Teaching guide: Programming concepts (Variables and constants)

This resource will help with understanding the use of variables and constants in programming concepts. It supports Section 3.2.2 of our current GCSE Computer Science specification (8520). This guide is designed to address the following learning aims:

- Appreciate the need for variables.
- Be able to declare and assign variables.
- Create trace tables to follow algorithms by hand.

Remembering data

Computer programs process data and this would be impossible if at least some of that data could not be remembered as the program progresses. Variables are created to store data so that the values can be accessed and changed as the program is executed by the computer.

For example, there are values that need to be stored and recalled in this simple number guessing game:

1. The setter thinks of a number between 1 and 100
2. The guesser has a guess at this number
3. Is this guess the same as the setter's number?
    a. YES then the game ends
    b. NO then setter tells the guesser if it is too high or too low
4. Go back to step 2

Even though this is a simple game it would be impossible to play unless the following values were stored:

- The number the setter thinks of in step 1
- The guess (which must be remembered between steps 2 and 3 each time)

The programmer should probably include at least these variables in their program. This section will cover how variables can be created and used.

Variable declaration and assignment

Variables are locations (slots) in memory where data is stored.

These slots have a number (literally an address) that is used by the computer to locate them (a bit like your postal address) but, because of the way computers operate, that number may be different every time the program is run and it also depends very highly on the computer used to run it. As a result these memory addresses are not used directly in the majority of programs.

When you program, every time you need to use a slot of memory to store data you declare a variable with a name instead of a number – you then let the computer decide which memory slot to use. Every time you want to access the value kept in that memory slot, or change it, you use the variable's name. This way you can always be sure which item of data you are referring to. To store the integer 14 in a memory slot called age, the code may be as simple as:

```
age ← 14
```

Which you can read as "the variable age is assigned the value 14".

Some programming languages require that you also say what type the variable must be which, in the previous example, is an integer. So the assignment statement would instead be something like:

```
age: integer ← 14
```

Both of these examples use a style of expressing programs called pseudo-code. This isn't actually a language but is a way to express programs generally that can be easily understood by programmers regardless of the language they are using. Although it does not have the strict rules that programming languages have you should always make your pseudocode clear, consistent and unambiguous.

The most significant difference between this pseudo-code and most programming languages is the ← symbol used for assignment. Languages typically (but not always) use the = symbol for assignment (and use the == symbol to mean 'is equal to').

Throughout these Teaching guides the arrow is used to make it obvious that a value is being assigned to the variable and the = symbol is used to mean 'is equal to' in the same way that it is used in mathematics.

This is how this pseudocode could be implemented in different programming languages:

Programming Language	Implementation
C#	`int age = 14;`
Java	`int age = 14;`
Pascal	`var` `age : integer = 14;`
Python	`age = 14`
VB.net	`Dim age as Integer` `age = 14`

Assigning new values

As the name suggests, the values of variables can change. If you were not able to do this then you would end up using many more memory slots than necessary and the programs would become very difficult to read and, when repeating code an unknown number of times (such as in the guessing game example) it would just not be possible at all. Reusing variables is very simple, you just assign an existing variable a new value using the ← symbol.

In the following example a programmer is defining the width of a paint brush – this is assigned a new value three times as the program executes.

```
# short program to show how variable values can change
brush_width ← 242
draw_square(brush_width)

brush_width ← 187
draw_square(brush_width)


brush_width ← brush_width − 20
draw_square(brush_width)
```

In the pseudo-code used in these resources, just as in the Python, lines beginning with a # symbol are comments. They are not considered part of the program but are very useful to let other humans (and yourself) know what your program is doing. In this example the comment is used to describe what the code below the comment does.

The second line in our program assigns the value 242 to the variable brush_width (you read lines in a program from top to bottom). After the instruction to draw a square the programmer changes the value of brush_width to 187. This is still the same variable but now it has a new value and from that line of code and until it is given another new value the variable will evaluate to 187.

The final assignment looks odd at first sight – it assigns itself its own value minus 20. There is nothing particularly strange about this line though if you evaluate the right hand expression first. You need to know the value of brush_width which is 187 at this particular point in the program, and subtract 20 from it to get 167. Now you know the value of the expression and you can assign this back to brush_width.

It is worth noting two things here about variable names in general:

- Give variables a name, called an identifier, which describes the value it holds. brush_width is a much better name than xyz even though both would normally be permissible in a language. This is an example of self-documenting code which is very beneficial to anyone who has to read your code, including yourself.
- The variable name contains no spaces. Different languages have different rules about naming identifiers but good rules of thumb are:
    - always start with an alphabetic character
    - leave spaces out
    - avoid punctuation (except the _ symbol)

  There are different conventions in various languages for leaving out spaces. Python commonly uses an underscore to signify a space such as brush_width whereas the Java convention is to capitalise the first letter of every new word such as brushWidth. The documentation for your language will help you out here.

Expressions involving variables

It was shown in the previous example that expressions can involve variables, possibly even the same variable that the value of the expression is being assigned, as in

```
brush_width ← brush_width - 20
```

Any expression can be used on the right hand side of an assignment statement as long as it evaluates to a value that has the correct type for the variable.

