1. Proposiciones lógicas

Las **proposiciones** (enunciados) son oraciones declarativas al las que se le pueden determinar su *valor de verdad*. Es una expresión lingüística del razonamiento, que se caracteriza por ser verdadera o falsa empíricamente, sin ambigüedades. Son proposiciones las oraciones aseverativas, las leyes científicas, las fórmulas matemáticas, las fórmulas y/o esquemas lógicos, los enunciados cerrados o claramente definidos.

El **valor de verdad** de una proposición depende no solamente de las relaciones entre las palabras del lenguaje y los objetos en el mundo, sino también del *estado del mundo* y del conocimiento acerca de ese estado. El valor de verdad de la oración *"Juan canta"* depende no solamente de la persona denotada en *Juan* y el significado del verbo *cantar*, sino también *del momento* cuando esta oración es expresada. Juan probablemente canta ahora, pero ciertamente que no siempre está cantando.

Se debe hacer una distinción entre la oración gramatical propiamente dicha, a la que se llama enunciado, y el contenido o significado del enunciado, que es la proposición.

Los siguientes *enunciados* representan en realidad a **la misma proposición**:

- En Buenos Aires hay mucha humedad.
- Buenos Aires es una ciudad muy húmeda.
- La humedad en Buenos Aires es bastante alta.

Ejemplos de expresiones las cuales **no son** proposiciones:

- El hombre más fuerte del mundo.
- ¿Quién ganará el Mundial de Pelota Panamá 2011?
- 13 + 7
- ¡Hable en voz baja!

1.1. Proposiciones atómicas y compuestas

Las **proposiciones atómicas** (o simples) son las que carecen de conectores lógicos, es decir que sólo se evalúa un valor de verdad. Por ejemplo *"Matías es profesor"* es una proposición simple la cual puede ser verdadera o falsa. En cambio las **proposiciones compuestas** (o moleculares) son aquellas que tienen dos o más proposiciones atómicas vinculadas entre sí por conectores. Por ejemplo, *"Matías es profesor* ***Y*** *Matías dicta el curso de programación"* es una proposición compuesta entre las proposiciones simples *"Matías es profesor"* y *"Matías dicta el curso de programación"* conectadas entre sí por una **conjunción**. Por lo tanto, **para que la proposición molecular sea verdadera, ambas deben serlo**.

A medida que avancemos en la construcción de algoritmos con sentencias de control veremos cómo aplicar estos conectores.

2. Sentencias y operaciones lógicas

Sabemos que un algoritmo es un modelo de resolución para un problema, el cual estaba caracterizado por ser secuencial, ordenado, preciso, definido y finito. Es decir que es una lista ordenada de instrucciones o pasos que debemos seguir para llegar a la solución del problema. Pero hay situaciones en las que debemos tomar decisiones, las cuales pueden llevarnos a tomar un camino u otro al momento de implementar la solución.

Las **sentencias de control**, son *instrucciones* que permiten romper la secuencialidad de la ejecución, esto significa que nos permiten la realización de algunas instrucciones y omitir otras, de acuerdo a la evaluación de una condición o expresión lógica. Por otra parte, hay sentencias que nos permiten volver a ejecutar un conjunto de instrucciones dada una condición.

Resumiendo, existen dos tipos de *sentencias de control*:

1. **Selectivas:** un camino entre dos o más opciones se ejecuta "por única vez".
2. **Repetitivas:** permiten ejecutar un conjunto de instrucciones "varias veces" dada una condición.

2.1. Implementación en el lenguaje C++

2.1.1. Variable booleanas

En C++ existe un tipo de dato denominado `bool` el cual solo puede tener uno de dos valores: `true` o `false`. Un ejemplo de declaración y asignación de dos variables de este tipo se vería así:

```
bool es_lunes = true;
bool es_martes = false;
```

2.1.2. Operadores de comparación

Existen operadores que permiten comparar dos valores, que usualmente conocemos de la matemática. Por ejemplo: $a > b$, $a \leq b$, $a = b$, $a \neq b$, etc.

Operadores de comparación en C++

- == : igual
- > : mayor
- < : menor
- != : distinto
- >= : mayor igual
- <= : menor igual

Veamos aplicaciones concretas sobre la aplicación de estos operadores:

```
int a = 2, b = 3;
bool es_a_mayor_b = (a > b);        // ¿a es mayor que b?
bool es_a_menorigual_b = (a <= b); // ¿a es menor o igual que b?
bool es_a_distinto_b = (a != b);    // ¿a es distinto que b?
```

2.1.3. Operadores lógicos

Los operadores lógicos son los ya mencionados **conectores**, qué sirven para poder programar *proposiciones compuestas*. Ya que por ejemplo es imposible hacer la siguiente expresión: "$0 < a < x$", ya que el operador de comparación < es un operador binario, es decir que tiene solo dos operandos.

