1. Lenguajes de programación

Como lo indica la palabra, un *lenguaje de programación* es fundamentalmente *un lenguaje* de los denominados **formales**. Sus reglas son más estrictas que las del *lenguaje natural* que se utiliza habitualmente para la comunicación entre personas. Estos tienen la característica de permitir a los programadores codificar algoritmos que serán procesados de alguna manera para que una computadora pueda ejecutarlos.

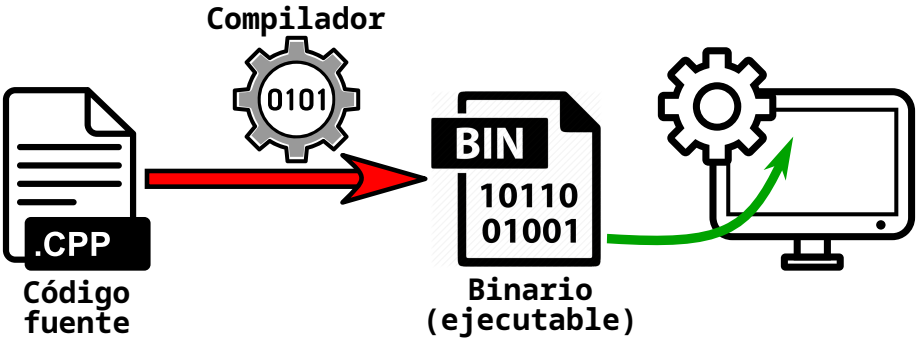
Así como cualquier lenguaje tiene sus reglas gramaticales:

- **Léxico:** conjunto de palabras o códigos (combinación de símbolos) que son válidos en el lenguaje. Por ejemplo, `casa` es un código válido en la lengua española y `c454` sería una combinación inválida. Al conjunto finito de códigos válidos se los conoce en el lenguaje de programación como **palabras reservadas** (*keywords*).
- **Sintaxis:** son las combinaciones gramaticalmente correctas para la formación de oraciones con las palabras validas y la construcción de unidades superiores. En un lenguaje de programación a estas oraciones se las conocen como **sentencias** y **expresiones**.
- **Semántica:** es el sentido o significado de las *expresiones* o *sentencias* para validarlas.

Los programas se escriben en archivos de *texto plano* que son diferentes a los que tienen formato como podrían ser los creados con un procesador de texto, donde puede cambiarse las fuentes, dar resaltado a voluntad (negritas, itálicas, subrayado, etc.). Estos archivos solo contienen texto y pueden ser escritos prácticamente con cualquier software de edición de texto. Aunque existen softwares especialmente dedicados para escribir en lenguajes de programación y tienen características que facilitan dicha tarea.

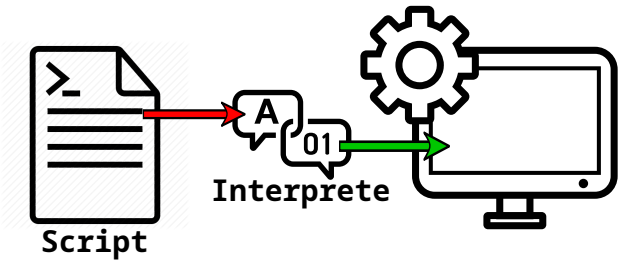
1.1. Compilado vs. interpretado

Existen, esencialmente, dos maneras en que las computadoras ejecutan los programas. La primera es utilizando **un compilador**, este es un software que traduce en su totalidad el código a **lenguaje máquina** (instrucciones en binario) que es lo que puede procesar la computadora.



Por otro lado, existen otros softwares denominados **interpretes**, que leen sentencia por sentencia y analizan expresión por expresión, decodificando a lenguaje máquina en

el momento (en tiempo real) cada una de las lineas de los programas. Usualmente a los lenguajes que son "interpretados" se los denomina también *lenguajes de scripting*.



Como analogía, podríamos entender que un libro puede ser traducido del inglés y ser editado en castellano. En ese caso el libro es "compilado", es posible acceder a la obra completa de una vez y el tiempo demorado está impuesto por nuestra capacidad de lectura. En cambio, si una persona toma el libro original en inglés y empieza a leerlo en voz alta en castellano, diremos que "interpreta" el libro. En este caso el tiempo demorado estará dictado por la capacidad del *interprete* en traducir y la comunicación oral, que suele ser más lenta.

