INTRODUCTION TO COMPUTER PROGRAMMING

Richard Pierse

Class 4:
Programming Fundamentals: Structuring and Subroutines

Designing Programs

Writing a computer program to perform a complex task can seem a daunting undertaking. Before getting down to write any code, it is important to spend time thinking about the design of the program and breaking the task down into its component steps. In this way, the task becomes more manageable and no detail gets forgotten. In the business world, large programming projects are generally managed by ***systems analysts*** who are concerned with designing programs and may do very little actual programming themselves.

Top-Down Program Design

Top-down program design is a technique to help plan the design of a program. The key to the top-down approach is to put off the details as long as possible, concentrating on the overall goal and breaking that up into component parts, and then each part into smaller parts and so on. It can be illustrated in the schema:

1. Determine the overall goal
2. Break that goal into a few component parts, each with its goal
3. Repeat step 2 for all sub-goals, holding back details as long as possible.

Developing Program Logic

Once the task has been broken down into its component parts, each component can be analysed in detail. Two tools which are useful in planning the logic of each component are pseudo-code and flow charts.

Using pseudo code

Pseudo-code is a form of English that is understandable without knowledge of any programming language, but that represents clearly the logic underlying the task. It is simple for a programmer to translate pseudo-code into actual code, making the final programming task very straightforward. Pseudo-code can be used at different levels of abstraction, and is useful in many areas outside computer programming e.g. instruction manuals.

Here is an example of some pseudo-code for a simple payroll calculation:

```
For each employee:
   If the employee worked 0 to 40 hours then
     net pay equals hours worked times rate
   Otherwise,
     If the employee worked between 40 and 50 hours then
        net pay equals 40 times the rate
        add (hours worked - 40) times the rate times 1.5
     Otherwise,
        net pay equals 40 times the rate
        add 10 times the rate times 1.5
        add (hours worked - 50) times twice the rate
   Deduct taxes from the net pay
   Print the pay cheque.
```

