b$
- La operación  $a = a / b$  se puede reescribir como  $a /= b$
- La operación  $a = a + 1$  se puede reescribir como  $a++$

#### 6.1.2.1 Operadores de condición

EN la computadora, **true** es 1 y **false** es 0. Este tipo de valores los obtenemos al realizar comparaciones, por ejemplo.

5 < 7

La anterior instrucción regresa **true** o 1. Lo podemos verificar con el siguiente programa sencillo

```
#include <stdio.h>

int main(){
    int x = 5 < 7;
    printf("%d\n", x);
}
```

Los operadores de condición son menor que (<), mayor que (>), menor o igual que (<=), mayor o igual que (>=), distinto (!=) e igual (==).

### 6.1.3. Múltiples condiciones

Los operadores lógicos nos sirven para poner dos o más condicines agrupadas tal que sólo obtengamos **true** o **false** al final. Entre condiciones, un doble *ampersand* es **AND**, y un doble *pipe* es **OR**. La negación se consigue con un simple **!**. Es decir, la expresión *es x mayor a 6 y es x diferente de 10* se puede escribir como

```
x >6 && x != 10
```

Estos son los principales operadores lógicos para condiciones.

## 6.2. Instrucciones básicas

Las siguientes tres son las instrucciones básicas para hacer un programa.

### 6.2.1. if ...else

Esto es lo principal de lo principal del mundo mundial. Le permite al programa tomar dos caminos distintos, en función del cumplimiento de una condición. La estructura es la siguiente:

```
int s = 2;
// Aquí le pasa algo a s, supongamos
if(s < 2){
    printf("s es más pequeña\n");
} else {
    printf("s aumentó o quedó igual\n");
}
```

**Nota:** Ese **else** no es necesario, en realidad. La sintáxis **if ...else** es la versión completa, pero si sólo queremos que el código ejecute un cierto bloque de instrucciones en caso de que la condición sea cierta, podemos únicamente usar **if** sin escribir la parte de **else**, es decir **if (s < 2) {printf("s es más pequeña\n");}**

### 6.2.2. for

Igual que como lo hicimos en *shell*, **for** nos permite ejecutar un bloque múltiples veces. La estructura es la siguiente:

```
for(int i = 0; i <10; i++) → for( int i = 0 ; int i <10 ; i++ )
```

Condición inicial                      Condición principal                      A ejecutar al final de cada ciclo



**Nota:** La condición inicial bien podría haber utilizado una variable local de la función en la que el ciclo `for` fue declarado, pero a mi parecer es mucho mejor declarar la misma variable (cuando es posible) dentro de la condición inicial. Esto evita que la variable sea usada fuera de este ciclo.

### 6.2.3. `while`

Es también muy útil, sirve para evaluar un bloque de código hasta que una condición deje de ser cierta.

```
int i = 0;
while( i < 10 ){
    printf("i sigue siendo menor a 10");
    i++;
}
```

## 6.3. Más variables, *dirección de memoria*

Cuando se compila un programa en C, el compilador busca todas las definiciones de memoria para poder *apartar* suficiente para la ejecución del programa. A esto se le conoce como *allocation*. Las variables están en la memoria RAM durante la ejecución del programa.

Para los que no sepan qué es la memoria RAM, es simplemente otro circuito a base de silicio cuya función principal es permitirle a la computadora almacenar información *temporal* de modo que pueda acceder a ella cuando quiera y de manera rápida. Podríamos imaginar esta memoria hecha por puras cajitas chiquitas, una al lado de la otra. Cada cajita tiene una dirección, que no es más que un número en binario. En esta suposición, todo lo que querramos guardar se puede dividir en pedacitos tal que cada pedacito está en una caja.

Para conocer la dirección de memoria exacta que una variable tiene durante la ejecución del programa, existe el operador `&`. Éste se coloca antes del nombre de una variable para obtener su dirección en lugar de su valor.

Como está explicado atrás, para la función `printf` existe un identificador para imprimir algo llamado *apuntadores*. Este es el identificador que necesitaremos para imprimir la dirección de memoria, en la siguiente sección explicaré por qué.

```
#include<stdio.h>

int main(){
    int x = 10;
    printf("El valor de v en la direccion %p es %d\n", &x, x);
    return 0;
}
```

### 6.3.1. Apuntadores, pesadilla inminente

Vamos ahora al tema más divertido de todos los posibles en C: apuntadores.

Un apuntador es un tipo de variable que en lugar de almacenar un valor como cualquier variable normal, almacena una dirección de memoria. Eso sí, el apuntador también tiene tipo, y no se puede usar un apuntador que apunta a enteros para apuntar a un flotante. La instrucción siguiente define un apuntador

```
int *x;
```

Así es, basta con anteponer un asterisco al nombre del apuntador. Un apuntador inicializado de esta manera tendrá una dirección

de memoria aleatoria. Hay que saber que para sacar el valor al que apunta un apuntador se debe anteponer un asterisco al nombre del apuntador. Es algo confuso pero debería quedar claro con el siguiente ejemplo.

```
#include<stdio.h>

int main(){
    // Declaración de variable
    int x = 2;
    // Declaración de apuntador
    int *p;
    // Asignación de dirección al apuntador
    p = &x;
    printf("El valor en x, segun p que apunta a %0.16x, es %d", p, *p);
    return 0;
}
```

Por si alguien ya se lo preguntó, sí, el apuntador tiene dirección de memoria también y se obtiene usando el operador `&`, pero no nos metamos mucho en problemas ahorita.

Y bien, ¿Para qué queríamos semejante funcionalidad en el lenguaje de programación? Es aquí en donde entran las funciones que pueden alterar el valor de una variable, para ello veremos un ejemplo.

```
#include<stdio.h>

void nofunciona(int x){
    x = 10;
}

void sifunciona(int *x){
    *x = 11;
}

int main(){
    int v = 2;
    printf("El valor de v inicial es %d\n", v);
    nofunciona(v);
    printf("El valor de v despues de nofunciona es %d\n", v);
    sifunciona(&v);
    printf("El valor de v despues de sifunciona es %d\n", v);
    return 0;
}
```

**Nota:** Antes de que se comiencen a confundir, noten porfavor que los nombres de las variables en los argumentos de las funciones son solo un *alias*. El hecho de que diga `int x` como argumento en ambas funciones sólo le dice a la función que le van a pasar un entero, y ella se puede referir a él como `x`.

¿Ya vieron por qué sirven los apuntadores? La función `nofunciona` fue completamente incapaz de cambiar el valor de la variable `v`. Esto es porque C puede pasar variables de dos formas, por *valor* (que fue lo que ocurrió) que es donde **se hace una copia del contenido de la variable y se le pasa a la función**, y por *referencia* (que ahorita vemos).

Ambas funciones pasan la variable por valor, pero nos es mucho más útil tener una copia de la dirección de memoria de la variable que su valor.

#### 6.3.1.1 Pase por *referencia*

Es claro que si pasamos un apuntador en lugar del valor de la variable, vamos a tener que estar poniendo un asterisco en cada línea en que querramos modificar el valor de la variable. La solución a esto se le llama *pasar la variable por referencia*. Esto **no** es exactamente lo mismo que pasar una apuntador y poner asterisco, pero para fines prácticos lo es. La siguiente función recibe una variable por referencia

```
void tambienfunciona( int &x ){
    x = 12;
}
```

No le den muchas vueltas al significado del *ampersand*, es mera notación en este caso.

**Nota:** Como seguramente ya se dieron cuenta, a C no le importan ni los saltos de línea ni los espacios, a diferencia de *shell*. Esto ayuda porque le podemos dar un buen formato al código.

## 6.4. Casting (no encontré traducción alguna)

Este tema es una de las partes más importantes de C. Para entender lo que *casting* significa hay que tener bien claro en la mente los tipos de variable. Ya que cada variable es un recipiente con un tipo asignado, ¿Qué pasa cuando queremos hacer operaciones entre distintos tipos de variable? Compilen y ejecuten el siguiente programa. Con ello será muy evidente el problema que hay.

```
#include<stdio.h>

int main(){
    float x = 5 / 2;
    printf("Division maligna: %f\n", x);
    return 0;
}
```

El resultado de esta operación será 2.000000, lo cual claramente es incorrecto. ¿Qué pasó? La operación 5 / 2 es una operación entre dos enteros, y su resultado es un entero. C automáticamente convierte de **int** a **float**, pero lo hace en el momento equivocado, después de la división.

Para decirle a C que tome el numerador como un flotante, se utiliza algo conocido como *casting*, y es muy sencillo. Basta con colocar el tipo de variable que queremos entre paréntesis antes del elemento que queremos convertir. En este caso, lo que C hace sería equivalente a

```
int x = (float) 5 / 2
```

Para hacerlo de manera correcta, hay que escribir

```
int x = ((float) 5) / 2
```

**Nota:** En este caso hacer *casting* es un poco *overkill*, bien se podría haber escrito esa fracción como 5.0 / 2 y con eso C entiende que el 5 es un doble.

## 6.5. Variable global y local

Veamos el siguiente programa

```
int x = 2;

int main(){
    int s = x;
    return 0;
}
```

¿Cuál es la diferencia entre `x` y `s`? La variable declarada fuera de función alguna es conocida como variable global. Las variables globales siempre se inicializan a un cero aritmético. Esto puede ser de utilidad si hay algo que queremos pasar a varias funciones y queremos ahorrarnos pasar el argumento, pues las funciones pueden hacer referencia a las variables globales.

La `s` es una variable local y sólo está disponible dentro de la función `main`. En general, es mucho más recomendado utilizar variables locales aunque eso implique pasar miles de apuntadores.

## 6.6. ¿Y los archivos `.h`?

No vamos a utilizar esta característica de C de manera directa, pero es importante que la entiendan. Los archivos `.h` se supone deben contener únicamente declaraciones de funciones, más no su implementación.

El propósito de esto es hacer el código reutilizable. Como vimos al principio, C puede generar el código objeto en la 3ra etapa de la compilación, y este deberá *conectarse* con los bloques restantes ya sea que se encuentren en una biblioteca o otras partes del mismo código. La idea es entonces conservar este código objeto y darle a los programas sólo una explicación de su valor de retorno, los valores que toma como argumentos y su nombre.