Los softwares que son compilados suelen estar más optimizados en velocidad y tamaño, pero llevan más tiempo elaborarlos, porque tienen reglas más estrictas para el programador. En cambio, los softwares interpretados suelen ser menos óptimos en tiempo y uso de recursos, pero en contrapartida suelen escribirse en lenguajes de programación más ágiles y sencillos para el programador.

1.2. Clasificación en Software privativo, Open Source y Libre

Al archivo o conjunto de archivos que contienen el o los algoritmos codificados (*programa codificado*) se lo denomina **código fuente**. Es la receta de cocina para *construir* el programa ejecutable. Dependiendo si el o la autora o autores del software deciden compartir el código fuente se clasificará en: **Software Privativo** en caso de que no lo hagan o **Software de Código Abierto** (*Open Source*) si deciden hacerlo.

Para que un Software sea denominado **Software Libre**, debe cumplir con las siguientes libertades:

0. La libertad de ejecutar el programa como se desee, con cualquier propósito.
1. La libertad de estudiar cómo funciona el programa, y cambiarlo para que haga lo que se desee.
2. La libertad de redistribuir copias para ayudar a otros.
3. La libertad de distribuir copias de sus versiones modificadas a terceros. Esto le permite ofrecer a toda la comunidad la oportunidad de beneficiarse de las modificaciones.

Para cumplir con las libertades 1 y 3 se debe tener acceso al código fuente, es decir, que el *Software Libre es de código abierto*, pero no necesariamente esta relación se da al revés.

1.3. Paradigmas

Los paradigmas de programación son maneras de pensar o *"encarar"* los algoritmos. Estos difieren unos de otros, en los conceptos y la forma de abstraer los elementos involucrados en un problema, así como en los pasos que integran su solución del problema.

Se presenta a continuación una clasificación *clásica* de los paradigmas, teniendo en cuenta que existen diferencias entre bibliografías.

1. **Paradigma Imperativo:** se basa en dar instrucciones a la computadora de *cómo hacer* las cosas en forma de algoritmos, en lugar de describir el problema o la solución. Dentro de éste podemos diferenciar:
   a) **Procedural u orientado a procesos:** este paradigma permite listar de forma estructuradas las operaciones necesarias a ser programadas, es posible agrupar estas ordenes en procedimientos y funciones, así como estructurar datos complejos o interconectados.
   b) **Orientado a objetos:** aquí se intenta reflejar la realidad, se describen clases que modelizan objetos y estos interactúan transmitiendo mensajes entre sí para solucionar el problema modelizado.
   c) **Orientado a eventos:** en vez de seguir un flujo secuencial de instrucciones, se espera a que *eventos* disparen porciones de código. Esto pude aplicarse tanto en programación procedural como en orientada a objetos.
2. **Paradigma declarativo:** está basado en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones.
   a) **Funcional:** está basada en la definición los predicados y es de corte más matemático.
   b) **Lógico:** se basa en el concepto de función, gira en torno al concepto de predicado, o relación entre elementos.
3. **Multiparadigmas:** hoy en día gran parte de los lenguajes de programación soportan diferentes paradigmas. Tienen un paradigma base o raíz, pero fracciones de código pueden estar escritos en otros por cuestiones de síntesis, facilidad al momento de planteare esa parte del algoritmo, etc.

1.4. Niveles

Oficialmente existen 2 clasificaciones de nivel para los lenguajes de programación: **bajo** y **alto**. Esto se refiere a la relación entre el lenguaje y la máquina (computadora). Los lenguajes de bajo nivel son los que están íntimamente relacionados con las instrucciones que puede ejecutar una computadora en particular y son el *lenguaje máquina* (binario) y el *assembler*. Cada procesador tiene su propio set de instrucciones y su propio assembler, por lo tanto, el programador debe tener un conocimiento muy detallado sobre el procesador que está programando y el sistema operativo donde va a ejecutarse su programa.

Los lenguajes de *alto nivel* abstraen al programador de la máquina, y pueden utilizarlos sin saber siquiera sobre que computadora correrá el programa. Ya que de eso se encarga el

compilador, que traduce todo el código a un lenguaje máquina específico, o del interprete que debe estar preparado para esa computadora en particular y traducir las instrucciones del script.

