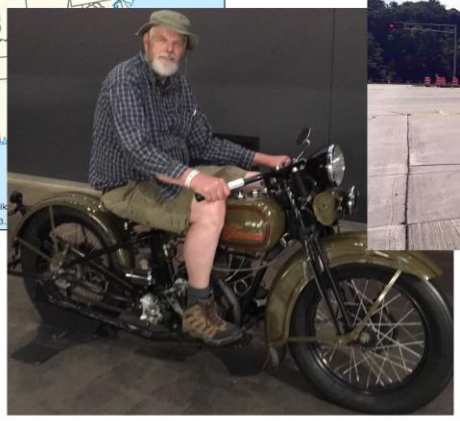
C/C++ Programming

if … else if … else
while loops, **do** loops
DeMorgan's Theorem
switch/case

Dan McElroy
June 2017

Welcome. This video discusses the **if … else if … else** control structure, some information on **while** and **do** loops, **DeMorgan's Theorem** and the C and C++ version of the **switch/case** statements.

Trip to Wisconsin



I went to visit my college roommate in Wisconsin and we took a trip to the Harley Davidson museum. That's me – sitting on a Harley in the museum. I wasn't worried about skinning my knees because the bike was bolted to the floor. I couldn't help taking a picture of this road sign. The county roads were identified by one or two characters. Check it out. Here is the intersection of roads PP and TA.

Program for R Y G

R = Red, display STOP

Y = Yellow, display CAUTION

G = Green, display GO

Q = Quit

Actually, what is this video about? I want to show how to organize a program with **if** and **else if** statements for a traffic light. This simple program inputs a single character R, Y or G for red, yellow and green. The program then displays a message STOP, CAUTION or GO. I also want the program to reject illegal inputs and treat upper-case and lower-case letters the same.

Version 1 – Only using **if** statements

```
char light;     // user enters R, Y or G  for red, yellow or green light

do              // loop until a 'Q' is entered
{
    cout << "Enter R, Y or G  for red, yellow or green light. Q to quit: ";
    cin  >> light;

    if (light == 'R' || light == 'r')
        cout << "STOP" << endl;
    if (light == 'Y' || light == 'y')
        cout << "CAUTION" << endl;
    if (light == 'G' || light == 'g')
        cout << "GO" << endl;
    if (light!='R' && light!='r'&& light!='Y' && light!='y' && light!='G' && light!='g')
        cout << "Illegal selection. Try again" << endl;
} while (light!='Q' && light!='q');
return 0;
```

Here is the C++ version of the code. The cout and cin statements need to be replaced with printf or scanf to make the program work in the C-language. At the top, **light** is defined as a character variable. The **do** loop encloses all of the code that inputs a character and displays the output. That way, the program can keep running for multiple inputs. The 'Q' character is used to quit the program.

(slide continued)

Version 1 – Only using **if** statements

```
char light;     // user enters R, Y or G  for red, yellow or green light

do              // loop until a 'Q' is entered
{
    cout << "Enter R, Y or G  for red, yellow or green light. Q to quit: ";
    cin  >> light;

    if (light == 'R' || light == 'r')
        cout << "STOP" << endl;
    if (light == 'Y' || light == 'y')
        cout << "CAUTION" << endl;
    if (light == 'G' || light == 'g')
        cout << "GO" << endl;
    if (light!='R' && light!='r'&& light!='Y' && light!='y' && light!='G' && light!='g')
        cout << "Illegal selection. Try again" << endl;
} while (light!='Q' && light!='q');
return 0;
```

A prompt is displayed on the screen, "Enter R, Y or G" and a character is input from the keyboard and stored in the **light** variable. This version of the program uses only the **if** statements. A separate **if** statement is used to check for either an upper-case or lower-case character input. The double equal operator == is used to compare the character in the variable named **light** to a single character literal. The single quotes ' in C and C++ are used to identify a single character literal. The double quotes " are used in C and C++ to identify a character string.

The last **if** statement checks to make sure that only legal inputs are made. If anything except R, r, Y, y, G or g is input an error message is displayed. The use of the logical AND && may look a little strange here, but more on this later in the video.

