G-PASCAL NEWS

Registered by Australia Post - Publication Number VBG 6589

Published by Gambit Games for Commodore 64 G-Pascal owners.

Mail: G-Pascal News, P.O. Box 124, Ivanhoe, Victoria 3079. (Australia)
Phone: (03) 497 1283

Gambit Games is a trading name of Gammon & Gobbett Computer Services Proprietary Limited, a company incorporated in the State of Victoria.

VOLUME 2, NUMBER 2 — April 1985 Annual Subscription $12

WHAT'S IN THIS ISSUE

This month we have two main articles — a G-Pascal tutorial for readers who would like to know more about Pascal programming, and a spelling checking program submitted by a reader. We also describe on this page a simpler method of backing up and customising your G-Pascal compiler, and there is a small program on page 9 which prints dollars and cents right justified.

NEXT ISSUE

Next issue we intend to proceed with the G-Pascal tutorial, print the syntax diagrams for G-Pascal, describe how to properly use the Debug and Trace feature, and describe Independent Modules in greater depth. Any suggestions for other articles, or questions to be answered are welcome.

CREDITS

Material in this issue by Nick Gammon, Sue Gobbett and Adrian Winn. Edited by Nick Gammon. Assembled by Cynthia Gammon. Typeset on a Brother EM-100 typewriter using 'Lori', 'OCR-B', and 'Script' daisy wheels. Typesetting software written in G-Pascal to handle proportional space typefaces. Written and printed in April 1985.

COPYRIGHT

BACKING UP YOUR G-PASCAL

Since our last issue was published a number of readers have written in to point out a simpler method of backing up your copy of G-Pascal. This method involves:

— Turn on your Commodore 64 and load G-Pascal as usual, but do **not** type RUN yet.
— Insert a new disk or cassette as appropriate.
— Use the Basic POKE command to make any changes (such as printer seconday address) that you normally make. **(CAUTION: see below for address offset.)**
— Type: SAVE "GPASCAL",8
to save your new copy.
(For cassette, leave out the ',8').

Backing up using this method means that the new copy will still auto load, the same as the original copy.

If you want to 'patch' your new copy then you must take into account the fact that the G-Pascal that you have just loaded from disk or cassette has not yet been relocated to its 'running' address. **You must subtract hexadecimal $77C7 from any addresses suggested in earlier copies of G-Pascal News.** To summarise the most useful ones:

Background colour: $23B1 (9137)
Foreground colour: $1512 (5394)
Printer secondary address: $2696 (9878)
Source start address: $843 (2115)
Default disk/cassette access: $249F (9375)

Decimal addresses are given in parentheses. Refer to the earlier issues of G-Pascal news to see the meanings of these locations. For example, to change the background screen colour to black, and your printer secondary address to 7, you would type:
POKE 9137,Ø : POKE 9878,7
before saving your new copy of G-Pascal.

G-PASCAL TUTORIAL - PART 1

Author: Nick Gammon

Fundamentals

Getting started with G-Pascal is easy — the simplest programs are ones that don't use variables, procedures or functions. (However, such programs are not as powerful as those that use these facilities). The only 'required' words in a G-Pascal program are the BEGIN and END that surround the main program, and the period following the final END. For example, this simple program will print out the results of an arithmetic calculation:

```
1 BEGIN
2 WRITELN (5 + 20 * 6)
3 END.
```

Line 1 signals the start of the 'executable statements'. Line 2 multiplies 20 by 6 and then adds 5, and writes the result to the screen using the 'write line' (WRITELN) statement. Line 3 indicates the end of the program, and as such must end with a period. The reason that G-Pascal does the multiplication before the addition is because the 'precedence' of multiplication is higher than addition. The precedence sequence for all of the arithmetical operations is described on page 25 of the G-Pascal Manual.

If you want to have more than one statement between the words BEGIN and END then they must be separated by semicolons. Notice the distinction between **separating** statements by semicolons, and **ending** them with semicolons. Statements are not ended with a semicolon, just separated by them.

For example, to add another line to our example program, we could say:

```
1 BEGIN
2 WRITELN ("Here is a test program");
3 WRITELN (5 + 20 * 6)
4 END.
```

Notice the semicolon at the end of the new line 2. Also notice how the editor has renumbered what was previously lines 2 and 3 to be lines 3 and 4, as we inserted a new line 2 (using the Editor 'Insert' command).

Constants

A useful aspect of Pascal is the ability to declare constants at the start of a program (or the start of a procedure or function). Constants are values that will never change during the execution of a program, but which can be assigned meaningful names. For example, if you want to have the maximum number of ships in a game to be three, then declare:

```
CONST maximum_ships = 3;
```

The constant declarations consist of the word CONST (this only appears once for a group of constants), then the name you choose — in this example it is MAXIMUM_SHIPS, then the "=" sign, then the value you are assigning to this constant, and lastly a semicolon. The value of the constant is usually represented by a number (such as 3), but can also be the name of a previously defined constant. Constant declarations appear before VAR, PROCEDURE and FUNCTION declarations (described later). A series of constant declarations would look like this:

```
1 CONST green = 5;
2       initial_colour = green;
3       true = 1;
4       false = 0;
5 BEGIN
6 END.
```

The above example is a complete program, although it does not 'do anything' because there are no statements between the BEGIN and END. Declaring TRUE and FALSE as above is useful in most programs so as to make Boolean variables work the same as in full Pascal. Notice how in line 2 the constant INITIAL_COLOUR is assigned the value GREEN which is itself a constant. Using constants carefully makes programs more readable (referring to GREEN and INITIAL_COLOUR later on in the program is more meaningful than writing "5"). Also, if you decide to change the initial colour (whatever that is) you only need to change one spot near the start of the program, rather than hunting through the program looking for the number 5.