Hay un neologismo que define lenguajes de *nivel intermedio*. Estos son los lenguajes de alto nivel que tiene facilidades para acceder a instrucciones de bajo nivel o que su abstracción no es tan elevada. Esto último indicaría que debemos tener consideraciones si se programa para una arquitectura especifica de computadora o sistema operativo en particular. Lo cual no ocurre con los de *alto nivel* en esta clasificación, los cuales pueden compilarse o ejecutarse en cualquier sistema sin la necesidad de tener consideraciones particulares.

2. Programación en C++

El lenguaje C desarrollado por Dennis Ritchie entre 1969 y 1972 en los Laboratorios Bell es un lenguaje estructurado, orientado a procesos, el cual se hizo muy popular y desplazó al *FORTRAN*. Como la *Programación Orientada a Objetos* (**POO**) estaba empezando a tener protagonismo, Bjarne Stroustrup diseño en 1979 una extensión del lenguaje para que soportara este paradigma denominándolo `C++`. El cual mantuvo gran parte de las características de C, pero no deben confundirse, ya que **son dos lenguajes diferentes**. Tanto `C` como `C++` son lenguajes que siguen vigentes al día de hoy, en grandes desarrollos donde la *performance* o el acceso al hardware a bajo nivel son esenciales, por ejemplo el desarrollo de Sistemas Operativos, Motores de Videojuegos, plataformas de virtualización, etc.

En este curso aprovecharemos varios aspectos de `C++`, pero iniciaremos utilizando únicamente sus características de `C`, es decir, para hacer programación estructurada. Aprender a diseñar algoritmos es el principal objetivo.

2.1. Salida Estándar

La mayoría de los lenguajes de programación están orientados al desarrollo de sistemas que eventualmente podrían o no tener una interfaz de usuario. Las interfaces de usuario (**UI - *User Interface***) podrían ser gráficas (**GUI**) tanto sean aplicaciones de celular, desktop, web, etc. o bien de **consola**. Ésta última es la que la mayoría de los lenguajes utiliza como salida y entrada estándar o predefinida, ya que los componentes gráficos son elementos que suelen ser más independientes o diversos (dependen del sistema operativo, biblioteca gráfica u otro elemento ajeno al lenguaje). Esta interfaz es aún ampliamente utilizada, especialmente en servidores y terminales de servicios.

2.1.1. La consola/terminal

Lo cierto es que la salida estándar suele darse por medio de un monitor o pantalla. Los primeros programas serán desarrollados para una consola o terminal de texto, también denominada interfaz de linea de comandos (**CLI - *Command Line Interface***).

Figura 1: Terminal de Linux

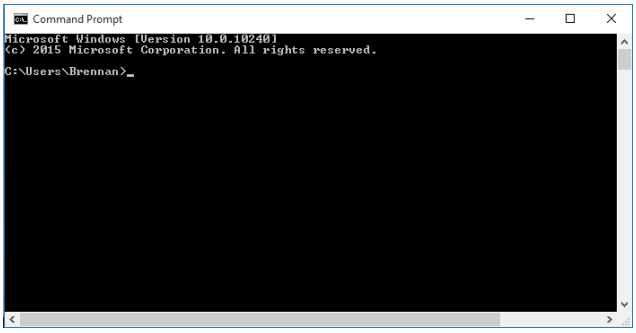


Figura 2: cmd.exe de MS Windows

2.1.2. El comando `cout` (*Console Out*)

Los programas codificados en C++ tienen determinadas características. Para empezar debemos utilizar un comando que le indique a nuestro programa que se utilizará una funcionalidad específica, esto se hace con el comando `#include<>`. Lo utilizaremos para indicar que queremos utilizar los flujos (*streams*) de entrada/salida estándar `iostream` (ver linea 1 del Programa 1). Esto nos permitirá acceder a el comando `std::cout` que permite mostrar textos hacia la pantalla. El prefijo `std::` indica que utilizaremos un comando dentro del ámbito "estándar", en este caso `cout` (***C**onsole **out***).