Es por ello que disponemos de 3 operadores lógicos:

Operadores lógicos en C++

- `and` : conjunción (Y)
- `or` : disyunción (O)
- `not` : negación (NO)

Por lo tanto si se deseara averiguar si una variable está dentro de determinado rango:

```
bool esta_entre_0y10 = (a > 0 and a < 10); // true: si 0 < a < 10
```

Otros ejemplos serían:

```
bool es_0o10 = (a == 0 or a == 10); // true: si a es 0 ó 10
bool b = true;
bool no_b = not b;                  // true: si b es false
```

3. Sentencias selectivas en C++

En este encuentro nos centraremos en las sentencias **selectivas** únicamente, que nos permitirán ejecutar porciones de código y omitir otras.

3.1. Condicional simple

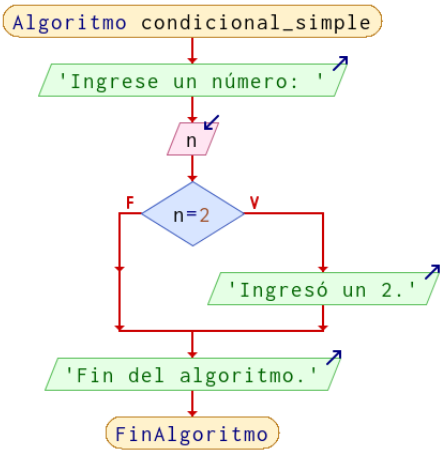
La sentencia de control condicional *"Si"* es la más básica de todas. Se evalúa una **expresión lógica**. En caso verdadero, las instrucciones declaradas se ejecutarán, pero en caso contrario, estas serán omitidas y se continuarán con las siguientes a continuación del bloque. En C++ se utiliza la palabra en ingles `if` que se traduce literalmente como *si*. Su implementación se puede observar en el Programa 1.

En el diagrama se detalla que *"Si n es igual a 2"* **entonces** se escribirá el texto *"Ingresó un 2."*. Lo cual se codifica en C++ con un `if` que a continuación tiene encerrada entre paréntesis la expresión lógica "`n == 2`" utilizando el operador correspondiente de **igualación**, el cual no debe confundirse con el de **asignación** que es un solo igual "=".

Inmediatamente después de la expresión a igualar se abren llaves (`{...}`) en las cuales irán encerradas las instrucciones que correspondan a la ejecución en caso de que la expresión **sea verdadera**, aquí únicamente se mostrará el texto *"Ingresó un 2."*. En caso de que no se ingrese un 2 por la linea de comandos, entonces este texto no se verá. Finalmente, sin importar si el caso resultara verdadero o falso se mostrará el texto *"Fin del algoritmo."*, ya que este se encuentra por fuera del bloque del **`if`**.

Programa 1: Condicional simple

```cpp
#include <iostream>
using namespace std;
int main() {
  cout << "Ingrese un número: ";
  double n;
  cin >> n;
  if(n == 2) {
    cout<<"Ingresó un 2."<<endl;
  }
  cout<<"Fin del algoritmo.";
}
```



¡Cuidado!

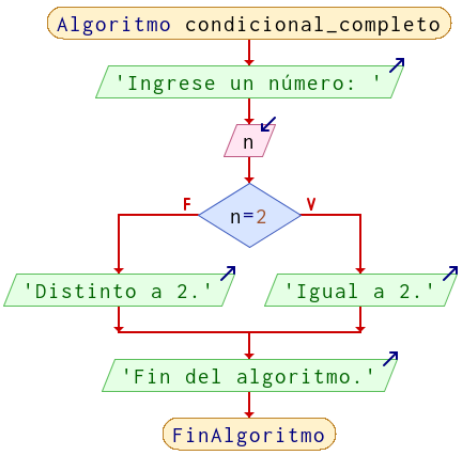
Es importante para el programador no descuidar la sintaxis del lenguaje, ser lo más claro y ordenado posible. Se debe prestar atención al momento de abrir una llave, el recordar ubicar correctamente la de cierre y que siempre estén emparejadas. Es por ello que se adopta la convención de dejar espacios cada vez que se abre un nuevo *"ámbito"*, para identificar visualmente –y rápidamente– qué instrucciones están dentro de qué estructura.