Variables

To use the full power of the Pascal

language we can use variables — these are named locations in the computer's memory that we use to hold data that varies during the execution of the program, hence their name. The names of variables are at the sole discretion of the programmer — yourself. (The same applies to naming constants, procedures and functions). You can choose a name of any length, to make the variable as descriptive as you like. All the characters in the variable name are 'significant' which means that, even using long variable names which differ only at the end, G-Pascal can tell the difference between them.

The only restriction in choosing your own names is that you cannot use a 'reserved' word — that is, one that G-Pascal already uses for its own use, and the variable name must start with a letter (A to Z), and then consist only of letters, numbers, or the underscore character. This means that you can choose names such as:

```
NUMBER_OF_KLINGONS_LEFT_IN_THE_GALAXY

TAX_RATE_FOR_1985
```

Of course really long names are a bit tedious to type in every time, but they do make the program easy to follow. One trick that you can use to reduce typing time is to initially use a short variable name (such as NKL) and then use the (R)eplace command in the Editor to expand the short name to a longer one once you have finished typing in the program. Just make sure that your short names are not ambiguous if you try this trick.

To use variables in your program you must first 'declare' them — this means telling G-Pascal that you are planning to use the name you have chosen as a variable, and in the process tells G-Pascal what sort of variable it is. A simple program with a variable declaration looks like this:

```
1 VAR number : INTEGER;
2 BEGIN
3 WRITE ("Enter a number: ");
4 READ (number);
5 WRITELN ("Twice ", number,
6          " is ", number * 2)
7 END.
```

In this example program we place our declaration **before** the initial word BEGIN, as declarations are not part of our executable statements, but merely impart information to the G-Pascal compiler. The example also illustrates the use of the READ statement to read information into a variable while the program is running. As is usual practice the program first prompts the person running the program by preceding the READ statement (on line 4) with a WRITE statement (on line 3), asking for a number.

Note also the use of **commas** in the WRITELN statement on line 5. Whilst semicolons are used to separate **statements,** commas are used to separate **parameters** which are part of a single statement. In this case the WRITELN statement on lines 5 and 6 is a single statement, but it actually writes to the screen four separate items, namely the word "Twice", the value contained in NUMBER, the word "is", and the result of computing NUMBER times 2. Commas are used to separate these individual items. The use of commas in this way occurs frequently in G-Pascal.

The other data type supported by G-Pascal is the CHAR type. The only difference between INTEGER and CHAR is that integers are three bytes long, and can hold a value roughly between plus and minus eight million (in fact, +/- 8,388,607), however a CHAR variable only holds one byte, and thus can hold a value from 0 to 255. CHAR variables also behave differently when used in READ statements, as they accept characters from the keyboard rather than numbers.

Loops

The fundamental feature of computer programming that makes computers a powerful tool is the ability to execute the same pieces of code repetitively — this is called 'looping'. There are three major types of loops in computer programming in general, and in Pascal in particular, namely:

1. Looping a pre-defined number of times.
2. Repetitively testing a condition and then executing some code if the condition is true (pre-loop test).
3. Repetitively executing some code and then testing a condition, until the condition is true (post-loop test).

The first technique, looping a pre-defined

number of times, is useful if the program knows in advance of executing the loop how many times to go through the loop. This is implemented in Pascal with the FOR statement, as follows:

```
1 VAR counter : INTEGER;
2 BEGIN
3 FOR counter := 1 TO 10 DO
4   WRITELN (counter, " squared is ",
5            counter * counter)
6 END.
```

In this case the WRITELN statement is executed 10 times, with the value of "counter" starting at 1 and incrementing by 1 each time through the loop until it reaches the value 10.

The number of times the FOR loop is executed does not necessarily have to be known when the program is actually written, as this example shows:

```
 1 VAR counter, times : INTEGER;
 2 BEGIN
 3 WRITE
 4 ("Enter number of times to do it: ");
 5 READ (times);
 6 FOR counter := 1 TO times DO
 7   BEGIN
 8   WRITE (counter, " times 8 is ");
 9   WRITELN (counter * 8)
10   END
11 END.
```

This last example also illustrates a couple of other interesting points, one being the multiple declarations of variables in the single VAR declaration. In this case, both COUNTER and TIMES are declared as integers. In fact, any number of variables can be declared in this way — they are just separated by commas.

What happens if you have too many to fit on one line, you ask? Well, this is one of the wonderful things about Pascal — program lines are "free format", you can start a new line or add extra spaces (within reason) whenever you want. To illustrate with an extreme example:

```
1 BEGIN
2 WRITELN
3 (
4 "hello there"
5 )
6 END
7 .
```

In this example, each line has exactly one Pascal "symbol" on it (as recognised by the compiler). This program could not be broken down any more, although extra blank lines could be inserted. You cannot, for example, break up the word BEGIN into BEG IN as the compiler would recognise BEG and IN as two separate words. Similarly, a quoted string (as in line 4) must start and end on the same line, otherwise the compiler thinks you must have left out the closing quote symbol and would give you an error message.