Como ejemplo vamos a hacerlo en un único archivo. En lugar de usar `include`, simplemente definimos una función sin decirle a C lo que pasa dentro de ella. La podemos *usar* de inmediato (o más bien, C) no se va a quejar). La implementación de la función la hacemos al final del código.

```
#include <stdio.h>

// Declaramos la función primero
void saluda(void);

// La utilizamos en main
int main(){
    saluda();
    return 0;
}

// Ahora implementamos la función
void saluda(void){
    printf("Holaaaa!");
}
```

## 6.7. Recibiendo información

En general vamos a tratar de pasar esto a C++ porque es un poco más fácil pero vamos a intentar hacerlo con C también.

### 6.7.1. Lectura por `stdin`

Compilen el siguiente programa y ejecútelo

```
#include <stdio.h>

int main(){
    int x;
    printf("Ingrese un numero: ");
    scanf("%d", &x);
    printf("Usted ingreso %d\n", x);
    return 0;
}
```

**Nota:** La función `scanf` puede ser fuente de ataques, uno conocido como *stack buffer overflow attack*. Sin embargo, nosotros le vamos a dar un uso muy básico y controlado.

`scanf` sigue la notación que `printf` entiende. Como ejemplo, si ustedes quisieran leer 2 números separados por un espacio, escribirían `scanf("%d %d", &var1, &var2)`.

### 6.7.2. Lectura y escritura de archivos

C tiene un tipo de variable llamado `FILE`. Este tipo de variable se puede utilizar para escribir cosas a un archivo

```
#include<stdio.h>
int main(){
    FILE *file = fopen("hola.txt", "w");
    fprintf(file, "Hola mundo!\n");
    fclose(file);
}
```

Ejecuten ese código y se va a crear un archivo llamado `hola.txt` que contiene la cadena de caracteres `Hola mundo!`.

La función `fopen` toma como primer argumento el nombre del archivo, y como segundo argumento un identificador de qué tipo de operación vamos a realizar (`r` para *read*, `w` para *write*). `fsclose` se encarga de cerrar el archivo y liberarlo. Siempre se debe llamar al acabar de utilizar el apuntador.

Para leer de un archivo, se utiliza la función `fscanf` la cual funciona idénticamente a `scanf`. Imaginemos que tenemos un archivo llamado `datos.txt` de 10 líneas, y cada línea tiene el mismo formato, por ejemplo:

```
1, 2
2, 6
3, 0
4, 9
5, 3
6, 8
7, 1
8, 1
9, 12
10, 10
```

Para leer el archivo vamos a usar un ciclo `while` pues vamos a leer el archivo línea por línea.

```
#include <stdio.h>

int main(){
    FILE *file = fopen("datos.txt", "r");
```

```

int x,y, counter = 0;
while(fscanf(file, "%d, %d", &x, &y) != EOF){
    printf("%d y %d estan en la lineea %d\n", x, y, counter);
    counter++;
}
fclose(file);
}

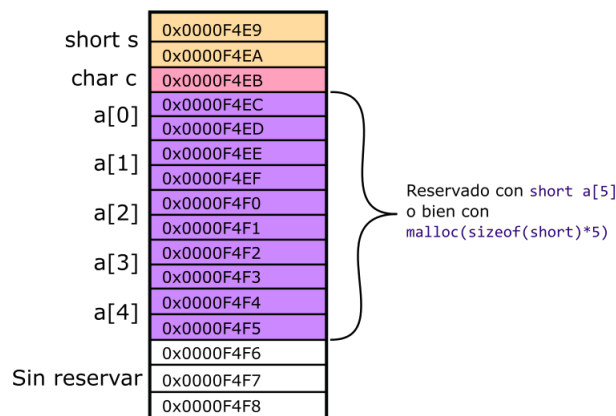
```

La función `fscanf` toma como argumentos el apuntador al archivo, la *estructura* de cada línea y los *recipientes* para guardar los valores extraídos. El valor de retorno de esta función tiene que ver con el éxito o fracaso de la ejecución de la instrucción. El caracter EOF es un caracter especial que identifica el final de un archivo, y `fscanf` regresa EOF cuando ha llegado al final del archivo.

## 6.8. Arrays, malloc, free y sizeof

Vamos a suponer que queremos almacenar no un valor, pero una *lista* de valores. En C hay dos maneras de completar esta tarea

- Usar notación de arreglos: `short a[5]` declara un arreglo de 5 variables tipo short. Se puede acceder a cada una de las entradas utilizando la misma notación, es decir, una vez declarado el arreglo `a` se accede a el miembro 1 con `a[0]`, al segundo con `a[1]` y así hasta el quinto, `a[4]`.



**Nota:** Los arreglos, vectores y caracteres en una cadena en C comienzan en 0. No en todos los lenguajes es lo mismo. Nunca olviden esto porque es común fuente de errores.

Usando esta *técnica*, la memoria se reserva en algo conocido como el *stack*, es un espacio especial para variables locales y llamadas a funciones. Su espacio es limitado, depende de la implementación del compilador pero puede ser de unos 8MB. Noten que si estuviera libre la memoria, con dificultad nos alcanzaría para hacer un array de  $10^6$  entradas. Además, esta memoria sólo está reservada mientras la función que declara el arreglo exista.

- Usar reservación de memoria: `short *a = (short*) malloc(sizeof(short)*5)`, se ve horrible pero no es complicado de entender. La función `malloc` reserva el número de *bytes* que se le da como argumento y regresa un apuntador del tipo `void *` apuntando a la primer posición de la memoria reservada. La función `sizeof` da el número de bytes que ocupa el tipo que se le pasa como argumento.

Así que en resumen, reservamos un espacio de `sizeof(short)*5` bytes, y hacemos un *cast* a `short *` del apuntador que regresa `malloc`. Para acceder a los elementos a los que este apuntador *apunta* se utiliza la misma notación de corchetes cuadrados, `a[2]` para el tercer elemento y así.

Eso sí, al final de su utilización **siempre debemos liberar la memoria reservada**. Esto se hace llamando a `free` pasando como único argumento el apuntador que obtuvimos al reservar la memoria.

Noten porfavor que en ambos casos la memoria es contigua.

## 6.9. Funciones matemáticas

C tiene soporte para funciones matemáticas. Para poder utilizarlas, hay que incluir el *header* `math.h`, y al compilar con `gcc` hay que añadir la opción `-lm` (esto le dice al compilador que utilice el código objeto de la biblioteca de matemáticas a la hora del *linking*).

Dentro de esta biblioteca van a encontrar funciones como `sin`, `cos` y `exp`. Para la lista completa porfavor visiten [este sitio](#).

Por si tienen curiosidad de cómo le hace C para calcular la funciones trigonométricas, ese será tema de su curso de física computacional. Pero, dependiendo de la implementación (es decir, la biblioteca `math` que venga con su compilador), se suelen utilizar polinomios de Taylor, o de Chebyshev, así como *reducción de argumento*.

### 6.9.1. $\pi$

El número  $\pi$  generalmente viene definido dentro de `math.h` como `M_PI`, pero en caso de no existir, simplemente pónganlo en una variable usando `4*atan(1)`.

## 7. C++

El lenguaje C en sí es muy, muy poderoso, pero podríamos decir (muy entre comillas) que no es de tan alto nivel, y es complicado hacer muchas cosas. En particular, hay una característica conocida como *programación orientada a objetos* que C simplemente no puede hacer. Para ello existe un lenguaje conocido como C++ que tiene como subconjunto a C (y sí, el ++ es para indicar que es como C pero aumentado en 1).

### 7.1. Programación orientada a objetos

Un lenguaje orientado a objetos es aquel que permite la creación de **objetos** que contienen la información en forma de campos o en la forma de métodos. Estos lenguajes nos son muy naturales de utilizar. Además de esto, los lenguajes de programación orientada a objetos contienen ciertas características que permiten eliminar redundancia del código. Esto lo vamos a ver hasta la próxima vez.

### 7.2. Las pequeñas diferencias

Ya que C es un subconjunto de C++, podemos seguir incluyendo las mismas bibliotecas y demás. Sin embargo, C++ tiene bibliotecas exclusivas como lo es `iostream` que provee funcionalidad para imprimir y leer de `stdin` (esto reemplaza a `stdio.h` en C).

Otra gran diferencia es que en el caso de usar la biblioteca matemática, ya no se necesita hacer un *linking* con `-lm`. Los archivos fuente terminan por `.cpp` y los *headers* a veces por `.hpp`. Otra diferencia *sustancial* es el comando para compilar, `g++` y no `gcc`.

### 7.3. Clases en C++

En C++ este elemento fundamental llamado objeto se declara por medio de la palabra `class`. Es decir, el elemento abstracto se llama *clase* y su instancia se llama *objeto*. Vamos a suponer que queremos declarar una clase que se llame `Particula` que tenga como variables miembro tres coordenadas

```
class Particula{
public:
    float x;
    float y;
    float z;

    void saludo(){
        // Así es como se imprime en terminal en C++, lo explico después.
        std::cout << "Aquí estoy!" << std::endl;
    }
};
```

**Nota:** La definición de clases **sí** debe llevar punto y coma al final, como si fuera una instrucción.

Bien, pues ahora podemos utilizar esta *clase* para declarar partículas, digamos que el nombre de la clase pasa a ser un tipo de variable y lo utilizaremos como tal.

```
Particula a;
// Asignar un valor a la coordenada y
a.y = 10.0;
// Llamar a la función miembro saludo
a.saludo();
```



Para acceder a cualquiera de las variables miembro o funciones miembro, se utiliza el punto (.) como operador. No nos vamos a meter muy a fondo en todas las posibilidades que C++ ofrece, pero vamos a hablar un poco de muchas de sus características. La palabra **public** que antecede la declaración de las variables y de la función le dice a C++ que los miembros que se encuentren después de esa palabra podrán ser accedidos por cualquiera en la ejecución del programa.

Existen otras dos posibilidades, **protected** y **private**. Para entender **protected** vamos a necesitar más elementos y queda para después. **private** le dice a C++ que los miembros sólo se pueden acceder desde una instancia de la clase **Particula**. Esta característica es útil para evitar que un usuario haga un uso indebido del código que hacemos (por ejemplo, podríamos hacer la posición un campo privado para evitar que un usuario los cambie de manera declarativa y sólo se puedan cambiar como consecuencia de un campo de fuerza, algo que también podemos escribir como un *objeto*).