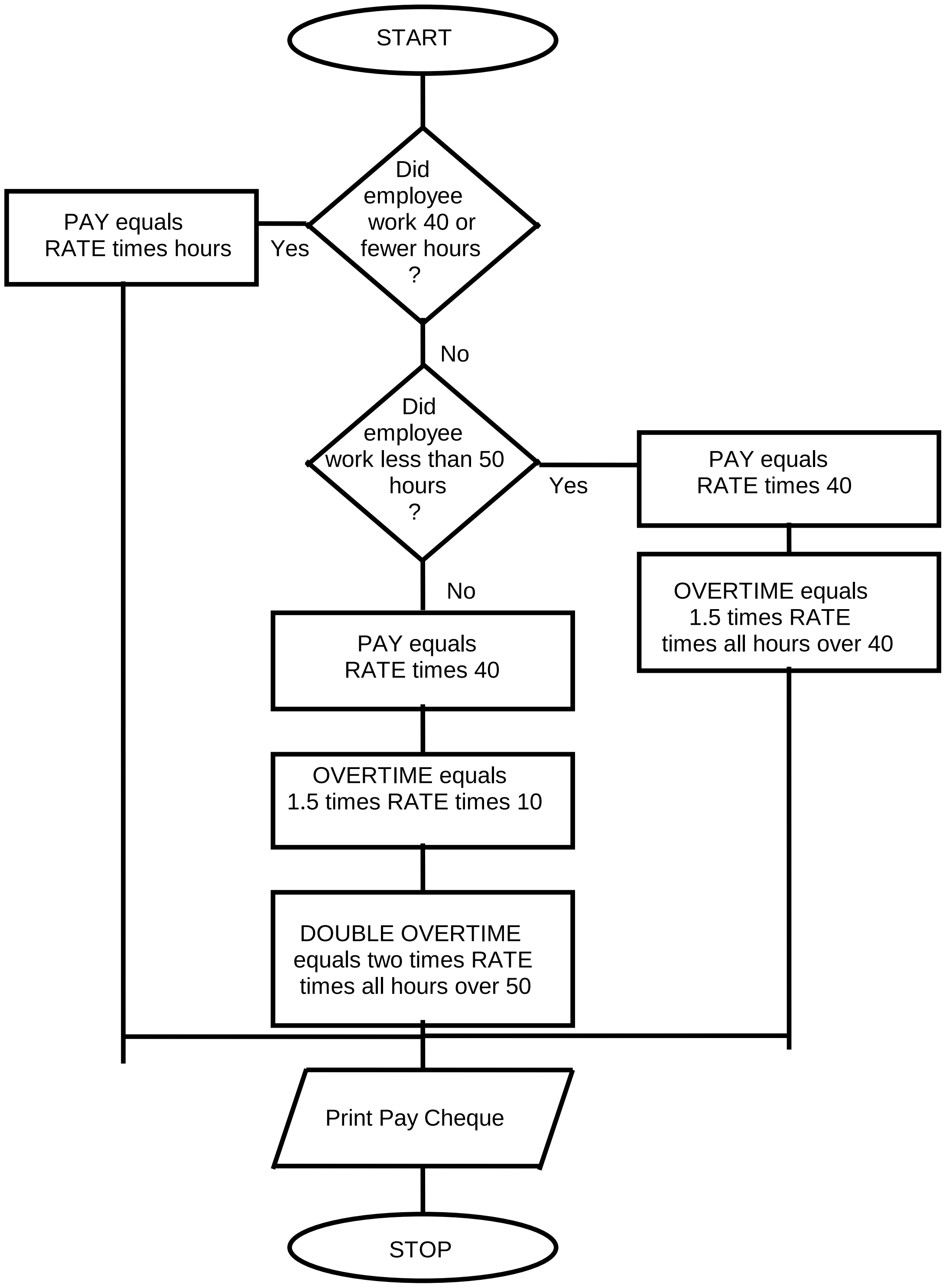
Flow Charts

An alternative to pseudo-code is the use of flow charts. The flow chart gives a pictorial representation of the logic of a program, using arrows to indicate the flow and boxes of various shapes to indicate different tasks: calculations, conditional branching, input / output etc.

Flow charts have been around since the early days of computing and are very useful in documentation and as a teaching tool. However, they are rarely used by practical programmers.

A Flow Chart

Here is a flow chart corresponding to the previous pseudo-code.


START
Did employee work 40 or fewer hours ?
Yes
PAY equals RATE times hours
No
Did employee work less than 50 hours ?
Yes
PAY equals RATE times 40
OVERTIME equals 1.5 times RATE times all hours over 40
No
PAY equals RATE times 40
OVERTIME equals 1.5 times RATE times 10
DOUBLE OVERTIME equals two times RATE times all hours over 50
Print Pay Cheque
STOP

The arrows indicate the flow. Note the different symbol shapes: a diamond for a question, a rectangle for a task, a rhombus to indicate input/output, an oval for start or stop.

Structured Programming

Structured programming is a philosophy stating that programs should be written in an orderly fashion without a lot of jumping around. The keys to structured programming are:

- sequence
- decision
- iteration

Sequence means that commands should follow in a natural order. *Decision* refers to the use of `if...else...endif` structures to control the conditional execution of commands. *Iteration* means that loops should be structured like the `for...next` and `while...endw` loops in *Liberty BASIC*.

Modern programming languages are designed to encourage structured programming and the block `if...endif` and the `for...next` and `while...endw` loops in *Liberty BASIC* are examples of structured commands.

To see an example of unstructured programming, we will look at a *Liberty BASIC* command not covered before: the `goto` statement. The format is:

```
    goto [label]
```

where `label` is a name or a number referring to a statement somewhere else in the program. The command instructs the program to find the labelled instruction and continue executing at that point. Consider the following example:

```
    i = 1
    [start]
    print i
    i = i+1
    if(i > 10) goto [finish]
    goto [start]
    [finish]
```

This code is equivalent to a `for i=1 to 10...next i` loop but is considerably more clumsy and difficult to understand. (Note also the use of an unstructured `if` statement in this example).

The wide use of `goto` statements and other unstructured commands quickly results in ***spaghetti code***, so-called because it flows and swirls all over the place. Although `goto` statements cannot always be avoided (e.g. to force premature exit from a loop), they should be avoided whenever possible.