The next sort of loop mentioned earlier is the "pre-loop test" loop, implemented in Pascal with the WHILE statement. In this case, the program tests the condition at the start of the loop, and then performs the loop if the condition is true. Then the condition is tested again, and so on. The following program illustrates this point:

```
1 BEGIN
2 WHILE MEMC [653] = 1 DO
3  WRITE ("hello ")
4 END.
```

The contents of location 653 will be "1" if the SHIFT key is pressed, so when you run this program the screen will quickly fill up with the word "hello", providing you hold down the SHIFT key as you press "R" to (R)un the program. As soon as you let go of the shift key the program will stop running. This is because the statement following the word "DO" is repeatedly executed while the condition (MEMC [653] = 1) is true. If you are not holding down the SHIFT key when you run the program then the word "hello" will not appear at all. This is because the condition test is done before the loop is executed. Contrast this to the REPEAT statement discussed below, in which the loop is always executed at least once.

Pascal uses the REPEAT statement to repetitively execute a loop, with the test for exiting the loop at the end of the loop — this is called the "post-loop test". Here is an example:

```
1 BEGIN
2 REPEAT
3   WRITE ("hello ")
4 UNTIL MEMC [653] = 0
5 END.
```

In this example also, the word "hello" appears as long as you hold down the SHIFT key. However, unlike the earlier example using the WHILE statement, the word "hello" will always appear at least once, even if you are not holding down the SHIFT key at all. This is because the test for leaving the loop is done at the end of the loop, not the start.

Here is another example:

```
1 BEGIN
2 REPEAT
3   WRITE ("hello ");
4   WRITE ("there ")
5 UNTIL 0
6 END.
```

The example above illustrates using the REPEAT statement to create an infinite loop. The value "0" in line 5 will never become true (the number zero is considered equivalent to the logical value "false" by G-Pascal) so the program will run indefinitely until you press the RUN/STOP key.

The example above also illustrates that you can have as many statements as you like between REPEAT and UNTIL, providing you separate them by semicolons, in a similar way to putting multiple statements between BEGIN and END to form a compound statement.

If you want to use multiple statements with a WHILE loop however, you must use BEGIN and END to indicate the extent of the loop, for example:

```
 1 VAR number : INTEGER;
 2 BEGIN
 3 number := 1;
 4 WHILE number <= 10 DO
 5   BEGIN
 6   WRITE (number, " squared is: ");
 7   WRITELN (number * number);
 8   number := number + 1
 9   END
10 END.
```

In this example the semicolons at the ends of lines 6 and 7 are statement separators. The entire compound statement bracketed by the BEGIN and END (in lines 5 and 9) is executed repetitively by the WHILE statement.

Procedures and Functions

The use of procedures and functions greatly adds to the power and flexibility of programming in Pascal. Broadly speaking, both procedures and functions allow blocks of code to be separated from the 'main' program, assigned a name, and executed when and where desired. There are two main reasons for doing this:

— To allow the same piece of code to be called (that is, executed) from more than one place in the program, thus saving unnecessary repetition; and

— To break the logical functions of a program up into named blocks of code, thus making the program easier to follow, and making debugging easier.

For example, in a game program you might have procedures called DISPLAY_SCORE and SHOOT_MISSILE. This means that if you want to display the game score from more than one place in the program, you only have to call the procedure 'DISPLAY_SCORE', rather than repeating the code in many different places. Also, the names 'DISPLAY_SCORE' and 'SHOOT_MISSILE' (if chosen carefully) imply what the procedure is attempting to do, without the need for further comments. This make the program easier to read and understand.

There are two steps to using procedures and functions.

The first is to 'declare' the procedure or function — that is, to tell the compiler what the procedure or function is going to do. This must be done before attempting to use it.

The second step is to 'invoke' the procedure or function. ('Invoke' literally means to 'call by name'). As the word 'invoke' implies, this is done by simply naming the procedure or function. The following example should make this clear:

```
 1 PROCEDURE say_hello;
 2 BEGIN
 3   WRITELN ("HELLO")
 4 END;
 5
 6 BEGIN
 7   say_hello;
 8   WRITELN ("HAVE A NICE DAY");
 9   say_hello
10 END.
```

The BEGIN on line 6 signals the start of the 'main' program. The procedure 'SAY_HELLO' is declared before the main program by writing the word PROCEDURE followed by the procedure name, in this case SAY_HELLO. The BEGIN on line 2 signals the start of the code for the procedure, and the END on line 4 indicates the end of the procedure.

Although the procedure is declared first, it does not actually execute until it is invoked — this is done at both lines 7 and 9 by simply writing its name.

Functions

The difference between procedures and functions is simply that a function returns a **value** and must be used in an expression, whereas a procedure does not return a value, and cannot be used in an expression. To illustrate this point, the following program calculates a table of Fahrenheit to Celsius temperatures, using an appropriate function:

```
 1 VAR temperature : INTEGER;
 2
 3 FUNCTION celsius (fahrenheit);
 4 BEGIN
 5  celsius := (fahrenheit - 32) * 5 / 9
 6 END;
 7
 8 BEGIN
 9 FOR temperature := 60 TO 80 DO
10   WRITELN (temperature, "F = ",
11         celsius (temperature), "C")
12 END.
```

This program illustrates a number of interesting points:

— Any CONST or VAR declarations that belong to the main program must appear **before** function and procedure declarations.
— A function or procedure may have parameters (numbers) passed to it — this is done by putting the parameters in parentheses both when the function or procedure is declarared (i.e. in line 3 above), and when the function or procedure is invoked (i.e. in line 11 above).
— The parameters do not have to have the same name — if there is more than one then they are matched by position, not name. In this case, the parameter is called TEMPERATURE in line 11, but inside the function it is referred to as FAHRENHEIT.
— As CELSIUS is a function it can appear inside a WRITELN statement, as in lines 10 and 11 above. A procedure invocation on the other hand can only appear on its own, not inside an expression.
— The function CELSIUS 'returns' the value that it has calculated in its own name. In the example above this happens on line 5.