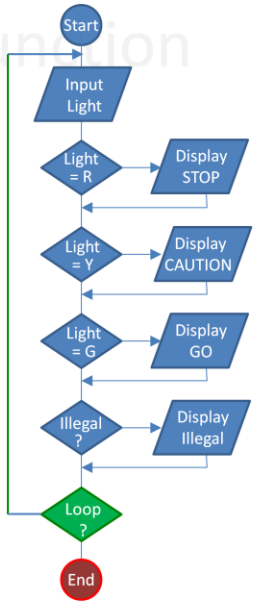
Version 2 –The toupper( ) function

```
char light;     // user enters R, Y or G  for red, yellow or green light

do              // loop until a 'Q' is entered
{
    cout << "Enter R, Y or G  for red, yellow or green light. Q to quit: ";
    cin  >> light;
    light = toupper(light);     // needs #include <cctype> in C++ or
                                //        #include <ctype.h> in C
    if (light == 'R')
        cout << "STOP" << endl;
    if (light == 'Y')
        cout << "CAUTION" << endl;
    if (light == 'G')
        cout << "GO" << endl;
    if (light!='R' && light!='Y' && light!='G')
        cout << "Illegal selection. Try again" << endl;
} while (light!='Q');
return 0;
```

The program can be simplified by converting the input variable to an upper-case character. The **if** statements now only need to test for upper-case characters R, Y and G.

Version 2 –The toupper( ) function

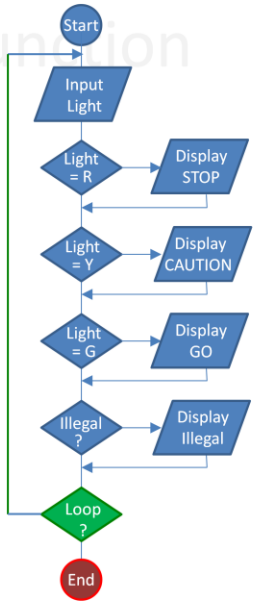
```
char light;     // user enters R, Y or G  for red, yellow or green light

do              // loop until a 'Q' is entered
{
    cout << "Enter R, Y or G  for red, yellow or green light. Q to quit: ";
    cin  >> light;
    light = toupper(light);      // needs #include <cctype> in C++ or
                                 //        #include <ctype.h> in C
    if (light == 'R')
        cout << "STOP" << endl;
    if (light == 'Y')
        cout << "CAUTION" << endl;
    if (light == 'G')
        cout << "GO" << endl;
    if (light!='R' && light!='Y' && light!='G')
        cout << "Illegal selection. Try again" << endl;
} while (light!='Q');
return 0;
```



Here is a flowchart showing the execution of the program. When the logical expression for each **if** statement evaluates to **true**, the **cout** statement executes and control flows down to the next statement. If the expression evaluates to **false**, the **cout** is skipped and control still flows down to the next statement.

Version 2 –The toupper( ) function

```
char light;     // user enters R, Y or G  for red, yellow or green light

do              // loop until a 'Q' is entered
{
    cout << "Enter R, Y or G  for red, yellow or green light. Q to quit: ";
    cin  >> light;
    light = toupper(light);      // needs #include <cctype> in C++ or
                                 //        #include <ctype.h> in C
    if (light == 'R')
        cout << "STOP" << endl;
    if (light == 'Y')
        cout << "CAUTION" << endl;
    if (light == 'G')
        cout << "GO" << endl;
    if (light!='R' && light!='Y' && light!='G')
        cout << "Illegal selection. Try again" << endl;
} while (light!='Q');
return 0;
```



Let's look again at the last **if** statement and see what would happen if it were changed to use the logical-OR || instead of the logical-AND &&. With the logical-OR, if any of the conditions are true, the resulting expression becomes true. So, if somebody input an 'X', the if statement would always be true because the first test is **light not equal 'R'** is true. But if somebody did input an 'R', the first test is false but the second test **light not equal 'Y'** is true. Since the entire expression is a collection of **logical-OR** tests, any of the comparisons being true will cause the entire expression to be true. The message "Illegal selection. Try again" is displayed even regardless of what character is input.

