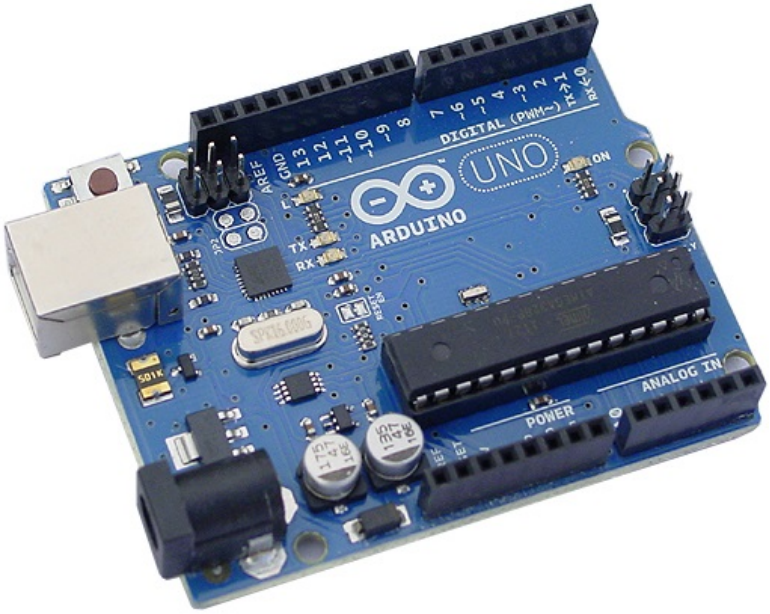
Trabajo Práctico
Programación con Arduino UNO y Lógica Booleana

BOT - Robótica Educativa

Receso de Invierno 2025



Ejercicios progresivos para simulación en Tinkercad

Índice

1. Introducción

Este trabajo práctico está diseñado para que ejercites y profundices los conceptos de programación con Arduino UNO que hemos visto en clase, integrándolos con lógica booleana y circuitos combinacionales. Los ejercicios están organizados en orden creciente de dificultad y están pensados para ser resueltos durante el receso de invierno.

1.1. Objetivos

- Aplicar conceptos de lógica booleana en circuitos prácticos con Arduino
- Utilizar todas las funciones básicas de Arduino: `pinMode()`, `digitalWrite()`, `digitalRead()`, `analogWrite()`, `analogRead()`, `map()`, `Serial.begin()`, `Serial.print()`, etc.
- Practicar la modularización de código usando archivos `.h` y `.cpp`
- Simular circuitos en Tinkercad antes de la implementación física
- Integrar múltiples sensores y actuadores con lógica combinacional

1.2. Herramientas necesarias

- Arduino IDE instalado en tu computadora
- Cuenta en Tinkercad (`https://www.tinkercad.com`)
- Placa Arduino UNO (física o simulada)
- Componentes varios: LEDs, resistencias, potenciómetros, botones, sensores

1.3. Instrucciones generales

1. **Simula primero:** Antes de armar cada circuito físicamente, créalo y pruébalo en Tinkercad
2. **Documenta tu código:** Agrega comentarios explicando la lógica booleana utilizada
3. **Modulariza:** A partir del ejercicio 8, utiliza archivos `.h` y `.cpp` para organizar tu código
4. **Verifica funcionamiento:** Cada ejercicio debe funcionar correctamente antes de pasar al siguiente

2. Recursos Adicionales

2.1. Repaso de tablas de la verdad de operadores lógicos básicos

En lógica booleana, una proposición atómica es una declaración simple que puede ser evaluada como verdadera (1) o falsa (0), sin posibilidad de subdivisión en componentes más simples. Por ejemplo, "el botón está presionado" o "la temperatura es mayor a 25°C" son proposiciones atómicas que en Arduino representamos mediante variables de tipo `bool`. Estas proposiciones pueden combinarse usando operadores lógicos para formar proposiciones más complejas que describan condiciones elaboradas del sistema.

Los operadores lógicos fundamentales (AND, OR, NOT) actúan como bloques constructivos que permiten crear cualquier función lógica imaginable. En Arduino, estos se implementan mediante los operadores `&&`, `||` y `!` respectivamente. Cuando combinamos estos operadores básicos, podemos construir funciones más complejas como NAND (que es simplemente NOT(AND)) o XOR, que aunque no existe como operador nativo en Arduino, puede implementarse usando la combinación `(A && !B) ||`

`(!A && B)`. Esta capacidad de combinar operadores simples para crear lógica compleja es fundamental para el control inteligente de sistemas con Arduino.