Local variables

One of the very useful aspects of using procedures and functions is the ability to declare 'local' variables — these are variables that are only active while the procedure or function is exeucting. Variables which are declared inside a procedure or function take precedence over a variable of the same name declared outside that procedure or function. This means that even if you use a variable name inside a procedure or function that is the same as one outside that procedure or function then you do **not** change the contents of the outer variable when executing inside that procedure or function.

This will be illustrated in the following program:

```
 1 VAR x, y : INTEGER;
 2
 3 PROCEDURE calculate (a, b, c);
 4 VAR x : INTEGER;
 5 BEGIN
 6   x := a + b;
 7   y := x * c
 8 END;
 9
10 BEGIN
11   x := 1;
12   calculate (5, 6, 7);
13   WRITELN ("X = ", x, " Y = ", y)
14 END.
```

If you compile and run the above program you will see that it will display:

X = 1 Y = 77

This proves that the 'global' variable X

declared on line 1 is still containing the value 1 which it was given on line 11.

The variable X used in the procedure was declared on line 4 and is completely different from the variable X declared on line 1.

This program also illustrates that procedures can refer to variables which are already declared outside themselves. In the example above the procedure CALCULATE uses the local variable X (because there was a local declaration for X), but uses the 'global' variable Y (because there was no local declaration for Y).

The normal practice is to use global variables (that is, ones that are declared at the start of the program) for values that are to be shared between procedures, functions and the 'main' program. On the other hand, you would use local variables for intermediate results such as counters and other temporary values that are only needed while a procedure or function is executing.

By doing this, the placement of a variable declaration immediately implies to the reader what type of use the program is putting it to.

Local Procedures and Functions

It is also possible to declare procedures and functions within other procedures and functions — again this makes the inner procedure 'local' to the outer one. This would normally be done if the inner procedure or function performed a task directly related to the outer procedure and could therefore be logically part of the outer procedure. The reason for doing this would basically be to make the purposes and inter-relationships between the procedures more obvious. For example:

```
 1 PROCEDURE type_hello_there;
 2
 3 PROCEDURE type_hello;
 4 BEGIN
 5   WRITE   ("Hello")
 6 END;
 7
 8 PROCEDURE type_there;
 9 BEGIN
10   WRITELN (" there")
11 END;
12
13 BEGIN  (* 'type_hello_there' *)
14   type_hello;
15   type_there
16 END;
17
18 BEGIN  (* 'main' program *)
19   type_hello_there
20 END.
```

Anything between '(*' and '*)' is considered a comment and ignored by the compiler.

Although the example above is rather trivial it does illustrate the mechanics of defining procedures within other procedures. In fact the same concept can be taken further by nesting procedures even more deeply. For example, between lines 8 and 9 above, other CONST, VAR, PROCEDURE and FUNCTION declarations can appear, these being local to the 'type_there' procedure.

Making decisions — the IF statement

Most programs would not be interesting if they did not make at least some decisions — the fundamental decision-making statement in Pascal is the 'IF' statement. (Other ones are the CASE statement — discussed next, also decisions about loops are made by REPEAT and WHILE).

The simplest form of the IF statement is:

IF *condition* THEN *statement*

The 'condition' can be any legal Pascal expression — if the expression evaluates to zero it is considered 'false', if the condition evaluates to non-zero it is considered 'true'. If the condition is true then the statement following the word THEN is executed. For example:

```
1 VAR number : INTEGER;
2 BEGIN
3   REPEAT
4     WRITE ("Enter a number: ");
5     READ (number);
6     IF number = 5 THEN
7       WRITELN ("You entered 5!")
8   UNTIL number = 99
9 END.
```

Conditions are frequently expressed using the relational operators: = (equal), <> (not equal), < (less than), > (greater than), <= (less than or equal) and >= (greater than or equal).

The result of using a relational operator is always 1 (true) or 0 (false). Conditions can be expressed in other ways, however — for

example the statement:

```
IF 1 THEN WRITE ("hi!")
```

would always execute, or the statement:

```
IF a + b - c THEN
  WRITE ("Result is non-zero")
```

would do the WRITE if the result of the arithmetic was not zero.

A more advanced form of the IF statement includes an ELSE clause, as follows:

```
IF condition THEN
  statement1
ELSE
  statement2
```

In this case 'statement1' (the first statement) is executed if the condition is true, and 'statement2' (the second statement) is executed if the condition is false. Obviously the two statements are mutually exclusive, that is, they won't both be executed, but one of them will be depending on the condition evaluation.

In both cases the 'statement' can be a compound statement — namely a block of statements delimited by BEGIN and END, for example:

```
 1 VAR number : INTEGER;
 2 BEGIN
 3   REPEAT
 4     WRITE ("Enter a number: ");
 5     READ (number);
 6     IF number = 5 THEN
 7       BEGIN
 8         WRITELN ("You entered 5!");
 9         WRITELN ("Try again ...")
10       END
11     ELSE
12       BEGIN
13         IF number < 5 THEN
14           WRITELN
15     ("Your number was < 5")
16         ELSE
17           WRITELN
18     ("Your number was > 5")
19       END
20   UNTIL number = 99
21 END.
```

In the example above both the 'true' and 'false' portions of the first IF statement are compound statements (using BEGIN and END). The second IF statement (line 13) does not use compound statements, and thus only a single statement is executed for the 'true' and 'false' portions.

