



# Conditional Statements

## Introduction

A program is a series of instructions that ask the computer (actually the compiler) to check some situations and to act accordingly. To check such situations, the computer spends a great deal of its time performing comparisons between values. A comparison is a Boolean operation that produces a true or a false result, depending on the values on which the comparison is performed.

A comparison is performed between two values of the same type; for example, you can compare two numbers, two characters, or the names of two cities. On the other hand, a comparison between two disparate values doesn't bear any meaning. For example, it is difficult to compare a telephone number and somebody's age, or a music category and the distance between two points. Like the binary arithmetic operations, the comparison operations are performed on two values. Unlike arithmetic operations where results are varied, a comparison produces only one of two results. The result can be a logical true or false. When a comparison is true, it has an integral value of 1 or positive; that is, a value greater than 0. If the comparison is not true, it is considered false and carries an integral value of 0.

The C++ language is equipped with various operators used to perform any type of comparison between similar values. The values could be numeric, strings, or objects (operations on objects are customized in a process referred to as Operator Overloading).

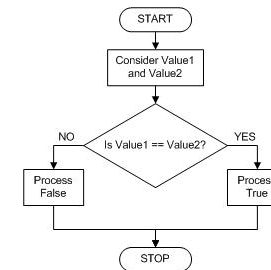
## Logical Operators

### The Equality ==

To compare two variables for equality, C++ uses the == operator. Its syntax is:

```
Value1 == Value2
```

The equality operation is used to find out whether two variables (or one variable and a constant) hold the same value. From our syntax, the compiler would compare the value of Value1 with that of Value2. If Value1 and Value2 hold the same value, the comparison produces a true result. If they are different, the comparison renders false or 0.



Most of the comparisons performed in C++ will be applied to conditional statements; but because a comparison operation produces an integral result, the result of the comparison can be displayed on the monitor screen using a cout extractor. Here is an example:

```

//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Value = 15;

    cout << "Comparison of Value == 32 produces " << (Value == 32) << "\n\n";
    cout << "\n\nPress any key to continue...";
    getch();
    return 0;
}
//-----
  
```

The result of a comparison can also be assigned to a variable. As done with the cout extractor, to store the result of a comparison, you should include the comparison operation between parentheses. Here is an example:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Value1 = 15;
    int Value2 = (Value1 == 24);

    cout << "Value 1 = " << Value1 << "\n";
    cout << "Value 2 = " << Value2 << "\n";
    cout << "Comparison of Value1 == 15 produces " << (Value1 == 15) << "\n\n";

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//-----
```

This would produce:

```
Value 1 = 15
Value 2 = 0
Comparison of Value1 == 15 produces 1

Press any key to continue
```

Very important

The equality operator and the assignment operator are different. When writing `StudentAge = 12`, this means the constant value 12 is assigned to the variable `StudentAge`. The variable `StudentAge` can change anytime and can be assigned another value. The constant 12 can never change and is always 12. For this type of operation, the variable `StudentAge` is always on the left side of the assignment operator. A constant, such as 12, is always on the right side and can never be on the left side of the assignment operator. This means you can write `StudentAge = 12` but never `12 = StudentAge` because when writing `StudentAge = 12`, you are modifying the variable `StudentAge` from any previous value to 12. Attempting to write `12 = StudentAge` means you want to modify the constant integer 12 and give it a new value which is `StudentAge`: you would receive an error.

`NumberOfStudents1 == NumberOfStudents2` means both variables exactly mean the same thing. Whether using `NumberOfStudents1` or `NumberOfStudents2`, the compiler considers each as meaning the other.

## The Logical Not Operator !

When a variable is declared and receives a value (this could be done through initialization or a change of value) in a program, it becomes alive. It can then participate in any necessary operation. The compiler keeps track of every variable that exists in the program being processed. When a variable is not being used or is not available for processing (in visual programming, it would be considered as disabled) to make a variable (temporarily) unusable, you can nullify its value. C++ considers that a variable whose value is null is stern. To render a variable unavailable during the evolution of a program, apply the logical not operator which is `!`. Its syntax is:

!Value

There are two main ways you can use the logical not operator. As we will learn when studying conditional statements, the most classic way of using the logical not operator is to check the state of a variable.

To nullify a variable, you can write the exclamation point to its left. When used like that, you can display its value using the `cout` extractor. You can even assign it to another variable. Here is an example:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Value1 = 250;
    int Value2 = 32;
    int Value3 = !Value1;

    // Display the value of a variable
    cout << "Value1 = " << Value1 << "\n";
    // Logical Not a variable and display its value
    cout << "Value2 = " << !Value2 << "\n";
    // Display the value of a variable that was logically "notted"
    cout << "Value3 = " << Value3 << "\n";

    cout << "\n\nPress any key to continue...";
    getchar();
    return 0;
}
//-----
```