Hagan el ejercicio: cambien la palabra **public** a **private** (u omítanla, es el comportamiento por defecto de las clases) e intenten compilar un programa donde llamen la función **saludo**.

### 7.3.1. La palabra **this**

La palabra **this** es una palabra reservada en C++. Sirve para hacer referencia a la instancia desde la cual se llama la función miembro de una clase. Léase, es un apuntador que tiene como valor la dirección de la instancia desde la que se llamó la función miembro. Pudimos hacer que la función **saludo** cambiara el valor de alguna de las coordenadas, y para eso podemos usar **this**

```
void saludo(){
    // Así es como se imprime en terminal en C++, lo explico después.
    std::cout << "Aquí estoy!" << std::endl;
    // Cambiamos el valor de la variable miembro x
    this->x = 0;
}
```

**Nota:** Hemos dicho que **this** es un apuntador, por lo que hacer **this.x** sería incorrecto. Lo correcto sería escribir **(\*this).x** pero C++ tiene *azúcar sintáctica* para esta operación, el operador flechita **->** que es equivalente.

Usualmente se puede evitar escribir **this** y el compilador reconoce de inmediato que hacemos referencia a una variable miembro, sin embargo esto puede generar problemas en presencia de variables globales o de argumentos proporcionados a la función que compartan el nombre de una variable miembro. Comúnmente se antepone un guión bajo a todas las variables miembro para seguir sin escribir **this**, pero usar **this** es lo más correcto.

### 7.3.2. Constructor

Una de las cosas más útiles en una clase es el *constructor*. Esta es una función que se llama durante la creación de una instancia de una clase. Por ejemplo, la partícula que habíamos descrito anteriormente no inicializa sus variables y por lo tanto **contienen basura**, pero ¿Qué tal si queremos darle un identificador y valores a las 3 coordenadas desde un inicio? Un constructor se identifica porque lleva por nombre el mismo que la clase y no son antecedidas por tipo alguno de variable.

Una clase puede tener varios constructores, y el compilador seleccionará automáticamente el correcto según el tipo de argumentos con el que una instancia es creada.

```
#include<iostream>

class Particula{
public:
    int id;
    float x, y, z;

    // Constructor que nada hace, deja todo en 0
```

```

Particula(){};

// Constructor más sofisticado
Particula(int id, float x, float y, float z){
    this->x = x;
    this->y = y;
    this->z = z;
    this->id = id;
}

// Esta función imprime los datos de la partícula
void donde(){
    std::cout << "Soy " << id << " y estoy en " << x << ", " << y << ", " << z << std::endl;
}
};

int main(){
    // Creamos una particula con el constructor chafa (es el mismo que por defecto)
    Particula a;
    // Creamos una partícula con id = 11, y posición (1,2,3)
    Particula b(11,1,2,3);
    a.donde();
    b.donde();
}

```

Ejecuten ese código y verán lo maravilloso que C++ puede llegar a ser. Así como existe el constructor, existe el destructor `~Particula` que se llama al eliminar una instancia de una clase, aunque esta característica es un poco avanzada y no la utilizaremos por el momento.

## 7.4. Variables estáticas y *namespaces*

Tampoco vamos a ir de fondo con esto, pero sólo lo voy a explicar para que entiendan qué significa la sintaxis `std::cout`, los cuatro puntos intermedios vaya.

Supongamos que tenemos una clase con un nombre tan común que las probabilidades de que exista en alguna otra biblioteca (sólo el nombre, no la funcionalidad) es muy alta. Digamos, **vector**. ¿Hay algo que podamos hacer para distinguirlos? La respuesta es sí, y es lo que la biblioteca estándar de C++ hace. `cout` es un objeto definido dentro de `iostream`, y se encuentra dentro de un *namespace* llamado `std` (para *estándar*). Nosotros no vamos a ver como definirlos, pero esto obliga a la persona que quiera usar estos objetos a anteceder su nombre con el *namespace* y cuatro puntos. Es decir, la clase **vector** que está dentro del archivo **vector** se tiene que acceder como `std::vector`, porque el autor decidió llamar a su namespace como `std`. Sólo es importante que entiendan lo que significa.

### 7.4.1. ¿Y las variables estáticas?

Una variable estática es una variable cuyo valor está ligado a la clase y no a sus instancias. Esto quiere decir que es como una *constante*. Un ejemplo sería el siguiente

```

class CosasMatematicas{
public:
    static float pi = 3.141592;
}

```

Ahora pueden acceder a esta variable con los 4 puntos, tal y como con los namespaces

```
float pi2 = CosasMatematicas::pi / 2
```

**Nota:** Las variables estáticas son parte de la clase y no de las instancias, por lo mismo no se necesita declarar una instancia de la clase para acceder a ella. También se pueden declarar funciones estáticas, con la condición de que, si hacen uso de variables u otros métodos miembro de la clase a la que pertenecen, todas esas variables y métodos **también** deben ser estáticos.

## 7.5. Cadenas de caracteres

Este tipo de variable es mucho, mucho más divertido aquí que en C. Si recuerdan, en C las cadenas no son más que un arreglo de caracteres. Aquí son una clase en sí, están en el archivo `string` (por lo que lo deben incluir con `#include`). Dentro de `main`, pueden declarar cadenas de la siguiente manera:

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

int main(){
    std::string cadenita = "Hola ";
    cadenita += "mundo!";
    std::cout << cadenita << std::endl;
    return 0;
}
```

Ahora sí podemos explicar mejor lo que hacen `cout`, `endl` y `<<`

`cout` y `cin` son objetos, conocidos como **streams**, que en este caso hacen referencia a `stdout` y `stdin` respectivamente. Por salud mental, pueden imaginar que esto es lo equivalente a cuando en la terminal ustedes ejecutan algo como `echo "hey!" > cout.txt`, es decir, redirige información *dentro* de un objeto. La funcionalidad real de `<<` es de hacer operaciones a nivel de bits, pero en este caso el operador está *sobrecargado* para detectar automáticamente los objetos que tiene a la izquierda y a la derecha.

Lo mismo pasa con el signo `+` que nos permitió concatenar cadenas de una manera limpia.

`endl` es casi igual a pasar `"\n"`, sólo que además vacía el *buffer* de la salida estándar.

Se puede acceder a los caracteres de la cadena tal y como lo hacíamos con los arreglos en C, usando el operador `[ ]`

### 7.5.1. La función `to_string`

¿Qué pasa si tenemos un flotante y queremos convertirlo en una cadena de caracteres? Desde C++11 existe la función estática `to_string`, la cual convierte enteros y flotantes a una cadena de caracteres.

```
std::string numero = std::to_string(10.18203);
```

### 7.5.2. Conversión inversa

Para las conversiones en sentido opuesto existen dos métodos, `stof` y `stoi` (*string-to-float* y *string-to-int*).

```
float pi = std::stof("3.141592");
int micalif = std::stoi("10");
```

## 7.6. Escritura y lectura

Ya vimos como escribir cosas a `stdout`, ahora vamos a ver cómo leer cantidades ingresadas por un usuario.

### 7.6.1. Lectura de `stdin`

Compile y ejecute el siguiente programa

```
#include<iostream>
#include<cmath>

double pi = atan(1)*4;

int main(){
    std::cout << "Ingrese el radio de un círculo: ";
    float r;
    std::cin >> r;
    std::cout << "Este círculo tiene área " << pi*r*r << std::endl;
    return 0;
}
```

El programa leerá un valor flotante de la terminal, escrito por un usuario, y lo guardará en `r`. Si quisiera un par de valores separados por espacios, la instrucción `std::cin >> a >> b` guardaría estos valores en `a` y `b` respectivamente.

### 7.6.2. Lectura y escritura de archivos

Gracias al cielo, la manera de escribir a archivos es idéntica a la de escribir a la salida estándar, sólo que en lugar de usar los objetos `cin` y `cout` usaremos *file streams*.

```
#include<fstream>

int main(){
    std::fstream myFile("archivito.txt", std::ios::out);
    myFile << "Soy un archivito :v\n";
    myFile.close();
    return 0;
}
```

Como pueden ver, es algo muy sencillo, el constructor de un `fstream` toma como argumento el nombre del archivo, y después el *tipo* de operación que realizará con el archivo. Es igual a `fopen` en C sólo que en lugar de escribir "`w`" y "`r`" se escribe `std::ios::out` y `std::ios::in` para escritura y lectura respectivamente.

El código anterior creará un archivo llamado *archivito.txt* que contendrá una línea. Para leer es un poco más complicado, pero se puede hacer de la siguiente manera

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

int main(){
    // Abrimos el archivo
    std::fstream inFile("archivito.txt", std::ios::in);
```

```

// Verificamos que se pudo abrir, a la mejor no existe :/
if(!inFile.is_open()){
    std::cout << "No se pudo abrir el archivo!" << std::endl;
    return -1;
}
// Aquí vamos a almacenar cada línea
std::string line;
// Ciclo para leer cada línea, la función getline está en string
while ( getline(inFile,line) ) {
    std::cout << line << std::endl;
}
// Siempre cerramos el archivo antes de terminar el programa.
inFile.close();
return 0;
}

```

## 7.7. Estructuras de datos

C++ viene con ciertas clases que nos ayudan a realizar diversas operaciones. Vamos a ver dos muy curiosas.

### 7.7.1. `std::vector`

Pues, literal es un vector pero que es capaz de almacenar distintos tipos de clases. Es un *array* ó arreglo pero de memoria dinámica, lo cual lo hace bellísimo. Tiene múltiples constructores. No voy a explicar lo que significan los paréntesis angulares (< y >), sólo recuerden la notación. Frente al nombre de la clase **vector**, se tiene que poner entre paréntesis angulares el tipo de variable que va a almacenar.

```
int myVector[10] ≈ std::vector<int> myVector(10)
```

Eso que he escrito es sólo para decir que lo de la izquierda, que está en C, sería la operación similar a lo de la derecha, que está en C++. He utilizado el constructor que toma como único argumento el número inicial de valores que va a tener el vector. El constructor por defecto crea un vector de 0 entradas.