It is interesting to see how it is possible to 'nest' a second IF statement inside another one.

The CASE statement

The CASE statement is a very powerful way of selection a course of action out of a list of possibilities. The CASE statement is a sort of 'extended IF'. It is particularly useful if you want to decide on a different course of action for a series of possibilities. An example of this might be a 'Star Trek' game, where you might enter M (Move), T (Torpedo), P (Phasor), S (Shields) and E (Energy). This could be coded as follows:

```
WRITE ("Enter action: ");
READ (action);
WRITELN (CHR(action));  (* echo it *)
CASE action OF
  "M", "m" : move_enterprise;
  "T"      : fire_torpedo;
  "P"      : fire_phasor;
  "S"      : set_shields;
  "E"      : change_energy
ELSE
  WRITELN ("Enter M, T, P, S or E")
END
```

The above is not a complete program — it assumes that procedures have been declared to accomplish the various actions (for example, MOVE_ENTERPRISE). It also assumes that ACTION has been declared as a variable of type CHAR.

The above example illustrates a number of points:

— More than one choice can be associated with a particular action — in the example above both 'M' and 'm' would invoke MOVE_ENTERPRISE. If more than one 'selector' is used they are separated by commas.

— Each choice executes a single statement, which must be followed by a semicolon, except for the last one. To do more than one thing, either use a compound statement or call a procedure.

— An ELSE clause can be optionally used to accomplish a default action if none of the previous selections are met.

— The final END is part of the CASE statement and is required, either after the statement following the ELSE, or following the final selection if ELSE is not used.

Obviously the CASE statement used in this way is very flexible — adding more commands to this game would be a simple exercise, namely writing the procedure to handle the command, and adding the command letter to the CASE statement.

Notice the (reasonably) helpful error message displayed if you choose an invalid action. Programs should always attempt to help the person running them as much as possible. Saying "Enter M, T, P, S or E" is more helpful than saying "Invalid action", but is less helpful than (for example): "Enter M — Move, T — Torpedo, P — Phasor, S — Shields or E — Energy". As far as practical the program should let the user know what is expected of him or her, detect invalid responses, and reply where appropriate with a suitably helpful response.

Next Issue . . .

Next issue we intend to cover more complex aspects of G-Pascal programming, such as independent modules, use of Trace and Debug, and any other areas that readers enquire about over the next month.

USING G-PASCAL TO PRINT DOLLARS AND CENTS

Although G-Pascal does not have a 'floating-point' data type as such, it is possible to utilise the fairly long integer size to simulate numbers with decimal points, as the program on the right illustrates.

Let us assume that you want to use G-Pascal to store some financial data (balance the cheque book perhaps?). By storing values internally as cents, and using the PRINT_AMT procedure on the right you can print out your results, including the decimal point, right-justified to a specified column width (LENGTH) — so that columns of figures will line up.

The procedure PRINT_AMT is self-contained, apart from assuming that TRUE and FALSE have been declared at the start of the program in the usual way.

Lines 49 to 58 in the program are just an example of using this procedure.

```
 1 (*                              *)
 2 (*  Prints a right justified    *)
 3 (*  dollar amount.              *)
 4 (*                              *)
 5 (*  Written by Sue Gobbett.     *)
 6 (*                              *)
 7
 8 CONST
 9   true=1; false=0;
10   clear_screen = 147;
11 VAR i :INTEGER ;
12
13 PROCEDURE print_amt (amount,
14 (*****************) length);
15
16 VAR i,j,print_sign : INTEGER ;
17   result : ARRAY [20] OF CHAR ;
18 BEGIN
19 IF amount < 0 THEN
20   print_sign := true;
21 FOR i := 0 TO length - 1 DO
22   BEGIN
23   IF (ABS (amount) = 0) AND
24      (i > 3) THEN
25     IF print_sign THEN
26       BEGIN
27       result[i] := "-";
28       print_sign := false
29       END
30     ELSE
31       result[i] := " "
32   ELSE
33   IF i = 2 THEN
34     result[i] := "."
35   ELSE
36   IF i = 6 THEN
37     result[i] := ","
38   ELSE
39     BEGIN
40     result[i] := "0" +
41       ABS (amount MOD 10);
42     amount := amount / 10
43     END
44   END ;
45 FOR i := length - 1 DOWNTO 0 DO
46   WRITE (CHR (result[i]))
47 END ;
48
49 BEGIN
50 WRITE (CHR (clear_screen));
51 REPEAT
52   WRITELN ;
53   READ (i);
54   CURSOR (CURSORY - 1,10);
55   IF i <> 0 THEN
56     print_amt(i,15)
57 UNTIL i = 0
58 END .
```

SPELLING CHECKER by Adrian Winn

[This program and its operating instructions was submitted by a G-Pascal owner from New South Wales, Adrian Winn. We are pleased to reprint it as an example of a useful program, and also to demonstrate how to do string handling in G-Pascal. — Ed.]

This spelling program was only written for practice at Pascal programming, but it might help people struggling with programs using disk access and strings.

Operating options:

(A)dd list —

This option will clear any previous lists in memory and replace with your new list. Enter your words when prompted. Enter a '*' when finished.