When a variable holds a value, it is "alive". To make it not available, you can "not" it. When a variable has been "notted", its logical value has changed. If the logical value was true, which is 1, it would be changed to false, which is 0. Therefore, you can inverse the logical value of a variable by "notting" or not "notting" it. This is illustrated in the following example:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Value1 = 482;
    int Value2 = !Value1;
```

```

cout << " Value1 = " << Value1 << "\n";
cout << " Value2 = " << Value2 << "\n";
cout << "!Value2 = " << !Value2 << "\n";

cout << "\nPress any key to continue...";
getchar();
return 0;
}
//-----

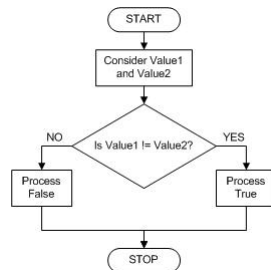
```

## For Inequality !=

As opposed to Equality, C++ provides another operator used to compare two values for inequality. This operation uses a combination of equality and logical not operators. It combines the logical not ! and a simplified == to produce !=. Its syntax is:

Value1 != Value2

The != is a binary operator (like all logical operators except the logical not, which is a unary operator) that is used to compare two values. The values can come from two variables as in Variable1 != Variable2. Upon comparing the values, if both variables hold different values, the comparison produces a true or positive value. Otherwise, the comparison renders false or a null value.



Here is an example:

```

//-----
#include <iostream.h>
#pragma hdrstop
//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Value1 = 212;
    int Value2 = -46;
    int Value3 = (Value1 != Value2);

    cout << "Value1 = " << Value1 << "\n";
    cout << "Value2 = " << Value2 << "\n";
    cout << "Value3 = " << Value3 << "\n\n";

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}

```

```

}
//-----

```

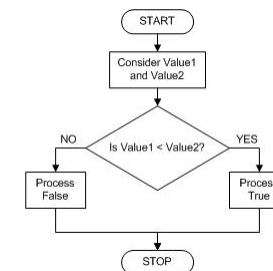
The inequality is obviously the opposite of the equality.

## A Lower Value <

To find out whether one value is lower than another, use the < operator. Its syntax is:

Value1 < Value2

The value held by Value1 is compared to that of Value2. As it would be done with other operations, the comparison can be made between two variables, as in Variable1 < Variable2. If the value held by Variable1 is lower than that of Variable2, the comparison produces a true or positive result.



Here is an example:

```

//-----
#include <iostream.h>
#pragma hdrstop
//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Value1 = 15;
    int Value2 = (Value1 < 24);

    cout << "Value 1 = " << Value1 << "\n";
    cout << "Value 2 = " << Value2 << "\n";

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----

```

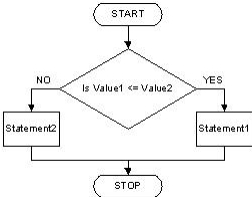
## Combining Equality and Lower Value <=

The previous two operations can be combined to compare two values. This allows

you to know if two values are the same or if the first is less than the second. The operator used is `&#60;&#61;`; and its syntax is:

Value1 <= Value2

The <= operation performs a comparison as any of the last two. If both Value1 and VBlue2 hold the same value, result is true or positive. If the left operand, in this case Value1, holds a value lower than the second operand, in this case Value2, the result is still true.



Here is an example:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Value1 = 15;
    int Value2 = (Value1 <= 24);

    cout << "Value 1 = " << Value1 << "\n";
    cout << "Value 2 = " << Value2 << "\n";

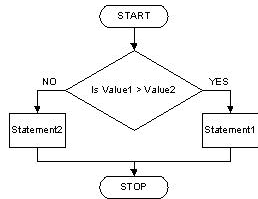
    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----
```

A Greater Value >

When two values of the same type are distinct, one of them is usually higher than the other. C++ provides a logical operator that allows you to find out if one of two values is greater than the other. The operator used for this operation uses the > symbol. Its syntax is:

Value1 > Value2

Both operands, in this case Value1 and Value2, can be variables or the left operand can be a variable while the right operand is a constant. If the value on the left of the > operator is greater than the value on the right side or a constant, the comparison produces a true or positive value . Otherwise, the comparison renders false or null.

