# 4. On the Move (Variables, Assignment and Declarations)

*"But it does move"*

Galileo (1632).

**Assignment and Variables**

Storage space is very important in programming. Programs process information and that information needs to be stored somewhere. Processing it involves moving it from place to place. Without the ability to move data around, computers would not be of much use. We will examine here the basic features of instructions for moving things around.

Consider the game of chess. Chess players normally record their moves as they make them so they can play the games through again and analyse them. Such written version of games between grandmasters often appear in books and newspaper columns. Also books on how to play often include fragments of games "openings" for example such as one called "Ruy Lopez" and "Queen's Gambit". What is such a written version of a chess game? It is an algorithm of how to get from the initial position to the final position, following the rules of the game (ie how the pieces can move). If you were just told the final position of a game of chess, it would not (usually) tell you much. The algorithm of how that position arose can be fascinating for chess players. So what does such a chess game algorithm actually involve? It is a series of moves of pieces from one position to another – a series of **assignments** in programming terms. An assignment is just an instruction to move data from one place to another. Each square of the chessboard is a storage space that can hold a single piece at any time. The pieces are playing the same role as data values in a program – the things that are stored. The squares are **variables**. A variable in a computer program is just a place where things (data) can be stored.

Each square on the chessboard is given a name. In computer parlance a variable name is referred to as an **identifier**. The most common way in chess is to name each with a letter giving the column it is in and a number giving the row it is in. Thus *b1* is the name used to refer to the square that the white Knight starts in, in row 2 column 1. All identifiers in chess are thus a letter from a to h followed by a number from 1 to 8. Programming languages similarly have rules for what makes a valid identifiers so you can tell when something is and is not an identifier. For example, the rule might be that an identifier is a sequence of digits and letters of the alphabet but cannot start with a digit.

If we wish to specify a move of the game in chess we must specify two things – the piece to be moved and the place it is to be moved to. There are many ways used to do this and different chess books use different ways. The most common involves making moves like R – e4 to mean move the Rook (that is perhaps at e1) to square e4. However this does not always work – there are two Rooks of each colour and in some board positions either could move to square e4. For example perhaps the other Rook was at square e5. It could also be the one we meant. We need a system that uniquely tells us which piece we want to move. An easy way to do this is not to give the name of the piece (which is not unique) but to give the name of the square. This is unique

since we have given a different name to each square and only one piece is on a square at any time). Thus the easiest way to specify a chess move is just to give the start square and the destination square. Thus in the above situation we would say e1-e4. Had we meant to move the other Rook we would have written e5-e4. Notice that we are using a special symbol "-" to mean move. Notice also that we do not mean move e5 itself (the square). We mean move the piece that is currently in that square. At different times in a game"e5" could refer to completely different pieces – at one point a pawn, later a Bishop, then another pawn. "e5" is an identifier of the variable. A whole game would be specified by giving a whole series of such moves. In essence we have created a language for writing chess algorithms in. The language used in chess books is usually a little more complicated – it includes instructions that mean "The player Resigns", use a "x" to mean move to a square and capture the piece that is there and also have a way of indicating whether white or black. Why has a special language been created? Why do the books not just use English? It is essentially for the reasons we discussed for computer languages – we need to remove **ambiguity** as just discussed. Such a language is also very concise —and once learnt is very easy to understand (though looks like meaningless lists of numbers and letters to someone who does not play chess).

A program is an algorithm for moving data (usually numbers) around in a computer rather than pieces around a chessboard. However the same principles apply. How do we specify moving something from one place to another? We need to specify what is to move and where it is to move to. Identifiers (i.e. the names of storage spaces) are used to refer to the thing being moved and also the place it is to move to. We need some symbol to indicate that we are talking about moving something from one place to another (like the "-" we used in the chess algorithms). Otherwise we would not know whether the instruction was to move the data or (for example) do a calculation on it. Different languages use different symbols (like "=" or ":=") to mean "assign". For example, to write an instruction to move a chess piece using programming language notation you might write something like.

```
e4 := e1
```

or

```
e4 = e1
```

rather than

```
e1-e4
```

to mean square e4 gets the piece currently at e1.

This use of different notations to mean the same thing in different languages is just like in human languages where different languages have different words for the same thing – French for example uses the symbols "bouger" to mean what in English is referred to using the symbol "move". Once you understand the concept however switching between languages is much easier.

Consider another example from the world of games. When I was at school we had a craze for playing the game of *Diplomacy*. It is a war game set on a board of Europe. Unlike other war games the fun is not just in moving pieces about but in, negotiation making secret deals with other players and double crossing (like real Diplomacy). Here we are interested in the actual moves. The board is divided into named areas corresponding to places like London or Bulgaria and seas like the North Sea or the Black Sea. On each round of the game, players move their pieces from one location to

an adjacent location. As with chess, each place can only hold one piece so there are rules about how you decide who wins when two pieces are supposed to be in the same square. The moves are intended to happen simultaneously, but obviously, in reality they are done one at a time with a period at the end of each turn for deciding the conflicts. To ensure no one has a chance to change their moves on seeing someone else move, each player writes their orders down. For example, the person playing England might write:
Lon-North S
Den-Kie
As with chess these are a series of assignments. Now the **variables** are the named areas. Their names like "London" shortened to "Lon" are identifiers. Again "-" means "move". The problem is similar to that for chess and the solution is basically the same. The movements are specified by giving the locations moved to and from.

There is one difference in the way assignment is usually written in programming languages that often confuses beginners. Many programming languages give the two locations apparently the wrong way round. e1-e4 in our chess language above was used to mean "Move the piece from e1 to square e4". In programming languages it is often the other way round: i.e. if you meant "Move the piece from e1 to square e4" you would write e4-e1. This looks bizarre but it makes sense if you read it as: "Square e4 gets the piece from square 1".