3.2. Condicional completo

Los condicionales completos son aquellos que poseen dos bloques de código, uno que se ejecuta en caso verdadero y otro que se ejecuta en el caso falso. El bloque ejecutado por el lado falso se indico con la palabra reservada `else` (*"sino"*):

Programa 2: Condicional completo

```cpp
#include <iostream>
using namespace std;
int main() {
  cout << "Ingrese un número: ";
  double n;
  cin >> n;
  if(n == 2) {
    cout<<"Igual a 2."<<endl;
  } else {
    cout<<"Distinto a 2."<<endl;
  }
  cout<<"Fin del algoritmo.";
}
```



En esta variante obtendremos dos tipos de salida dependiendo del valor ingresado:

```
Ingese un número: 2
Igual a 2.
Fin del algoritmo.
```

```
Ingese un número: 3
Distinto a 2.
Fin del algoritmo.
```

Al final de la llave de cierre del camino a ejecutar ante una condición verdadera se escribe la palabra reservada **`else`**, la cual abre una nueva llave donde se ubicarán las instrucciones a ejecutar si la condición del **`if` es falsa**.

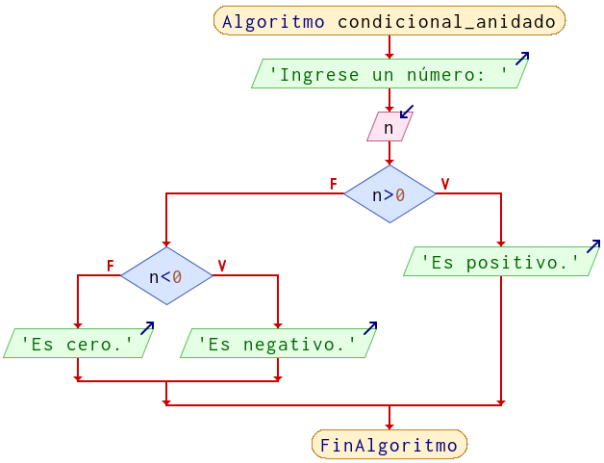
Los `else` no deben tener condiciones, es decir, no hay una condición encerrada entre paréntesis como en el `if`, ya que justamente el bloque se ejecuta cuando esta primera es falsa. *Nunca habrá un `else` sin un `if` previo.*

3.3. Condicionales anidados

Muchas veces debemos corroborar algo que no puede resolverse con una sola expresión lógica. Por ejemplo, saber si un número es positivo, negativo o cero. Aquí tenemos 3 posibilidades. Para ello, podemos pasar por más de una instancia de evaluación **adentro** de otra. En este sencillo algoritmo observamos que luego de un `else` es posible colocar

Programa 3: Condicional anidado

```cpp
#include <iostream>
using namespace std;
int main() {
  cout << "Ingrese un número: ";
  double n;
  cin >> n;
  if(n > 0) {
    cout<<"Es positivo."<<endl;
  } else if (n < 0) {
    cout<<"Es negativo."<<endl;
  } else {
    cout<<"Es cero."<<endl;
  }
}
```



un nuevo `if` con una nueva condición. El nuevo condicional puede ser simple o completo dependiendo de la necesidad, es decir que el último `else` puede estar o no dependiendo de la necesidad del algoritmo.

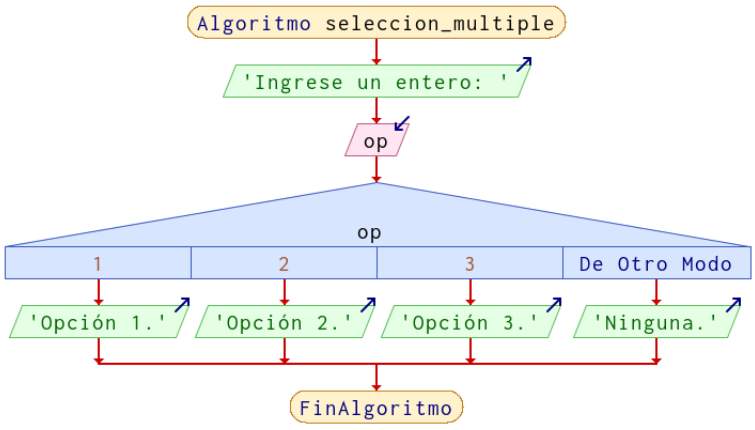
Es posible poner un nuevo `if` luego de un `else` todas las veces que se requieran:

```cpp
if(/* condición 1 */) {
  // ...
} else if(/* condición 2 */) {
  // ...
} else if(/* condición 3 */) {
  // ...
} else { // Opcional...
  // ...
}
```

3.4. Selección múltiple

Frecuentemente nos encontraremos con casos en los cuales debemos tomar un camino dependiendo de un valor fijo. El caso más típico es al crear un *menú de opciones*. En el cual debemos seleccionar un valor predeterminado y preestablecido. Para ello podremos evaluar una ***variable*** numérica y ejecutar el bloque de código correspondiente al valor de la misma.