DeMorgan's Theorem

The dome light is **ON** if,
the driver's door is **OPEN**
OR
the passenger door is **OPEN**

The dome light is **ON** if,
NOT (
the driver's door is **NOT OPEN**
AND
the passenger door is **NOT OPEN**
)

The same result is achieved if all of the inputs and the output is complemented with an AND changed to an OR, or an OR changed to an AND.

It might be easier to think of this negative logic using car doors as an example. If either of the doors are open, the dome light turns on. This is true even if only one of the doors is open. The logical-OR evaluates to TRUE if either condition is TRUE.

Instead of just saying what happens to turn the light ON, I am going to use DeMorgan's Theorem to get the same result by complementing (changing to the opposite state) each of the input and output conditions and changing the OR to an AND. The dome light is ON if NOT(both the driver's door is NOT OPEN AND the passenger door is NOT OPEN). The result is exactly the same.

DeMorgan's Theorem

```
bool domeLightOn (bool driverDoorOpen, bool passengerDoorOpen)
{
    // returns true if light is on, or false if the light is off
    return (driverDoorOpen || passengerDoorOpen);

    // same result
    return ( ! (!driverDoorOpen && !passengerDoorOpen) );
}
```

Here it is in code that uses the **bool** data type. The **bool** data type only has two values, **true** and **false**. The function named **domeLightOn** has two input parameters, both of type bool. The function returns a true to indicate the light is on, and a false to indicate that the light is off. The first return statement is probably easier to understand. It returns TRUE if either the driverDoorOpen is TRUE or the passengerDoorOpen is TRUE. The extra spaces around the parentheses ( ) are only there to help make the code a little easier to read. They are not required to make the program work.

DeMorgan's Theorem

```
bool domeLightOn (bool driverDoorOpen, bool passengerDoorOpen)
{
    // returns true if light is on, or false if the light is off
    return (driverDoorOpen || passengerDoorOpen);

    // same result
    return ( ! (!driverDoorOpen && !passengerDoorOpen) );
}
```

The dome light is **ON** if,

NOT (the driver's door is **NOT OPEN**
AND
the passenger door is **NOT OPEN**)

The second return statement has been totally DeMorganized. The exclamation-point operator **!** indicates a logical NOT in C and C++.

The inner parentheses state

(!driverDoorOpen && !passengerDoorOpen)

the driver's door is NOT OPEN AND the passenger door is NOT OPEN. This condition would show the light being in the OFF state, so we complement that entire expression with an exclamation-point for the inner parentheses and change it into defining when the light is ON instead of OFF.

DeMorgan's Theorem

```
bool domeLightOff (bool driverDoorClosed, bool passengerDoorClosed)
{
    // returns true if light is on, or false if the light is off
    return (driverDoorClosed && passengerDoorClosed);


}
```

The dome light is **OFF** if,
the driver's door is **CLOSED**
AND
the passenger door is **CLOSED**

Here is a slightly different version that returns TRUE if the light is OFF. Changing the name of the function to **domeLightOff** does not do anything, but it helps identify what will be happening inside. The dome light is OFF if both the driver door is closed and the passenger door is closed.

DeMorgan's Theorem

```
bool domeLightOff (bool driverDoorClosed, bool passengerDoorClosed)
{
    // returns true if light is on, or false if the light is off
    return (driverDoorClosed && passengerDoorClosed);

    // same result
    return ( ! (!driverDoorClosed || !passengerDoorClosed) );
}
```

The second return statement has been DeMorganized and returns the same result.

You might look that one over to see what is going on, but I am not going to describe it.