One way to think of variables is just that they are storage boxes, each big enough to hold just one thing. Assignment is just an instruction to move something from one box to another. However, as each box can only hold one thing, if something else is already there, that thing is first thrown away unretrievably to make room for the new. Assignment in programming languages is a little bit more subtle than that though and this is a way it differs from moves in chess or Diplomacy. The thing being moved is actually copied first and it is the copy that is put in the new box. What this means is that not only is the thing copied placed in the new box, but an identical thing is also left in the original. This is as though the rules of chess were such that if you moved a pawn forward a square, the new position would have a pawn both in the original and new positions.

The memory buttons on calculators are providing an assignment operation of this copy form. They are used to store temporary results in the middle of calculations. This is exactly what a variable in a computer program is for: storing something for later manipulation. The memory is thus a variable: its identifier on most calculators is M. However, when you use the memory button to move a value from the memory to the display, the value stays in the memory too, so can be used again. Some calculators have multiple memories: ie several variables with names like M1, M2 and so on. When you press the memory button you do an assignment to the memory. You are basically saying put the number currently on the display into the memory. Again you are specifying (at least implicitly) the place to get the value from and the place to move it to. As the display is the only place that is allowed as the source of the assignment on calculators you do not have to explicitly specify it. If there are multiple memories you will need to indicate which is to be used – usually by pressing a different memory button.

Many phones have a way of storing commonly used numbers so that they can be dialled quickly with only one or two button presses. For example I often phone my parents and my wife's work. It is useful to have those numbers stored and so be able to phone them quickly. Each of the quick dial locations is a variable in computer science terms – it is a place where numbers can be stored until they are needed. The label on the button is its identifier. The identifiers might just be numbers or sometimes they are special buttons with identifiers like M3. How do I store a number in a quick dial button? The instruction book gives me an algorithm.

1. Press the quick dial button (Q)
2. Press the numbered button where the phone number will be stored
3. Dial in the telephone number to be stored.
4. Press the "enter" (E) button.

This is giving an algorithm for typing in an assignment command. Suppose I wanted to store in quick dial button 3 the number 020-1234567. Following the algorithm I would press the following keys:

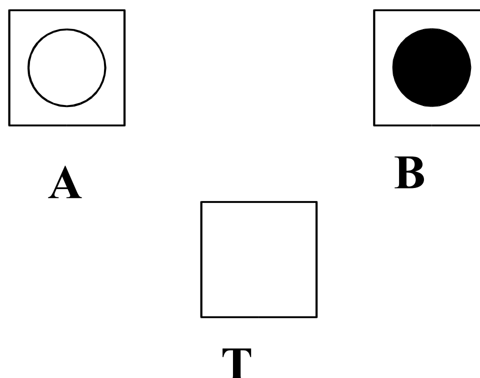
Q 3 0201234567 E

This is just another notation for assignment! Another way of writing this might be:

3 ← 0201234567.

That is the same instruction – just written in a different language – perhaps for a phone with a key marked ← to mean store in a memory key and a key "." to mean "enter". The first Q can be thought of as being a symbol to mean we are doing an assignment. The letter is the place to assign to and the following digits are the value being assigned to that key (i.e. being stored there). The final E can be thought of as punctuation –like a full stop in a sentence. It indicates that we have finished the command. Computer languages have similar punctuation characters. In some languages a full stop is used to mean the end of the current command, in others a semi-colon is used.

Swapping the positions of two things is a common operation that when broken down into its basic steps is a sequence of assignments. We will use it later when we look at sorting things. Try the following simple puzzle (using coins) that requires you to devise a swap algorithm. (One answer is at the end of the chapter).




A board consists of three squares named A, B and T, and two pieces, one black and one white. The black piece is placed on square A, and the white piece is placed on square B. Pieces can be moved from any square to any other. However, if a piece is moved to a square where the other piece sits, then that piece is removed permanently

from the board. Work out a sequence of moves (and write them out in English) that lead to the positions of the two pieces being swapped. Now write down your sequence of moves using the notation $x \leftarrow y$ to mean "square $x$ gets the piece currently on square $y$".
Here is one way to do it.

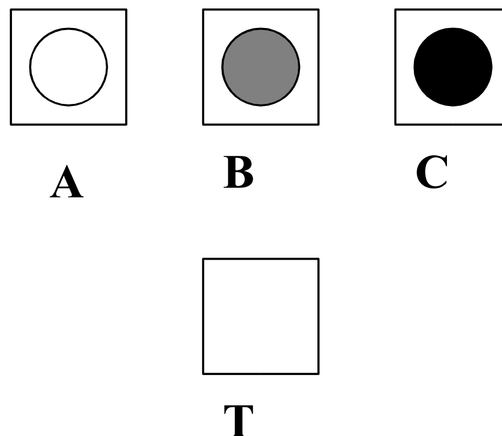
1. Move the piece at square A to square T
2. Move the piece at square B to square A
3. Move the piece at square T to square B

Notice that this takes three moves. You cannot do it in two as you would remove a piece if you for example tried to move the piece on square A straight to square B. We will look at why that goes wrong later but check it for yourself now. Writing the above moves out in our invented notation.

1. $T \leftarrow A$
2. $A \leftarrow B$
3. $B \leftarrow T$

Remember to check you wrote the assignments the right way round to mean for example "square T gets the piece on square A"

Rotating the positions of things needs a similar algorithm to swapping. Try the following similar puzzle.