Operador AND (&&): Verdadero únicamente cuando ambas entradas son verdaderas.

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

Operador OR (||): Verdadero cuando al menos una de las entradas es verdadera.

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

Operador NOT (!): Invierte el valor lógico de la entrada.

A	!A
0	1
1	0

Operador XOR (OR Exclusivo): Verdadero únicamente cuando las entradas tienen valores diferentes.

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Implementación de XOR en Arduino:

```
// XOR puede implementarse como: (A && !B) || (!A && B)
bool A = digitalRead(2);
bool B = digitalRead(3);
bool resultado_XOR = (A && !B) || (!A && B);
```

2.2. Repaso de funciones básicas de Arduino IDE

Arduino IDE proporciona un conjunto de funciones fundamentales que nos permiten interactuar con el microcontrolador y sus componentes externos. Estas funciones constituyen la base de cualquier programa de Arduino y son esenciales para controlar entradas, salidas, comunicación y temporización. Dominar estas funciones básicas te permitirá crear proyectos cada vez más complejos y sofisticados.

Antes de explorar las funciones, es importante comprender los tipos de datos básicos que Arduino maneja. Cada variable en nuestros programas debe tener un tipo específico que define qué clase de información puede almacenar y cómo se comporta en las operaciones. Los cinco tipos de datos fundamentales en Arduino son `int` para números enteros, `float` para números decimales, `bool` para valores verdadero/falso, `char` para caracteres individuales, y `String` para cadenas de texto. La elección correcta del tipo de dato no solo optimiza el uso de memoria, sino que también previene errores comunes en la programación.

2.2.1. Tipos de datos básicos

Tipo int (Entero): Almacena números enteros desde -32,768 hasta 32,767. Es ideal para contadores, pines, y valores que no requieren decimales.

```
// Declaracion e inicializacion de variables int
int ledPin = 13;           // Pin del LED
int contador = 0;          // Contador inicializado en 0
int temperatura = 25;      // Temperatura en grados Celsius
int valorSensor;           // Variable sin inicializar (valor indefinido)
```

Tipo float (Punto flotante): Almacena números decimales con precisión limitada. Perfecto para mediciones, cálculos matemáticos y valores que requieren decimales.

```
// Declaracion e inicializacion de variables float
float voltaje = 3.3;          // Voltaje de alimentacion
float temperatura = 23.5;     // Temperatura con decimales
float promedio = 0.0;         // Inicializado en cero
float pi = 3.14159;           // Constante matematica
```

Tipo bool (Booleano): Almacena únicamente dos valores: `true` (verdadero) o `false` (falso). Fundamental para lógica de control y estados.

```
// Declaracion e inicializacion de variables bool
bool ledEncendido = false;     // Estado del LED
bool botonPresionado = true;   // Estado del boton
bool sistemaActivo;            // Sin inicializar (false por defecto)
bool alarmaActivada = false;   // Sistema de alarma
```

Tipo char (Carácter): Almacena un solo carácter ASCII. Útil para comandos simples y caracteres individuales.

```
// Declaracion e inicializacion de variables char
char comando = 'A';           // Comando de control
char grado = 'C';             // Simbolo de grados Celsius
char letra = 'X';             // Caracter individual
char respuesta;               // Sin inicializar
```

Tipo String (Cadena de texto): Almacena secuencias de caracteres. Ideal para mensajes, nombres y texto en general.

```
// Declaracion e inicializacion de variables String
String nombreProyecto = "Sistema de Alarma";
String estado = "Activo";
String mensaje = "";              // String vacio
String comandoCompleto = "RESET_SISTEMA";
```

2.2.2. Funciones de configuración y control de pines

pinMode(pin, modo): Configura un pin específico como entrada (`INPUT`), salida (`OUTPUT`), o entrada con resistencia pull-up interna (`INPUT_PULLUP`).

```
void setup() {
  pinMode(13, OUTPUT);        // Pin 13 como salida (LED)
  pinMode(2, INPUT);          // Pin 2 como entrada
  pinMode(3, INPUT_PULLUP);   // Pin 3 con resistencia pull-up
}
```

digitalWrite(pin, valor): Escribe un valor digital (HIGH o LOW) en un pin configurado como salida. HIGH representa 5V y LOW representa 0V.

```
// Encender y apagar un LED
digitalWrite(13, HIGH);   // Encender LED (5V)
delay(1000);              // Esperar 1 segundo
digitalWrite(13, LOW);    // Apagar LED (0V)
```

digitalRead(pin): Lee el estado digital de un pin configurado como entrada. Retorna HIGH o LOW según el voltaje presente.

```
// Leer estado de un boton
bool estadoBoton = digitalRead(2);
if (estadoBoton == HIGH) {
  Serial.println("Boton presionado");
}
```

