ME 120: If-then-else structures in Arduino
May 22, 2021
Gerald Recktenwald
gerry@pdx.edu

1 Making Decisions in Arduino Programs

It's incredibly useful that computer programs allow devices respond to changing conditions: a light comes on when it's dark, the furnace comes on when it's cold, a robot balances a load as it moves over rough terrain. The ability to make decisions is called program logic, which means that the program makes a choice based on some information. We are constantly making logical choices: if I'm thirsty, get a drink; if I'm code, put on a jacket; if it's dark, turn on a light.

The general term *conditional execution* refers to the logic that causes specific program statements to be executed only when a prescribed condition is met. When a sequence of logical choices are made, the program (and the system it controls) changes, or flows, from one state to the next. At all times, the flow of execution is determined by a decision (or series of decisions) that depend on the current set of data in the program. When the data changes, the flow of execution changes. The data could come from external sensors or it could come from internally generated data such as system clock that indicates the time elapsed since a prior event occurred.

Consider the situation depicted in Figure 1. A level of water in a tank needs to maintained. Perhaps this is a pet dish, or a tank for livestock. Connecting an Arduino to a water level sensor provides data to determine whether the water supply valve should be opened.

1.1 Introduction to Logic Structures

There are a handful of ways to make blocks of code execute only when prescribed conditions are met. The most common conditional execution technique uses the basic "`if`" structure and its variants, "`if-else`", "`if-else if`", etc. When there are several discrete choices, a "`switch`" structure can be convenient. When a decision needs to be repeated a "`while`" structure is helpful. In the following, we will concentrate on the `if` structure and its variants.

As programmers, we need to anticipate the sequence of events that might happen at a future time when the program is running. We write code that tests for a condition and then executes an appropriate set of actions when those conditions occur. Therefore, we need to consider two main features, the logical test — "is condition A true" — and the response to those tests — "take action X if condition A is true, otherwise take action Y".

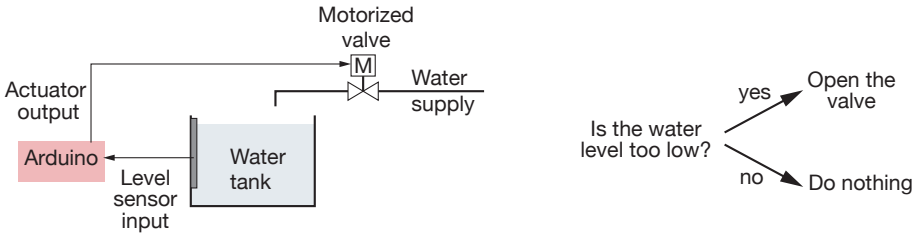

Figure 1: Automatic filling of a water tank. Schematic of the hardware (left) and logic (right).

1.2 True and False Values

Before making decisions, we need to establish a convention identifying whether a logic test is true or false. Here, although logic is an important part of philosophy, we are not engaging deep philosophical questions about the existence or meaning of truth. Rather, we merely need to establish a way to designate the outcome of a test performed by the computer.

In a digital computer, the decision-making, no matter how complex, is reducible to one or more yes/no decisions. The decision-making questions are called *logical expressions* and are discussed in the next section.

When asking a question with digital logic, a "yes" answer is equivalent to `True`, and "no" is equivalent to `False`. In other words, the outcome of a logical expression will be binary: true or false, yes or no. For convenience, `True` and `False` are pre-defined Arduino values.

The values of `True` and `False` are the result of evaluating a logical expression and those values can also be stored in variables. A *boolean*[1] variable can take on only two states, `True` or `False`. `True` is a boolean value equal to one or any value not numerically equal to zero. `False` is a boolean value equal to zero. Ordinary numerical variables can also be used in logical expressions. When interpretting a numerical value in a logical expression, the numerical value zero is considered to be `False`, and any numerical value that is not zero (1, 2, 25, -13.23) is considered to be `True`.