A board consists of four squares named A, B, C and T, and three pieces, one black, one grey and one white. The black piece is placed on square A, the grey piece on square B and the white piece is placed on square C. Pieces can be moved from any square to any other. However, if a piece is moved to a square where the other piece sits, then that piece is removed permanently from the board. Work out a sequence of moves (in English) that lead to the positions of the three pieces being rotated one square to the right (so that the black piece is on square A, the white piece is on square B and the grey piece is on square C. Now write down your sequence of moves using the notation $x \leftarrow y$ to mean "square $x$ gets the piece currently on square $y$" as before.

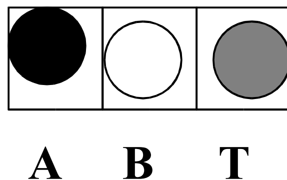
Here is one algorithm to do this rotation in English

1. Move the piece at square A to square T
2. Move the piece at square B to square A
3. Move the piece at square C to square B
4. Move the piece at square T to square C

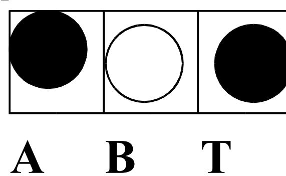
and in the special notation:

1. T ← A
2. A ← B
3. B ← C
4. C ← T

In most of the above we have talked of *moving* things from one place to another. That is not quite what computers do. Rather than move things they just copy them. When you move something it is no longer in the place where it was – so the above swap puzzle, moving a piece means there is no longer a piece in the place it came from. Computer assignment leaves a copy of the original in the original place. Variables can never be “empty”, the may contain 0 (but that is still a thing) or you may not know what is in them (but something will still be there!). To see how a computer actually swaps things lets look at a variation on the swap puzzle. There are now three squares with identifiers a, b and t as before. In this puzzle they start with a piece in each, one black, one white and one grey.




The aim is again to swap the pieces in A and B, so that now A is white and B is black. Now you cannot just move pieces. Instead you can just copy them. You have a supply of other black, white and grey pieces. Each “move” now involves doing the following: pick a square, note the piece in that square, get another piece of the same colour and finally place that new piece in some new square discarding whatever was already there. So for example, now the instruction T ← A would mean look at what is in square A (its black at the moment), get another black piece and put it square T throwing away the grey piece that is there. The new board position is:

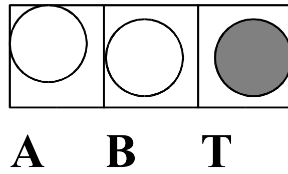



Write down a sequence of moves that would lead to A being white and B black – so they are swapped over (we do not care what T ends up with). In particular the following algorithm does not work, why not (try it and see).

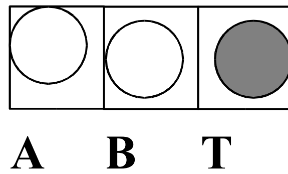
1. A ← B (A gets a copy of B)
2. B ← A (B gets a copy of A)

Each “move” in this puzzle is just like an assignment in a simple programming language, the squares are storage spaces – variables and the assignment copies whatever is in one variable (square) to some other square.

Why did the above 2 step algorithm work? It is because the first step makes a copy of B and puts it in A:




There are now no black spaces on the board. When you do the second step you are now making a copy of A and putting it in B, but A is white so you just throw away the old white piece and put in a different but identical looking white piece. The board looks exactly the same after the second step:




As there are no black pieces after the first move there are no blacks to make a copy of, so there can not do any move that will get a black after.

What is the correct algorithm? It is exactly the same one as for the original puzzle. We must make a copy of one of the pieces in T first so that we do not lose it when we copy something different into its position.

1. T ← A (T gets a copy of A)
2. A ← B (A gets a copy of B)
3. B ← T (B gets a copy of T)

Let us return to our design of the Imp computer – a computer that works using Imps to do the computation. Storage space in our Imp computer consists of boxes. There are two kinds of Imps: memory Imps and instruction Imps. The instruction Imps are the middle managers giving orders. The memory Imps are the workers – who pass data around. Each memory Imp owns a box, capable of storing an object or perhaps a piece of information. The boxes act as the memory of the Imp computer (in our world Imps themselves are very forgetful so for the computer to be reliable it is important that their own memories are not relied on). Each Imp that is required to remember something is referred to by name in instructions concerning that Imp. This is the way individual Imps are identified: it is their identifier.

The instruction Imps do not have boxes, but instead have pieces of paper containing the instruction they must follow if ever it is their turn. We will return to how we determine when it is a particular elves turn later. Here let us look at the individual instructions. An Assignment elf is one holding an instruction to make a copy of a piece of data held in the box of one Memory Imp and put a copy of it in some other memory Imp's box. An example instruction might be "Joe gets a copy of whatever is in Fred's box". The Assignment elf with this instruction would go to Fred, copy down onto a new piece of paper whatever was in the box and put it into Joe's box, hiding forever what had been in that box. The two elves Joe and Fred here are acting as variables. Their identifiers are the names "Joe" and "Fred".

Let us return to the puzzle above of swapping the positions of two counters but look at it in terms of an Imp computer. Our aim now is to swap the things that two Imps are looking after (storing in their boxes). Suppose the two Imps concerned are called Alf and Bridgit. Alf has the number 6 on a piece of paper in his box. Bridgit has the number 42 in her box. We need Alf to pass his number to Bridgit and Bridgit to pass her number to Alf. However they can not do it simultaneously – one has to go first but that would leave one Imp temporarily holding two things. Imps only have one box and refuse (due to Union rules and their poor memories) to remember more than one thing at once. We therefore need another Imp to temporarily store one of the pieces of information. We there for rope in another Imp, Trevor, to help. What is the program to do this on the Imp computer:

1. Trevor gets Alf's value.
2. Alf gets Bridgit's value.
3. Bridgit gets Trevor's value.

If we agree that an arrow means the Imp named on the right of the arrow passes a copy of their value to the Imp on the left who puts the number in the box, discarding whatever was there before then we can use the following shorthand.