2.2.3. Funciones de control analógico

analogRead(pin): Lee un valor analógico de los pines A0 a A5. Retorna un valor entre 0 y 1023, donde 0 representa 0V y 1023 representa 5V.

```
// Leer valor de un potenciometro
int valorPotenciometro = analogRead(A0);
Serial.print("Valor del potenciometro: ");
Serial.println(valorPotenciometro);
```

analogWrite(pin, valor): Genera una señal PWM en pines específicos (3, 5, 6, 9, 10, 11). El valor debe estar entre 0 (0 % duty cycle) y 255 (100 % duty cycle).

```
// Controlar brillo de un LED
int brillo = 127;              // 50% de brillo
analogWrite(9, brillo);        // Aplicar PWM al pin 9

// Ejemplo con LED RGB
analogWrite(9, 255);   // Rojo al maximo
analogWrite(10, 0);    // Verde apagado
analogWrite(11, 128);  // Azul al 50%
```

2.2.4. Función de mapeo

map(valor, fromLow, fromHigh, toLow, toHigh): Convierte un valor de un rango a otro proporcionalmente. Extremadamente útil para conversiones entre diferentes escalas.

```
// Convertir lectura analogica a voltaje
int lectura = analogRead(A0);
float voltaje = map(lectura, 0, 1023, 0, 500) / 100.0;

// Convertir potenciometro a angulo de servo
int valorPot = analogRead(A1);
int anguloServo = map(valorPot, 0, 1023, 0, 180);

// Convertir sensor a temperatura
int sensorValue = analogRead(A2);
int temperatura = map(sensorValue, 0, 1023, -10, 50);
```

2.2.5. Funciones de comunicación serie

Serial.begin(baudRate): Inicializa la comunicación serie con una velocidad específica. Comúnmente se usa 9600 baudios para debugging básico.

```
void setup() {
  Serial.begin(9600);  // Inicializar comunicacion a 9600 baudios
  Serial.println("Sistema iniciado");
}
```

Serial.print(datos): Envía datos al monitor serie sin agregar un salto de línea al final. Útil para mostrar múltiples valores en la misma línea.

```
int temperatura = 25;
Serial.print("Temperatura: ");
Serial.print(temperatura);
Serial.print(" grados Celsius");
// Resultado: "Temperatura: 25 grados Celsius"
```

Serial.println(datos): Similar a `Serial.print()`, pero agrega un salto de línea al final. Ideal para separar líneas de información.

```
Serial.println("=== INICIO DEL PROGRAMA ===");
Serial.println("Leyendo sensores...");
Serial.println("Sistema listo");
// Cada mensaje aparece en una linea separada
```

2.2.6. Funciones de temporización

delay(milisegundos): Pausa la ejecución del programa por el tiempo especificado en milisegundos. Bloquea completamente el programa durante este tiempo.

```
// Parpadeo basico de LED
digitalWrite(13, HIGH);   // Encender LED
delay(1000);              // Esperar 1 segundo
digitalWrite(13, LOW);    // Apagar LED
delay(500);               // Esperar medio segundo
```

millis(): Retorna el número de milisegundos transcurridos desde que el Arduino comenzó a ejecutarse. No bloquea el programa y permite temporización no bloqueante.

```
// Parpadeo sin bloquear el programa
unsigned long tiempoAnterior = 0;
unsigned long intervalo = 1000;    // 1 segundo

void loop() {
  unsigned long tiempoActual = millis();

  if (tiempoActual - tiempoAnterior >= intervalo) {
    // Cambiar estado del LED
    digitalWrite(13, !digitalRead(13));
    tiempoAnterior = tiempoActual;
  }

  // El programa puede hacer otras tareas aqui
}
```

2.2.7. Creación de funciones personalizadas

Las funciones son bloques de código reutilizable que realizan tareas específicas. Una función puede recibir parámetros (datos de entrada) y puede retornar un valor. Los parámetros son variables que permiten pasar información a la función para que pueda procesarla.