Programa 1: Saludo

```
1 #include<iostream> // permite utilizar std::cout
2
3 int main() {
4   std::cout << "¡Hola!" << std::endl;
5 }
```

Como se observa en el Programa 1, hay un cuerpo principal denominado `main()` el cual inicia con en la *linea 3* indicado por la llave de apertura `'{'` y termina en la *linea 5* con el carácter `'}'`. Todo lo que coloquemos dentro del cuerpo principal se ejecutará, en orden y secuencialmente. En este caso solo hay una instrucción, `std::cout` en la *linea 4*, seguido de << y el texto `¡Hola!`, luego otro << seguido de `std::endl` (***End L**ine*). Además es importante destacar que cada sentencia que escribamos terminará con un `';'`.

El operador << indica que deseamos *pasarle* algo a `cout`, en este caso es un texto fijo que deseamos mostrar en pantalla. Los textos fijos los escribiremos entre comillas dobles. Cuando ejecutemos nuestro programa veremos el texto `¡Hola!` en la pantalla. ¿Entonces qué es `endl`? Este comando agregará un *salto de linea* inmediatamente después del texto. Sería algo como apretar un *Enter*, lo que quiere decir que si agregáramos otro texto este se escribiría debajo del "¡Hola!";

Al principio del programa, luego del `#include<>` hay lo que se conoce como un **comentario**, que se indica empezando con una doble barra `//`. Todo lo que se escriba a continuación no formará parte de los comandos ejecutables, sino que serán una referencia o aclaración para el programador que esté leyendo el código. Otra manera de iniciar un comentario es utilizando `/*`, pero en ese caso, el comentario finalizará al encontrar la secuencia `*/`, por eso a la doble barra se la conoce como *comentario de linea*, porque solo ocupa una linea, y a la secuencia `/* */` se lo conoce como *comentario multilinea*.

Como a continuación utilizaremos varios elementos que se encuentran dentro del *ámbito estándar*, agregaremos un comando para evitar anteponer el prefijo `std::`. Como se observa a continuación al agregar la declaración `using namespace std` en la *linea 3*, nos permite escribir simplemente `cout` y `endl` en la *linea 6*.

Programa 2: Saludo v2.0

```
1 #include<iostream>
2 // Nos permite evitar el 'std::' antes de cout
3 using namespace std;
4
5 int main() {
6   cout << "¡Hola!" << endl;
7 }
```

2.2. Entrada Estándar y Variables

2.2.1. Memoria

Todas las computadoras poseen al menos una *memoria* en la que pueden almacenar **datos** y/o **instrucciones**. Las mismas son ***finitas***, por lo tanto los programas y la cantidad de datos que podemos almacenar en ellas también lo son. La memoria (por lo general), es el recurso más escaso en una computadora, por ello sería conveniente intentar hacer algoritmos pequeños y utilizar la menor cantidad de *variables* posibles. Sin embargo, no es algo de lo que nos preocuparemos por el momento.

Las **variables son espacios de la memoria** en las cuales podemos almacenar *datos* y son denominadas así, porque esos datos pueden *cambiar*. Es decir que podemos ***asignarles*** un dato y luego alterarlo según sea necesario. Estos datos pueden ser, un número entero, números con coma, un carácter (una letra) o una cadena de caracteres (*string*).

Nota

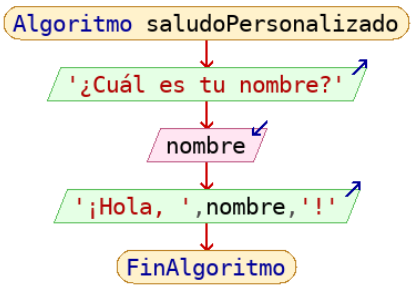
Las cadenas de caracteres son *textos* usualmente encerrados entre comillas (dobles o simples) dependiendo del lenguaje de programación. En caso de `C++` se utiliza comilla doble " " para *strings* y comillas simples ' ' para los caracteres.

Como se observa en el Programa 3, hay una variable llamada ***"nombre"*** a la cual el *usuario*, que ejecuta nuestro algoritmo debe asignarle un valor utilizando la instrucción `cin` (***C****onsole* ***In***) utilizando el operador >> seguido del nombre de la variable. En ese momento nuestro programa quedará en estado de *espera* a que el usuario ingrese un valor y luego presione la tecla `ENTER` para indicar que ha finalizado el ingreso. Luego es mostrada con la instrucción `cout` combinando un *string* fijo ("`¡Hola, `") con el valor ingresado y contenido en *nombre* y luego finaliza con un "`!`". A esta acción se la conoce como **concatenación** y se logra separando los textos, variables y comandos especiales (como `endl`) con el operador << en un `cout`.