Hay 3 métodos principales que deberíamos no olvidar:

- **size**, el cual no toma argumentos y regresa el número de elementos que el vector contiene
- **push\_back**, el cual toma como argumento un valor y lo añade *al final del vector*
- **pop\_back**, el cual no toma argumentos y elimina la última entrada del vector

Se puede acceder a los elementos del vector de la misma manera que a un arreglo, con el operador [ ].

```

#include<vector>
#include<iostream>

int main(){
    std::vector<int> v = {6, 4, 1, 5};
    std::cout << "Los elementos del vector son : ";
    for(unsigned int i = 0; i < v.size(); i++) std::cout << v[i] << ((i!=v.size()-1) ? ", " : "");
    std::cout << std::endl;
    return 0;
}

```

**Nota:** Ese último elemento de la instrucción dentro del `for` es una manera muy elegante de poner un `if ...else` en una línea. Esa instrucción es equivalente a `if(i!=v.size()-1){ return ", ";}else{ return ""};`

### 7.7.2. `std::set`

Este es un conjunto. Existen métodos para hacer las operaciones entre conjuntos (unión, diferencia, intersección) pero no los usaremos, eso se lo vamos a dejar al siguiente lenguaje que aprenderemos. Lo básico que haremos con un `std::set` (contenido en el archivo `set`) es tener un conjunto de elementos únicos. El constructor por defecto crea un conjunto vacío, aunque de todos modos hay que especificar el tipo de elementos que va a contener

```
std::set<char>myCharSet;
```

Hay tres métodos importantes a recordar

- `size`, el cual no toma argumentos y regresa el número de elementos que el conjunto contiene
- `insert`, el cual toma como argumento un valor y lo añade al conjunto en caso de no estar ya incluido
- `erase`, el cual toma un argumento y lo elimina del conjunto en caso de estar contenido

### 7.7.3. No son los únicos

C++ tiene más estructuras de datos, por ejemplo `std::map` que sirve para hacer un mapeo de un objeto a otro objeto (son conocidos como diccionarios).

En general, las estructuras de datos son cruciales para la computación, pues permiten muchas veces hacer un uso más eficiente de los recursos disponibles. Las búsquedas pueden mejorar su eficiencia rápidamente cuando se usa una estructura de datos correcta, como un árbol de búsqueda binaria.

## 8. Graficando con C++

Bueno, pues viendo que Python ya se ha convertido en una utopía, un horizonte, el tema ha quedado ya fuera del curso. Lo del horizonte lo digo por esta bella cita de Fernando Birri

*La utopía está en el horizonte. Camino dos pasos, ella se aleja dos pasos y el horizonte se corre diez pasos más allá. ¿Entonces para que sirve la utopía? Para eso, sirve para avanzar*

Léase con acento Argentino para magnificar el impacto de la frase. En fin, ahora hay que ver para adelante y seguir caminando. Es por eso que nos vamos a auxiliar de un *header* superútil para graficar con Python, pero sin usar Python.

### 8.1. matplotlib.cpp

Esta biblioteca la encontré en [internet](#), y me encuentro superagradecido con el autor pues nos está facilitando la vida. La biblioteca se llama `matplotlibcpp`, y sirve para poder llamar funciones de la biblioteca `matplotlib` de Python desde C++. A esto se le conoce como un *wrapper*.

Esto en particular es fácil porque Python está escrito en C++, y los mismos desarrolladores hacen también las bibliotecas para poder interactuar entre estos dos lenguajes.

#### 8.1.1. Dependencias

Hay un par de cosas que debemos dejar listas para poder utilizar `matplotlibcpp`. Tanto en WSL como en las distribuciones de Linux (preferentemente Ubuntu), necesitamos instalar las bibliotecas de desarrollo de Python 2.7

```
sudo apt install python-dev
```

Además de eso, necesitamos las bibliotecas de `matplotlib` y `numpy`.

```
sudo apt install python-matplotlib python-numpy
```

Ahora sí estamos más que listos.

#### 8.1.2. Un código de ejemplo

El código que está aquí es nuestro ejemplo mínimo de cómo usar la biblioteca. Vamos a colocarlo en un archivo llamado `main.cpp` para poder dar las instrucciones de cómo compilar

```
#include"matplotlibcpp.h"
#include<vector>

// matplotlibcpp es lo equivalente a std, pero decidido por el autor del header
namespace plt = matplotlibcpp;

int main(){
    plt::backend("Agg");

    // Data vectors
```

```

std::vector<float> x, y;
x.push_back(0); x.push_back(1);
y.push_back(0); y.push_back(1);
// Now we plot
plt::plot(x, y);
plt::save("example.png");
return 0;
}

```

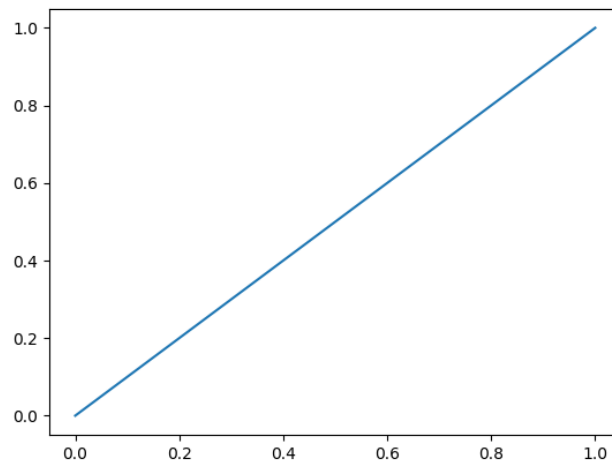
**Nota:** Esta vez el comando para el precompilado tiene comillas en lugar de brackets (me refiero al `#include`), esto es para que la búsqueda del *header* sea dentro del folder en el que se compila en lugar de `/usr/include` y el resto de rutas por defecto.

Para compilar el código, vamos a tener que decirle a `g++` en dónde están los headers de desarrollador de `Python`. Para ello está la opción `-I` (de manera similar, podemos especificar rutas diferentes para encontrar bibliotecas con la opción `-L`).

Además de decirle esto, le vamos a decir que el código correspondiente a los headers en `/usr/include/python2.7` se encuentra en el archivo `libpython2.7.a`, lo cual espero se hace con la opción `-l` e ignorando el `lib` y la terminación `.a`

```
g++ main.cpp -I/usr/include/python2.7 -lpython2.7
```

El resultado de ejecutar el `a.out` resultante de este código será una imagen con el nombre `example.png`. En este caso, con el siguiente contenido



Les daré una tablita de los comandos más útiles que pueden usar con `matplotlibcpp`, y haré un único ejemplo más pero bastante más complejo

Hay más funciones pero esas les deberían ser suficientes para fines prácticos. La función principal, `plt::plot` no la puse en la tabla porque su funcionamiento es más *complejo*, y lo explico aquí.

Esa función puede ser llamada de varias formas, pero nos vamos a enfocar sólo en dos. Primero, en llamarla como `plt::plot(x,y)`, donde `x` y `y` son del tipo `std::vector<float>` (o puede ser `double`, `int`, `long`, los que sean que sean numéricos). Esto grafica el vector `x` contra el vector `y`, entrada por entrada. Es decir, grafica los puntos  $\{(x_i, y_i)\}_{i=0}^{n-1}$  donde el tamaño de cada vector es  $n$ .

Ahora, bien podría ser que queramos graficar con un color en específico, o poner una gráfica de dispersión (es decir, no unir los puntos, sólo graficar los puntos).



Función	Efecto...
<code>plt::figure_size(w, h)</code>	Establece el tamaño de la <i>figura</i> en $w \times h$ pixeles
<code>plt::save("nombre.png")</code>	Guarda la <i>figura</i> bajo el nombre <code>nombre.png</code>
<code>plt::title("Grafiqueishon")</code>	Pone el título <i>Grafiqueishon</i> a la figura
<code>plt::xlabel("Tiempo")</code>	Pone la etiqueta <i>Tiempo</i> al eje x
<code>plt::ylabel("Velocidad")</code>	Pone la etiqueta <i>Velocidad</i> al eje y
<code>plt::grid(true)</code>	Cuadricula el espacio (o no si se usa <i>false</i> )
<code>plt::xlim(l, r)</code>	Define el rango de graficado como $[l, r]$
<code>plt::ylim(d, u)</code>	Define el rango de graficado como $[d, u]$

## 9. Números Pseudoaleatorios

¿Cómo le hacemos para generar números aleatorios? Y antes que eso, para qué demonios queremos números aleatorios.

### 9.1. Para qué

Aún están muy pequeños para entender la necesidad de numeros aleatorios, pero en semestres más adelantados van a conocerla. Como casos particulares, hay fenómenos físicos en donde es imposible hacer un modelo simple que describa con precisión el comportamiento de un sistema de muchísimas variables. Encima, hay fenómenos para los que sólo podemos encontrar una descripción probabilística respecto a su comportamiento. Para estos casos se vuelve necesario poder hacer un muestreo de una distribución en específico.

### 9.2. Cómo

Lo primero que hay que hacer es dar una definición muy filosófica respecto a lo que aleatorio significa. ¿Qué significa que un fenómeno es aleatorio? Un poco laxos, significa que no tenemos manera de predecir el resultado o la evolución de un fenómeno cuando éste es aleatorio.

Es justo esta definición la que nos va a permitir generar números *aleatorios* en la computadora.

De esa definición se puede seguir que es nuestra ignorancia la que convierte a un fenómeno en aleatorio. Lanzar una moneda, en principio, no es un fenómeno aleatorio. Para alguien que tiene un poder de cómputo infinito y un conocimiento absoluto y preciso de todas las variables involucradas en el giro de la moneda, esto no sería un evento aleatorio. En la física, sólo hay un conjunto de fenómenos que son considerados 100% aleatorios, y ellos son los fenómenos de la física cuántica (pues la misma descripción está basada en *espacios de probabilidad*). Y aún así, eso es lo que pensamos hasta ahora, es *probable* que haya otro modelo que pueda predecir con exactitud los fenómenos cuánticos.

En todo caso, para una computadora no es sencillo producir fenómenos cuánticos para obtener variables aleatorias. Existen periféricos que generan números aleatorios ayudados de fenómenos cuánticos, pero esto no es un elemento estandar de las computadoras comerciales. Es por ello que vamos a conformarnos con generar una secuencia de números cuya regla es desconocida para nosotros.

#### 9.2.1. Método de Lehmer

La regla más simple que veremos (y la única) es la siguiente: haremos una cadena de markov (esto quiere decir, una serie de números que sólo dependen del número anterior) con operaciones elementales y que se encuentre acotada por un intervalo de enteros positivos más el cero.