Estructura básica de una función:

```
tipoRetorno nombreFuncion(tipoParametro1 parametro1, tipoParametro2 parametro2) {
  // Codigo de la funcion
  return valor; // Solo si la funcion retorna algo
}
```

Función void (sin retorno): Ejecuta acciones pero no devuelve ningún valor.

```
// Funcion para encender LED por tiempo especifico
void encenderLED(int pin, int tiempo) {
  digitalWrite(pin, HIGH);
  delay(tiempo);
  digitalWrite(pin, LOW);
```

```
}

// Funcion para mostrar estado de sensor
void mostrarSensor(String nombreSensor, int valor) {
  Serial.print("Sensor ");
  Serial.print(nombreSensor);
  Serial.print(": ");
  Serial.println(valor);
}

// Uso de las funciones
void loop() {
  encenderLED(13, 500);           // Encender LED pin 13 por 500ms
  mostrarSensor("Temperatura", 25); // Mostrar temperatura
}
```

Funciones con valor de retorno: Procesan información y devuelven un resultado.

```
// Funcion que retorna temperatura en Fahrenheit
float celsiusAFahrenheit(float celsius) {
  float fahrenheit = (celsius * 9.0 / 5.0) + 32.0;
  return fahrenheit;
}

// Funcion que retorna el promedio de tres valores
float calcularPromedio(int valor1, int valor2, int valor3) {
  float suma = valor1 + valor2 + valor3;
  float promedio = suma / 3.0;
  return promedio;
}

// Funcion booleana para verificar rango
bool estaEnRango(int valor, int minimo, int maximo) {
  return (valor >= minimo && valor <= maximo);
}

// Uso de las funciones
void loop() {
  float tempC = 25.0;
  float tempF = celsiusAFahrenheit(tempC);

  int promedio = calcularPromedio(10, 20, 30);

  bool dentroRango = estaEnRango(promedio, 15, 25);

  Serial.print("Temperatura: ");
  Serial.print(tempC);
  Serial.print("Celsius = ");
  Serial.print(tempF);
  Serial.println(" Farenheit");
}
```

Parámetros de función: Los parámetros son variables especiales que reciben los valores que se pasan a la función cuando se la llama. Permiten que la misma función trabaje con diferentes datos cada vez que se ejecuta.

```
// Funcion con multiples parametros de diferentes tipos
void configurarSistema(bool activar, int tiempoEspera, String mensaje) {
  if (activar) {
    Serial.println("Activando sistema...");
    Serial.println(mensaje);
    delay(tiempoEspera);
    Serial.println("Sistema activo");
  } else {
    Serial.println("Sistema desactivado");
```

```
  }
}

// Llamadas a la funcion con diferentes parametros
void setup() {
  Serial.begin(9600);

  configurarSistema(true, 2000, "Iniciando modo normal");
  configurarSistema(false, 0, "");
  configurarSistema(true, 1000, "Modo de emergencia");
}
```

2.3. Componentes simulados en Tinkercad

- Arduino UNO
- LEDs de diferentes colores
- Resistencias (220Ω, 10kΩ)
- Potenciómetros
- Botones (pushbuttons)
- Buzzer
- LED RGB
- Protoboard para conexiones

2.4. Consejos para el desarrollo

1. **Planifica antes de programar:** Dibuja la lógica booleana en papel
2. **Prueba paso a paso:** No intentes hacer todo de una vez
3. **Usa el monitor serie:** Es tu herramienta de debugging principal
4. **Comenta tu código:** Explica especialmente la lógica booleana
5. **Backup regular:** Guarda versiones funcionando antes de hacer cambios grandes

3. Ejercicios Nivel Básico - Lógica Booleana Fundamental

Ejercicio 1: Compuerta AND con LEDs

Objetivo: Implementar una compuerta AND usando dos botones como entradas y un LED como salida.

Componentes:

- 2 botones (pines digitales 2 y 3)
- 1 LED (pin digital 13)
- Resistencias de pull-up internas
- Resistencia de 220Ω para el LED

Lógica booleana: El LED se enciende únicamente cuando ambos botones están presionados simultáneamente.

Tabla de verdad:

Botón A	Botón B	LED
0	0	0
0	1	0
1	0	0
1	1	1

Consignas:

1. Diseña el circuito en Tinkercad
2. Implementa el código usando `digitalRead()` y `digitalWrite()`
3. Verifica que el comportamiento coincida con la tabla de verdad
4. Agrega salida por puerto serie indicando el estado de las entradas y salida

Ejercicio 2: Compuerta OR con LEDs de colores

Objetivo: Implementar una compuerta OR usando dos botones y dos LEDs de diferentes colores.