Las variables en `C++` deben indicar de *que tipo son*, es decir, que clase de información estarán almacenando. Esto se hace anteponiendo el tipo antes de nombrar la variable,

Programa 3: Saludo con una variable

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6    cout << "¿Cuál es tu nombre? ";
7    string nombre;
8    cin >> nombre;
9    cout<<"¡Hola, "<<nombre<<"!"<< endl;
10 }
```



como se observa en la *linea 7* del Programa 3. En este caso, como almacenaríamos un texto, se debió especificar la palabra `string` antes de la variable *nombre*. A esto se lo denomina **declarar la variable**.

Existen diversos **tipos de datos**, pero a continuación se listarán los que estaremos utilizando por el momento y qué pueden almacenar:

Tipos de datos

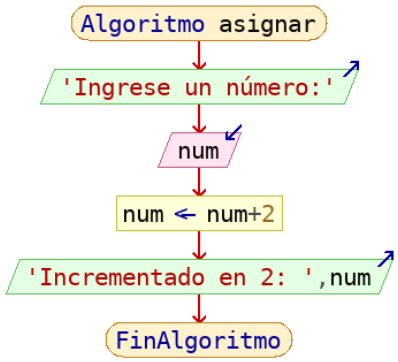
- `int` (enteros)
- `float` o `double` (números con coma)
- `char` (carácter)
- `string` (cadena de texto)
- `bool` (valores booleanos)

2.2.2. Dar valores a una variable

Muchas veces necesitamos modificar o "crear" datos sin depender exclusivamente del *usuario*. En ese caso, la asignación no será a través de la instrucción "**cin**" sino a través del operador de asignación que en pseudocódigo suele indicarse con una flecha hacia la izquierda "←" o "<-". En C++ se utiliza simplemente el operador "=" ubicando la variable a izquierda y lo que se asignará (lo que se guardará) en ella a la derecha.

Programa 4: Asignación

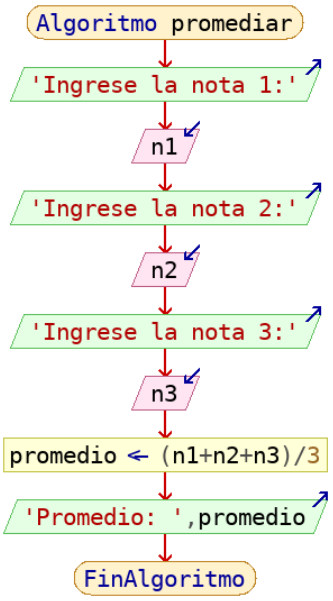
```
1  #include <iostream>
2  using namespace std;
3  int main() {
4    cout << "Ingrese un número: "
5    int num;
6    cin >> num;
7    num = num + 2;
8    cout << "Incremento en 2: " << num;
9    cout << endl;
10 }
```



En este caso, tomamos el valor en la variable *"num"* y lo incrementamos en 2, guardando el nuevo valor en la misma variable (en el mismo espacio de memoria). Veamos el ejemplo del calculo de un promedio de notas.

Programa 5: Asignación