The Geniuses of Boolean Algebra

Augustus De Morgan

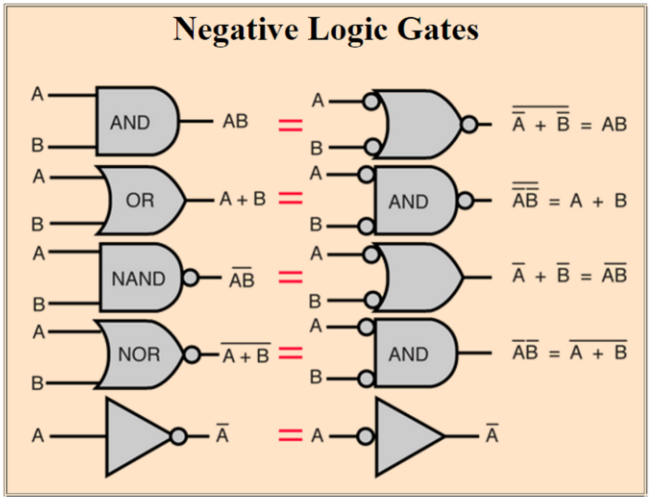
George Boole

Muḥammad ibn
Mūsā al-Khwārizmī

Images from Wikipedia

DeMorgan's Theorem is named after Augustus De Morgan a 19th century British mathematician who was instrumental in developing some of the math behind Boolean Algebra which in turn is named after George Boole, another 19th century mathematician. Algebra is derived from the Arabic word al-jabr and comes from the writings of a Persian mathematician Muhammad ibn Mūsā al-Khwārizmī.

DeMorgan's Theorem Applied to Logic Gates



Computer circuits jump back and forth between positive and negative logic all the time when implementing AND or OR functions. In the first circuit shown, the AND gate performs an OR function if the inputs are inverted.

Diagram from http://hyperphysics.phy-astr.gsu.edu/hbase/Electronic/gate.html at Georgia State University

DeMorgan's Theorem is used extensively in computer hardware to reduce the amount of circuits needed to implement the logic. Computer circuits jump back and forth between positive and negative logic all the time when implementing AND or OR functions. In the first circuit shown, the AND gate performs an OR function if the inputs are inverted.

Although it would not be very efficient, an entire computer could be built using only NAND gates.

DeMorgan's Theorem

```
char light;     // user enters R, Y or G  for red, yellow or green light

do              // loop until a 'Q' is entered
{
    cout << "Enter R, Y or G  for red, yellow or green light. Q to quit: ";
    cin  >> light;
    light = toupper(light);     // needs #include <cctype> in C++ or
                                //        #include <ctype.h> in C
    if (light == 'R')
        cout << "STOP" << endl;
    if (light == 'Y')
        cout << "CAUTION" << endl;
    if (light == 'G')
        cout << "GO" << endl;
    if (light!='R' && light!='Y' && light!='G')
        cout << "Illegal selection. Try again" << endl;
} while (light!='Q');
return 0;
```

Back to our program for the traffic signal. This section of code uses a single **if** statement to reject illegal inputs.

Programming with DeMorgan's

R = Red, display STOP
Y = Yellow, display CAUTION
G = Green, display GO

```
// reject if not any of the acceptable inputs
if (light!='R' && light!='Y' && light!='G')
   cout << "Illegal selection. Try again" << endl;
```

If the character input is not 'R' and the input is not 'Y' and the input is not 'G' then the user typed something illegal. If the logical-OR operator || were used instead of the AND && the expression inside the **if** statement would always be true and an error message would always appear. Think that one over. If the user typed a 'Y' then light!='R' would be TRUE. With OR operators, if any of the selections are true then the entire expression evaluates to a TRUE.

Programming with DeMorgan's

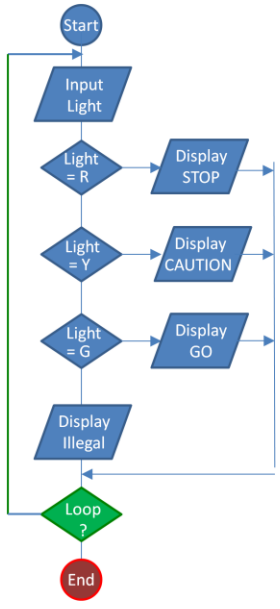
R = Red, display STOP
Y = Yellow, display CAUTION
G = Green, display GO

```
// reject if not any of the acceptable inputs
if (light!='R' && light!='Y' && light!='G')
   cout << "Illegal selection. Try again" << endl;

// test for good inputs, reject if not good
if (! (light=='R' || light=='Y' || light=='G') )
   cout << "Illegal selection. Try again" << endl;
```

The logical expression inside the **if** could use the OR-operator, but the entire expression would need to be DeMorganized.