Functions and Subroutines

We came across functions in the first class in the form of standard mathematical and character functions such as `log(x)`, `exp(x)` and `char$(n)`. You can easily define your own functions to perform common tasks. Functions can take one or more arguments (or no arguments) and return a single value. As an example, consider the following function to compute the area of a circle:

```
     function area(r)
       a=3.141592654*r*r
       area=a
     end function
```

This function takes a single argument, the radius `r`, and returns the value of the variable `area`, the circle area πr^2. The value returned by a function always has the same name as the function itself. Note that the variable `a`, used within the function, is local and is not accessible outside the function. The function can be referenced fro the main program as in the command:

```
     a = area(8)-area(5)
```

Subroutines

Subroutines or *procedures* (known as ***subprograms*** in *Liberty BASIC*) differ from functions in that they do not return a value. They can take any number of parameters. The following example defines a subroutine called `maxmin` with three parameters that prints the maximum and minimum of the three arguments `p1`, `p2` and `p3`. The subroutine definition starts with the `sub` statement and ends with the `end sub` statement.

```
     sub maxmin p1, p2, p3
       maxp = max(max(p1,p2),p3)
       minp = min(min(p1,p2),p3)
       print ″ max ″; maxp; ″ min ″; minp
     end sub
```

Variables defined within a subroutine are local to that subroutine and inaccessible outside it. Similarly, the subroutine arguments are also local so that, if changed within the subroutine, remain unchanged outside it. A subroutine can be called in the main program using the `call` command as in

```
     call maxmin x1/x4, log(x2), abs(x3)
```

Programming Errors

Programmers inevitably make mistakes. An important part of programming is to ***debug*** a program to remove all the errors. There are three types of programming error:

- syntax errors
- runtime errors
- logical (semantic) errors

Syntax errors are caused by illegal instructions in the program source code. These are mainly caused either by typing errors as in `endif` (should be two words) or by failing to follow the rules as in

```
if (4 > 3)
```

where the keyword `then` has been omitted. Syntax errors are almost always spotted by the compiler. When the *Liberty BASIC* compiler tries to run a program containing syntax errors, it highlights the first offending line of code in blue. This makes it easy to locate the error and fix it.

Runtime errors are errors that occur during the running of a program, generally causing it to crash. The most common cause of runtime errors is an illegal mathematical operation such as division by zero or taking the logarithm of a negative number. Two other runtime errors are infinite loops and code over-writing caused by overflowing array bounds. Runtime errors won't be spotted by the compiler, even when they are blatant as in the line `x = 4 / 0`. The best way to avoid runtime errors is to program cautiously and include careful checks before any potentially dangerous operations such as division.

Logical errors are probably the most common errors but also the most difficult to spot. This is because a program with logical errors may well compile and run successfully. The problem is that the program just doesn't work correctly because of a fault in the program logic. Consider the following example to print the minimum of 10 numbers and see if you can spot the error.

```
for i=1 to 10
   input ”enter number: ”; x
   if(x < minx) then
      minx = x
   end if
next i
print ”minimum is ”; minx
```

Debugging Programs

Logical errors can be hard to locate. *Liberty BASIC* offers a tool to help programmers locate logical errors and debug their programs.

Using the Debugger

The Liberty BASIC debugging tool is started by clicking on the debug button on the toolbar (the one with a bug on it) or by selecting ***debug*** from the ***run*** menu. Two new windows open: the familiar output window which will display any output as the program executes, and a debug window with two panes and several buttons. The lower pane shows the source code with the next line to be executed highlighted in blue. The upper pane displays the contents of all variables in the program so that we are able to track their values as the execution progresses.

Single Stepping through Code

Probably the most useful feature of the debugger is the single-step feature. This allows the code to be executed a single line at a time. Click on the ***step*** button in the debug window to start single stepping. The highlighted line in the lower pane moves on to the next line. The upper pane allows us to inspect data variables and see how their values have changed after the execution of the line. Continue to step through the code. As you step through loops and conditional statements, the highlighted line will jump about, following the flow of the program.

The ***walk*** and ***run*** buttons allow you to run the program at half or full speed until you click on the ***step*** button to pause execution again. This allows you to move quickly to a particular area of the program code which you want to examine in detail.

The *Liberty BASIC* debugger is quite simple compared with some other debugging tools but is still very useful. Running the problem program from the previous page through the debugging tool should make clear why that program doesn’t always work as intended.

Testing Programs

Programs should always be fully tested to ensure that all bugs have been eliminated. Beta testers are used for large programs.

Solutions to last week’s exercise

A program to guess a user’s number between 1 and 100

```
print "Think of a number (1-100)"
gmax=100
gmin=1
tries=0
ans$="h"
while (ans$ <> "=")
    guess=int((gmax-gmin)/2)+gmin
    print "I guess "; guess
    tries=tries+1
    print "Is it higher(h), lower(l) or OK(=) ";
    input ans$
    if (ans$ = "h") then
        gmin=guess+1
    else
        gmax=guess-1
    end if
wend
print "Your number was "; guess
print "I got it in "; tries; " tries"
end
```