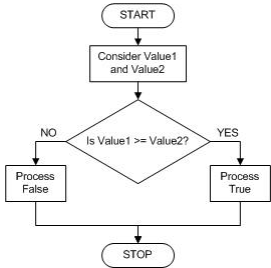


Greater or Equal Value >=

The greater than or the equality operators can be combined to produce an operator as follows: >=. This is the "greater than or equal to" operator. Its syntax is:

Value1 >= Value2

A comparison is performed on both operands: Value1 and Value2. If the value of Value1 and that of Value2 are the same, the comparison produces a true or positive value. If the value of the left operand is greater than that of the right operand,, the comparison produces true or positive also. If the value of the left operand is strictly less than the value of the right operand, the comparison produces a false or null result.



Here is a summary table of the logical operators we have studied:

Operator	Meaning	Example	Opposite
==	Equality to	a == b	!=
!=	Not equal to	12 != 7	==
<	Less than	25 < 84	>=
<=	Less than or equal to	Cab <= Tab	>
>	Greater than	248 > 55	<=
>=	Greater than or equal to	Val1 >= Val2	<

## Conditions

When programming, you will ask the computer to check various kinds of situations and to act accordingly. The computer performs various comparisons of various kinds of statements. These statements come either from you or from the computer itself, while it is processing internal assignments.

Let's imagine you are writing an employment application and one question would be, "Do you consider yourself a hot-tempered individual?" The source file of such a program would look like this:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    char Answer;

    cout << "Do you consider yourself a hot-tempered individual? ";
    cin >> Answer;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----
```

Some of the answers a user would type are y, yes, Y, Yes, YES, n, N, no, No, NO, I don't know, Sometimes, Why are you asking?, and What do you mean? The variety of these different answers means that you should pay attention to how you structure your programs, you should be clear to the users.

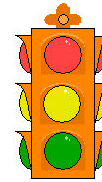


A better version of the line that asks the question would be:

```
cout << "Do you consider yourself a hot-tempered individual? (y=Yes/n=No)";
```

This time, although the user can still type anything, at least you have specified the expected answers.

## Introduction to Conditional Statements



There are three entities that participate on a traffic light: the lights, the human beings who interact with the light, and the law. The road provides a platform on which these components come together.

### The Traffic Light

Everything taken into consideration, a traffic light is made of three light colors: Green – Yellow/Orange – Red. When the light is green, the road is clear for moving in. The red light signals to stop and wait. A yellow light means, "Be careful, it is not safe to proceed right now. Maybe you should wait." When it is not blinking, the yellow light usually serves as a transition period from green to red. There is no transition from red to green.

### The Drivers

There are two main categories of people who deal with the traffic light: the drivers and the walkers. To make our discussion a little simpler, we will consider only the driver. When the light is green, a driver can drive through. When the light is red, the driver is required to stop and wait.

### The Law

Rules and regulations dictate that when a driver does not obey the law by stopping to a red light, he is considered to have broken the law and there is a consequence.

The most independent of the three entities is the traffic light. It does not think, therefore it does not make mistakes. It is programmed with a timer or counter that directs it when to act, that is, when to change lights. The second entity, the driver, is a human being who can think and make decisions based on circumstances that are beyond human understanding. A driver can decide to stop at a green light or drive through a red light...

A driver who proceeds through a red light can get a ticket depending on one of two circumstances: either a police officer caught her "hand-in-the-basket" or a special camera took a picture. Worse, if an accident happens, this becomes another story.

The traffic light is sometimes equipped with a timer or counter. We will call it Timer T. It is equipped with three lights: Green, Yellow, and Red. Let's suppose that the light stays green for 45 seconds, then it turns and stays yellow for 5 seconds, and finally it turns and stays red for 1 minute = 60 seconds. At one moment in the day, the timer is set at the beginning or is reset and the light is green: T = 0. Since the timer is working fine, it starts counting the seconds 1, 2, 3, 4, ... 45. The light will stay green from T = 0 to T = 45. When the timer reaches 45, the timer is reset to 0 and starts counting from 0 until it reaches 5; meanwhile, Color = Yellow.

## Practical Learning: Introduction to Conditional Statements

- 1. Create a new console application named Conditions1
- 2. Create a C++ source file name Exercise
- 3. To apply what we have learned, change the file as follows:

```
//-----  
#include <iostream.h>  
#pragma hdrstop  
//-----  
  
#pragma argsused  
int main(int argc, char* argv[])  
{  
    char Light;  
  
    cout << "What is the current light color(g=Green/y=Yellow/r=Red)? ";  
    cin >> Light;  
  
    cout << "\nThe current color of the light is " << Light << "\n\n";  
  
    cout << "\nPress any key to continue...";  
    getchar();  
    return 0;  
}  
//-----
```

- 4. Test the program
- 5. Return to your programming environment.
- 6. Save the project.

if a Condition is True

In C++, comparisons are made from a statement. Examples of statements are:

- "You are 12 years old"
- "It is raining outside"
- You live in Sydney"


When a driver comes to a traffic light, the first thing she does is to examine the light's color. There are two values the driver would put together: The current light of the traffic and the desired light of the traffic.

Upon coming to the traffic light, the driver would have to compare the traffic light variable with a color she desires the traffic light to have, namely the green light (because if the light is green, then the driver can drive through). The comparison is performed by the driver making a statement such as "The light is green".

After making a statement, the driver evaluates it and compares it to what must be true.

When a driver comes to a traffic light, she would likely expect the light to be green. Therefore, if the light is green (because

that is what she is expecting), the result of her examination would receive the Boolean value of TRUE. This produces the following table:

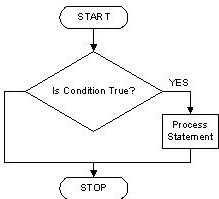
Color	Statement	Boolean Value
	The light is green	true

One of the comparisons the computer performs is to find out if a statement is true (in reality, programmers (like you) write these statements and the computer only follows your logic). If a statement is true, the computer acts on a subsequent instruction.

The comparison using the if statement is used to check whether a condition is true or false. The syntax to use it is:

if(Condition) Statement;

If the Condition is true, then the compiler would execute the Statement. The compiler ignores anything else:



If the statement to execute is (very) short, you can write it on the same line with the condition that is being checked.

Consider a program that is asking a user to answer Yes or No to a question such as "Are you ready to provide your credit card number?". A source file of such a program could look like this:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    char Answer;

    // Request the availability of a credit card from the user
    cout << "Are you ready to provide your credit card number(1=Yes/0=No)? ";
    cin >> Answer;

    // Since the user is ready, let's process the credit card transaction
    if(Answer == '1')cout << "\nNow we will need your credit card number.\n";

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----
```

You can write the if condition and the statement on different lines; this makes your program easier to read. The above code could be written as follows:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    char Answer;

    // Request the availability of a credit card from the user
    cout << "Are you ready to provide your credit card number(1=Yes/0=No)? ";
    cin >> Answer;

    // Since the user is ready, let's process the credit card transaction
    if(Answer == '1')
        cout << "\nNow we will get your credit card information.\n";

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----
```

You can also write the statement on its own line if the statement is too long to fit on the same line with the condition.

Although the (simple) if statement is used to check one condition, it can lead to executing multiple dependent statements. If that is the case, enclose the group of statements between an opening curly bracket "{" and a closing curly bracket "}". Here is an example:

```
//-----
#include <iostream.h>
```

```
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    char Answer;
    char CreditCardNumber[40];

    // Request the availability of a credit card from the user
    cout << "Are you ready to provide your credit card number(1=Yes/0=No)? ";
    cin >> Answer;

    // Since the user is ready, let's process the credit card transaction
    if(Answer == '1')
    {
        cout << "\nNow we will continue processing the transaction.";
        cout << "\nPlease enter your credit card number without spaces: ";
        cin >> CreditCardNumber;
    }

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----
```

If you omit the brackets, only the statement that immediately follows the condition would be executed.

When studying logical operators, we found out that if a comparison produces a true result, it in fact produces a non zero integral result. When a comparison leads to false, its result is equivalent to 0. You can use this property of logical operations and omit the comparison if or when you expect the result of the comparison to be true, that is, to bear a valid value. This is illustrated in the following program:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Number;

    cout << "Enter a non zero number: ";
    cin >> Number;

    if(Number)
        cout << "\nYou entered " << Number << endl;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----
```

## Practical Learning: The if Statement

1. To apply the if condition to our traffic light study, click on the right side of the cin>> Light line and press Enter twice.

2. Change the line to display:

```
if ( Light == 'g' ) cout << "You can proceed and drive through.";
```

- 3. Test the program.
- 4. When prompted, test the program by typing g and press Enter. Notice the sentence that is displaying.
- 5. Press any key to get back.
- 6. Test the program again. This time, type anything else but g and press Enter. Notice that nothing particular displays.
- 7. Return to your programming environment.
- 8. To send the statement to the next line, change the file as follows:

```
if( Light == 'g' )  
    cout << "You can proceed and drive through.";
```



- 9. Test the program and return to your programming environment.
- 10. To process more than one statement for the if condition, change the if section as follows:

```
if( Light == 'g' )  
{  
    cout << "\nThe light is green";  
    cout << "\nYou can proceed and drive through.\n";  
}
```

- 11. Test the program and return to your programming environment.
- 12. Save your project.