Version 3 –**if - if else - else**

```
char light;     // user enters R, Y or G  for red, yellow or green light

do              // loop until a 'Q' is entered
{
    cout << "Enter R, Y or G  for red, yellow or green light. Q to quit: ";
    cin  >> light;
    light = toupper(light);

    if (light == 'R')
        cout << "STOP" << endl;
    else if (light == 'Y')
        cout << "CAUTION" << endl;
    else if (light == 'G')
        cout << "GO" << endl;
    else  // default condition
        cout << "Illegal selection. Try again" << endl;
} while (light!='Q');
return 0;
```



Here is a flowchart showing the **if … else if** control structure. A circle represents an entry or exit to the routine, a parallelogram represents input or output and a diamond represents a decision.

Version 3 –**if - if else - else**

```
char light;     // user enters R, Y or G  for red, yellow or green light

do              // loop until a 'Q' is entered
{
    cout << "Enter R, Y or G  for red, yellow or green light. Q to quit: ";
    cin  >> light;
    light = toupper(light);

    if (light == 'R')
        cout << "STOP" << endl;
    else if (light == 'Y')
        cout << "CAUTION" << endl;
    else if (light == 'G')
        cout << "GO" << endl;
    else  // default condition
        cout << "Illegal selection. Try again" << endl;
} while (light!='Q');
return 0;
```

It might be easier and would make better coding to provide a default condition so that we test for all the valid inputs using **if** and **else if** statements. At the end of the block of **if** and **else if** statements, we place a final **else** statement without any test expressions (an else cannot have a logical expression) to collect anything that is left over. This is referred to as the default condition. This is much better programming. We don't need to worry about modifying the final **if** in the previous example in the event that something new was added such as processing an **'F'** character for flashing-red.

The only problem with this code is that entering a **'Q'** to quit the program also causes the "Illegal selection…" message to appear. An extra **else if** statement should be added to process the **'Q'**.

Version 4 – Infinite Loop

```
char light;     // user enters R, Y or G  for red, yellow or green light

while (true)    // endless loop - needs to exit somehow
{
    cout << "Enter R, Y or G  for red, yellow or green light. Q to quit: ";
    cin  >> light;
    light = toupper(light);     // needs #include <cctype>

    if (light == 'R')
        cout << "STOP" << endl;
    else if (light == 'Y')
        cout << "CAUTION" << endl;
    else if (light == 'G')
        cout << "GO" << endl;
    else if (light == 'Q')
        break;  // exit out of the while loop
    else  // default condition
        cout << "Illegal selection. Try again" << endl;
};
return 0;
```

I did something else here that is very common in embedded systems, where code becomes part of the product such as a microwave oven. I created a infinite-loop using **while (true)**. Other than embedded systems, it is usually not considered a good idea to use infinite loops. This loop will never end unless something causes it to break out of the loop. The **break** statement is executed when the letter **'Q'** is entered at the keyboard. The **break** causes the program to exit the loop, but not exit the program.

Version 4 – **switch/case** Statements

```
char light;     // user enters R, Y or G  for red, yellow or green light

do              // loop until a 'Q' is entered
{
    cout << "Enter R, Y or G  for red, yellow or green light. Q to quit: ";
    cin  >> light;
    light = toupper(light);

    switch (light)
    {
    case 'R':
        cout << "STOP" << endl;
        break;
    case 'Y':
        cout << "CAUTION" << endl;
        break;
    case 'G':
        cout << "GO" << endl;
        break;
    case 'Q':
        break;  // the 'Q' will be processed by the while statement
    default:
        cout << "Illegal selection. Try again" << endl;
    } // end switch statement

} while (light!='Q');
return 0;
```

The **switch/case** statements provide another way of implementing **if…else if…else** logic. Although some languages have implemented a more robust version of the case statement, C and C++ can only test for an 'integral' value, such as bool, char, and int. The C and C++ languages cannot test for float or double or test for a range of values.