1.3 Logical Expressions

A *logical expression* is at the core of any conditional execution. You can think of a logical expression as a formula. However, unlike a numerical formula, which results in a numerical value, a logical expression results in either `True` or `False`. Figure 2 shows a comparison between a numerical and a logical expression. Note that in both cases the variables involved are integers (`int`s).

Logical expressions are created with the *logical operators* listed in Table 1. Most of the logical operators take two operands. For example, the expression `x < y` is compares the magnitudes of the operands `x` and `y`. The result of evaluating `x < y` is either `True` or `False`, depending on the values currently stored in `x` and `y`.

The `!` or "not" operator modifies the result of a logical expression to change the value of the expression to its opposite. The `!` operator takes only one operand, which is the logical expression that it modifies. In Figure 2, the statement `v = !w` assigns a value to `v` that is the logical opposite of what is stored in `w`. Thus, if `True` is stored in `w`, the expression `v = !w` stores `False` in `v`.

[1] Named after mathematician and logician George Boole. See, for example the Wikipedia article, `https://en.wikipedia.org/wiki/Boolean_data_type`.

```
int x,y,z;

x = 3;
y = 2;
z = x/y;

Result: 1 is stored in z
```

```
int v,w,x,y;

x = 3;
y = 2;
v = x < y;
w = !v;

Result: 0 is stored in w
```

Figure 2: Contrast between numerical expressions (left) and logical expressions (right).

Table 1: Logical operators.

Operator	Meaning	Example
`==`	is equal to	`if ( x==y ) { ... }`
`!=`	is not equal to	`if ( x!=y ) { ... }`
`<=`	is less than or equal to	`if ( x<=y ) { ... }`
`>=`	is greater than or equal to	`if ( x>=y ) { ... }`
`<`	is less than	`if ( x<y ) { ... }`
`>`	is greater than	`if ( x>y ) { ... }`
`!`	not (unary operator)	`z = !x`

1.4 Flow Charts

A flow chart is a graphical tool for representing the sequence of operations in a computer code. Flow charts are especially helpful in visualizing the sequence of steps in a condition execution.

Think of a flow chart as a high level map of a program. The computer code is represented by blocks connected by arrows. Labels on the blocks are written in plain English and use phrases to summarize a decision or a calculation. Flow charts allow us to focus on the logic without getting lost in syntax and other details of code.

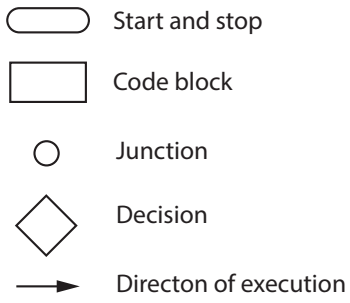

Figure 3: Flow chart symbols.

Figure 3 shows the symbols used to create flow charts. The rounded rectangles that designate start and stopping points are simple markers. As their name suggest, the start and stop symbols identify the beginning and end of an algorithm.

Code blocks are the major steps in the execution of the program. Think of a code block as a chunk of code that achieves a specific task like turning on an LED, reading a sensor, or calculating the desired speed of a motor. When translated to an Arduino sketch, the instructions in a code block may take one or more lines of code.

Junctions indicate points where two or more streams of logic merge. *Decision blocks* represent logical tests that determine which of two branches of code to execute. A decision block usually is translated to an `if` statement. *Arrows* indicate the direction of the code, and connect divergent branches at junctions.

Figure 4 shows two examples of flow charts. The flow charts make it possible to understand how those codes work without needing to read an Arduino sketch. Of course, to create a useful program the ideas expressed in the flow chart must ultimately be expressed in working code.

The flow chart on the left side of Figure 4 is for the familiar "blink" program that flashes an LED on and off. The flow chart on the right side of Figure 4 is for a nightlight that turns on a light when the ambient light level is low.