```cpp
#include <iostream>
using namespace std;
int main() {
  int n1, n2, n3;
  cout << "Ingrese la nota 1: ";
  cin >> n1;
  cout << "Ingrese la nota 2: ";
  cin >> n2;
  cout << "Ingrese la nota 3: ";
  cin >> n3;
  int promedio = (n1+n2+n3) / 3;
  cout << "Promedio: " << promedio;
  cout << endl;
}
```



En el Programa 5, creamos 3 variables *"n*"* al principio del programa, pero al declarar la variable *promedio* además usamos el *operador de asignación*. Es decir que **inicializamos** la variable al momento de declararla. Nótese además, que este algoritmo calcula promedios enteros (sin decimales).

Nota

En el C++ disponemos de las operaciones matemáticas más comunes:

- `+` (suma)
- `-` (resta)
- `*` (multiplicación)
- `/` (división)
- `%` (resto de la división entera)

2.3. Interpretación de enunciados

Como aclaramos en el primer encuentro, la programación es la codificación de los algoritmos en un lenguaje de programación. Pero es preciso comprender los enunciados para pensar una solución y como articular la misma, es decir *cómo* diseñaremos nuestro algoritmo (*modelo de solución*). Interpretar los enunciados es casi tan importante (*¡o más!*) que la solución a los mismos. Ya que podemos buscar a alguien que lo solucione por nosotros o que nos oriente, buscar en internet, etc.. Pero, si no comprendemos el problema o lo entendimos mal, llegaremos a soluciones erróneas o a ninguna en absoluto.

Ejemplos:

- Elabore un algoritmo que pida ingresar un número, determine su anterior y muéstrelo en pantalla.

- Elabore un algoritmo que pida ingresar un número, determine su doble y muéstrelo en pantalla.
- Elabore un algoritmo que pida ingresar un número, determine el doble del siguiente y lo muéstrelo en pantalla.
- Elabore un algoritmo que pida la cantidad de pesos argentinos para hacer una compra de divisas en dólares y muestre la cantidad que se pueden comprar.

2.4. ¿Enteros o decimales?

Al momento de codificar un algoritmo es importante definir qué tipo de variable es la que mejor se adecúa al dato que queremos almacenar. Así pues, si necesitamos almacenar un texto usaremos `string`, en caso de guardar una única letra `char` y especialmente al momento de utilizar números debemos optar por alguna de las dos posibilidades `int` o `float/double`.

El hecho de elegir un tipo de dato con, o sin coma, estará dado por la naturaleza del problema a resolver por el algoritmo. Por ejemplo, si el problema es repartir una cantidad dada de caramelos entre una *x cantidad* de personas, estamos ante un claro ejemplo de uso de números enteros. No es posible, en principio, repartir $4\frac{3}{4}$ caramelos o tener $2\frac{1}{2}$ personas. Y, en caso de necesitar almacenar o procesar un dato como una temperatura en grados centígrados o el volúmen de un líquido en litros, posiblemente debamos utilizar números con decimales.

2.4.1. Operaciones entre enteros y números con decimales

Como mencionamos, existen dos maneras de *declarar* números: enteros (`int`) y con decimales (`double`) debemos tener especial cuidado al momento de hacer operaciones entre éstos. Al operar mezclando valores enteros con decimales, los enteros serán *promovidos* a números con decimales, es decir que el 3 (`int`) será considerado como 3.0 (`double`):

Programa 6: Operaciones mixtas

```cpp
#include <iostream>

using namespace std;

int main() {
   int a = 3;
   double b = 2.4;
   cout << "La suma entre 3 y 2.4 es: " << a + b << endl;
}
```

Así, "`a`", al momento de operar es *transfomado* en `double` para realizar la operación, y al ejecutar este programa veremos en pantalla el texto:

```
La suma entre 3 y 2.4 es: 5.4
```

En el caso de los literales, cuando operamos entre enteros (`int`) no hay problemas de hacer sumas, restas o multiplicaciones. Pero debemos prestar atención cuando dividamos, por ejemplo:

Programa 7: Promedio con error

```
#include <iostream>

using namespace std;

int main() {
  double promedio = (5+5+6)/3;
  cout << "El promedio de 5, 5 y 6 es " << promedio << endl;
}
```

Para sorpresa de algunos, se observa que al ejecutar este programa veremos por pantalla el texto:

```
El promedio de 5, 5 y 6 es 5
```

Sabemos que el resultado real es $5.\widehat{3}$, ¿por qué falló la computadora? Bueno, en realidad el que ha fallado es el programador, ya que ha utilizado números **constantes enteros** 5 (dos veces), 6 y 3. Todos ellos *literales enteros*, por lo tanto el resultado de la operación $(5 + 5 + 6) \div 3$ da como resultado un número entero: **5**. Basta con que transformemos a alguno de ellos en un número decimal para que todos los demás sean promovidos a `double`.

Programa 8: Promedio corregido