1. Trevor ← Alf
2. Alf ← Bridgit
3. Bridgit ← Trevor

This is the same algorithm as we saw earlier (just with different identifiers for the variables). Notice that in each of these instructions, the two Imps named have to do something different.
In the instruction:

Trevor ← Alf

Alf must make a copy of his value to pass it on (All Imps come with a supply of rice paper and a pen to do this). Trevor on the other hand is not being told to do any writing, just to take the piece of paper given and put it in his box, throwing away whatever was there. (Imps actually eat the paper rather than throw it away as Imps always use rice paper to write on – once eaten the value can never be recovered).

If on the other hand the instruction had been

Alf ← Trevor

then Trevor would be doing the writing and Alf would eat his old value.

**Initialisation**

The point of variables is to store things that can then be moved around or manipulated as we discussed. Before you start doing this you have to **initialise** the variables. By this we just mean to set starting values. In chess, we initialise the squares with pawns on the second row, and Rook, Knight, Bishop etc on consecutive pieces of the first row. You cannot start a new game until you have initialised the board. At any one time there are always some squares that are blank. Those squares have not been initialised with a value. Obviously you cannot give a move that moves a piece from an empty square so you cannot in the first move refer to any of those squares as the

source of the move. A first move of e3-e4 is nonsensical as there is nothing in square e3 to move at the start. You could of course move a piece into e3 first and then move it out again on a later move.

It is possible to buy chess puzzle books. They give chess positions that are 2 moves away from checkmate. The puzzle is to work out what the moves are for that position. Each puzzle uses a different initialisation of the board. Somehow the book has to tell you which pieces too put where. This is usually done with a picture of the board, but this is giving the same two bits of information a square (a variable) and a piece (value) to go in that square at the start.

A chessboard can be used to play draughts. It is just initialised in a different way (even though the board itself is identical). Different pieces are placed on different squares to start the game.

My phone has a feature that allows me to store in special "quick dial" buttons numbers I use frequently like those of my parents. The idea is that you can phone common numbers with the press of just two buttons. What would have happened if I pressed one of these buttons when I first bought the phone before I had stored any numbers? There are several possibilities. One is that the buttons could do nothing, another is that they dial a random number, another is that an error message might be spoken in the handset. None of these options would result in a sensible number being dialled – pressing the button before they have been set is not useful. It most probably would be a mistake on my part. Before I use those buttons I must initialise them by doing an "assignment" as we saw earlier.

The Diplomacy board also has to be initialised. The instruction book does it saying what piece goes where. For example England is initialised with:
F London
F Edinburgh
A Liverpool
i.e., Fleets are placed at London and Edinburgh and an Army at Liverpool – my apologies to the Scots (and Welsh) about the above – the Diplomacy Board really does have Scotland (and Wales) as regions of "England". These instructions are again just assignments, but this time rather than moving a value from one place to another we are moving a value not currently at another location to a location on the board. We use a notation that allows us to specify the place moved from and the value (i.e., piece) to be moved there. A similar facility is provided in programming languages –though the notation used is normally the same as for assignment. You thus might write something like

```
London = F
```

rather than F London to mean "London gets a Fleet".

The above example is different to earlier examples in that rather than just moving existing things from one storage location to another, we are putting a new piece or value into a storage space. We thus have a notation not only for referring to the storage locations (the variables) but we also need a notation for referring to values. In Diplomacy there are only two kinds of piece – Fleets and Armies so we use F to mean a value of the former and A to mean a value of the latter in our notation. Suppose we wished to give an algorithm for initialising a chessboard. Normally, this would be

done using a picture. To give it as a series of instructions (an algorithm) we need a distinct (i.e. unambiguous) way of referring to each piece. What are the values of a chessboard? – Pawns, Rooks, Knights, Bishops, Kings and Queens. The normal convention used by chess players is for the pieces to be named by a single letter – P, R, N, B, K, Q. However that on its own is not enough. We also need to indicate whether the piece is black or white otherwise our notation would be ambiguous. We might therefore add an extra letter (W or B) to each to indicate the colour. The algorithm might be then written something like:

a1 = WR
a2 = WB
etc

We have thus given ourselves a notation for referring to the chess pieces, the values. This raises an important issue with respect to readability. The above notation is only readable once you have used it enough to be familiar with the code. It would be much more readable if we had actually used longer identifiers.

a1 = White_Rook
a2 = White_Bishop
etc

That way anyone who was familiar with chess pieces would immediately understand the notation. It is similarly important with programs that names chosen both for values and variables are easily understandable without explanation. This is known as good style. A program is not wrong if it uses obscure names for things in the sense that it will (possibly) do the correct thing. However it is bad if it is hard for other people to understand.

When we think in terms of chessboards, moving a piece from an empty square seems a silly idea. However, a common fault in programs is to do precisely that – forget to initialise a variable and then give an instruction to move a value from it when one does not exist. Different programming languages vary on what happens when you try this but in all cases it would mean the program was not as intended.

When you switch on a calculator, it automatically initialises the display to 0. Some programming languages automatically initialise variables to zero. More commonly they start with some random and unpredictable value unless the program explicitly contains an instruction to put something else there.

In terms of our Imp computer, initialisation is just a question of ensuring all the memory Imps being used have been given something to store in their box at the point when they are told they are needed for a particular program run. If they are not then their box will have whatever number happened to be there from the last time the box was used (which could be anything). If this Imps value was used before anything new was put in the box, then the Imp computer would have an unpredictable result as it would depend on the old value being used.