The night light program has one conditional execution feature. The ambient light level is compared to a user-specified threshold value that determines whether the light is turned on or off. After

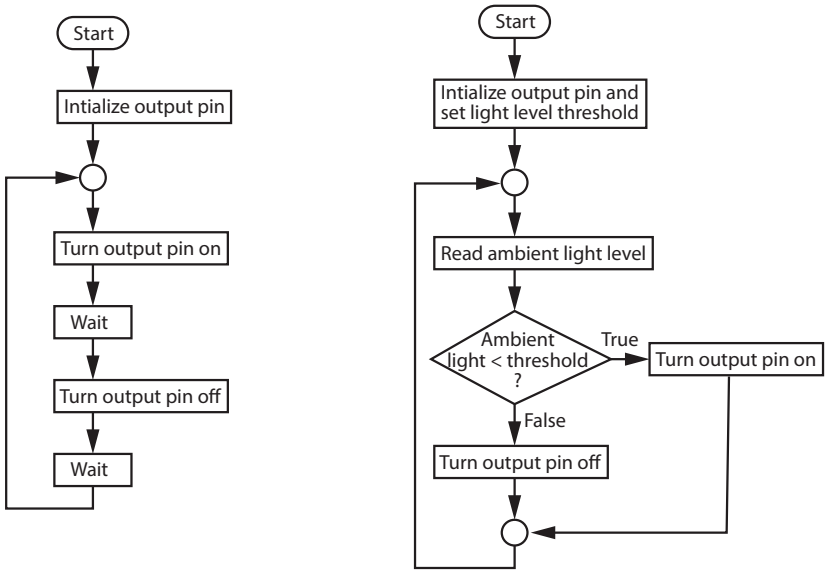

Figure 4: Flow charts for the "blink" program (left) and the "night light" program (right).

the decision block, the program loops back to the top, where the light level is measured again.

Flow charts can be used during code development, and as a method of documenting code after it is finished. In the early stages of code development, especially when you are trying to figure out the logical sequence necessary to achieve a goal, a flow chart can be a helpful way to organize ideas. Unless you are a very experienced programmer and the steps in the program are fairly straightforward, it's usually much faster to sketch (and re-sketch) a flow chart than it is to write and debug code line-by-line.

Flow charts are also useful for documenting code. Logical concepts can usually be expressed more succinctly in a flow chart than in written English. When using flow charts to document code, it is a good idea to make the flow chart compact enough to fit on a single page. That is not an argument for using small fonts and cramming details into a small space. Rather, when creating a flow chart to document a sketch, focus on the highest levels of logic and use plain English to express ideas that may, in fact, take many lines of code.

2 `if` and Its Variants

There are several structures that use an `if` statement to initiate a logical decision. We will start with the basic `if` structure, and then add options.

2.1 The Basic `if` Structure

The basic `if` structure allows a single block of code to be executed when a logical expression is `True`. Figure 5 shows the syntax and flow chart for a basic `if` structure. In Figure 5, "test" represents logical expression that results in a `True` or `False` value. The *`conditional execution block`* can be one or more lines of code that are evaluated only if the `test` expression is `True`. Whether or not the `test` expression is `True`, the program continues execution immediately after the closing brace ("}").

```
if ( test ) {

   conditional execution block

}
```

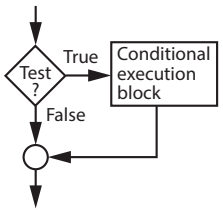

Figure 5: Skeleton code and flowchart for a basic `if` structure.

Example 1 Warning for a floating point error

In standard numerical computation, attempting to take the square root of a negative number causes an error. On an Arduino board, evaluating `y = sqrt(x);` when $x < 0$ causes a value of `NaN` (not a number) to be store in `x`. The following code snippet shows how an `if` statement could be used to generate a warning message when `x` becomes negative.

```
x = ...

if ( x<0 ) {
  Serial.println("WARNING: x is negative");
}
```

□

Example 2 Blink when input is below a threshold

Suppose you wanted a warning indicator when a voltage level on your system fell below a threshold value. Perhaps the voltage was the output of a temperature sensor and the warning light was used to show that the system temperature was too cold. The following code snippet shows how an `if` statement could be used to blink an LED when the reading is lower than a threshold. In this example, nothing happens when the reading is above the threshold.

```
threshold = ...    //  Set equal to a reasonable value

reading = analogRead(inputPin);

if ( reading < threshold ) {
  digitalWrite(warningPin, HIGH);
  delay(500);
  digitalWrite(warningPin, LOW);
  delay(500);
}
```

□

2.2 The `if-else` Structure

The `if-else` structure implements and either-or kind of decision. Each of two outcomes has its own block of code. If the test is true, code block 1 is executed. Otherwise, code block 2 is executed.