Using the Logical Not

When a driver comes to a light that he expects to be green, we saw that he would use a statement such as, "The light is green". If in fact the light is green, we saw that the statement would lead to a true result. If the light is not green, the "The light is green" statement produces a false result. This is shown in the following table:



Color	Statement	Boolean Value
	The light is green	true
	The light is green	false

As you may realize already, in Boolean algebra, the result of performing a comparison depends on how the Condition is formulated. If the driver is approaching a light that he is expecting to display any color other than green, he would start from a statement such as "The light is not green". If the light



IS NOT green, the expression "The light is not green" is true (very important). This is illustrated in the following table:

Color	Statement	Boolean Value
	The light is green	true
	The light is not green	true

The "The light is not green" statement is expressed in Boolean algebra as "Not the light is green". Instead of writing "Not the light is green", in C++, using the logical Not operator , you would formulate the statement as, !"The light is green". Therefore, if P means "The light is green", you can express the negativity of P as !P. The Boolean table produced is:

Color	Statement	Boolean Value	Symbol
	The light is green	true	P
	The light is not green	false	!P

When a statement is true, its Boolean value is equivalent to a non-zero integer such as 1. Otherwise, if a statement produces a false result, it is given a 0 value. Therefore, our table would be:

Color	Statement	Boolean Value	Integer Value
	The light is green	true	1
	The light is not green	false	0

Otherwise: if...else

The if condition is used to check one possibility and ignore anything else. Usually, other conditions should be considered. In this case, you can use more than one if statement. For example, on a program that asks a user to answer Yes or No, although the positive answer is the most expected, it is important to offer an alternate statement in case the user provides another answer. Here is an example:

```
//-----  
#include <iostream.h>  
#pragma hdrstop  
  
//-----  
  
#pragma argsused  
int main(int argc, char* argv[])  
{  
    char Answer;
```



```

cout << "Do you consider yourself a hot-tempered individual(y=Yes/n=No)? ";
cin >> Answer;

if( Answer == 'y' ) // First Condition
{
    cout << "\nThis job involves a high level of self-control.";
    cout << "\nWe will get back to you.\n";
}
if( Answer == 'n' ) // Second Condition
    cout << "\nYou are hired!\n";

cout << "\nPress any key to continue...";
getchar();
return 0;
}
//-----

```

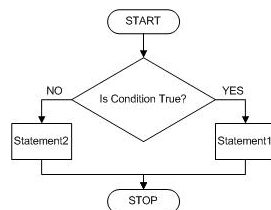
The problem with the above program is that the second if is not an alternative to the first, it is just another condition that the program has to check and execute after executing the first. On that program, if the user provides y as the answer to the question, the compiler would execute the content of its statement and the compiler would execute the second if condition.

You can also ask the compiler to check a condition; if that condition is true, the compiler will execute the intended statement. Otherwise, the compiler would execute alternate statement. This is performed using the syntax:

```

if(Condition)
    Statement1;
else
    Statement2;

```



The above program would better be written as:

```

//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    char Answer;

    cout << "Do you consider yourself a hot-tempered individual(y=Yes/n=No)? ";
    cin >> Answer;

    if( Answer == 'y' ) // One answer
    {
        cout << "\nThis job involves a high level of self-control.";
        cout << "\nWe will get back to you.\n";
    }
    else // Any other answer
        cout << "\nYou are hired!\n";
}

```

```

cout << "\nPress any key to continue...";
getchar();
return 0;
}
//-----

```

## Practical Learning: The if...else Statement

1. To consider when the traffic light's color is other than green, change your program as follows:

```

//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    char Light;

    cout << "What is the current light color(g=Green/y=Yellow/r=Red)? ";
    cin >> Light;

    if ( Light == 'g' )
    {
        cout << "\nThe light is green";
        cout << "\nYou can proceed and drive through.\n";
    }
    else
        cout << "\nPlease wait!\n";

    cout << "\nPress any key to continue...";
    getchar();
    getchar();
    return 0;
}
//-----

```

2. Test the program and return to your programming environment
3. Save your project.

## The Conditional Operator (?:)

The conditional operator behaves like a simple if...else statement. Its syntax is:

Condition ? Statement1 : Statement2;

The compiler would first test the Condition. If the Condition is true, then it would execute Statement1, otherwise it would execute Statement2. When you request two numbers from the user and would like to compare them, the following program would do find out which one of both numbers is higher. The comparison is performed using the conditional operator:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    signed Num1, Num2, Max;

    cout << "Enter two numbers: ";
    cin >> Num1 >> Num2;