Escribiéndolo ya de manera formal, la secuencia es la siguiente (conocido como método de Lehmer)

$$X_{i+1} = aX_n + c \mod(n)$$

El número  $X_0$  es conocido como el número semilla, pues una vez que se conoce, la secuencia queda determinada de manera única.

### 9.2.1.1 Sobre la longitud de la secuencia

Tomemos  $n = 7, a = 3, c = 5$  y  $X_0 = 2$  con lo que llegamos a la secuencia

$$\{2, 4, 3, 0, 5, 6, 2, \dots\}$$

Es casi una secuencia bendita, pues cubre casi todos los números posibles en el rango que el módulo permite, osea entre el 0 y el 6. EL único número que queda, es el 1, con el que la secuencia apesta porque  $(3(1) + 5) \bmod(7) = 8 \bmod(7) = 1$ . Esta secuencia es todo menos buena. Entonces el número semilla no puede ser cualquiera.

No será nuestro tema de estudio, pero hay *buenas* y *malas* elecciones de  $a, c$  y  $n$  que permiten utilizar el rango en toda su extensión.

Sólo como ejemplo, los parámetros  $n = 8, a = 17, c = 5$  generan una mejor secuencia (comenzando con  $X_0 = 1$ )

$$\{1, 6, 3, 0, 5, 2, 7, 4, 1, \dots\}$$

De aquí sale un pequeño problema. ¿Qué tal si nosotros queríamos forzosamente se quería una variable aleatoria  $X \in [0, 6]$ ?

### 9.2.2. Números reales

¿Qué tal si lo que queremos es una variable real aleatoria también? Por la misma naturaleza de los números en una computadora, los números ya se van a encontrar acotados. Muy fácilmente podemos utilizar el método de Lehmer para obtener un conjunto de *reales* en un intervalo. Si optamos por el intervalo  $[0, 1]$  (*que siempre es un buen intervalo*), bastaría con tomar en cuenta la secuencia  $\{r_i = X_i/(n-1)\}$  en donde  $X_i = aX_{i-1} + c \bmod(n)$ . Usando la última secuencia exitosa, nosotros obtendríamos la secuencia

$$\{0.1428, 0.8571, 0.4285, 0, 0.7142, 0.2857, 1, 0.5714, 0.1428, \dots\}$$

### 9.2.3. Cambiando el rango

Vamos primero a ver el caso de los reales, pues esto es más fácil. Si  $X$  es una variable aleatoria que toma valores de manera uniforme en el intervalo  $[0, 1]$ , entonces se puede conseguir una función de variable aleatoria  $Y$  que toma valores en el rango  $[a, b]$  definiendo

$$Y(X) = (b - a)X + a$$

#### 9.2.3.1 Caso entero

El caso de las variables aleatorias enteras no es tan sencillo, supongamos que tenemos nuestro generador uniforme maravilloso de Lehmer que avienta números entre el 0 y el 7. No debería ser muy increíble notar que si hacemos 7k mediciones (con  $k \in \{1, 2, \dots\}$ ) la distribución va a ser perfecta.

Un primer instinto para limitar el rango es aplicar un módulo, pero esto NO es una buena idea. Si aplicamos un módulo, vamos a duplicar la frecuencia de aparición del 0 (pues todas las veces que salga el número 7, se irá hacia el 0).

No les voy a decir cómo se soluciona esto. *Les dejo sólo promesas para su 7mo semestre.*

**Nota:** En general no tienen que preocuparse por esto, C++ tiene maneras de ajustar el rango y *garantizar* la uniformidad de la distribución.

## 9.3. Generación de números pseudoaleatorios en C/C++

### 9.3.1. C

Para generar números pseudoaleatorios en C se utilizan dos funciones, `rand()` que devuelve números enteros pseudoaleatorios en  $[0, \text{RAND\_MAX}]$  (`RAND_MAX` está definido al preproceso en `stdlib.h`, al igual que las dos funciones) y la función `srand(int s)` que sirve para definir el número semilla  $X_0$ . Es una buena costumbre utilizar como semilla lo que devuelve la función `time()` evaluada en 0 (que está definida en `time.h`)

### 9.3.2. C++

Para generar números pseudoaleatorios en C++ la historia es un poco más complicada. Existe el header `random` en donde hay unas clases definidas. Un ejemplo de código para generar variables enteras y continuas es el siguiente

```
#include<iostream>
#include<random>
#include<chrono>

int main(){
    unsigned long int seed = static_cast<unsigned long int>(
        std::chrono::system_clock::now().time_since_epoch().count()
    );
    std::default_random_engine g{seed};

    // Distribución entera entre 0 y 6
    std::uniform_int_distribution<int> uniform(0, 6);
    // Distribución continua entre 0 y 1
    std::uniform_real_distribution<float> uniformR(0, 1);

    // Generar el entero aleatorio
    std::cout << uniform(g) << std::endl;
    // Generar el flotante aleatorio
    std::cout << uniformR(g) << std::endl;
    return 0;
}
```

### 9.3.3. Números no deterministas

La computadora tiene un mecanismo aún más complejo para generar números un tanto más aleatorios. La computadora toma *entropía* (es la manera en que se le llama) de movimientos del mouse o de las teclas que se han apretado. En principio, estas dos cosas son bastante complicadas de predecir por lo que son un buen candidato a generar números aleatorios. EN C++ esto lo hace la clase `std::random_device`.

## 9.4. Un experimento chistoso

Vamos a hacer un *ejercicio numérico* para ver cuál es la probabilidad de que dos personas cumplan años el mismo día en un grupo de unas  $n$  personas.

Debe resultar un poco obvio que si  $n = 365$  la probabilidad es 1, y si  $n = 1$  la probabilidad es 0. Lo que nos interesa es ver cómo se comporta esto entre esos dos números. Vamos a usar la clase `std::set` para este ejercicio de una manera muy sencilla. La mecánica será la siguiente:

Vamos a repetir el experimento `limit` veces, y en cada una hacemos lo siguiente: Dado un número de personas `n` metemos sus fechas de cumpleaños en un `std::set`, el cual no almacena repeticiones de elementos. Es decir, si el tamaño del set es más pequeño que el número de personas, quiere decir que alguien cumplió años el mismo día que otra persona. Al final, contamos en cuántos de los grupos aleatorios hubo personas que cumplieron años el mismo día, y en cuántos no.

Vamos a repetir esto para  $n = \{1, 2, \dots, 365\}$ . El código que hace esto, y además grafica, es el siguiente:

```
#include<iostream>
#include<random>
#include<chrono>
#include<set>

#define WITHOUT_NUMPY
#include"matplotlibcpp.h"

namespace plt = matplotlibcpp;

int main(){
    plt::backend("Agg");
    std::vector<int> enes; //vector de las enes
    std::vector<float> probs; // vector de las probabilidades

    unsigned long int seed = static_cast<unsigned long int>(
        std::chrono::system_clock::now().time_since_epoch().count()
    );
    std::default_random_engine g{seed};

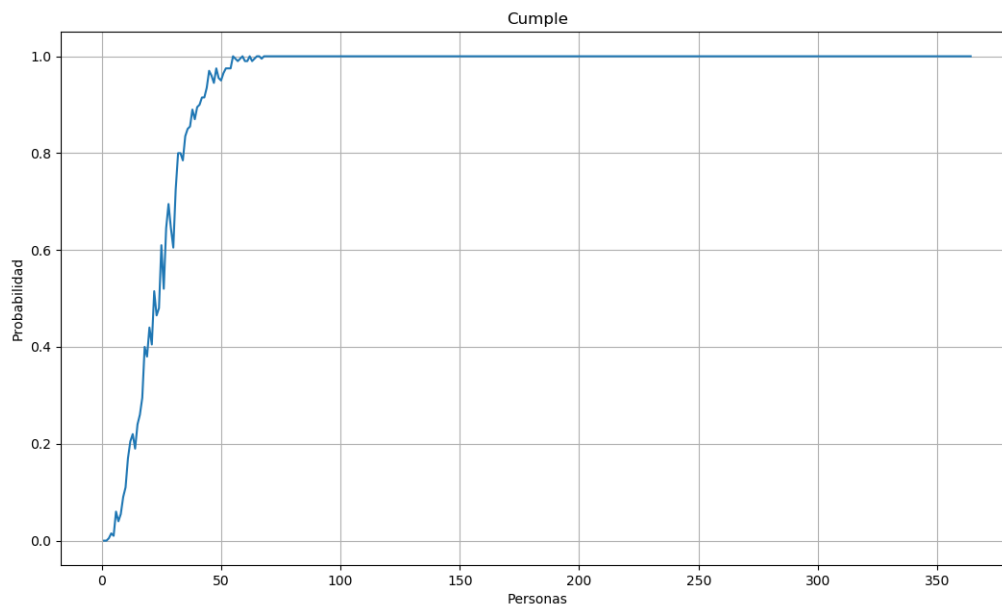
    // Distribución entera entre 0 y 6
    std::uniform_int_distribution<int> uniform(0, 365);

    int limit = 5000;
    for(int i = 1; i < 101; i++){
        enes.push_back(i);
        int cnt = 0;
        for(int rep = 0; rep < limit; rep++){
            std::set<int> cumple;
            for(int j = 0; j < i; j++){
                cumple.insert(uniform(g));
                //Checamos en cada ronda si alguien ya cumplió mismo día
                if(cumple.size() <= j){
                    cnt++;
                    break;
                }
            }
        }
        probs.push_back(((float) cnt) / limit);
        if(i%20 == 0){
            std::cout << i << ": " << probs[probs.size()-1] << std::endl;
        }
    }

    // Ahora sí el momento de graficar
    plt::figure_size(1280,720);
    plt::plot(enes, probs);
    plt::title("Cumple"); plt::xlabel("Personas"); plt::ylabel("Probabilidad"); plt::grid(true);
```

```
plt::save("fig.png");
}
```

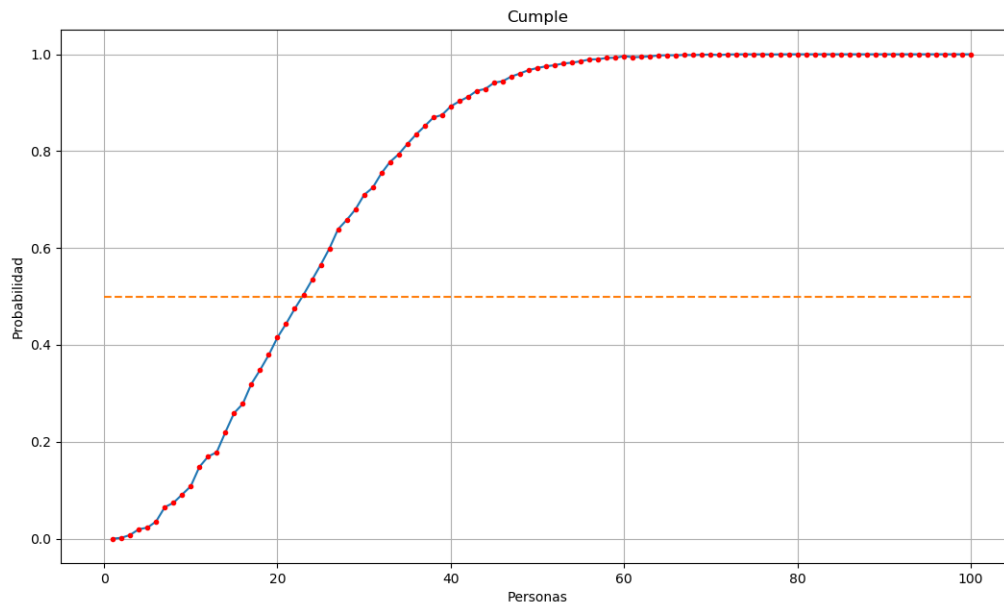
Para derivar la primer gráfica, hacemos 200 repeticiones para cada grupo de personas.