Compare the flow charts in Figure 5 and Figure 6. The flow chart for the `if-else` structure makes it clear that when conditional block 1 ("if"-is-true block) is chosen, the code *skips over* conditional block 2 (the else block).

```
if ( test ) {

   conditional execution block 1

} else {

   conditional execution block 2

}
```

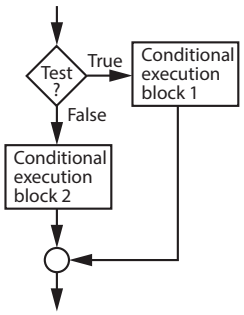

Figure 6: Skeleton code and flowchart for an `if-else` structure.

Example 3 Warning for a floating point error – version 2

We expand on Example 1 by adding and `else` block that allows the square root operation to proceed, and to supply a default value for the case when `x` is negative.

```
x = ...

if ( x<0 ) {
  Serial.println("WARNING: x is negative");
  y = 0.0;
} else {
  y = sqrt(x);
}
```

□

Example 4 Turn on a warning light when input is below a threshold

In Example 2 an LED blinks once when a voltage level falls below a threshold value. Instead of quickly blinking an LED, it may be more effective to leave a warning light on constantly as long as the voltage is too low. The following code snippet shows one way to implement that feature.

```
threshold = ...

reading = analogRead(inputPin);

if ( reading < threshold ) {
  digitalWrite(warningPin, HIGH);
} else {
  digitalWrite(warningPin, LOW);
}
```

□

2.3 The `if-elseif` Structure

A second `if` can immediately follow an `else` statement. This gives two closely related structures: the `if-elseif` and the `if-elseif-else`. The difference is subtle, but important.

The `if-elseif` structure creates two choices. Figure 7 shows the skeleton code and flow chart for an `if-elseif` structure. Note that if both tests are `False`, neither code block is executed. Compare the flowchart in Figure 7 to the flowchart in Figure 6 to see the important difference.

Example 5 uses a furnace control (a thermostat) as an example of how an `if-elseif` structure could be used.

```
if ( test1 ) {

   conditional execution block 1

} else if ( test2 ){

   conditional execution block 2

}
```

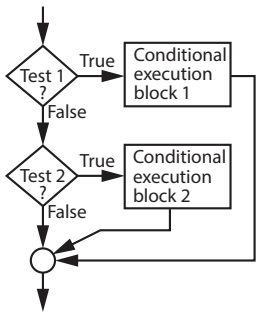

Figure 7: Skeleton code and flowchart for a `if-elseif` structure.

Example 5 Furnace control with an `if-elseif` structure

Consider the logical decision-making used to turn on a furnace. The thermostat has a lower temperature setting, say 18 °C, that determines whether heat needs to be added. The thermostat also has an upper temperature setting, say 22 °C, that determines whether the room temperature is warm enough that the furnace should turn off. Here is the logic for the thermostat expressed in words.

> If the temperature is less than 18 °C, turn on the furnace.
>
> If the temperature is greater than 22 °C, turn off the furnace.
>
> Otherwise, do nothing.

Figure 8 shows an incomplete Arduino sketch and the corresponding flowchart for implementing the thermostat control with an `if-elseif` structure.

The furnace thermostat is a somewhat unusual example of a logic structure where the `if-elseif` does not also have a closing `else` block. In most cases involving `if-elseif`, a closing `else` block is a good idea, as is discussed in the next section.