    Max = (Num1 < Num2) ? Num2 : Num1;

    cout << "\nThe maximum of " << Num1
         << " and " << Num2 << " is " << Max;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----
```

## Conditional Statements: if...else if and if...else if...else

The previous conditional formula is used to execute one of two alternatives. Sometimes, your program will need to check many more than that. The syntax for such a situation is:

```
if(Condition1)
    Statement1;
else if(Condition2)
    Statement2;
```

An alternative syntax would add the last else as follows:

```
if(Condition1)      if(Condition1)
    Statement1;      Statement1;
else if(Condition2) else if(Condition2)
    Statement2;      Statement2;
else                else if(Condition3)
    Statement-n;      Statement3;
                    else
                    Statement-n;
```

The compiler will check the first condition. If Condition1 is true, it will execute Statement1. If Condition1 is false, then the compiler will check the second condition. If Condition2 is true, it will execute Statement2. When the compiler finds a Condition-n to be true, it will execute its corresponding statement. If that Condition-n is false, the compiler will check the subsequent condition. This means you can include as many conditions as you see fit using the else if statement. If after examining all the known possible conditions you still think that there might be an unexpected condition, you can use the optional single else.

A program we previously wrote was considering that any answer other than y was negative. It would be more professional to consider a negative answer because the program anticipated one. Therefore, here is a better version of the program:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    char Answer;

    cout << "Do you consider yourself a hot-tempered individual(y=Yes/n=No)? ";
    cin >> Answer;

    if( Answer == 'y' ) // Unique Condition
    {
        cout << "\nThis job involves a high level of self-control.";
        cout << "\nWe will get back to you.\n";
    }
    else if( Answer == 'n' ) // Alternative
        cout << "\nYou are hired!\n";
    else
        cout << "\nThat's not a valid answer!\n";

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----
```

## Practical Learning: if...else if and if...else if...else

1. To consider various answers that the user could give, change your program as follows:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    char Light;

    cout << "What is the current light color(g=Green/y=Yellow/r=Red)? ";
    cin >> Light;

    if( Light == 'g' )
    {
        cout << "\nThe light is green";
        cout << "\nYou can proceed and drive through.\n";
    }
    else if( Light == 'y' )
        cout << "\nBe careful!\n";
```

```

    else if( Light == 'r' )
        cout << "\nPlease Stop!!!";
    else
        cout << endl << Light << " is not a valid color.\n";

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----

```

2. Test the program.
3. Type a letter and press Enter. See the result.
4. Test the program again and provide different answers, those that are valid and those that are not.
5. Return to your development environment and save your project.

## The switch Statement

When defining an expression whose result would lead to a specific program execution, the switch statement considers that result and executes a statement based on the possible outcome of that expression, this possible outcome is called a case. The different outcomes are listed in the body of the switch statement and each case has its own execution, if necessary. The body of a switch statement is delimited from an opening to a closing curly brackets: "{ " to "}". The syntax of the switch statement is:

```

switch(Expression)
{
    case Choice1:
        Statement1;
    case Choice2:
        Statement2;
    case Choice-n:
        Statement-n;
}

```

The expression to examine is an integer. Since an enumeration (enum) and the character (char) data types are just other forms of integers, they can be used too. Here is an example of using the switch statement:

```

//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Number;

    cout << "Type a number between 1 and 3: ";
    cin >> Number;

    switch (Number)

```

```

{
    case 1:
        cout << "\nYou typed 1";
    case 2:
        cout << "\nYou typed 2";
    case 3:
        cout << "\nYou typed 3";
    }

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----

```

The program above would request a number from the user. If the user types 1, it would execute the first, the second, and the third cases. If she types 2, the program would execute the second and third cases. If she supplies 3, only the third case would be considered. If the user types any other number, no case would execute.

When establishing the possible outcomes that the switch statement should consider, at times there will be other possibilities other than those listed and you will be likely to consider them. This special case is handled by the default keyword. The default case would be considered if none of the listed cases matches the supplied answer. The syntax of the switch statement that considers the default case would be:

```

switch(Expression)
{
    case Choice1:
        Statement1;
    case Choice2:
        Statement2;
    case Choice-n:
        Statement-n;
    default:
        Other-Possibility;
}

```

Therefore another version of the program above would be

```

//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Number;

    cout << "Type a number between 1 and 3: ";
    cin >> Number;

    switch (Number)
    {
        case 1:
            cout << "\nYou typed 1";
        case 2:
            cout << "\nYou typed 2";

```

```

    case 3:
        cout << "\nYou typed 3";
    default:
        cout << endl << Number << " is out of the requested range.";
    }