Bueno, sólomente de ver esto, nos resulta obvio que se alcanza el 1 (o nos acercamos mucho) muy rápidamente.

Como una buena observación a la gráfica, es incorrecto que la función que encontramos no sea no decreciente. Evidentemente, añadir una persona debería incrementar siempre la probabilidad de que dos personas cumplan años el mismo día. Esto es meramente *ruido estadístico*. Por ejemplo, sabemos que si aventamos una moneda un número par de veces, en expectativa vamos a ver cada cara el mismo número de veces. Pero si aventamos la moneda sólo tres veces, es posible que salga 3 veces la misma cara (de hecho, la probabilidad es 0.125). Si hicieramos el experimento infinitas veces, ese ruido desaparecería.

Vamos a repetir la simulación para el rango  $n \in \{1, 2, \dots, 100\}$ , pero para que haya un poco menos de ruido estadístico, vamos a repetir el experimento 5000 veces por cada tamaño del grupo de personas.



Increíblemente, la probabilidad de que dos personas cumplan años el mismo día es más de 0.5 en cuanto se tienen al menos 23 personas. Este resultado se puede derivar de manera analítica sin mucho problema (aunque hay números muy grandes involucrados), pero no necesitamos expresar explícitamente la probabilidad para tener una buena idea del resultado.

## 10. Aplicación I

### 10.1. Integración estocástica (*Monte Carlo*)

Imaginemos el siguiente problema. Tenemos dos figuras geométricas, una contenida en su totalidad dentro de la otra, y la que está dentro tiene un área un tanto complicado de calcular. Conociendo el área de la figura externa ¿Podemos estimar el área de la figura dentro? La respuesta es sí.

Supongamos temporalmente que esas figuras son un cuadradito de lado 1, y un cuarto de un círculo de radio 1 cuyo centro coincide con una de las esquinas del cuadro. Supongamos que nosotros no podemos calcular tan fácilmente el área del círculo, Lo que vamos a hacer es arrojar puntos aleatorios dentro del cuadro, y vamos a llevar la cuenta de cuántos quedan dentro del círculo (además de cuántos hemos aventado en total).

Hay un resultado teórico (que se encuentra en el anexo del final) que nos dice que

$$\mathbb{E}\left[\frac{n_{\text{dentro}}}{n_{\text{total}}}\right] = \frac{\text{Área interior}}{\text{Área total}}$$

Donde el  $\mathbb{E}$  significa *número esperado*, es decir, lo que la fracción va a valer después de hacer el ejercicio infinitas veces. Tomando esa ecuación como cierta, entonces podemos ver que

$$\text{Área interior} = \frac{n_{\text{dentro}}}{n_{\text{total}}}(\text{Área total})$$

Caso particular del cuarto de círculo, de área  $\pi r^2/4 = \pi/4$  y el cuadrado de área 1, tenemos que

$$\pi = 4 \frac{n_{\text{dentro}}}{n_{\text{total}}}$$

Así es: vamos a intentar utilizar este método de aproximación de áreas para determinar el valor de  $\pi$ .

**Nota:** Hay otras maneras de realizar integrales en la computadora. Una técnica usada comúnmente (para bajas dimensiones) es una integración geométrica, en donde el espacio se discretiza de una manera conveniente, y luego se hace algo así como una suma de Riemann.

El código para hacer esta *simulación* estará escrito en C++.

```
#include<random>
#include<chrono>
#include<cmath>

int main(){
    unsigned long int seed;
    seed = std::chrono::system_clock::now().time_since_epoch().count();

    std::default_random_engine g{seed};
    std::uniform_real_distribution<float> uniform(0,1);

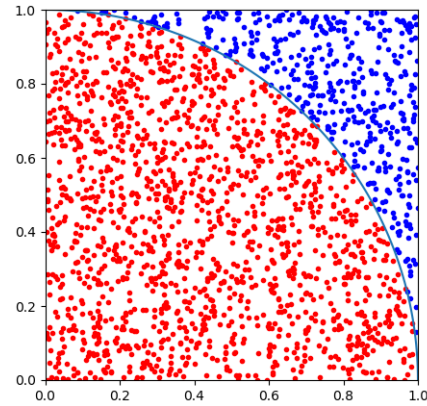
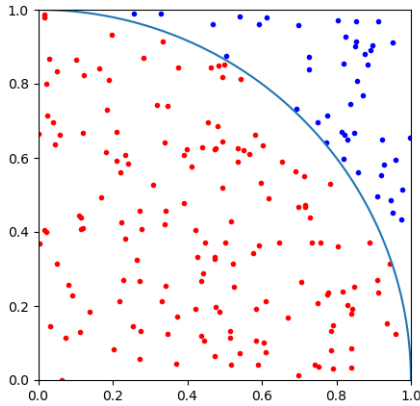
    int n = 200, count = 0;
    std::vector<float> xin, yin;
    std::vector<float> xout, yout;
    float x,y;
```

```

for(int i = 0; i < n; i++){
    x = uniform(g);
    y = uniform(g);
    if(x*x+y*y < 1){
        xin.push_back(x);
        yin.push_back(y);
        count++;
    } else {
        xout.push_back(x);
        yout.push_back(y);
    }
}
//std::cout << count << ", de un total de " << n << std::endl;
return 0;
}

```

Para la primer ocasión, se tienen 161 puntos dentro del círculo, de un total de 200. En una segunda ocasión, obtenemos 1575 de 2000 tiradas, es decir, aumentamos un poquito la resolución de nuestro experimento (la figura izquierda es para 200 puntos, la derecha para 2000)



Las aproximaciones correspondientes de  $\pi$  son 3.22 y 3.15 para el segundo caso.

Para obtener un resultado un poco más preciso, probamos tirando 100,000 puntos 200 veces, y se promedia la fracción de puntos dentro del círculo. Obtenemos que el 78.539725 % de los puntos caen dentro del área. Multiplicar esto por 4 nos da 3.141589, que ya comienza a ser muy satisfactorio.

## 10.2. Anexo teórico

Vamos a pensar en que se generan puntos con una distribución uniforme en el intervalo  $[a, b]$ . Tomemos pues  $a < c < d < b$  para que haya un intervalo  $[c, d]$  contenido en el  $[a, b]$ . Las mediciones las vamos a escribir formalmente como mediciones a una variable aleatoria  $X$  con una función de distribución de probabilidad  $p(x)$  uniforme en el intervalo  $[a, b]$ , definida como:

$$p(x) = \begin{cases} \frac{1}{b-a} & \text{si } x \in [a, b] \\ 0 & \text{o.c.} \end{cases}$$

De esta manera, la probabilidad de que una medición de la variable aleatoria  $X$  (es decir, la posición del puntito) esté en algún lado sería



$$\mathbb{P}[X \text{ está en algún lado}] = \int_{-\infty}^{\infty} p(x)dx = \int_a^b \frac{dx}{b-a} = 1$$

Queremos encontrar el número esperado de puntos dentro del intervalo, es decir:

$$\mathbb{E}[\# \text{ de puntos en el intervalo } [c, d]]$$

La probabilidad de que 1 punto  $x_0$ , resultado de la medición de  $X$ , esté en el intervalo  $[c, d]$  (cuya longitud denotaremos por  $\ell = d - c$ ) es sencillamente

$$\mathbb{P}[x_0 \in [a, b]] = \int_c^d p(x)dx = \int_c^d \frac{dx}{b-a} = \frac{d-c}{b-a} = \frac{\ell}{b-a}$$

Para facilitar notación, también bautizamos  $L := b - a$ . Ahora, la probabilidad de que  $k$ -mediciones  $\{x_i\}_{i=0}^{k-1}$  consecutivas caigan en el intervalo es

$$\mathbb{P}[x_i \in [c, d], \quad \forall i = \{0, \dots, k-1\}] = \prod_{i=0}^{k-1} \mathbb{P}[x_i \in [c, d]] = \left(\frac{\ell}{L}\right)^k$$

La probabilidad de que  $n - k$  mediciones no caigan en el intervalo, es simplemente  $(1 - \frac{\ell}{L})^{n-k}$ . Con ello, concluimos que

$$\mathbb{P}[x_i \in [c, d], \quad \forall i = \{0, \dots, k-1\} \wedge x_i \notin [c, d], \quad \forall i = \{k, \dots, n-1\}] = \prod_{i=0}^{k-1} \mathbb{P}[x_i \in [c, d]] = \left(\frac{\ell}{L}\right)^k (1 - \frac{\ell}{L})^{n-k}$$

Nos falta tomar en cuenta todas las permutaciones de este resultado. El total de maneras de *hacer el muestreo* de las  $n$  variables es  $n!$ . Vamos a contemplar las permutaciones. Hay  $n!/(n-k)!$  maneras de que  $k$  puntos caigan en el intervalo  $[c, d]$ . Ya que esto incluye *el orden en el que hacemos las mediciones*, nos deshacemos de ello multiplicando por  $(k!)^{-1}$ . Con ello, nuestra probabilidad final queda como

$$\mathbb{P}_{n,k} := \frac{n!}{(n-k)!k!} \left(\frac{\ell}{L}\right)^k (1 - \frac{\ell}{L})^{n-k} = \binom{n}{k} \left(\frac{\ell}{L}\right)^k (1 - \frac{\ell}{L})^{n-k}$$

