Facilities of Programming Languages

In the next few lessons we will explore the operations and functions that are used in all programming languages to allow us to manipulate the data within a program.

The Assignment Operator

All programs use variables. A variable can be given different values throughout the execution of a program. When we set the value of a variable, we assign a value to it. The assignment operator is usually the equals sign (=), and is used as below:

\(<\text{myVariable}> = <\text{value or expression}>\)

For example:

```vbnet
Dim x As Integer
Dim y As Integer

x = 273
y = (8 * x) / 3
```

In this example, the value of 273 is assigned to the integer variable `x`. The value of `x` multiplied by 8 and then divided by 3 is then assigned to the integer variable `y`.

Important Points to Remember

- The assignment operator is NOT the same as the equality operator
- The variable being assigned to must always be on the left of the assignment operator

In VB, the assignment and equality operators use the same symbol (=). This can be confusing for the programmer if he doesn’t know what he is doing! Generally it is fairly obvious whether the = symbolises assignment or equality...

If it is being used inside a conditional statement, it is the equality operator, determining whether two values are the same. For example:

```vbnet
If (x = 3) then    While (x <= 7)    Do
...                  ...                  ...
End If              End While           While (x = 1)
```

If it is being used with a function call, or in the normal flow of the program, it will be the assignment operator. For example:

```vbnet
x = addNumbers(y, z)    x = 3 + 4
```
Arithmetic Operations

Arithmetic operators are used to manipulate numeric data (integers and real numbers). The numbers that are manipulated are called **operands**. You should be familiar with most of these already! An important point to note is that arithmetic operations require the use of the assignment operator if their results are to be made available to the program; an arithmetic operation can only be used as part of an expression in a statement that tells the program what to do with the result.

**Unary Operators**

- Require only 1 operand.
- Operator can come before or after operand

  *e.g.*  
  
  \[- i\]

**Binary Operators**

- Require two operands
- Operator usually written between the two operands

  *e.g.*  
  
  \[i - j\]

Remember that the same operand can have different uses... in the examples above, the – sign is used first as a unary operator to express \(i\) as a negative value and then as a binary operator to represent subtraction of the value of \(j\) from the value of \(i\).

**Familiar Arithmetic Operators**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unary / Binary</th>
<th>Symbol</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>Binary</td>
<td>+</td>
<td>Add 2 numbers</td>
<td>(x = 3 + 4)</td>
</tr>
<tr>
<td>Subtraction</td>
<td>Binary</td>
<td>-</td>
<td>Subtract 2 numbers</td>
<td>(x = 7 - 5)</td>
</tr>
<tr>
<td>Multiplication</td>
<td>Binary</td>
<td>*</td>
<td>Multiply 2 numbers</td>
<td>(x = 4 \times 7)</td>
</tr>
<tr>
<td>Division</td>
<td>Binary</td>
<td>/</td>
<td>Divide 2 numbers</td>
<td>(x = 6 / 4)</td>
</tr>
<tr>
<td>Increment</td>
<td>Unary</td>
<td>++</td>
<td>Add one to number</td>
<td>(i ++)</td>
</tr>
<tr>
<td>Decrement</td>
<td>Unary</td>
<td>--</td>
<td>Subtract one from number</td>
<td>(i --)</td>
</tr>
<tr>
<td>Negation</td>
<td>Unary</td>
<td>-</td>
<td>Express number as negative value</td>
<td>(-8)</td>
</tr>
<tr>
<td>Assignment</td>
<td>Binary</td>
<td>=</td>
<td>Assign result of expression to a variable</td>
<td>(x = 4 + 5)</td>
</tr>
</tbody>
</table>
Not So Familiar Arithmetic Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Unary / Binary</th>
<th>Symbol</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quotient</td>
<td>Binary</td>
<td>DIV</td>
<td>Operands are integers. Result is the integer part of the result</td>
<td>13 DIV 2 = 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15 DIV 3 = 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 DIV 9 = 0</td>
</tr>
<tr>
<td>Remainder</td>
<td>Binary</td>
<td>MOD</td>
<td>The operands are integers and the result is the remainder when the first number is divided by the second.</td>
<td>13 MOD 5 = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15 MOD 3 = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2 MOD 9 = 2</td>
</tr>
</tbody>
</table>

Relational Operators

Relational (comparison) operators are used to compare values to produce a result that can either be True or False, depending on how the data compared relate to each other.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Is Equal to</td>
<td>= (or ==)</td>
</tr>
<tr>
<td>Is NOT Equal to</td>
<td>&lt;&gt; (or !=)</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
</tr>
<tr>
<td>Less than OR equal to</td>
<td>&lt;=</td>
</tr>
<tr>
<td>Greater than OR equal to</td>
<td>&gt;=</td>
</tr>
</tbody>
</table>

- Binary operators
- Operator is written between the two operands
- Operands should be of the same data type
- End result is always a Boolean (true / false)
- Generally used in conditional statements
Boolean Operators

Boolean (logical) operators compare data of *Boolean* type. The three main operands are AND, OR and NOT. Truth tables are used to show all possible combinations of operands and their result. You will most likely have come across these already in Maths or Physics.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a AND b</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>a OR b</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>NOT a</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

Operator Precedence

You have probably used BODMAS in maths to help you remember which operators to evaluate first. You’ll be pleased to know that it’s the same in computing... but with a couple of extras!