□

```
float Tlow  = 18.0;
float Thigh = 22.0;

void loop() {

  //  Read temperature; details elsewhere
  float T = getTemperatureReading();

  if ( T<Tlow ) {

    //   Turn the furnace on
    digitalWrite(heaterCircuitControl, HIGH);

  } else if ( T>Thigh ) {

    //   Turn the furnace off
    digitalWrite(heaterCircuitControl, LOW);
  }
}
```

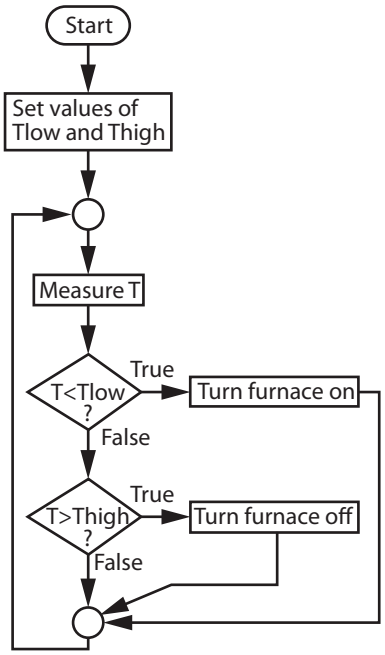

Figure 8: Incomplete Arduino code (left) and flowchart (right) for a thermostat

2.4 The `if-elseif-else` Structure

The `if-elseif-else` structure adds an important *default* condition to the the `if-elseif` structure. The default condition handles cases when none of the `if` conditions are true.

Notice that in Example 5, no special action is taken when the temperature is between the limits of `Tlow` and `Thigh`. For the furnace example, that either-or choice is OK. However, in many (if not most) cases it is a good idea to also include an `else` block when `if-elseif` is used. Figure 9 shows the skeleton code and flow chart for an `if-elseif-else` structure.

Example 6 extends the furnace control logic from Example 5 by adding indicator lights to display the status of the furnace. In this case three states of the furnace are of interest: cooling on, heating on, and furnace off. The furnace off condition is the goldilocks case of the temperature being neither too hot or too cold.

Example 7 gives an example `if-elseif-else` structure along with three examples of how the flow of execution depends on the data supplied to the test cases in the structure. Although the logic in Example 7 is based on some arbitrary computations, it is important to understand how the flow logic depends on the values of `s` and `x`.

Example 8 shows how an sequence of `if-elseif-else` statements can be used to create an Arduino function to evaluate a piecewise continuous mathematical function. Study this example carefully because (1) it is practical, and (2) because it shows that the sequence of tests (progressing from small x to large x) means that on "greater than x_1 and less than x_2" type of test is necessary.

```
if ( test1 ) {

   conditional execution block 1

} else if ( test2 ) {

   conditional execution block 2

} else {

   conditional execution block 3

}
```

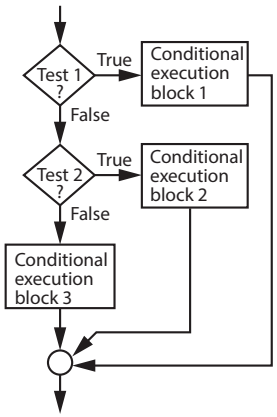

Figure 9: Skeleton code and flowchart for a `if-elseif-else` structure.

Example 6 Furnace control with Status Indicator Lights

Let's extend Example 5 by adding indicator lights to show the status of the heating system.

> If the temperature is less than 18 °C, turn on the furnace and turn on the blue LED to indicate that the temperature is too cold.
>
> If the temperature is greater than 22 °C, turn off the furnace and turn on the red LED to indicate that the temperature is too warm.
>
> Otherwise, turn on the green LED to indicate that the temperature is within the dead band, i.e., the temperature is OK.

The furnace control and indicator lights would be implemented with an `if-elseif-else` structure, as shown in the following code snippet. Note that in addition to turning on the appropriate indicator light, we have to be sure that the other indicator lights are off.