**Expressions**

So far we have seen assignments where a literal value is moved to a storage location. We have also seen how rather than referring to a value we can refer to a storage location to indicate which thing we wish to move. A third thing that we can do is to move the value of a calculation to a variable. Instructions to do calculations are referred to as **expressions**. Often the calculations we wish computers to perform are

on numbers – arithmetical calculations. For example, suppose I am told the time of my train is at 17 hundred hours. The first thing I do is mentally convert it to a 12 hour clock as that is what I am most familiar with. What I do is a calculation – I subtract 12 from the 24-hour time given. How might I write an instruction to do such a calculation for someone else to follow – I just use the notation from school

(17 – 12)

giving the answer 5 if followed. At school we learnt notation for writing more complicated calculations such as how to work out the average of several numbers.

How would our Imp computer deal with expressions? We would require an Imp who was expert in each of the arithmetical operations – an addition Imp, a subtraction Imp and so on. Whenever a calculation (say addition) needed to be done the memory Imps holding the values concerned would pass their values to the appropriate Imp who would do the calculation and pass the result to wherever the result was to be stored. For example Arthur might be the addition Imp, doing any addition needed, Sol might be the subtraction Imp.

For example, suppose the instruction to be followed was

Joe gets the answer of Arthur adding Bridget and Alf's numbers.

perhaps written in our short hand notation as:

Joe ← Bridget + Alf

Arthur here is the addition Imp referred to using the symbol +. This means get a copy of the value held by Bridget in her box and a copy of the value held by Alf, pass them to Arthur who will add them (the only thing he is useful for) and pass his answer on to Joe to store in his box.

The Imps would be able to cope with more complicated calculations by working together. For example if the instruction were:

Joe ← (Bridget + Alf) - Conor

This would mean that first Arthur would be passed the values from Bridget and Alf as before. However, rather than passing the result to Joe he would pass it to Sol (the subtraction expert) who would also get the other value from Conor (an Irish Imp) and do the subtraction, before passing the final result to Joe.

If we needed our Imp computer to keep a count (perhaps of the number of times a dice had been rolled), then we would need an instruction to add 1 to the count. Suppose Zack was charged with keeping count then he would do this by keeping the current count in his box. Every time the dice was rolled, Zack would need to look in his box, note the number, add one to it and store the new answer back in his box. However, adding one is an addition calculation and Arthur is the only one who can do that (there are poor levels of numeracy amongst Imps generally though Union rules also mean only registered addition Imps can do addition). The instruction we write to tell Zack how to get the addition done is thus:

Zack ← Zack + 1

What does this tell Zack to do? He must pass a copy of the value in his box to Arthur. Arthur will add 1 to it and pass the result back to Zack, who will then put it back in his box. The result: Zack has added 1 to his count without knowing how to add 1 himself!

Remember at the start we talked about how in the Second World War, the British broke the German Enigma codes? This allowed them to read all their encrypted communications. This was done by having teams of people working in separate Huts. Each did a specialist part of the calculation and passed on the results to someone else in another Hut who could do the next part. The result was that together they cracked the codes, without understanding anything other than their own part. Only a few Mathematicians such as Alan Turing understood the whole process. In essence those people were just acting like an Imp computer with Turing their programmer.

**Boolean Expressions**

A calculator is used specifically to do calculations on numbers. The answers it produces are also numbers. The expressions use operators such as + and -. However expressions do not have to only be about numbers. We also frequently use expressions that give **boolean** answers. A boolean is just one of the values **true** or **false**. Quizes and tests often use "true or false" questions. Here are some examples that might be used on a Radio quiz:

True or False:

1. Freddy Mercury is alive.
2. Britney Spears is alive.
3. David Beckham is dead.
4. Elvis Presley is dead.
5. Clint Eastwood is alive.

At the time of writing the answers to these questions are

1. False    2. True    3. False    4. True    5. True.

Here is another set that might appear on a school test.

True or False:

1. Glasgow is the capital of Scotland.
2. Sidney is the capital of Australia.
3. Diamonds are made of crushed Carbon.
4. Sheffield is in South Yorkshire.
5. Rubber comes from a plant.

The answers to these questions are:

1. False    2. False    3. True    4. True    5. True.

The answers are not numbers as for questions calculators are used to answer but either true or false. They are thus questions with **boolean** answers. As we will see later booleans are very important in computer programming, as they allow computers to make decisions. Notice that to get a question with a true or false answer it has to be phrased in a particular way. You cannot ask "Is Freddy Mercury Dead or Alive?" The answer to that question is either "Dead" or "Alive" not "true" or "false". Programming languages similarly require questions to be phrased in a particular way to ensure the answers are either true or false.

Expressions are made up by applying operators to values: 6+2 for example applies the arithmetic operator + to the numbers 6 and 2. We can and do build boolean expressions in a similar way. The operators in boolean expressions are things like **and** and **or**. We can use these words to convert true/false questions into more complicated true/false questions. For example the following question uses the logical connective **and**:

True or False:

Clint Eastwood was in the film The Good, the Bad and the Ugly **and** Harrison Ford was in the film Mad Max.

This is made up of two separate questions: one about Clint Eastwood and the other about Harrison Ford. Depending on the answer to the parts you may need to know the answer to both to be able to answer the question. Only if both separate parts are true can the whole statement be true. If either were false the whole thing would be false. In fact it is false because it is not true that Harrison Ford was in Mad Max.

A similar question using the connective **or** is a different question all together:

True or False:

Clint Eastwood was in the film The Good, the Bad and the Ugly **or** Harrison Ford was in the film Mad Max.

This is true because Clint Eastwood is in The Good, The Bad and the Ugly even though Harrison Ford was not in Mad Max.