Componentes:

- 2 botones (pines digitales 2 y 3)
- 1 LED rojo (pin digital 12) - indica estado de entrada A
- 1 LED verde (pin digital 11) - indica estado de entrada B
- 1 LED azul (pin digital 13) - indica resultado OR

Lógica booleana: El LED azul se enciende cuando al menos uno de los botones está presionado.

Consignas:

1. Crea el circuito en Tinkercad con LEDs de diferentes colores
2. Los LEDs rojo y verde deben mostrar el estado individual de cada botón
3. El LED azul debe implementar la función OR
4. Usa el monitor serie para mostrar la tabla de verdad en tiempo real

Ejercicio 3: Compuerta XOR con análisis completo

Objetivo: Implementar una compuerta XOR (OR exclusivo) y analizar su comportamiento.
Componentes:

- 2 botones (pines digitales 2 y 3)
- 1 LED amarillo (pin digital 13)
- Display serie para análisis

Lógica booleana: El LED se enciende únicamente cuando los botones tienen estados diferentes.
Consignas:

1. Implementa la función XOR usando operadores lógicos de C++
2. Crea una función que imprima la tabla de verdad completa al inicio
3. Agrega un contador que muestre cuántas veces se activó la salida
4. Implementa un sistema de reset usando un tercer botón

4. Ejercicios Nivel Intermedio - Entradas Analógicas y Lógica Combinacional

Ejercicio 4: Control de LED RGB con lógica booleana

Objetivo: Controlar un LED RGB usando potenciómetros y aplicar lógica booleana para crear patrones de colores.

Componentes:

- 3 potenciómetros (pines analógicos A0, A1, A2)
- 1 LED RGB (pines PWM 9, 10, 11)
- 2 botones para modos (pines digitales 2 y 3)

Lógica booleana: Los botones determinan el modo de operación:

- Modo 00: Control directo RGB
- Modo 01: Solo colores primarios (rojo, verde, azul)
- Modo 10: Solo colores secundarios (amarillo, magenta, cian)
- Modo 11: Blanco cuando todos los potenciómetros $> 50\,\%$

Consignas:

1. Usa `analogRead()` para leer potenciómetros
2. Usa `map()` para convertir valores 0-1023 a 0-255
3. Implementa lógica booleana para determinar qué colores mostrar
4. Muestra por serie el modo actual y valores RGB

Ejercicio 5: Sistema de alarma con múltiples sensores

Objetivo: Crear un sistema de alarma que combine múltiples entradas usando lógica booleana.

Componentes:

- 1 sensor de temperatura (simulado con potenciómetro en A0)
- 1 sensor de luz (simulado con potenciómetro en A1)
- 1 sensor de movimiento (simulado con botón en pin 2)
- 1 LED rojo (alarma - pin 13)
- 1 buzzer (pin 8)
- 1 botón de reset (pin 3)

Lógica combinacional: La alarma se activa cuando:

- (Temperatura $> 30°$ C AND Movimiento detectado) OR
- (Luz $< 20\,\%$ AND Movimiento detectado) OR
- (Temperatura $> 40C$)

Consignas:

1. Simula sensores usando potenciómetros con rangos realistas
2. Implementa la lógica booleana usando operadores `&&`(operador AND), `||`(operador OR), `!`(operador NOT)
3. Implementa un sistema de reset que requiera mantener presionado el botón por 3 segundos

Ejercicio 6: Contador binario de 4 bits con display

Objetivo: Implementar un contador binario que muestre números 0-15 usando 4 LEDs y display serie.

Componentes:

- 4 LEDs (pines 10, 11, 12, 13) - representan bits 0, 1, 2, 3
- 2 botones: incrementar (pin 2) y decrementar (pin 3)
- 1 botón reset (pin 4)

Lógica booleana: Cada LED representa un bit del número binario. Usar operaciones bit a bit para extraer cada bit.

Consignas:

1. Implementa funciones para conversión decimal a binario
2. Usa operadores bit a bit (`&`, `«`, `»`) para extraer bits individuales
3. Agrega debouncing para los botones usando `millis()`
4. Muestra en serie el número decimal, binario y hexadecimal
5. Implementa overflow y underflow (15+1=0, 0-1=15)

5. Ejercicios Nivel Avanzado - Modularización y Sistemas Complejos

Ejercicio 7: Sistema de Semáforo Inteligente con Lógica Booleana

Objetivo: Implementar un sistema de semáforo que utilice lógica booleana para controlar el flujo de tráfico con sensores de peatones y emergencias.