The following incomplete Arduino code uses the `if-elseif-else` structure. Creating the corresponding flowchart is left as an exercise.

```
float Tlow  = 18.0;
float Thigh = 22.0;

void loop() {

  float T = getTemperatureReading();  // Read temperature; details elsewhere

  if ( T<Tlow ) {
    digitalWrite(heaterCircuitControl, HIGH);   //  Turn the furnace on
    digitalWrite(blueLED, HIGH)                 //  Turn on the blue LED
    digitalWrite(redLED, LOW)                   //  Make sure the red LED is off
    digitalWrite(greenLED, LOW)                 //  Make sure the green LED is off
  } else if ( T>Thigh ) {
    digitalWrite(heaterCircuitControl, LOW);    //  Turn the furnace off
    digitalWrite(redLED, HIGH)                  //  Turn on the red LED
    digitalWrite(blueLED, LOW)                  //  Make sure the blue LED is off
    digitalWrite(greenLED, LOW)                 //  Make sure the green LED is off
  } else {
    digitalWrite(greenLED, HIGH)                //  Turn on the green LED
```

```
    digitalWrite(redLED, LOW)                  //  Make sure the red LED is off
    digitalWrite(blueLED, LOW)                 //  Make sure the blue LED is off
  }
}
```

□

Example 7 Alternative flows through an `if-elseif-else` structure

Conditional execution allows a program to respond to changes in data. Figure 10 shows an example `if-elseif-else` structure. The flow through the structure depends on the data, which in this example are the values stored in `s` and `x`. The bottom half of Figure 10 shows possible paths through this structure for different values of `s` and `x`.

□

Example 8 Creating a piecewise linear "hat" function.

An `if-elseif-else` structure can be used to evaluate a piecewise function. Consider the "hat" function shown in Figure 11. The goal is to write the Arduino code that evaluates $y = f(x)$ for any x when $f(x)$ is defined by the piecewise continuous "curve" in Figure 11.

We assume that a user specifies values of x_1, x_2, x_3, y_1 and y_2. Those values define the function, i.e., specify the shape of the hat. We also assume that the horizontal line segments for $x < x_1$ and $x > x_2$ continue indefinitely.

A little algebra shows that the inclined line segments are defined by these formulas for $f_1(x)$ and $f_2(x)$

$$f_1(x) = \frac{y_2 - y_1}{x_2 - x_1}\, x - \frac{y_1 x_2 - y_2 x_1}{x_2 - x_1} \tag{1}$$

$$f_2(x) = \frac{y_1 - y_2}{x_3 - x_2}\, x - \frac{y_2 x_3 - y_1 x_2}{x_3 - x_2} \tag{2}$$

The following code snippet evaluates the piecewise function in Figure 11, using the formulas in Equation (1) and Equation (2).

```
if ( x<x1 ) {

  y = y1;

} else if ( x<x2 ) {

  y = ( (y2-y1)*x + y1*x2 - y2*x1 )/(x2-x1);

} else if ( x<x3 ) {

  y = ( (y1-y2)*x + y2*x3 - y1*x2 )/(x3-x2);

} else {

  y = y1;

}
```

For this use case (evaluation of a function), it would make sense to put the preceding code snippet in a function that would return `y` for input values of `x`, `x1`, `x2`, `y1` and `y2`.

□

Base Code:

```
if ( s<5 ) {
   x = x*x;
} else if ( s<10 ) {
   x = x - 10;
} else {
   x = 0;
}
Serial.println(x);
```

```
x = 25;
s = 1;
if ( s<5 ) {          True
   x = x*x;
} else if ( s<10 ) {
   x = x - 10;
} else {
   x = 0;             Continue
}
Serial.println(x);
   prints  625
   ( = 25*25)
```

```
x = -5;
s = 7.35;
if ( s<5 ) {          False
   x = x*x;
} else if ( s<10 ) {  True
   x = x - 10;
} else {
   x = 0;             Continue
}
Serial.println(x);
   prints  -15
   ( = -5 - 10 )
```

```
x = 3;
s = 12.2;
if ( s<5 ) {          False
   x = x*x;
} else if ( s<10 ) {  False
   x = x - 10;
} else {
   x = 0;             Continue
}
Serial.println(x);
   prints  0
```

Figure 10: An example `if-else` `if-else` structure. The bottom three examples show the paths through the logic for different values of `x` and `s`. The base code is the same for all three examples.