```
#include <iostream>

using namespace std;

int main() {
  double promedio = (5+5+6)/3.0;
  cout << "El promedio de 5, 5 y 6 es " << promedio << endl;
}
```

Al transformar el divisor a un `double` (3,0), los demás automáticamente serán transformados también. Ahora veremos que correctamente se observa por pantalla:

```
El promedio de las notas 5, 5 y 6 es 5.33333
```

2.4.2. Redondeo y Truncamiento

Hay dos operaciones que son muy comunes dentro de las computadoras al operar y debemos utilizarlas correctamente. El redondeo y el truncamiento.

Hay cálculos donde podemos aproximar el valor desestimando los números seguidos desde la cifra decimal deseada, a esta operación se la denomina truncamiento. Simplemente consta en deshacerse de los números decimales que no son necesarios. Por ejemplo, si queremos 2 decimales para truncar, debemos multiplicar por 100 para pasar los decimales

deseados a la parte entera y asignarlos a una variable de este tipo, deshaciéndonos de los decimales no deseados. Luego se vuelve a pasar a `double` dividido por 100 para devolver los decimales al lugar correspondiente.

$$23{,}12357 \times 100 \rightarrow 2312{,}\cancel{357} \div 100 \rightarrow 23{,}12 \quad (1)$$
$$54{,}25786 \times 100 \rightarrow 5425{,}\cancel{786} \div 100 \rightarrow 54{,}25 \quad (2)$$

Programa 9: Truncar a 2 decimales

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6    double pi = 3.14159;
7    cout << "PI sin truncar: " << pi << endl;
8    int aux = pi * 100; // aux <- 314
9    pi = aux / 100.0;   // !!
10   cout << "PI truncado a 2 decimales: " << pi << endl;
11 }
```

Ha de notarse que es importante que en la *linea 9* la división sea por 100.0 (un `double` para que `aux` que es una variable entera sea *promovida* a `double` y el resultado de la división sea correcto.

Por último, el redondeo es la técnica de aproximación numérica donde se toman en cuenta los decimales anteriores al número de decimales deseados. Por ejemplo, si deseamos 2 decimales para mostrar un resultado, entonces el proceso sería el de multiplicarlo por 100 para que esos decimales queden en la parte entera, sumar 0,5 para redondear al entero más próximo y truncar la parte decimal. Finalmente volver a dividir por 100 para retornar al número original ya redondeado:

$$23{,}12357 \times 100 \rightarrow 2312{,}357 + 0{,}5 \rightarrow 2312{,}\cancel{857} \div 100 \rightarrow 23{,}12 \quad (1)$$
$$54{,}25786 \times 100 \rightarrow 5425{,}786 + 0{,}5 \rightarrow 5426{,}\cancel{286} \div 100 \rightarrow 54{,}26 \quad (2)$$

Programa 10: Redondeo a 4 decimales

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main() {
6    double pi = 3.14159;
7    cout << "PI sin redondear: " << pi << endl;
8    int aux = (pi * 10000) + 0.5; // aux <- 31416
9    pi = aux / 10000.0;           // !!
10   cout << "PI redondeado a 4 decimales: " << pi << endl;
11 }
```

2.5. Nombrando variables

Es importante utilizar nombres para las variables que indiquen claramente cual es su rol dentro del programa. Además existen diferentes convenciones. En general, acordaremos que las variables siempre deben empezar con una letra alfabética en minúscula.

Los nombres de las variables no pueden tener espacios, solo caracteres alfanuméricos (letras y números) sin utilizar vocales con tildes o `ñ`.

Si el nombre de una variable se compone con dos o más palabras puede optar por separarlas con guiones bajos[1], por ejemplo `mejor_cancion` o empezando la siguiente palabra con una letra mayúscula[2], por ejemplo `mejorCancion`.

Las variables del tipo *booleanas* deben empezar preferentemente con el verbo ser/estar, tener o poder: `es_mayor`, `estaApagado`, `esta_sano`, `pudeLeer`, `puede_escribir`, `tieneComentarios`, `tiene_subtitulos`, `esFragil`, etc. También podría ser aceptable si es obvio el contexto, pero siempre que sea posible optaremos por esta convención adoptada del inglés.

