INTRODUCTION TO COMPUTER PROGRAMMING

Richard Pierse

Class 3: Controlling Programs: Conditionals and Looping

Conditionals

Conditional statements allow your program to make choices and perform one action or another depending on the value of a condition.

The `if` command

Liberty BASIC, like almost all other programming languages, uses the `if` command to allow the conditional execution of statements. The format is:

```
if(condition) then
    first block of statements
else
    second block of statements
end if
```

The first block of statements is executed if the condition is true. Otherwise, the second block of instructions is executed. The `else` clause is optional. If omitted then no instructions are executed when the condition is false. The `end if` command must always be present and signals the end of the conditional block(s).

Conditions are specified with relational expressions using one of the relational operators in the following table:

Operator	*Description*	*Example*
`<`	*Less than*	`if (sales < maxsales) then`
`>`	*Greater than*	`if(amount > 100.0) then`
`=`	*Equal to*	`if(age = 21) then`
`>=`	*Greater than or equal to*	`if(grade >= 70) then`
`<=`	*Less than or equal to*	`if(price <= 1.0) then`
`<>`	*Not equal to*	`if(year <> 2004) then`

Relational expressions are always either ***true*** or ***false***.

It is possible to nest one `if` statement within another as in the examples:

```
    if(age < 18) then
        if(age > 12) then
            print ”teenager”
        else
            print ”child”
        end if
    else
        print ”adult”
    end if
```

or

```
    if(age > 18) then
        print ”adult”
    else
        if(age > 12) then
            print ”teenager”
        else
            print ”child”
        end if
    end if
```

These two examples both do the same thing. In the first, an `if...end` block is nested within an outer `if` loop between the `if` and `else` clauses. In the second, the inner `if` loop occurs between the outer `else` and `end if` statements.

Note the use of indentation in the examples to make clear the level of nesting in the conditional statements. While not necessary, indentation increases clarity and is strongly recommended as good programming practice.

There are some limitations on the nesting of `if` statements in *Liberty BASIC*, compared with other languages and higher order nesting than that shown in these examples is not possible.

Looping Statements

Looping statements cause blocks of code to be executed repeatedly, making it possible to perform repetitive tasks, over and over and again.

Liberty BASIC supports two looping commands: the `for...next` loop and the `while...wend` loop. The first executes a block of code a fixed number of times, while the second executes a block of code repeatedly as long as a particular condition is satisfied.

The `for...next` loop

This loop is best explained through an example. Consider the following code:

```
for i = 1 to 5
    print i;
    print ": log = "; log(i)
next i
```

The `for` statement starts the loop and the `next` statement ends it. Each `for` loop has an associated *control variable*, in this case `i`. The statements between the `for` and `next` are executed a fixed number of times for values of `i` going from `1` to `5` in increments of `1`. This loop prints the value `i` and its natural logarithm, `log(i)` for the values: `1`, `2`, `3`, `4` and `5`.

It is possible to write `for` loops in which the control variable increments in values other than `1` by using the `step` clause. Consider this example:

```
for k = 0.5 to 0.2 step -0.1
    print k
next k
```

In this loop the control variable `k` goes from the value `0.5` down to the value `0.2` in negative increments of `–0.1`.

Loops can be *nested* within each other as in the following example:

```
for i = 1 to 4
    for j = 1 to 3
        print i*j
    next j
next i
```

In this example the block within the `for i` loop contains another `for` loop, for variable `j`. The entire `for j` loop is executed for each value of `i`. Note that the control variables in nested loops must be different, and that the inner loop(s) must finish before the outer loop. Loops in *Liberty BASIC* can be nested to any depth.

The `while...wend` loop

The `while...wend` loop allows a block of commands to be executed repeatedly, while a condition continues to hold. For example, the following code:

```
while (x <= 120)
    x=x*10
wend
```

repeatedly multiplies the number `x` by `10` until it is greater than `120` when the loop terminates. What would happen if the initial value of `x` were zero or negative? In this case, the condition would always be true so that the loop would never terminate. This is an example of an *infinite loop*. Programmers should take care to avoid inadvertently writing infinite loops.

`while...wend` loops can be used to create programs that run repeatedly until the user chooses to terminate them. This is illustrated in the following example:

```
ans$=”Y”
while(ans$ = ”Y”)
    input ”Enter a number: ”; x
    if (x > 0.0) then
        print ”Square root is ”; x^0.5
    else
        print ”Non-positive number”
    end if
    input ”continue (Y/N)? ”; ans$
wend
```

The loop prints the square root of positive numbers as long as the user continues to respond `”Y”` to the question. Note that the first line initialises the character variable `ans$` to `”Y”` so that the loop always executes at least once.