Buses sometimes have signs using boolean connectives. They give tests that must be applied by the driver to determine if the bus is full. For example, the sign may say:

35 sitting **and** 10 standing

What this means is the driver must ask the question of his or herself before letting a passenger on if the bus appears to be close to full: Are there already 35 people sitting and 10 people standing. If the answer is yes (so the original statement is true) then the driver must not allow anyone on the bus. Notice that boolean tests are special kinds of actions that gather information rather than actually doing anything. The bus driver evaluates the test so that they can make a decision about which action to take next. If the bus is full then the action they must take is to refuse to allow another passenger on. If it is not yet full their next action can be to take the fare of the next person. We will be looking in more detail at how the results of such tests determine our actions in subsequent chapters.

The film *Anna and the King*, which is about an English schoolteacher who arrives in Siam (now Thailand) to be the teacher of the King of Siam's children. It had been agreed before she arrived that she would be given a house to live in outside the palace. However, despite being there for months she is still in the palace. One day she is taken by surprise when the King finally gives her a house of her own. Taken aback she asks

"*Is this because of our agreement **or** are you simply trying to get rid of me?*"

Wishing to be evasive, the King answers

"Yes".

This highlights yet another ambiguity in English as well as being an example of boolean logic. Anna of course was asking which of the two options was the true one, expecting to be told "Its because of our agreement". However, the King has quite legitimately answered as though he was asked a boolean true/false question. If it is true that he is doing it because of the agreement or it is true that he wants rid of her, (or both) then the whole statement is true. It *is* either because of the agreement **or** because he is trying to get rid of her.

In a similar way if I am asked "Are you going to do the washing up **or** not" I can legitimately (and irritatingly) answer "Yes" – I am going to do one or the other (probably the latter).

We can build arbitrarily large questions using *and*, *or* and some simple basic true/false questions. For example, the rule of when you have won a game of chess can be thought of like this. You win if the other person resigns **or** their King is in check and all positions the King could move to also leave it in check.
On each move you ask the true/false question:

> True or False:
> the other person has resigned **or**
> their King is in check **and**
> all positions the King could move to also leave it in check.

Each part of this is a true/false question in its own right.

Returning to the bus example, more modern buses have spaces that can take wheelchairs. The presence of a wheelchair perhaps takes up two spaces. The sign about the test for when the bus is full will need to take this into account.

> 35 sitting **and** 10 standing **or**
> 1 wheelchair **and** 34 sitting **and** 9 standing

Menus in restaurants often have boolean like expressions in them to give descriptions of the meals:

> Fish and Chips.

Treat this as a true/false statement:

> The meal has fish **and** the meal has chips.

What they are saying is that this true false statement will be true for the meal you get.
Sometimes they get into trouble due to the ambiguity of language:

> Fish and Chips or Baked Potato and Salad.

If you were served just Baked Potato and Salad would you have cause for complaint?
That depends on where the brackets are supposed to be

> (Fish and Chips) or (Baked Potato and Salad)

is a different statement to

> Fish and (Chips or Baked Potato) and Salad

Since there are no brackets, unless an order of precedence has been agreed the statement could be interpreted as meaning a meal without fish. I have had arguments with waitresses over menu entries similar to that. The waitresses concerned have just assumed I was mad (probably a true statement anyway). Just as the position of brackets can change the result of an arithmetic calculation, they can also change the result of a true/false calculation.

The sign on the bus was written with the intention of the brackets being as follows:

> (35 sitting **and** 10 standing) **or**
> (34 sitting **and** 9 standing **and** 1 wheelchair)

With the brackets in different places it would

> ((35 sitting **and** 10 standing) **or** (34 sitting **and** 9 standing))
> **and** 1 wheelchair

then it would suggest that there was room for a wheelchair even if there were 35 people sitting and 10 standing.

Boolean expressions can have variables in them just as arithmetic ones can. If we think of a variable as something whose value can change with time then "The Queen of Great Britain" is like a variable. Its value (ie the actual person concerned) is

different at different times. (Coronations are then just assignments putting a new person to the post!) Similarly "the Prime Minister" can be treated as a variable.

*The Queen of Great Britain* is Elizabeth **and** *the Prime Minister* is Tony Blair is a true/false statement whose value changes over time. In the year 2000 it is true. In the year 1980 it was false as then the variable "the Prime Minister" had the value Margaret Thatcher. The answer to a true/false statement changes if the variables within it change their value. Similarly in our bus sign stating "35 sitting **and** 10 standing", "sitting" and "standing" can be thought of as variables holding numbers that vary as people get on and off the bus. As a person gets on the bus and sits down the number of people sitting goes up by one – so in effect that variable does in the mind of the bus driver if they are keeping track of how full they are.

In programming languages tests have to be written in a rigid form. Remember programming languages are designed to remove ambiguity and do this by restricting how you are allowed to say things. In our bus example, we wrote "10 standing". In a programming language we would probably be required to write this in a form similar to:

standing **equals** 10

using the **equality** operator. It is true if the two things given with it are equal – here we are testing whether the number of people standing is equal to the number 10 or not. Different programming languages would use different words or symbols for equals for example one (such as the language Pascal) might require you to write

standing = 10

and another language (such as the language Java) requiring you to write

standing = = 10.

In these two cases the meaning is intended to be the same – its just that the different languages designers have chosen different words to mean "equals". The full sign would need to be written something like:

(sitting **equals** 35 **and** standing **equals** 10) **or**
(sitting **equals** 34 **and** standing **equals** 9 **and** wheelchair **equals** 1)

Similarly if we were trying to write in the style of a programming language we might write for our earlier examples:

Glasgow **equals** the capital of Scotland

*The Queen of England* **equals** Elizabeth **and**
*the Prime Minister* **equals** Tony Blair

where before we were using the word "is" to mean "equals.

Whenever you are writing a test to go for a program you must look for situations where you are saying two things are equal and convert it to this form of asking whether it is true that one thing equals another.