(E)dit list —

This option will present you with your current list to correct mistakes, add words and delete words. When the word appears make any changes to the word and hit RETURN. Clear the word and hit RETURN to delete it. Type a '*' when finished.

(S)ave list —

Saves current list to disk or cassette using the supplied filename.

(L)oad list —

Loads a previously saved list from disk or cassette using the supplied filename.

(T)est —

This option tests you on the words of the current list. By pressing any key a word from the list will flash on the screen. After you type in your guess either 'right' or 'wrong' will appear. The program will go onto the next word if your guess was right. If your guess was wrong you will be asked the word again — this will repeat until you get it right or it takes you more than three tries. Your score and high score are displayed at the top of the screen.

(P)rintout list —

This gives a hard copy of the current list.

```
 1 (*   spelling program      *)
 2 (*                          *)
 3 (*   author: Adrian Winn    *)
 4 CONST
 5   color=1;  point=2;
 6   multi=3;  expandx=4;
 7   expandy=5; behind=6;
 8   active=7;on=1;off=0;
 9   areg=$2b2;
10   xreg=$2b3;
11   yreg=$2b4;
12   cc=$2b1;
13   setlfs=$ffba;
14   setnam=$ffbd;
15   loadrout=$ffd5;
16   saverout=$ffd8;
17   register=$6a;
18   (* graphics symbols used *)
19   bar = $c0;
20   inv = $12;
21   norm = $92;
22   cnr1 = $b0;
23   cnr2 = $ae;
24   cnr3 = $ad;
25   cnr4 = $bd;
26   hbar = $dd;
27   clear_screen = 147;
28 VAR
29   temp_word,
30   next_word : ARRAY [20] OF CHAR ;
31   word_list : ARRAY [410] OF CHAR ;
32   medium,
33   i,
34   temp_score,
35   score,
36   hiscore      : INTEGER ;
37   t            : CHAR ;
38
39 PROCEDURE sprite_setup;
40 (*********************)
41
42 BEGIN
43 DEFINESPRITE (135,
44   $0fc3f0, $186618, $102408,
45   $000000, $0781e0, $0fc3f0,
46   $0fc3f0, $0ec3b0, $0cc330,
47   $048120, $0300c0, $000000,
48   $000000, $100008, $080010,
49   $060060, $0300c0, $01e780,
50   $003800);
51 DEFINESPRITE (136,
52   $000000, $000000, $0781e0,
53   $000000, $0781e0, $0fc3f0,
54   $0fc3f0, $0fc3f0, $0cc330,
55   $048120, $0300c0, $000000,
56   $000000, $003800, $00ee00,
57   $030180, $0600c0, $080020,
58   $100010);
```

```
 59 SPRITE (1,expandx,on,
 60         1,expandy,on);
 61 POSITIONSPRITE (1,200,180)
 62 END ;
 63
 64 PROCEDURE menu;
 65 (*************)
 66
 67 VAR i : INTEGER ;
 68 BEGIN
 69 WRITE (CHR (clear_screen));
 70 CURSOR (1,13);
 71 WRITELN ("Pascal Speller");
 72 CURSOR (2,13);
 73 FOR i := 1 TO 14 DO
 74 WRITE (CHR (bar));
 75 CURSOR (5,5);
 76 WRITELN ("<A>dd list");
 77 CURSOR (7,5);
 78 WRITELN ("<S>ave list");
 79 CURSOR (9,5);
 80 WRITELN ("<L>oad list");
 81 CURSOR (11,5);
 82 WRITELN ("<T>est");
 83 CURSOR (13,5);
 84 WRITELN ("<E>dit list");
 85 CURSOR (15,5);
 86 WRITELN ("<Q>uit program");
 87 CURSOR (17,5);
 88 WRITELN ("<P>rintout list");
 89 CURSOR (20,3);
 90 WRITELN (">");
 91 REPEAT
 92   CURSOR (20,4);
 93   READ (t)
 94 UNTIL (t="a")
 95    OR (t="s")
 96    OR (t="l")
 97    OR (t="q")
 98    OR (t="p")
 99    OR (t="e")
100    OR (t="t")
101 END ; (* of menu *)
102
103 PROCEDURE press_any_key;
104 (**********************)
105
106 VAR i :CHAR ;
107 BEGIN
108 CURSOR (22,14);
109 WRITE (CHR (inv),"Press Any Key");
110 WRITELN (CHR (norm));
111 REPEAT
112 UNTIL GETKEY
113 END ;
114
115 PROCEDURE loadit (len,addr);
116 (************************)
117
118 BEGIN
119 MEMC [areg] := 1;
120 MEMC [xreg] := medium;
121 MEMC [yreg] := 0;
122 CALL (setlfs);
123 MEMC [areg] := len;
124 MEMC [xreg] :=
125     ADDRESS (temp_word[len]);
126 MEMC [yreg] :=
127     ADDRESS (temp_word[len])SHR 8;
128 CALL (setnam);
129 MEMC [areg] := 0;
130 MEMC [xreg] := addr;
131 MEMC [yreg] := addr SHR 8;
132 CALL (loadrout)
133 END ;
134
135 PROCEDURE saveit (len,start,
136                    finnish);
137 (************************)
138
139 BEGIN
140 MEMC [areg] := 1;
141 MEMC [xreg] := medium;
142 MEMC [yreg] := 0;
143 CALL (setlfs);
144 MEMC [areg] := len;
145 MEMC [xreg] :=
146    ADDRESS (temp_word[len]);
147 MEMC [yreg] :=
148    ADDRESS (temp_word[len])SHR 8;
149 CALL (setnam);
150 MEMC [register] := start;
151 MEMC [register + 1] := start SHR 8;
152 MEMC [areg] := register;
153 MEMC [xreg] := finnish;
154 MEMC [yreg] := finnish SHR 8;
155 CALL (saverout)
156 END ;
157
158 PROCEDURE load_init;
159 (******************)
160
161 VAR len,i,addr :INTEGER ;
162 BEGIN
163 WRITE (CHR (clear_screen));
164 CURSOR (5,1);
165 WRITE ("Filename>*");
166 CURSOR (7,1);
167 WRITELN ("<return> to abort...");
168 CURSOR (5,10);
169 READ (next_word);
170 IF next_word[0] <> "*" THEN
171   BEGIN
172   len := 0;
173   REPEAT
174     len := len + 1
```