Ahora, vamos bien a calcular el número esperado de puntos en el intervalo, dado que hicimos  $n$  mediciones

$$\mathbb{E}[\# \text{ de puntos en el intervalo } [c, d]] = \sum_{j=0}^n j \mathbb{P}_{n,j} = \sum_{j=0}^n \binom{n}{j} j \left(\frac{\ell}{L}\right)^j (1 - \frac{\ell}{L})^{n-j} = \sum_{j=1}^n \binom{n}{j} j \left(\frac{\ell}{L}\right)^j (1 - \frac{\ell}{L})^{n-j}$$

En el último paso nos hemos deshecho de un término, el de  $j = 0$  pues vale 0. Notemos primero que

$$j \binom{n}{j} = \frac{j n!}{j!(n-j)!} = \frac{n(n-1)!}{(j-1)!(n-j)!} = \frac{n(n-1)!}{(j-1)!((n-1)-(j-1))!} = n \binom{n-1}{j-1}$$

Ahora podemos arrastrar la suma de regreso al cero, además de factorizar un  $\ell/L$ .

$$\sum_{j=0}^{n-1} n \binom{n-1}{j} \left(\frac{\ell}{L}\right)^{j+1} (1 - \frac{\ell}{L})^{n-j-1} = n \frac{\ell}{L} \sum_{j=0}^{n-1} \binom{n-1}{j} \left(\frac{\ell}{L}\right)^j (1 - \frac{\ell}{L})^{n-1-j}$$

Para el mágico paso final, usamos que  $(x + y)^k = \sum_{j=0}^k \binom{k}{j} x^j y^{k-j}$

$$n \frac{\ell}{L} \sum_{j=0}^{n-1} \binom{n-1}{j} \left(\frac{\ell}{L}\right)^j \left(1 - \frac{\ell}{L}\right)^{n-1-j} = n \frac{\ell}{L} \left(\frac{\ell}{L} + 1 - \frac{\ell}{L}\right)^{n-1} = n \frac{\ell}{L} (1)^{n-1} = n \frac{\ell}{L}$$

Maravillosamente, si dividimos entre  $n$ , que es el número esperado de puntos que caerán en el intervalo  $[a, b]$ , obtenemos que:

$$\frac{\mathbb{E}[\# \text{ de puntos en el intervalo } [c, d]]}{\mathbb{E}[\# \text{ de puntos en el intervalo } [a, b]]} = \frac{\ell}{L}$$

Se puede hacer exactamente la misma generalización para dos dimensiones, en donde el resultado sería la división de áreas en lugar de longitudes. Léase:

$$\frac{n_1}{n_2} = \frac{A_1}{A_2}$$

## 11. Aplicación II

Seguimos con otra de las aplicaciones, esta vez orientada a determinar los parámetros que mejor se ajustan a un cierto tipo de modelo.

### 11.1. Regresión Lineal

#### 11.1.1. Modelos lineales

Para nuestros fines, que son los más sencillos e inmediatos, podemos conformarnos con definir que algo es *lineal* si satisface que es descrito por la ecuación para una recta:

$$y = mx + b \quad (1)$$

Cuando uno hace una *sugerencia educada* sobre el tipo de comportamiento que presenta un cierto fenómeno (con fines descriptivos, analíticos o predictivos) se dice que uno está *modelando* el fenómeno (el tipo de modelo depende justo de las herramientas que se empleen). Pues bien, cuando uno hace un modelo matemático y supone que una pareja de variables  $(x, y)$  asociadas al fenómeno están interrelacionadas vía una ecuación lineal, entonces se dice - con justa razón - que es un *modelo lineal*. El tener un modelo lineal básicamente significa que la razón entre los incrementos de ambas variables es constante, así que a un incremento unitario en  $x$  le corresponde un incremento fijo en  $y$ , independiente del valor de  $x$ , y cualquier otro incremento puede obtenerse esencialmente por una regla de 3; en contraste a lo que ocurriría con una función como  $x^2$  cuyos incrementos dependen del valor de  $x$ .

Naturalmente ninguna medición hecha en el mundo real tiene una precisión infinita, y justo por la misma razón, la probabilidad de que dadas más de tres parejas ordenadas  $\{(x_i, y_i)\}_{i=1}^N$ , éstas caigan sobre una misma recta es *casi seguramente* cero. ¿Significa esto que un modelo lineal nunca será válido? De ser el caso, debido a la imperfección de nuestras mediciones, necesariamente quedaríamos imposibilitados para modelar cualquier fenómeno que contara con una mínima componente estocástica. No, en su lugar sacrificamos el apego estricto a la ecuación anteriormente mencionada y decimos que *en promedio* los datos caen sobre una recta.

Esto quiere decir que, si bien las parejas de puntos no satisfacen la ecuación de la recta; lo que sí se cumple es que, al llamarle  $\hat{y}(x)$  a nuestra propuesta de lo que creemos que debería valer  $y$  (pero que nuestro muestreo no da exactamente por nuestras limitantes), se cumple que:

$$E(y - \hat{y}) = \sum_{i=1}^N (y_i - \hat{y}(x_i)) = 0 \quad (2)$$

Heurísticamente significa que "*las distancias entre los puntos por encima de la recta sugerida se cancelan con las distancias de los puntos por debajo*". A una cantidad que pretende describir de manera aproximada la distribución de nuestros datos le vamos a llamar un *estimador*. En particular si se cumple que el promedio/esperanza de un estimador respecto a la variable que estima es nula, como simplificamos en nuestro caso, se dice que el estimador es *insesgado*.

Si lo piensan bien, para cada conjunto de parejas de valores pueden existir varias rectas que satisfagan la condición anterior (tan solo piensen en dos puntos que no tengan el mismo valor de  $y_i$  y consideren tanto la recta que pasa por ellos como aquélla que es  $y = \frac{y_1+y_2}{2}$ ). ¿Cómo elegimos a la mejor entre ellas? La cuestión es que, si bien las distancias con signo se cancelan, lo que realmente buscamos es que la suma de todas esas distancias positivas sea lo más pequeña posible. A esta cantidad se le llama el *error cuadrático medio* del estimador (ECM) y suele considerarse que a menor ECM, mejor la calidad del estimador.

$$ECM = E((y - \hat{y})^2) = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}(x_i))^2 \quad (3)$$

A todo esto, los valores de  $x_i$  y  $y_i$  ya están fijos (pues fueron obtenidos de una tabla o experimento). ¿Entonces de qué es función el ECM? Si se escribe de forma explícita nuestro estimador lineal para  $y$  se ve claramente:

$$ECM(m, b) = \frac{1}{N} \sum_{i=1}^N (y_i - (mx_i + b))^2 \quad (4)$$

Así que, aunque típicamente  $m$  y  $b$  son *constantes* de la recta, como en este caso estamos variando sobre muchas rectas entonces  $m$  y  $b$  son las variables independientes del ECM, al que en lo sucesivo nos referiremos como  $S(m, b)$ . Como debieron ver en su curso de Cálculo I, para minimizar (o maximizar) una función de una variable,  $g(z)$ , lo suficientemente bien portada (como lo es nuestro caso pues es un polinomio), se recurre a derivar, igualar a cero y despejar los valores de la variable que satisfagan la ecuación obtenida, pues éstos, llamados *valores críticos* serán donde  $g$  tenga oportunidad de ser un máximo o mínimo (aunque sea local). Por ejemplo, la función  $g(z) = z^2 - 6z + 8$  al derivarla resulta en  $g'(z) = 2z - 6$ , que igualándola a cero y despejando  $z$  da como único resultado  $z = 3$ ; de modo tal que se concluye que en  $z = 3$  alcanza un valor crítico (en este caso, un mínimo). Al proceso de encontrar los valores que minimizan o maximizan un problema uno le conoce como *optimización*.

No discutiremos formalmente el por qué sabemos que el punto crítico que encontremos será un mínimo, ni el por qué éste se encuentra como estamos por describir pero ya tendrán tiempo y herramientas para hacerlo después. Por ahora convenzámonos lo siguiente:  $S$  no depende de una sino de dos variables ( $m$  y  $b$ ), pero si aplicamos el proceso de optimización de una variable a cada una por separado (haciendo de cuenta que la otra fuera constante) y ambas tienen un mínimo en común, entonces este mínimo es "*doblemente mínimo*" y será el que buscamos.

Así pues, de derivar a  $S$  considerando a  $m$  como la variable, a  $b$  constante, e igualar a cero, se obtiene:

$$\frac{dS}{dm} = \frac{2}{N} \sum_{i=1}^N (y_i - mx_i - b)(-x_i) = 0 \quad (5)$$

De derivar a  $S$  considerando a  $b$  como la variable, a  $m$  constante, e igualar a cero, se obtiene:

$$\frac{dS}{db} = \frac{2}{N} \sum_{i=1}^N (y_i - mx_i - b)(-1) = 0 \quad (6)$$

Podemos tirar las constantes innecesarias al otro lado de la igualdad y sacar a  $m$  y  $b$  de la suma pues no son dependientes del índice  $i$ . Y obtenemos las siguientes ecuaciones:

$$-\sum_{i=1}^N x_i y_i + m \sum_{i=1}^N x_i^2 + b \sum_{i=1}^N x_i = 0 \quad (7)$$

$$-\sum_{i=1}^N y_i + m \sum_{i=1}^N x_i + bN = 0 \quad (8)$$

Si aún lucen algo intimidantes por la presencia de las sumas, podemos llamarles  $A = \sum x_i y_i$ ,  $C = \sum x_i^2$ ,  $D = \sum x_i$ ,  $E = \sum y_i$ . Y la inocente apariencia es ahora:

$$mC + bD = A \quad (9)$$

$$mD + bN = E \quad (10)$$

Que si buscamos una pareja  $(m, b)$  que las satisfaga simultáneamente, ya debieron haber visto en su clase de álgebra que esto se puede hacer por muchos métodos; no importa el que elijan, la solución es la siguiente:

$$m = \frac{AN - ED}{CN - D^2} = \frac{N(\sum x_i y_i) - (\sum x_i)(\sum y_i)}{N(\sum x_i^2) - (\sum x_i)^2} \quad (11)$$

$$b = \frac{CE - DA}{CN - D^2} = \frac{(\sum x_i^2)(\sum y_i) - (\sum x_i)(\sum x_i y_i)}{N(\sum x_i^2) - (\sum x_i)^2} \quad (12)$$

Hemos encontrado así la mejor recta que se aproxima a los datos que nos proporcionan. A este método también se le conoce como el de *mínimos cuadrados*.