You can use as many variables and operators as you need just as long as the purpose of the expression can be reasonably worked out by a human reader. If you find that you are writing an expression which looks frighteningly complicated you may want to separate it into separate sub-expressions and assign each of these values to separate variables. For instance, if we have a quadratic equation of the form

$$ax^2 + bx + c = 0$$

and we know the values of a, b and c we can calculate the possible values of x. One of the solutions for this quadratic equation is given by this formula:

$$\frac{-\mathrm{b} + \sqrt{\mathrm{b}^2 - 4\mathrm{ac}}}{2\mathrm{a}}$$

If you were to write this out as one assignment statement it might look like this

```
solution ← (-b + SQRT(b^2 - 4*a*c)) / (2*a)
```

The ^ symbol is used to indicate 'to the power of' and SQRT is a subroutine that calculates the positive square root.

Whereas writing exactly the same code split over two lines and using an intermediate variable called discriminant is easier to read and understand:

```
discriminant ← b^2 - 4*a*c
solution ← (-b + SQRT(discriminant)) / (2*a)
```

The following program uses variables in the expressions and also reassigns values to variables. It calculates the area of three rectangles where one of the sides increases by 1 every time.

```
length ← 10
width ← 5
area ← length * width
OUTPUT area
width ← width + 1
area ← length * width
OUTPUT area
width ← width + 1
area ← length * width
OUTPUT area
```

This program uses three variables to store the data needed as the program executes. If you take a look at the expressions on the right hand side of every assignment statement in this program, you can work out their value and then assign this to the variable. The OUTPUT statement means just that – output the value of the variable presumably, but not necessarily, to a screen. (The Teaching guide - Programming concepts (Iteration) covers a way to write programs that repeat statements using a loop.)

Trace tables

It is often difficult to keep track of the value of variables as a program executes. You can use technological tools to help and these will be explored in another resource. You can also use a hand-written technique called a trace table to keep track of how variables change in simple programs (as you become more practiced at programming you will find yourself capable of calculating quite complex trace tables mentally).

The example above uses three variables called `length`, `width` and `area` and these become the column headings in the trace table:

length	width	area

You then work down the program from top to bottom working out the values of the variables and adding their values to the table. So after the first three lines of our program, the table will look like this:

length	width	area
10	5	50

The next instruction in our program is `OUPUT area` so you know by looking at our trace table that this will output the value `50`. The fifth line is `width ← width + 1` which you work out by evaluating the expression on the right of the arrow first which in turn requires finding out what the current value of width is which you can do by looking at the trace table:

`width + 1` simplifies to `5 + 1` which evaluates to `6`

The value `6` is then assigned to the variable `width` and you show this in the trace table by writing the new value underneath the previous one. The trace table now looks like this:

length	width	area
10	5	50
	6	

If you work through this program to the end then the trace table will have the following values in it:

length	width	area
10	5	50
	6	60
	7	70

The following algorithm calculates the first five Fibonacci numbers, created by starting with two 1s and then adding together the previous two values to generate the next number:

```
a ← 1
OUTPUT a
b ← 1
OUTPUT b
temp ← a
a ← b
b ← temp + a
OUTPUT b
temp ← a
a ← b
b ← temp + a
OUTPUT b
temp ← a
a ← b
b ← temp + a
OUTPUT b
```

The completed trace table for this algorithm will look like this:

a	**b**	**temp**
1	1	1
1	2	1
2	3	2
3	5	

The golden rule with variable assignment is to evaluate the right hand side first and then assign that value to the variable on the left of the arrow as the last thing you do.

This means:

- you can only have a variable name to the left of the arrow
- the code to the right of the arrow must either be a value or an expression that evaluates to a value

Writing variable assignment the wrong way around is a common mistake when you first start to program. For instance, if the programmer wants to evaluate `b + 2` and assign this value to a they may write:

```
b + 2 ← a
```

Whereas they meant to write:

```
a ← b + 2
```

Another common mistake is to use a variable in an expression before you declared it or assigned it a value. For example, the following program will almost certainly result in an error in a programming language because the first line needs the value of `a` to evaluate the expression `a + 1` and the variable `a` is only declared and assigned a value on the following line.

```
b ← a + 1
a ← 2
```

Variables vs constants

Variables are the names given to slots in memory that help store and refer back to data as our programs execute. Constants do exactly the same thing except, as the name suggests, their value does not change.

An advantage of using constants as well as variables may be a small increase in speed when your program runs on a computer but it also has the advantage that it makes your programs clearer and easier to understand. It can also make your program far easier to maintain in the future. If you declare a constant you are telling the people who read your program that this value never changes within the program and so whenever it is accessed it will be the same.

The mathematical value π (pi) is a constant value and if you were to use π in your programs you wouldn't want to have to type out 3.14159 (or more decimal places) every time you needed it, nor would you ever want to give it any other value. These Teaching guides will use the following pseudo-code to declare a constant:

```
constant PI ← 3.14159
```

This constant can be used in an expression in exactly the same way as a variable. For example, to work out the circumference of a circle with a radius `10`:

```
r ← 10
circumference ← 2*PI*r
```

Your programming language may support declaring constants directly. Other languages, such as Python, do not have a keyword such as `constant` or `final` but use the convention that if you declare a variable name using all upper case characters then it is to be used as a constant. These resources will also use upper case letters when defining constants.