Note: Page 12 missing from archival copy.

```
291   WRITELN ("Spelling List.");
292   WRITELN ("--------------");
293   WRITELN ;
294   pull_word(i);
295   REPEAT
296     WRITE ((i + 1),">");
297     print_word;
298     WRITELN ;
299     i := i + 1;
300     pull_word(i)
301   UNTIL (next_word[0]="*")
302         OR (i>19);
303   WRITELN ;
304   WRITELN ("----------------");
305   PUT (0);
306   CLOSE (2)
307   END
308 END ;
309
310 PROCEDURE delete(word);
311 (*********************)
312
313 VAR
314   i,t :INTEGER ;
315 BEGIN
316 i := word;
317 REPEAT
318   FOR t := 0 TO 19 DO
319     word_list[(i*20) + t] :=
320       word_list[((i + (1))*20)+ t];
321   i := i + 1
322 UNTIL (i=20)
323    OR (word_list[(i*20)] = "*");
324 word_list[(i - 1)*20] := "*"
325 END ;
326
327 PROCEDURE add_list;
328 (******************)
329
330 VAR
331   i,counter :INTEGER ;
332 BEGIN
333 WRITE (CHR (clear_screen));
334 counter := 0;
335 clear_mem;
336 REPEAT
337   CURSOR (8,1);
338   WRITE ("input word ",
339          counter + (1)," >");
340   input;
341   CURSOR (8,12);
342   WRITELN
343     ("[26 spaces]");
344   push_word(counter);
345   counter := counter + 1
346 UNTIL (counter=20)
347    OR (temp_word[0]="*");
348 hiscore := 0;
349 press_any_key
350 END ;
351
352 PROCEDURE edit_list;
353 (******************)
354
355 VAR exflg,i,t,counter :INTEGER ;
356 BEGIN
357 counter := 0;
358 WRITE (CHR (clear_screen));
359 CURSOR (1,8);
360 WRITELN ("Edit List");
361 CURSOR (2,8);
362 FOR i := 1 TO 9 DO
363   WRITE (CHR (bar));
364 WRITELN ;
365 exflg := 1;
366 WHILE exflg DO
367   BEGIN
368   CURSOR (5,1);
369   WRITELN ("word ",counter + (1));
370   CURSOR (6,1);
371   WRITELN ("[15 spaces]");
372   CURSOR (6,1);
373   pull_word(counter);
374   IF (next_word[0]<>0) AND
375      (next_word[0]<>"*") THEN
376     print_word;
377   CURSOR (6,1);
378   input;
379   IF counter=19 THEN
380     exflg := 0 ;
381   CASE temp_word[0] OF
382     " " :delete(counter);
383     "*" :exflg := 0
384   ELSE
385     BEGIN
386     push_word(counter);
387     counter := counter + 1
388     END
389   END
390 END ;
391 press_any_key
392 END ;
393
394 PROCEDURE right;
395 (***************)
396
397 VAR t :INTEGER ;
398 BEGIN
399 SPRITE (1,point,135,
400         1,color,1);
401 FOR t := 1 TO 4 DO
402   BEGIN
403   SPRITE (1,active,on);
404   SOUND (3,10);
405   SPRITE (1,active,off);
406   SOUND (3,10);
```

```
407   END ;
408 CURSOR (18,1);
409 WRITELN ("Right.");
410 SOUND (3,90);
411 CURSOR (18,1);
412    WRITELN ("[6 spaces]");
413 score := score + 1;
414 CURSOR (6,12);
415 WRITE (score);
416 temp_score := 0;
417 i := i + 1
418 END ;
419
420 PROCEDURE wrong(t);
421 (*****************)
422
423 VAR counter :INTEGER ;
424 BEGIN
425 SPRITE (1,point,136,
426         1,color,1);
427 FOR counter := 1 TO 4 DO
428   BEGIN
429   SPRITE (1,active,on);
430   SOUND (3,10);
431   SPRITE (1,active,off);
432   SOUND (3,10)
433   END ;
434 CURSOR (18,1);
435 WRITELN ("wrong");
436 CURSOR (20,1);
437 WRITE ("Word was>");
438 print_word;
439 CURSOR (17,12 + t);
440 WRITE ("↑");
441 SOUND (3,200);
442 CURSOR (18,1);
443 WRITELN ("[5 spaces]");
444 CURSOR (20,1);
445 WRITELN
446   ("[20 spaces]",
447    "[19 spaces]");
448 CURSOR (17,1);
449 WRITELN
450   ("[20 spaces]",
451    "[19 spaces]");
452 temp_score := temp_score + 1;
453 IF temp_score=3 THEN
454   BEGIN
455   temp_score := 0;
456   i := i + 1;
457   END
458 ELSE
459   BEGIN
460   CURSOR (19,1);
461   WRITELN ("TRY AGAIN");
462   SOUND (3,95);
463   CURSOR (19,1);
464   WRITELN ("[9 spaces]")
465   END
466 END ;
467
468 FUNCTION checkit;
469 (***************)