Arrays

Arrays are used for storing several items of the same type, such as product prices for 30 different firms, or names or test scores for a class of students. Without arrays, we would need to use different variable names for each item in the group e.g.

```
student1score = 56
student2score = 74
student3score = 65
```

With an array, a single name can be used to refer to the whole group of items and individual items can be referred to using indices e.g.

```
scores(1) = 56
scores(2) = 74
scores(3) = 65
```

Arrays are very powerful and are naturally manipulated with looping commands. The student scores in the previous example can be printed by the loop:

```
for i = 1 to 3
    print ″student ″; i; ″ mark ″; scores(i)
next i
```

Arrays normally need to be ***declared*** before they are first used so that the computer knows how much array space is required. This is accomplished by the `dim` command. For example, the command

```
dim scores(30)
```

reserves space for `30` numeric elements in the array `scores` , while the command

```
dim names$(100)
```

reserves space for `100` character variables in the array `names$`.

Liberty BASIC does allow small arrays (10 or less elements) to be used without having being declared beforehand. However, it is good practice *always* to declare arrays.

Liberty BASIC also supports two-dimensional arrays, which are rectangular sets of elements declared as e.g

```
dim m(10,4)
```

with individual elements referred to as `m(10,1)`, `m(1,4)` etc.

The following code reads a set of `n` numbers (maximum of 100) into the array `x` and computes their mean and variance:

```
dim x(100)
input ”How many numbers? ”; n
if(n < 2) then
    n = 2
else
    if(n > 100) then
        n = 100
    end if
end if
sum = 0.0
sumsq = 0.0
for i=1 to n
    input ”Enter next number: ”; num
    x(i) = num
    sum = sum + x(i)
    sumsq = sumsq + x(i)*x(i)
next i
mean = sum/n
variance = (sumsq - n*mean*mean)/(n-1)
print ”Mean is ”; mean
print ”Variance is ”; variance
```

The program first asks for the value of `n` and checks for illegal values (`n < 2`). If `n` is too large, it is reset to `100`. Then the variables `sum` and `sumsq` are initialised to zero and the loop starts, prompting the user to enter each number and cumulating the sum and sum of squares. Note that each number is read initially into the variable `num` before being stored in the array `x`. This is because *Liberty BASIC* does not allow input directly into array elements. (This is a particular limitation of *BASIC* and does not apply to most other programming languages.)

Example: A Number Guessing Game

The following example illustrates the use of the looping and conditional commands from this week’s class in a program in which the user has to guess a random number chosen by the computer. The second line uses the `rnd()` function to generate a random number between zero and one. This is transformed into an integer between `1` and `100`, which the user then has to guess. If the guess is too low or too high, the user is prompted to guess higher or lower. The variable `tries` counts the number of guesses. Finally, when the user correctly guesses the number, the program prints the answer and the number of tries. Note that the variable `guess` is initialised to the impossible value of `–1` so that the `while` guessing loop always gets executed.

```
print ”Guess my number”
num = int(rnd(1)*100)+1
tries = 1
guess = -1
while (guess <> num)
  input ”What is your guess (1-100)? ”; guess
  if(guess < num) then
    print ”Higher”
  else
    print ”Lower”
  end if
  tries = tries + 1
wend
print ”My number was ”; num
print ”You got it in ”; tries; ” tries”
end
```

Solutions to last week’s exercises

Exercise 1: calculate a firm’s profits

```
input ”fixed costs? ”; fixed
input ”unit variable costs? ”; variable
input ”product price? ”; price
input ”product quantity? ”; quantity
profits = (price-variable)*quantity-fixed
print ”profits are ”; profits
```

Exercise 2: annual compounded interest

```
input ”lump sum? ”; X
input ”annual interest rate as %? ”; r
input ”number of years? ”; n
print ”value is ”; X*(1+r/100)^n
```

Exercise 3: continuously compounded interest

```
input ”lump sum? ”; X
input ”annual interest rate as %? ”; r
input ”number of years? ”; n
print ”value is ”; X*exp(r/100)^n
```

Exercise 4: split first 3 words from a string

```
input ”enter string ”; str$
length=len(str$)           ' length of string
sp1=instr(str$,” ”)        ' position of 1st space
word1$=left$(str$,sp1-1)   ' first word
str$=right$(str$,length-sp1)  ' remove word 1 from str$
sp2=instr(str$,” ”)        ' position of 2nd space
word2$=left$(str$,sp2-1)   ' second word
str$=right$(str$,length-sp1-sp2)  ' remove word 2
sp3=instr(str$,” ”)        ' position of 3rd space
word3$=left$(str$,sp3-1)   ' third word
print word1$; ” ”; word2$; ” ”; word3$
```

Exercise 5: area of a circle

```
pi=3.1415926
input ”radius? ”; r
print ”area is ”; pi*r*r
```