Cabe mencionar que este método también puede ser usado, aunque con cierta sutileza, para estimar modelos no lineales; basta con que las variables sean "separables", en el sentido de que si se tiene una dependencia del estilo de:

$$g(y) = mf(x) + b$$

Para funciones  $f$  y  $g$ , entonces puede hacerse un *cambio de variable* del estilo de  $Y = g(y)$  y  $X = f(x)$  y ahora la ecuación sería idéntica a la del modelo lineal.

$$Y = mX + b$$

No ahondaremos en las implicaciones, pero el carácter no lineal es aún visible en dichos modelos al analizar los *momentos* de la distribución de  $Y$  y  $X$ .

Un ejemplo de la aplicación de este método es en la estimación de la frecuencia de un oscilador armónico (o resorte simple...). Pues sabemos de la Ley de Hooke que la fuerza de restitución que ejerce un resorte de constante de restitución  $k$  es:

$$F = -kx \tag{13}$$

Y que por la Ley de Newton, una masa  $m$ , fija al extremo del resorte tendría una aceleración:

$$F = ma = m \frac{d^2 x}{dt^2} \tag{14}$$

De modo que igualando ambas expresiones y despejando a la segunda derivada, se obtiene que:

$$\frac{d^2 x}{dt^2} = -\frac{k}{m}x = -\omega^2 x \tag{15}$$

A la cantidad  $\omega$  se le conoce como la frecuencia del resorte y es justo la pendiente de la recta que se le ajustaría a un modelo lineal al que le alimentáramos la posición y la aceleración del resorte. Este ejemplo, junto con la generación de las derivadas (velocidad y aceleración) a partir sólo de los datos de posición se ilustra en el script "*oscilador.cpp*" que les será enviado.

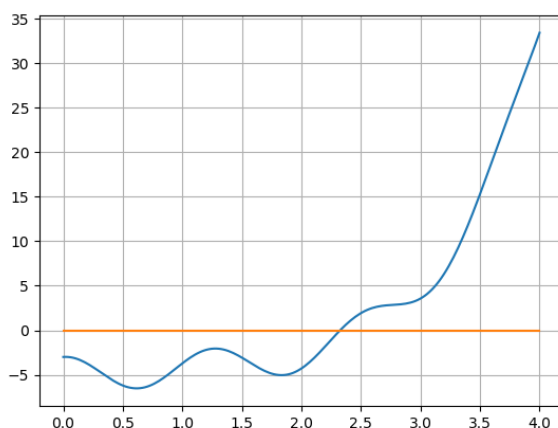
## 12. Aplicación III

### 12.1. Búsqueda de raíces en funciones

Imaginemos el siguiente problema: Tenemos una función que tiene cruces con la línea  $y = 0$ , por ejemplo,  $f(x) = x^2 - 3$ . Ahora, queremos encontrar en dónde es que la función cruza el cero. Para ello escribimos podemos igualar la función a 0, y encontramos la raíces.

$$0 = x^2 - 3 \rightarrow x_0 = -\sqrt{3}, x_1 = \sqrt{3}$$

Bueno, esta tarea fue particularmente fácil por el grado del polinomio, sin embargo, no siempre vamos a tener las cosas en *charola de plata*. Tomemos como ejemplo la función  $f(x) = 2\cos(5x) + e^x - x^2 - 6$



Es claro que anda por ahí del 2.3, pero no tenemos una respuesta exacta y hacer las cosas a ojo no es la mejor idea.

#### 12.1.1. Método de la bisección

El método de la bisección es el más sencillo, consiste en definir un intervalo en el que se sabe que hay una raíz. Eso se hace al encontrar un cambio de signo. Es decir, si  $f(x_0) < 0$  y  $f(x_1) > 0$  (y además resulta que  $f$  es *continua*), entonces  $f$  debe tener una raíz en el intervalo  $[x_0, x_1]$ . Esto no es más que el teorema del valor intermedio.

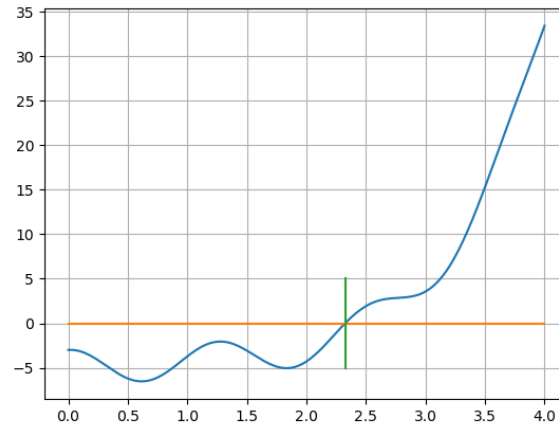
El algoritmo de búsqueda de raíces será el siguiente pues:

- Se toma el intervalo  $[x_0, x_1]$ , en donde se observa un cambio de signo en la función  $f$  a la que se quiere extraer las raíces.
- Se toma el punto medio  $\mu = (x_0 + x_1)/2$ . Si  $f(\mu) < \epsilon$ , detenemos aquí el algoritmo.
- Si resulta que  $f(\mu)f(x_0) > 0$  se sabe que no ha habido un cambio de signo entre  $x_0$  y  $\mu$ , esto quiere decir que podemos reducir el intervalo de búsqueda al  $[\mu, x_1]$ . Es decir, se toma  $x_0 = \mu$ .
- Si resulta que  $f(\mu)f(x_0) < 0$ , entonces el cambio de signo se dio en el intervalo  $[x_0, \mu]$ , por lo que podemos tomar  $x_1 = \mu$ .
- Se regresa al primer paso.

Esto se escribe muy fácil. De manera recursiva, basta con esta función

```
float bisection(float a, float b, float epsilon){
    float mu = (a+b)/2;
    float fmu = f(mu);
    if(abs(fmu) < epsilon) return mu;
    else if(fmu*f(a) > 0) return bisection(mu, b, epsilon);
    else return bisection(a, mu, epsilon);
}
```

El resultado de ejecutar esta función en el intervalo  $[0, 4]$  tomando  $\epsilon = 0.01$  es el de la siguiente figura



**Nota:** En general no tienen que preocuparse por esto, C++ tiene maneras de ajustar el rango y *garantizar* la uniformidad de la distribución.

## 13. Aplicación IV

¿Cómo podemos resolver una ecuación diferencial con ayuda de una computadora? Es algo complicado pues todo el cálculo está basado en la continuidad de funciones. Es decir, no hay precisión suficiente para representar una variable continua en una computadora (pues toda precisión será finita).

### 13.1. Integración numérica de ecuaciones diferenciales ordinales

¿Cómo entonces le hacemos para calcular una derivada? Tomemos el siguiente ejemplo.

```
#include<iostream>
#include"matplotlibcpp.h"
#include<cmath> //Aquí está M_PI definido

namespace plt = matplotlibcpp;

int main(){
    // Siempre va esto primero
    plt::backend("Agg");

    int n = 200;
    // Hacemos dos vectores de n entradas para almacenar los pares {x,y} de una función
    std::vector<double> x(n), y(n);
    for(int i = 0; i < n; i++){
        float t = ((float) i) / n * 2*M_PI;
    }
    return 0;
}
```

#### 13.1.1. Derivación numérica

Recordemos la definición de derivada pues

$$f'(x) := \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Ya sabemos que la computadora tiene precisión finita pues... bueno, intuitivamente vamos a ver el comportamiento de esta función si quitamos el límite. Llamemos a esta aproximación de la derivada *f gorrito que depende de h*, y hagamos el caso específico del seno:

$$\hat{f}_h(x) := \frac{\sin(x+h) - \sin(x)}{h}$$

A continuación, tabulamos los valores de esta función para  $x = 1$  y  $h = \{1, 0.1, 0.01, 0.001\}$

$h$	$\hat{f}_h(x)$
1	0.067826
0.1	0.497363
0.01	0.536085
0.001	0.539881
0.0001	0.540250



Es ahora que recordamos que  $f'(x) = \cos(x)$ , y que  $\cos(1) \approx 0.540302$ . Pues tal parece que no es *completamente necesario* tomar el límite. Computacionalmente vamos a conformarnos con tomar  $h$  lo suficientemente pequeña. ¿Qué tan pequeña debe ser  $h$ ? Eso será tema de su curso de física computacional. La respuesta dependerá de qué tanto error están dispuestos a tolerar en este cálculo.

### 13.1.2. Una ecuación diferencial

Hagamos lo siguiente: bauticemos  $f_1 := f(x+h)$  y  $f_0 := f(x)$ . Tomemos de momento la siguiente ecuación diferencial, donde  $a$  es una constante

$$\frac{\partial v}{\partial t} = a$$

Y vamos a sustituir mañosamente nuestra aproximación de la derivada

$$\frac{v_1 - v_0}{h} = a \quad \rightarrow \quad v_1 = v_0 + ha$$

Y vamos a hacer lo mismo para la posición, cuya derivada en el tiempo es la velocidad

$$\frac{\partial x}{\partial t} = v_0 \quad \rightarrow \quad \frac{x_1 - x_0}{h} = v_0 \quad \rightarrow \quad x_1 = x_0 + hv_0$$

#### 13.1.2.1 Casi lo mismo, pero con viento

Ya lo verán en sus cursos más avanzados, pero la fuerza de resistencia del viento de un objeto depende de la geometría del mismo, densidad del aire y demás (que podríamos decir son constantes). Hay un factor del que depende que no es constante, y ese es el cuadrado de la velocidad, con lo que la ecuación diferencial se ve de esta manera

$$f = a - \eta v^2$$

Haciendo el mismo truco, veremos que nuestras reglas sólo cambian un poquito

$$\begin{aligned} x_1 &= x_0 + hv_0 \\ v_1 &= v_0 + h(a - v_0^2) \end{aligned}$$