In this example, the **switch** statement selects the value to be tested. The **case** statements identify the values to be compared against the value tested. It is important to place a **break;** statement end each **case**. If the **break;** is missing, flow just continues to the next statement. Some people consider it good programming practice to place a **break;** statement at the end of the default code, although it is not required since program flow will fall out of the switch block anyway.

Multiple Conditions in the **switch** Statement

```
light = toupper(light);

switch (light)
{
case 'R':
    cout << "STOP" << endl;
    break;
case 'Y':
    cout << "CAUTION" << endl;
    break;
case 'G':
    cout << "GO" << endl;
    break;
case 'Q':
    break;  // the 'Q' will be processed by the while
default:
    cout << "Illegal selection. Try again" << endl;
} // end switch statement
```

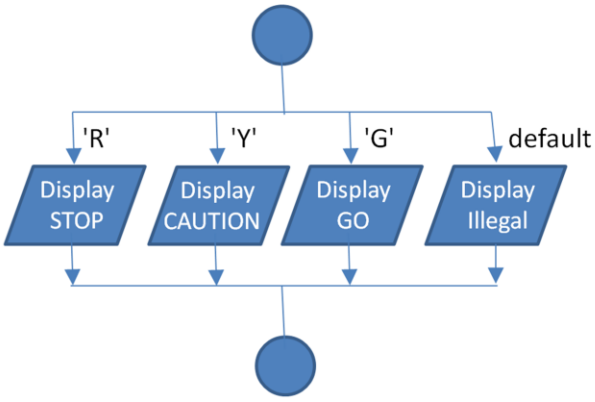
~~light = toupper(light);~~

```
switch (light)
{
case 'R':
case 'r':
    cout << "STOP" << endl;
    break;
case 'Y':
case 'y':
    cout << "CAUTION" << endl;
    break;
case 'G':
case 'g':
    cout << "GO" << endl;
    break;
case 'Q':
case 'q':
    break;  // the 'Q' will be processed by the while statement
default:
    cout << "Illegal selection. Try again" << endl;
} // end switch statement
```

Since the program would keep flowing down, you can use multiple **case** statements for each condition as shown in this example. Instead of converting the **light** variable to upper-case, just use a **case 'R':** followed by a **case 'r':** If either the 'R' or 'r' is matched with the **light** variable, flow keeps going down, outputting the word **STOP** and finally exiting the switch statement when the **break;** is reached. If you forget the **break;** after the cout to display "STOP", flow would keep going down and display "CAUTION" after displaying "STOP".

switch/case Flowchart

```
light = toupper(light);

switch (light)
{
case 'R':
    cout << "STOP" << endl;
    break;
case 'Y':
    cout << "CAUTION" << endl;
    break;
case 'G':
    cout << "GO" << endl;
    break;
case 'Q':
    break;  // the 'Q' will be processed by the while statement
default:
    cout << "Illegal selection. Try again" << endl;
} // end switch statement
```



When using a flowchart to diagram the case statement, the selections are usually graphed horizontally instead of vertically.

The End

As a review, the following topics were discussed in this video. I'm just hoping that I helped make things more clear rather than causing more confusion.

- **if … else if … else**
- **while** loops, **do** loops
- **DeMorgan's Theorem**
- **switch/case**

As a review, the following topics were discussed in this video.

- **if … else if … else**
- **while** loops, **do** loops
- **DeMorgan's Theorem**
- **switch/case**

I'm just hoping that I helped make things more clear rather than causing more confusion. Back to pretending that I'm on a Harley!

The End

Photo and Image Credits

- PP & TA sign – Dan McElroy
- Auto with open doors - by DreamTime.com Royalty free stock photo
- Logic Gates - hyperphysics.phy-astr.gsu.edu/hbase/Electronic/gate.html
- All other photos and images – en.wikipedia.org