Como vimos anteriormente, esto se puede lograr concatenando una serie de `if-else-if`, pero esto no se estila porque existe la sentencia de control de selección `switch-case`.



Programa 4: Selección múltiple

```cpp
#include <iostream>

using namespace std;

int main() {
  // Se le indica al usurio lo que debe hacer:
  cout << "Ingrese el entero 1, 2 ó 3: ";
  // Se lee la opción ingresada por el usuario
  int op;    // La variable evaluada debe ser un entero.
  cin >> op;

  switch(op) {
    case 1:     // Caso op == 1
      cout << "Opción 1." << endl;
      break;    // FinCaso
    case 2:     // Caso op == 2
      cout << "Opción 2." << endl;
      break;    // FinCaso
    case 3:     // Caso op == 3
      cout << "Opción 3." << endl;
      break;    // FinCaso
    default:    // op != 1 and op != 2 and op != 3
      cout << "Ninguna." << endl;
  }
}
```

La sentencia empieza con la palabra reservada `switch` que entre paréntesis evaluará una valor. Inmediatamente después dentro del bloque delimitado por las llaves se evaluarán los `case` compuestos por un valor constante (*literal*: 1, 2, 3, etc.) seguido de '`:`'.

Los bloques `case` a diferencia de los otros, no empiezan y terminan con llaves, sino que terminan al encontrar la palabra reservada `break`. Así podemos incluir todas las instrucciones necesarias para cada bloque.

Por último, hay un bloque opcional, el cual puede estar o no (como el último `else` en un bloque de condicionales) llamado `default`, el cual se ejecuta cuando la evaluación no coincide con ninguno de los *casos*.

El `switch` tiene limitaciones, ya que la variable a ser evaluada debe ser un entero, no se admiten otros tipos de datos. Además solo podemos ejecutar *"casos"* donde esa variable sea un valor específico (1, 2, -3, 0, etc.), no podemos evaluar expresiones como `(op > 5)` o similares. Por lo tanto, no se podrá utilizar en algoritmos donde debemos evaluar rangos.

Además no pueden declararse variables dentro de los casos, de ser necesario declarar variables éstas deben estar antes de los casos:

```
switch(op) {
  int a; // declaración de la variable 'a'
  int b; // declaración de la variable 'b'
  case 1:
    a = 1; // utilización de las variables dentro del switch
    b = 2;
    // etc...
    break;
  case 2:
    b = 3; // no siempre se utilizan todas las variables
    // etc...
    break;
  default:
    // etc...
}
```

4. Ejercitación

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

1. Pida ingresar un número, indique si es par o impar.
2. Ingrese 2 números, muestre la distancia entre ambos.
3. Pida ingresar una temperatura en grados centígrados, muéstrelos en fahrenheit. Si supera o iguala los 113°F muestre la cadena "Alerta roja por altas temperaturas".
4. Pida ingresar un número y muestre un texto que diga si el número es positivo, negativo o cero.

5. Pida ingresar 2 nombres, muéstrelos ordenados alfabeticamente (de forma ascendente: A→Z).

6. Pida ingresar un ángulo y verifique que esté entre 0° y 180°, muestre el texto "ángulo válido" en caso afirmativo y "ángulo inválido" de lo contrario.

7. Pida ingresar un ángulo entre 0° y 180°, determine y muestre si es: "nulo", "agudo", "recto", "obtuso", "llano" o "no válido".

8. Pida ingresar 2 ángulos internos de un triangulo, muestre el que falta e indique si es un triangulo equilátero.

9. Extienda el algoritmo del ejercicio anterior para que indique si el triangulo es rectángulo.

10. Pida ingresar 2 números y determine cuál es el mayor.

11. Pida ingresar 3 números y determine cuál es el menor.

12. Pida ingresar 3 nombres, muéstrelos ordenados de manera descendente (Z→A).

13. Muestre 3 opciones:

    1 - Doble

    2 - Triple

    3 - Mitad

    Pida ingresar una de las opciones, luego un número y muestre el doble, triple o la mitad del mismo según corresponda. Si no se ingresó un opción válida, muestre el mensaje de error: *"Opción no válida"*.