Componentes:

- 6 LEDs (2 rojos, 2 amarillos, 2 verdes) - Semáforos Norte y Sur
- 2 botones (pines digitales 2 y 3) - Botón peatones y emergencia
- 6 resistencias de 220 para LEDs
- 2 resistencias de 10k para botones

Lógica booleana aplicada:

- **Cambio de semáforo:** (Tiempo transcurrido AND Peatón esperando) OR Emergencia activa
- **Flujo de tráfico:** NOT emergencia AND Semáforo correspondiente verde
- **Estados mutuamente excluyentes:** Norte verde XOR Sur verde

Modularización básica requerida:

- `semaforo.h` - Declaraciones de funciones de control
- `semaforo.cpp` - Implementación de funciones de cambio de estado
- Programa principal en `.ino`

Consignas:

1. Implementa la lógica booleana usando operadores `&&`, `||`, `!`
2. Crea funciones simples para `cambiarSemaforo()`, `verificarBotones()`
3. Usa `millis()` para temporización sin bloquear el programa
4. Muestra en monitor serie el estado actual y las transiciones
5. Simula el circuito completamente en Tinkercad antes de implementar

Ejercicio 8: Control de Temperatura con Múltiples Sensores

Objetivo: Crear un sistema de control ambiental que tome decisiones basadas en múltiples sensores usando lógica booleana compleja.

Componentes:

- 3 potenciómetros (A0, A1, A2) - Sensores de temperatura simulados
- 1 servo motor (pin 9) - Ventilador
- 4 LEDs indicadores (pines 5, 6, 7, 8)
- 1 buzzer (pin 4) - Alarmas
- Resistencias según corresponda

Lógica combinacional para control:

- **Ventilador ON:** Temperatura promedio $> 25C$ OR Cualquier sensor $> 30C$
- **Alarma crítica:** Diferencia entre sensores $> 5C$ OR Temperatura $> 35C$
- **Estado normal:** Todos los sensores en rango 18°C - 28°C

Modularización requerida:

- `sensores.h/.cpp` - Funciones para lectura y procesamiento de sensores
- `control.h/.cpp` - Funciones de lógica de control y actuadores
- Programa principal minimalista

Consignas:

1. Implementa funciones para promediar lecturas de sensores
2. Usa la función `map()` para convertir valores de potenciómetros a temperatura
3. Crea funciones separadas para cada tipo de decisión de control
4. Implementa histéresis simple para evitar oscilaciones
5. Documenta claramente la lógica booleana en comentarios del código

Ejercicio 9: Monitoreo Ambiental Inteligente

Objetivo: Integrar múltiples sensores en un sistema de monitoreo que tome decisiones automáticas usando lógica booleana jerárquica.

Componentes:

- 5 potenciómetros (A0-A4) - Simulan sensores de temperatura, humedad, luz, calidad del aire, presión
- 8 LEDs indicadores (pines 2-9) - Estados del sistema
- 1 servo motor (pin 10) - Ventilador automático
- 1 buzzer (pin 11) - Alertas del sistema
- 2 botones (pines 12, 13) - Configuración manual y reset

Lógica booleana jerárquica:

- **Sistema OK:** Todos los sensores en rango normal (AND múltiple)
- **Alarma:** Algún sensor fuera de rango PERO no crítico (OR con condiciones)
- **Emergencia:** Cualquier sensor en rango crítico (OR con prioridad máxima)
- **Control automático:** Decisiones basadas en combinaciones de sensores

Modularización requerida:

- `sensores.h/.cpp` - Lectura y procesamiento de todos los sensores
- `evaluacion.h/.cpp` - Funciones de lógica booleana para evaluación de condiciones
- `actuadores.h/.cpp` - Control de LEDs, servo y buzzer

- `configuracion.h` - Constantes y umbrales del sistema

Consignas:

1. Implementa funciones separadas para cada tipo de evaluación (normal, alarma, emergencia)
2. Usa lógica booleana para priorizar acciones (emergencia ¿alarma ¿normal)
3. Crea un sistema de umbrales configurables usando `#define` en archivo de cabecera
4. Implementa un protocolo simple de comunicación serie para mostrar el estado del sistema
5. Agrega funcionalidad de reset que restaure todos los estados a valores normales
6. Simula completamente el sistema en Tinkercad con todos los componentes

¡Que tengas un excelente receso de invierno practicando!