Equality is a **relational operator**. It relates two things. Other relational operators, familiar from School Maths lessons are "less than", "greater than", "less than or equal to" and "greater than or equal to". These are normally written by mathematicians as $<$, $>$, $\leq$, $\geq$. The bus sign was written in a way so that the test is true when the bus is full. Alternatively the sign could have been written to be true when the bus still had space. The bus is legal if:

sitting $\leq$ 35 **and** standing $\leq$ 10

## Declarations

Here is a recipe:

***Fiorentina Pizza***

**Ingredients**

Yeast
Water
Flour
Tomato Puree
Cheese
Spinach
1 egg

1. Mix yeast and water, add flour and stir
2. Knead for 10 minutes.
3. Leave to rise for 30 minutes.
4. Roll out the dough.
5. Place in a large round pizza tin.
6. Spread with tomato puree, and cheese and spinach.
7. Crack an egg into the middle.
8. Bake in oven for 25 minutes.

This recipe for pizza, starts with a list of ingredients. They are not part of the algorithm as such since they are not instructions of how to do something. The algorithm consist of the number sequence of instructions. They could be given in any order. Why do recipes universally start with an ingredients list? It lists the resources that will be needed for the instructions to be followed. However, all that information is contained within the instructions themselves. On seeing an instruction, crack the egg into the bowl, I can see I need an egg. The advantage is that if the ingredients are listed at the start, you can ensure that you have everything you need to hand before you start.

Craft books, from how to make wooden toys to how to make necklaces use a similar list of resources. Here the "ingredients" are the materials:

**Materials**

5mm plywood (160 x 100mm)
2 20mm round-head woodscrews
PVA glue
paint

In addition, unlike recipe books craft books often include also a list of tools with each separate set of instructions.

**Tools**

Drill
Fretsaw
Paintbrush

This is just a different kind of resource being listed. That would be the equivalent of a recipe book also listing the pots and pans needed (something I would often find useful as I frequently run out of pans).

The script of a play also has the same structure. A play is just a series of instructions to actors playing different parts. Before starting you need to allocate parts to actors, so you need to know what parts this play has. My copy of Shakespeare's *Macbeth* has such a list on the first page before the actual play itself.

Persons Represented

> Duncan, King of Scotland.
> Malcolm, Son of Duncan.
> Donalbain, Son of Duncan.
> Macbeth, General of the King's army.
> etc

Programs have something similar to an ingredients list: a list of declarations. They also list the resources that a program needs. For programs the resources that matter are places to store data. Having declarations allows the compiler to ensure that enough storage space is available. Declarations in programs have other purposes as well, however. One is to give labels or names to the resources so that they can be referred to later in the program and we will know which resource is being referred to.

Consider another source of algorithms in everyday life: the instructions that come with construction kits (whether children's toys or flat-pack do-it-yourself book shelves). They also normally have a list of parts at the start. However, those parts are also given a label: often just a letter. The purpose of the letter is so that that part can be referred to without ambiguity later on in the instructions. For example, the list of parts might be as follows:

> 6 x **A**: 200x1000x20mm
> 2 x **B**: 200x2000x20mm
> 1 x **C**: 1000x2000x3mm

The instructions then might be:

1. Place part **C** face up on the floor.
2. Place the 2 parts **A** side by side against the edges of part **C**.
3. etc.

Thus declarations give a list of resources needed, but also give names to each resource that can be referred to within the algorithm itself. These names are referred to as **identifiers**: they are used to identify a specific resource. In the shelf construction algorithm above, three identifiers were used: A, B and C. Using letters for identifiers removes ambiguity but make algorithms harder to understand. Looking at the shelf instructions it is not immediately easy to see which pieces of wood are the shelves, which the back and which the sides. We could easily solve this problem by using more meaningful identifiers in our instructions (and also of course in the declarations). For example, the following instructions would be much easier to follow:

> **Parts**
> 6 x **shelf**: 200x1000x20mm
> 2 x **side panels**: 200x2000x20mm
> 1 x **back panel**: 1000x2000x3mm
> etc.
>
> **Instructions**

1. Place the **back panel** face up on the floor.
2. Place the 2 **side panels** side by side against the edges of part **back panel**.
3. etc.

Text books use glossaries and lists of acronyms for a similar purpose to allow a label to be used to refer to something else. A glossary is just a list of unusual terms used in the book. It gives a name for each unusual word or phrase in the book, by looking at the glossary you can find out what is meant by a term. Acronyms are just shortened versions of long phrases. By putting a list of acronyms at the start of a book, we can then use the acronym throughout the book in place of the term.

**Variable Declarations**
The declarations in programs are not quite the same as those in recipes in that they list a slightly different thing: storage spaces indicating the kind and size. It is as though a recipe also gave a list of storage jars, pots, pans and dishes needed for a recipe at the start. This would not be that silly a thing for them to do even though I have never come across it. On several occasions, part way through a recipe I have run out of pans, because I, for example, used a large pan for a cheese sauce, only to find I needed it later to fit the amount of vegetables I had to cook. Mid recipe I have a panic whilst I transfer things from one pan to another, and wash things – the last thing I need with guests arriving in 15 minutes. The point of declarations is to allow all the things you need to be gathered and organised before you start – precisely to avoid that kind of crisis.

Computers move and manipulate data: patterns of 1s and 0s. All of this data needs to be stored somewhere whilst it is processed. That is what a variable is used for: a storage jar for data. Each jar has an associated identifier: a name as we saw above. It is as though each jar is labelled with a name so that in the instructions of the algorithm, we can refer to each jar by name and be sure of which one we are referring to.

In our Imp computer, declarations would be used to gather the storage Imps together as they were needed. Without declarations, we might get part way through a calculation and find another Imp was needed. If they had not been booked in advance using a declaration they would unlikely to be available – most likely they would be down the Pub and in no state to do anything.