```
1  int cantidadc; // MAL:¿Qué significa la c después de 'cantidad'?
2  int cantidad_clientes; // BIEN: Es claro de qué es la cantidad
3  int i; // DEPENDE: BIEN si el uso es trivial, sino MAL
4  int indice; // DEPENDE: BIEN si es obvio de qué es el índice
5  int puntos_totales; // DEPENDE: BIEN si es obvio que se está
       puntuando, MAL si es ambiguo
6  int _cuenta; // MAL: Evite nombres que comiencen con guines bajos
7  int cuenta; // MAL: ¿Qué se está contando o es el resultado de
       una cuenta?
8  int datos; // MAL: ¿Qué tipo de datos datos?
9  int tiempo; // MAL: ¿Tiempo de qué?¿En horas, minutos o segundos?
10 int minutosTranscurridos; // BIEN: Es descriptivo dentro del
       contexto
11 int valor1, valor2; // MAL: ¿valores de qué? Es dificil
       diferenciarlos.
12 int cant_manzanas; // BIEN: es descriptivo
13 int monstruos_eliminados; // BIEN: Descriptivo.
14 int x, y; // DEPENDE: Bien si el uso es trivial, sino MAL.
15 bool ganado; // MAL: debería empezar con es, esta o puede
16 bool estaPartidaTerminada; // BIEN: es claro que es una variable
       booleana y cuando es true y cuando false.
17 bool tiene_cartucho; // BIEN: es claro dentro del contexto
```

[1] Convención conocida como *snake_case*.
[2] Convención conocida como *lowerCamelCase*.

3. Ejercitación

3.1. Intente resolver los siguientes problemas planteando los algoritmos en diagramas y codifíquelos en C++:

1. Ingrese un número y muestre la mitad del siguiente.
2. Ingrese 2 números y muestre la suma en pantalla.
3. Ingrese 3 números y muestre el producto de los dos primeros sumado al tercero.
4. Ingrese 2 números, muestre la diferencia entre ambos.
5. Ingrese su año de nacimiento y muestre la edad que alcanzará en 2030.
6. Ingrese la temperatura en grados centígrados y muéstrelos en grados Fahrenheit.
7. Ingrese dos ángulos internos de un triangulo y muestre el tercero.
8. Pida ingresar un cantidad de canicas **rojas**, y sabiendo que hay canicas rojas, amarillas y trasparentes, que las rojas son el doble que las transparentes y las amarillas el triple de las rojas, muestre cuantas canicas de cada color hay y el total.
9. Ingrese un número con coma, multiplíquelo por 3.125 y muestre el resultado redondeado en 2 decimales.
10. Ingrese el valor de lista de un producto y muestre el valor sumando el IVA (21 %), redondee a 2 decimales.
11. Ingresando la cotización del dólar estadounidense y la cantidad que se desea comprar, calcule y muestre la cantidad de pesos argentinos necesarios para dicha compra redondeado a 2 decimales.
12. Juan salio a comer una pizza con sus dos amigos de la primaria y desea determinar de cuánto dinero tiene que disponer cada uno. Si se reparten los gastos por igual: ingrese el total de la cuenta y determine cuánto dinero pagará cada uno.
13. Mariano quiere un algoritmo que al ingresar el dinero que posee determine cuántos alfajores puede comprar para él y sus 3 compañeros de trabajo, sabiendo que cada alfajor cuesta $15.25.
14. Pablo quiere diseñar un algoritmo en el que pueda ingresar el dinero en su bolsillo, ya que desea comprar caramelos para 3 amigos y quedarse con al menos 1 para él. Sabiendo que los caramelos cuestan $1.25, el algoritmo debe producir las siguientes salidas:
    a) Cantidad de caramelos que puede comprar.
    b) Cantidad de caramelos repartidos en total.
    c) Cantidad de caramelos que se queda cada uno de sus amigos.
    d) Cantidad de caramelos que se queda Pablo.

3.2. Indique si los nombres elegidos para las siguientes variables son correctos o no y porqué:

- `double suma; // Asuma que es obvio lo que se está sumando.`
- `int _manzanas;`
- `double VALOR;`
- `double dinero depositado;`
- `int clientes_totales;`
- `int cantFrutas;`
- `double metros_de_caño;`
- `double metrosDeTuberia;`
- `bool salir;`
- `bool puedeCorrerConZapatillas;`
- `bool puedeCorrer_descalzo;`
- `bool esta_sobre_hielo;`