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----

```

## Practical Learning: Switching Cases

1. Change the file as follows:

```

//-----
#include <iostream.h>
#pragma hdrstop
//-----

#pragma argsused
int main(int argc, char* argv[])
{
    char Light;

    cout << "What is the current light "
         << "color (g=Green/y=Yellow/r=Red)? ";
    cin >> Light;

    switch( Light )
    {
    case 'g':
        cout << "\nThe light is green";
        cout << "\nYou can proceed and drive through...";
        cout << "\nNow, passing from Green to Yellow...\n";

    case 'y':
        cout << "\nThe light has passed from Green to Yellow\n";
        cout << "Timer = 0. Yellow Light - Be Careful!\n";
        cout << "Timer = 1. Yellow Light - Be Careful!\n";
        cout << "Timer = 2. Yellow Light - Be Careful!\n";
        cout << "Timer = 3. Yellow Light - Be Careful!\n";
        cout << "Timer = 4. Yellow Light - Be Careful!\n";
        cout << "Timer = 5. Yellow Light - Be Careful!\n";
        cout << "\nYellow light ended.\n";

    case 'r':
        cout << "\nRed Light - Please Stop!!!";
        cout << "\nTimer:\t 1 2 3 4 5 6 7 8 9 10";
        cout << "\n\t11 12 13 14 15 16 17 18 19 20";
        cout << "\n\t21 22 23 24 25 26 27 28 29 30";
        cout << "\n\t31 32 33 34 35 36 37 38 39 40";
        cout << "\n\t41 42 43 44 45 46 47 48 49 50";
        cout << "\n\t51 52 53 54 55 56 57 58 59 60";
        cout << "\nEnd of Red Light.";
    }

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----

```

2. Test the program and return to your programming environment.
3. Save your project.

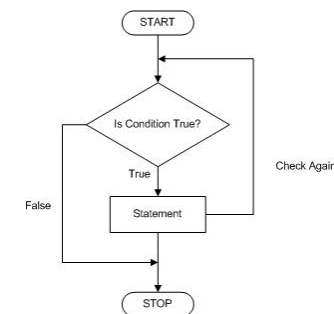
## Counting and Looping

The C++ language provides a set of control statements that allows you to conditionally control data input and output. These controls are referred to as loops.

## The while Statement

The while statement examines or evaluates a condition. The syntax of the while statement is:

while(Condition) Statement;



To execute this expression, the compiler first examines the Condition. If the Condition is true, then it executes the Statement. After executing the Statement, the Condition is checked again. AS LONG AS the Condition is true, it will keep executing the Statement. When or once the Condition becomes false, it exits the loop.

Here is an example:

```

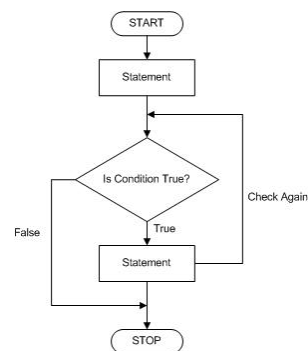
int Number;

while( Number <= 12 )
{
    cout << "Number " << Number << endl;
    Number++;
}

```

To effectively execute a while condition, you should make sure you provide a mechanism for the compiler to use a get a reference value for the condition, variable, or expression being checked. This is sometimes in the form of a variable being

initialized although it could be some other expression. Such a while condition could be illustrated as follows:



An example would be:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Number;

    while( Number <= 12 )
    {
        cout << "Number " << Number << endl;
        Number++;
    }

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----
```

## Practical Learning: Using the while Statement

1. Change the file as follows:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Timer = 0;

    while(Timer <= 5)
```

```

{
    cout << Timer << ". Yellow Light - Be Careful!\n";
    Timer++;
}
cout << "\nYellow light ended. Please Stop!!!\n";

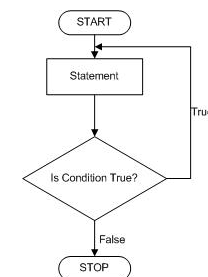
cout << "\nPress any key to continue...";
getchar();
return 0;
}
//-----
```

2. Test the program and return to your programming environment.

## The do...while Statement

The do...while statement uses the following syntax:

do Statement while (Condition);



The do...while condition executes a Statement first. After the first execution of the Statement, it examines the Condition. If the Condition is true, then it executes the Statement again. It will keep executing the Statement AS LONG AS the Condition is true. Once the Condition becomes false, the looping (the execution of the Statement) would stop.

If the Statement is a short one, such as made of one line, simply write it after the do keyword. Like the if and the while statements, the Condition being checked must be included between parentheses. The whole do...while statement must end with a semicolon.

Another version of the counting program seen previously would be:

```
//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Number = 0;

    do
```

```

        cout << "Number " << Number++ << endl;
    while( Number <= 12 );

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----

```

If the Statement is long and should span more than one line, start it with an opening curly bracket and end it with a closing curly bracket.

The do...while statement can be used to insist on getting a specific value from the user. For example, since our ergonomic program would like the user to sit down for the subsequent exercise, you can modify your program to continue only once she is sitting down. Here is an example on how you would accomplish that:

```

//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    char SittingDown;

    cout << "For the next exercise, you need to be sitting down\n";
    do {
        cout << "Are you sitting down now(y/n)? ";
        cin >> SittingDown;
    }
    while( !(SittingDown == 'y') );

    cout << "\nWonderful!!!";
    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----

```

## Practical Learning: Using The do...while Statement

1. To apply a do...while statement to our traffic program, change the file as follows:

```

//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    char Answer;

```

```

do {
    cout << "Check the light: is it green yet(1=Yes/0=No)? ";
    cin >> Answer;
} while(Answer != '1');

    cout << "\nNow, you can proceed and drive through!\n";
    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----

```

2. Test the program. Here is an example:

```

Check the light: is it green yet(1=Yes/0=No)? 5
Check the light: is it green yet(1=Yes/0=No)? b
Check the light: is it green yet(1=Yes/0=No)? y
Check the light: is it green yet(1=Yes/0=No)? s
Check the light: is it green yet(1=Yes/0=No)? 0
Check the light: is it green yet(1=Yes/0=No)? 1

Now, you can proceed and drive through!
Press any key continue...

```

3. Return to your programming environment.
4. Save your project.

## The for Statement

The for statement is typically used to count a number of items. At its regular structure, it is divided in three parts. The first section specifies the starting point for the count. The second section sets the counting limit. The last section determines the counting frequency. The syntax of the for statement is:

for( Start; End; Frequency) Statement;

The Start expression is a variable assigned the starting value. This could be Count = 0;

The End expression sets the criteria for ending the counting. An example would be Count < 24; this means the counting would continue as long as the Count variable is less than 24. When the count is about to reach 24, because in this case 24 is excluded, the counting would stop. To include the counting limit, use the <= or >= comparison operators depending on how you are counting. The Frequency expression would let the compiler know how many numbers to add or subtract before continuing with the loop. This expression could be an increment operation such as ++Count.

Here is an example that applies the for statement:

```

//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    for(int Count = 0; Count <= 12; Count++)

```

```

        cout << "Number " << Count << endl;

        cout << "\nPress any key to continue...";
        getchar();
        return 0;
    }
    //-----

```

The C++ compiler recognizes that a variable declared as the counter of a for loop is available only in that for loop. This means the scope of the counting variable is confined only to the for loop. This allows different for loops to use the same counter variable. Here is an example:

```

//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    for(int Count = 0; Count <= 12; Count++)
        cout << "Number " << Count << endl;
    cout << endl;

    for(int Count = 10; Count >= 2; Count--)
        cout << "Number " << Count << endl;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----

```

Some compilers do not allow the same counter variable in more than one for loop. The counter variable's scope spans beyond the for loop. With such a compiler, you must use a different counter variable for each for loop. An alternative to using the same counter variable in different for loops is to declare the counter variable outside of the first for loop and call the variable in the needed for loops. Here is an example:

```

//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Count;

    for(Count = 0; Count <= 12; Count++)
        cout << "Number " << Count << endl;
    cout << endl;

    for(Count = 10; Count >= 2; Count--)
        cout << "Number " << Count << endl;

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}

```

```

    }
    //-----

```

## Practical Learning: Using the for Statement

1. To apply the for statement to our traffic program, change the file as follows:

```

//-----
#include <iostream.h>
#pragma hdrstop

//-----

#pragma argsused
int main(int argc, char* argv[])
{
    int Timer;

    for( Timer = 0; Timer <= 5; ++Timer)
        cout << Timer << ". Yellow Light - Be Careful!\n";
    cout << "\nYellow light ended. Please Stop!!!\n";

    cout << "\nPress any key to continue...";
    getchar();
    return 0;
}
//-----

```

2. Test the program:

```

0. Yellow Light - Be Careful!
1. Yellow Light - Be Careful!
2. Yellow Light - Be Careful!
3. Yellow Light - Be Careful!
4. Yellow Light - Be Careful!
5. Yellow Light - Be Careful!

Yellow light ended. Please Stop!!!

Press any key to continue...

```

3. Return to your programming environment.
4. Save your project.