470
471 VAR
472   i :INTEGER ;
473 BEGIN
474 i := 0;
475 WHILE (temp_word[i]
476        =next_word[i])
477   AND (next_word[i] <> 13) DO
478   i := i + 1;
479   IF (next_word[i]=13) AND
480      (temp_word[i]=13) THEN
481     checkit := 255
482   ELSE
483     checkit := i
484 END ;
485
486 PROCEDURE scr_setup;
487 (******************)
488
489 VAR i : INTEGER ;
490 BEGIN
491 WRITE (CHR (clear_screen));
492 CURSOR (1,14);
493 WRITE (CHR (cnr1));
494 FOR i := 1 TO 6 DO
495   WRITE (CHR (bar));
496 WRITE (CHR (cnr2));
497 CURSOR (2,14);
498 WRITE (CHR (hbar)," Test ");
499 WRITELN (CHR (hbar));
500 CURSOR (3,14);
501 WRITE (CHR (cnr3));
502 FOR i := 1 TO 6 DO
503   WRITE (CHR (bar));
504 WRITE (CHR (cnr4));
505 CURSOR (5,17);
506 WRITE (CHR (cnr1));
507 FOR i := 1 TO 19 DO
508   WRITE (CHR (bar));
509 WRITE (CHR (cnr2));
510 CURSOR (6,17);
511 WRITE (CHR (hbar));
512 WRITE (" Previous Best >[3 spaces]");
513 WRITELN (CHR (hbar));
514 CURSOR (7,17);
515 WRITE (CHR (cnr3));
516 FOR i := 1 TO 19 DO
517   WRITE (CHR (bar));
518 WRITE (CHR (cnr4));
519 CURSOR (5,2);
520 WRITE (CHR (cnr1));
521 FOR i := 1 TO 12 DO
522   WRITE (CHR (bar));
```

```
523 WRITE (CHR (cnr2));
524 CURSOR (6,2);
525 WRITE (CHR (hbar));
526 WRITE (" Score[2 spaces]>[3 spaces]");
527 WRITELN (CHR (hbar));
528 CURSOR (7,2);
529 WRITE (CHR (cnr3));
530 FOR i := 1 TO 12 DO
531   WRITE (CHR (bar));
532 WRITE (CHR (cnr4));
533 CURSOR (14,1);
534 WRITE ("Word is>")
535 END ;
536
537 PROCEDURE test;
538 (*************)
539
540 VAR
541   t :INTEGER ;
542 BEGIN
543 i := 0;
544 temp_score := 0;
545 pull_word(i);
546 score := 0;
547 scr_setup;
548 CURSOR (6,34);
549 WRITE (hiscore);
550 REPEAT
551   CURSOR (16,12);
552   WRITELN ("[13 spaces]");
553   CURSOR (10,10);
554   WRITELN ("Press any key for");
555   CURSOR (12,13);
556   WRITELN ("Word > ",i + 1);
557   REPEAT
558   UNTIL GETKEY ;
559   CURSOR (10,10);
560   WRITELN ("[17 spaces]");
561   CURSOR (12,13);
562   WRITELN ("[10 spaces]");
563   CURSOR (14,1);
564   WRITE ("Word is>");
565   print_word;
566   SOUND (3,42);
567   CURSOR (14,9);
568   WRITELN ("[19 spaces]");
569   CURSOR (16,1);
570   WRITE ("Your guess>");
571   input;
572   t := checkit;
573   IF t=255 THEN
574     right
575   ELSE
576     wrong(t);
577   pull_word(i)
578 UNTIL (next_word[0]="*")
579    OR (i=20);
580 IF score > hiscore THEN
581   hiscore := score;
582 CURSOR (6,34);
583 WRITE (hiscore);
584 press_any_key
585 END ;
586
587 (* MAIN PROGRAM ***************)
588
589 BEGIN
590 sprite_setup;
591 hiscore := 0;
592 WRITE (CHR (clear_screen));
593 CURSOR (10,1);
594 WRITELN ("Using disk (Y/N)?");
595  REPEAT
596   CURSOR (10,18);
597   READ (t)
598 UNTIL (t="y")
599    OR (t="n");
600 IF t= "y" THEN
601   medium := 8
602 ELSE
603   medium := 1;
604 REPEAT
605   menu;
606   CASE t OF
607     "a" :add_list;
608     "e" :edit_list;
609     "t" :test;
610     "l" :load_init;
611     "s" :save_init;
612     "p" :printout
613     END
614 UNTIL t="q"
615 END .
```

Please note that on lines that have a space count inside quote symbols (for example line 343, where the listing says [26 spaces]) you should type the nominated number of spaces. This is done to save you the trouble of trying to count the number of spaces that appear within the quotes.

As mentioned in the previous issue, do not bother typing in reserved words in upper case — they are printed that way to make the program easier to read. Also be careful to distinguish between the letter 'O' and the number zero.

The symbol on line 440 is the 'up-arrow', on the keyboard to the left of the RESTORE key.