**Types**
Consider the following list of things.

Red
42
Blue
Bus
a
Car
64
b
c
Bicycle

Now group them into sensible categories. The chances are you grouped them in the same way as I did:
42, 64
Red, blue
a, b, c
Car, Bicycle, Bus
It is arguably a human instinct to divide the world into categories in this way. It also turns out to be very useful in programs – letting the computers into the secret of what our categories are.

Declarations, as we saw, are used to give information at the start of a set of instructions about the resources that will be needed to complete the instructions. For programs the resources are places to store data. Often by data we just mean numbers – a persons age, the salary of a person. However, data can represent more than just numbers: letters of the alphabet (eg representing a students grade) words or sentences (like a person's name) colours (such as the colour to paint Laura Croft's vest), and many more. An important piece of information about data is what is known as its **type**. The type of something in the computer science sense is just what kind of object it is: in fact everything of interest about it apart from its actual value. Common programming examples of types include integers (whole numbers), floating-point numbers (decimals), characters (letters and symbols), and strings (sequences of characters like words or sentences).

The children's game of Categories is based on the fact that we group things together. In the version for young children, it involves the father writing down a category and the child having to think of something that belongs in that category not thought of by another player. Part of the point of the game (at least from the father's point of view) is to teach the child what things are grouped together. Think of a bird: "Penguin". Think of a city: "Sheffield". For older children, the game is made harder by requiring the thing to start with a given letter of the alphabet.

Categorising things allows us to classify the world and draw conclusions about the properties of objects just from knowing their category. If it is a bird then without seeing it I can guess it has feathers as that is one of the properties of birds. In real life the categories blur. If it is a bird then it must be able to fly is not always true for example – categories are used more as rules of thumb. Computers require preciseness however, so the categories they use are more hard and fast. "If it is a character then it definitely does fit into a byte". Knowing the type (i.e. category) of things in the program allows the **compiler** to make appropriate preparations for use of the piece of data such as how much space it will need. It also allows the compiler to check that things are used in the program in the way intended. You do not throw a pig out of an aeroplane and expect it to fly in the real world as ability to fly unaided is not a property of pigs. Neither do you include instructions in a program to do multiplication on strings of letters as ability to be multiplied is not a property of strings.

Let us return to food and recipes. We group kinds of food into categories like "pasta" or "cheese". There are many different cheeses and many different kinds of pasta. What use are these labels? One reason why they are useful is that knowing something is pasta, immediately tells us a whole series of things about it. In particular it tells us which actions can be performed on it in a recipe and also which operations cannot be

performed. What can we do to (dry) pasta? We can boil it or bake it in the oven within a Lasagna like dish. What else do recipes do to pasta? My wife looked at me as though I was mad when I asked her that question! Why? Because it would be weird to think of doing anything else. If when we were cooking dinner together I passed her the packet of spaghetti and asked her to "sieve that" she would assume I was losing my grip. She would not get out a sieve. Why? Because you do not perform the action "sieve" on things of type "pasta". Similarly, there are sensible ("grate") and silly ("sieve" again) actions to do on cheese. The types of ingredients can thus be used to check for errors when looking at recipes. If I invented a new type of pasta "Taggliaroni" – just by telling you it was pasta, you would automatically know many of the things you could and should not with it. You could immediately substitute it into a whole range of recipes and get edible results. Similarly, if you know how to make Blackberry and Apple Crumble, then it is a fair bet that Raspberry and Apple Crumble would work too as both Blackberries and Raspberries are berries.

Types in programs serve a similar purpose – they allow errors to be spotted. You can not do arithmetic on colours for example – it is meaningless. You might on the other hand convert a colour to a number, do arithmetic on that and then convert it back to a colour in some way. For example you could arbitrarily decide that Blue converts to 1 and Red converts to 2. Blue could be converted to a number, 1 added to that number. Converting the resulting number back would give you the colour red. Such an operation that converts things from one type to another is known as a **cast** operation.

In an Imp computer, different types correspond to Imps holding different sized boxes. An integer Imp has a box big enough to hold a number. A character Imp has a smaller box big enough only for a character. Variable declarations ensure that the correct mix of Imps is obtained and that each Imp has the correct kind of box. If ever an Imp was given something of the wrong type for their box, they would mangle the result trying to fit it in.

**Enumerated Types**

Most computer languages come with some predefined types. By this we mean they just know about some values and how they are grouped together – which categories they belong to. These are usually the things that most programs will need to use like numbers and characters. Many languages also allow the programmer to write instructions that create new values and put them into new categories. That is the languages allow instructions for creating new types. What does this involve? If we are to create a new type we will need to give it a name and we will need to say which things are in the type. The simplest way of doing this is just listing them. For example if I was the manager of a consumer electronics shop and had a new trainee assistant, I would have to give him instructions about his job. Suppose he was very slow and so did not know much about televisions and videos. The first thing I might do would be to explain to him about the different things he had to sell – as he would have to explain the choices to the customers. The first thing I might do is say something like:

> "One thing we sell is *video recorders*. We sell the Technics-501, the ThornE54 and the Sony-P45". "We also sell *televisions*. We sell the Sony-T100 and the Ferguson170." etc

What we have introduced to the trainee is two new categories of things: two new type: televisions and videos. We have then explained what those types are by listing the things: the **values** that belong to that type one after another. The values of type television are the Sony-T100 and the Ferguson170, for example. A type defined in this

way by listing all the things in the type and giving this collection a name is known as an **enumerated type**. Once the type has been defined, then instructions can be given that use the type (in the above case instructions of how to get a customer to exchange money for one of the items of that type!) Notice however that the type definition is not strictly part of the algorithm itself. It is not an instruction of how to do something. It is rather a declaration of information needed to understand the algorithm. Type declarations are thus similar to variable declarations in this way.