1. Brackets  
   Do stuff inside brackets first
2. Order  
   Work out any exponents e.g. $2^3$
3. Unary Operators  
   - (negation), NOT
4. Division / Multiplication  
   *, /, DIV, MOD
5. Addition / Subtraction  
   +, -
6. Relational Operators  
   <, >, <=, >=, = (equality), <>
7. Boolean Operators  
   AND, OR
8. Assignment  
   = (assignment)

If confronted by a bunch of operators with the same precedence, your best bet is to evaluate them from left to right. Better still, they should be placed in brackets to begin with so that there is absolutely no confusion possible! Start with the brackets on the inside!

*e.g.*  
$x = 67 - 12 + 10$  
// is the result 65 or 45?
String Manipulation

When dealing with text there are a number of standard functions that are present in most programming languages to allow us to manipulate strings...

Concatenation

This is simply joining two or more strings together. For example:

String title = "Mr"
String initials = "R"
String surname = "Sole"

String output1 = title + initials + surname
String output2 = title + " " + initials + " " + surname

d debug.print(output1)
d debug.print(output2)

Output 1: MrRSole  Output 2: Mr R Sole

* Notice that it was necessary to include blank strings (" ") in order to create white space between the concatenated variables

Extracting Parts of a String

These functions allow you to strip out specific parts of a string. They only differ in which part of the String you start from, and in which direction you go.

LEFT

LEFT (<String>, <number of characters>)

e.g. newString = LEFT("Rock n Roll Ain’t Noise Pollution", 11)
d debug.print(newString)

OUTPUT: Rock n Roll

RIGHT

RIGHT (<String>, <number of characters>)

e.g. newString = RIGHT("Rock n Roll Ain’t Gonna Die", 15)
d debug.print(newString)

OUTPUT: Ain’t Gonna Die
**MID**

MID (<String>, <start position>,<number of characters>)

e.g. newString = MID("Discombobulated", 7, 3)
   debug.print(newString)

**OUTPUT:**  
   bob

**Locating**

This is simply searching for one string inside of another. In the example above, “bob” is a substring of “discombobulated”. Different programming languages have different functions to do this and may be called things like LOCATE, FIND, INSTR, SUBSTRING or POSITION_OF.

For example:

FIND(<search string>, <main string>)

This function will try to locate the search string within the main string. If it finds it, it will return the starting position of the substring within the main string as an integer. If the search string is not found, it will return 0.

e.g. position = inStr ("bob", "I’m rather discombobulated!")
   debug.print(position)

**OUTPUT:**  
   18

A slight problem occurs when the search string appears more than once in the main string. For example, if, instead of searching for “bob”, we searched for “at”, the function would return the integer value 6, but there is also an occurrence of the same substring starting at position 23 in the main string!

**String Length**

This function is pretty much standard for all programming languages:

LENGTH(<string>)

This function returns an integer value, which is the total number of characters in the string. Spaces also count as characters in most (if not all) cases.

e.g. strLen = length ("I’m rather discombobulated!")
   debug.print(strLen)

**OUTPUT:**  
   27
Have a Go...

ROT13 is an encryption code often used on web pages to hide the punch lines of jokes or movie spoilers. Letters A to M are replaced by the letter in the alphabet which is 13 places ahead of them, and letters N to Z are replaced by the letter which is 13 letters behind. Here is an algorithm for a function that takes a string of uppercase characters A-Z and encrypts it using ROT13. It uses many of the string manipulation functions we have discussed:

```vbnet
FUNCTION Convert (oldString As String): String

    stringLength = length(oldString)
    position = 0
    newString = ""

    For position = 1 To stringLength
        oldCharacter = mid(oldString, position, 1)
        oldCode = ASCII(oldCharacter)

        If (oldCode < ASCII(‘N’)) Then
            newCode = oldCode + 13
        Else
            newCode = oldCode - 13
        End If

        newCharacter = char(newCode)
        newString = newString + newCharacter
    Next position

    Return(newString)

END FUNCTION
```

Your Tasks

1. Identify the string manipulation functions that have been used in this algorithm.

2. Write comments explaining what the computer does at each step of the algorithm.

3. Implement the algorithm in Visual Basic. You will need to adapt the algorithm so that you are using String functions available in VB!
String Comparison

Strings are made up of characters that are represented by their numeric character code... basically each character has its own unique id number. When comparing strings, many languages will compare individual characters by their numeric character code.

Have a look at this example:

```java
string1 = “BLOCK”
string2 = “BLACK”

boolean sameString = strcmp(string1, string2)
```

The 2 strings are compared using the `strcmp` function. If the two strings are the same, the function will return a Boolean value of true, otherwise it will return false.

The `strcmp` function will look at the first character of each string and compare them using their ASCII character code.

They are the same so it moves on to the next character in each string...