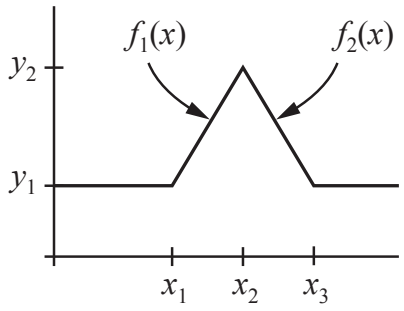

Figure 11: Hat function composed of piecewise continuous line segments

3 To Do

- Add discussion and examples of using compound `&&` and `||` operators in logical expressions.
- Add a summary section.

Exercises

1. What is the value of `z` after each of the following code snippets is executed?

a.
```
x = 2;
y = 5;
z = 0;
if ( x < y ) {
  z = y { x;
}
```

b.
```
x = 2;
y = 5;
z = 0;
if ( x > y ) {
  z = y { x;
}
```

c.
```
x = 2;
y = 5;
z = 0;
if ( (y-x)<= 3 ) {
  z = y/x;
}
```

d.
```
x = 2;
y = 5;
if ( (y-x)<= 3 ) {
  z = y/x;
} else {
  z = 0.0;
}
```

2. Write the flow chart that corresponds to the following code snippet. Assume that `redLED` and `yellowLED` are digital I/O pin numbers (between 0 and 13) and that those digital I/O pins are connected to LED circuits for a red LED and a yellow LED, respectively.

```
v = analogRead(A0);    //  Value between 0 and 1023 corresponding
                       //  the voltage on analog input pin A0
if ( v<=100 ) {
  digitalWrite(redLED, HIGH);
  digitalWrite(yellowLED, LOW);
}
if ( v>=900 ) {
  digitalWrite(yellowLED, HIGH)
  digitalWrite(redLED, LOW);
}
```

   Is this code/logic an `if-elseif` structure, and `if-elseif-else` structure or something different? If the preceding code snippet is in the `loop` function of an Arduino sketch, what happens with the value of `v` is greater than 100 and less than 900?

3. Create a flow chart that uses an `if-elseif-else` structure for a modified version of the code from preceding exercise. The modified code should implement the following logic

   If `v` is less than or equal 100, turn on the red LED only.

   If `v` is greater than or equal 900, turn on the red yellow only.

   Otherwise, all LEDs should be off.

4. Write an Arduino code that implements the logic from the preceding exercise. Build a circuit to test the code, using a potentiometer to create the input values.

5. Create a flow chart for the `if-elseif-else` structure that describes the thermostat code in Example 6.

6. In Example 6, what happens when $T_{\text{low}} < T < T_{\text{high}}$? From a practical perspective this behavior is desirable. Can you explain why?

7. Create a flow chart for the `if-elseif-else` structure that evaluates the hat function in Example 8.

8. Using the code in Example 8 as a starting point, write an arduino function called `hatfun` that evaluates the hat function for any input values of x_1, x_2, x_3, y_1 and y_2. Test your function with the code on the following page. The values of `x1`, `x2`, and `x3` can be chosen to give a symmetric function, but are not required to produce a symmetry function.

```
void setup() {

  float dx, x, xmin=-1.5, xmax=1.5, y;
  float x1=-1.0, x2=0.0, x3=1.0, y1=0.0, y2=1.0;

  Serial.begin(9600);
  Serial.println("\nTest of hatfun\n\n  x    f(x)");

  dx = 0.25;
  for ( x=xmin; x<=xmax; x+=dx ) {
    y = hatfun(x,x1,x2,x3,y1,y2);
    Serial.print(x);  Serial.print("  ");  Serial.println(y);
  }

}
```