They are the same so it moves on to the next character in each string...

These characters are different, so the strings must also be different. The `strcmp` function exits at this point, returning `false` to the routine from which it was called.

This can cause unexpected results...

```java
boolean sameString = strcmp(“bob”, “Bob”)
```

**OUTPUT:**  `false`

ASCII Code for B = 66
ASCII Code for b = 98
Top Tips When Comparing Strings

1. Ensure strings to be compared have a consistent format and same length
2. Convert all letters (A-Z) into upper case before comparison.
3. Add leading zeros or spaces to numeric strings
   (When compared as strings: 10 is less than 2 but greater than 02)
4. Use validation to ensure data is input in correct format to enable comparison
5. Replace accented characters to non-accented characters before comparison

Character Codes

It is not necessary to learn all of the character codes, but it is useful to know that in most character sets, the general order of the most common characters (from lowest to highest code) is as shown in the table below:

<table>
<thead>
<tr>
<th>LOWEST</th>
<th>HIGHEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>The SPACE character &amp; most common punctuation symbols</td>
<td></td>
</tr>
<tr>
<td>Digits 0-9 in order</td>
<td></td>
</tr>
<tr>
<td>Upper case letters A – Z in order</td>
<td></td>
</tr>
<tr>
<td>Lower case letters a – z in order</td>
<td></td>
</tr>
<tr>
<td>Letters with accents such as á</td>
<td></td>
</tr>
</tbody>
</table>

Input & Output Facilities

A program must be able to accept input data and output information or data in a specified format for it to be useful. How data is input or output will depend on:

- The type of user interface (GUI or command line, for example)
- Input & Output devices available

When creating a GUI, the type of data to be input will usually determine the most appropriate means of collecting that data, for example, for a yes/no choice we would use a checkbox and not a drop-down menu or textbox. Choosing the correct means of input will also allow for data validation, to ensure that data entered is of the correct type and size or within a permitted range of values.

When user interface objects are used to input and output data, it is important to be aware of the data types of the data being used. For example, if you are using a textbox to input an integer value, then you should be aware that the program may treat this value as a String rather than an integer. If your program tries to perform arithmetic operations on a String, you might get some rather unexpected results!
Validation & Verification

Data validation checks that the data is sensible before it is processed. Some methods used for data validation are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Range Check</strong></td>
<td>This checks that the data lies within a specified range of values.</td>
</tr>
<tr>
<td></td>
<td>e.g. the month of a person's DOB should lie between 1 and 12</td>
</tr>
<tr>
<td><strong>Presence Check</strong></td>
<td>This checks that important data is actually there and has not been missed out.</td>
</tr>
<tr>
<td></td>
<td>e.g. Customers may be required to have their telephone numbers.</td>
</tr>
<tr>
<td><strong>Type Check</strong></td>
<td>A check that data is of the right type.</td>
</tr>
<tr>
<td></td>
<td>e.g. number, text etc</td>
</tr>
<tr>
<td><strong>Length Check</strong></td>
<td>Checks that fields have the correct number of characters.</td>
</tr>
<tr>
<td></td>
<td>e.g. A bank account number may always be 10 digits long.</td>
</tr>
<tr>
<td><strong>Check Digit</strong></td>
<td>Used for numerical data. An extra digit is added to a number which is calculated from the other digits. The computer checks this calculation when data is entered.</td>
</tr>
<tr>
<td></td>
<td>e.g. The ISBN number on a book. The last digit is a check digit.</td>
</tr>
<tr>
<td><strong>Batch Total</strong></td>
<td>Checks for missing records. Numerical fields may be added together for all records in a batch. The batch total is entered and the computer checks that the total is correct.</td>
</tr>
<tr>
<td></td>
<td>e.g. Add the 'Total Cost' field of a number of transactions together.</td>
</tr>
<tr>
<td><strong>Hash Total</strong></td>
<td>This is just a batch total done on a meaningless field.</td>
</tr>
<tr>
<td></td>
<td>e.g. Add the Telephone Numbers together for a number of Customers.</td>
</tr>
</tbody>
</table>

Verification is used to prevent errors occurring when data is copied from one medium to another. (e.g. paper to disk, disk to disk, memory to disk) Some of the methods used for verification are:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Double keying</strong></td>
<td>Used to check for transcription errors. The data is entered twice (by two different people). The computer will only accept the data for processing if the two versions are identical.</td>
</tr>
<tr>
<td><strong>Visual check</strong></td>
<td>Checking for errors by looking through the data. e.g. Proof-reading a typed document.</td>
</tr>
<tr>
<td><strong>Parity</strong></td>
<td>Used to check for transmission errors over networks or between memory and disk. An extra bit is added to each binary number before it is transmitted. Even parity systems make sure that each number has an even number of '1' bits. After transmission, each binary number is checked to see if it still has an even number of '1' bits.</td>
</tr>
</tbody>
</table>
Example (Even parity)

If 11010111 is transmitted ... (6 '1' bits)
and 11010011 is received ... (5 '1' bits)

then the computer knows a corruption of the data has occurred.