STRUCTURE OF A PASCAL PROGRAM

The general structure of a Pascal program is:

program name;

declaration part

begin
    main program statements
end.

• It is optional to begin your program with the keyword 'Program', followed by your program name. name is the name the programmer gives to the Pascal program being written. It is an identifier. Identifiers begin with a letter, then followed by any digit, letter or the underscore character (_).

Quiz: Which of the following are valid Pascal identifiers?

<table>
<thead>
<tr>
<th>birthday</th>
<th>Too_hot?</th>
<th>First_Init</th>
<th>grade</th>
<th>1stprogram</th>
<th>down.to.earth</th>
</tr>
</thead>
<tbody>
<tr>
<td>see you</td>
<td>Old_Name</td>
<td>case</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• The declaration part consists of
  - const definition of constants;
  - type definitions of types;
  - var definitions of variables;
  - definitions of functions and procedures

‘Var’ declares variables that you will use in your program.

'Type' declares any variable structures - explained later.

'Const' declares any constant values to use throughout your program. These are anything which is always the same, such as the number of days in the week. Alternatively if you use a set value throughout your program, it is a good idea to make this a constant value so that it can easily be changed if you later decide to do so.

The 'Uses' keyword allows your program to use extra commands. These extra commands are stored together in what is called a module or library. These modules have names such as CRT, or GRAPH. Each modules contains several extra commands. These commands will be related in some way. Eg. GRAPH contains extra commands to do with graphics. CRT contains many general commands. (Even though CRT stands for Cathode Ray Tube - ie. the screen)
• After this, comes the keyword 'begin'. This indicates the beginning of the main part of your program.
• After this, comes the program code (the main program statements). Program statements are commands or instructions to the computer which perform various tasks. All program statements are terminated with a semi-colon, except the ‘begin’ and ‘end’ keywords. Program statements preceding an end statement do not require a semi-colon.
• The end of the program is indicated by the keyword ‘end.’ Note the full stop after the word 'end'.

Comments

It is a good idea to comment your code so you can understand what it is doing if you look at it later. It is also important so that other people can understand it also.

In Pascal you can comment your code in two different ways.

Either:

```pascal
{ to start the comment and } to end the comment or

(* to start the comment and *) to end the comment.
```

cg.

```pascal
Program Hello;     {This line is optional}
(*This program prints Hello on the screen*)
begin
    writeln('Hello');
    readln;{This holds the output screen until you press enter}
end.
```

The keyword `writeln` writes to the console screen. The text to be displayed is written inside `single quotes`. After printing the text inside the single quotes, the cursor is positioned to the beginning of the next line.

---

**EXERCISE:**

Write a program to print the following words on the console screen.

Hello. How are you?
I'm just fine.
PASCAL VARIABLES AND DATA TYPES

Pascal Variables
Variables store values and information. They allow programs to perform calculations and store
data for later retrieval. Variables store numbers, names, text messages, etc.
A Data Type defines the type of data that will be stored in a variable.

Every variable must be assigned a data type and given a unique name (identifier). Declaration of variables must be done in the VAR heading section.

Format:

Identifier : data-type;

Overview of Pascal Data Types:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type of Data</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Holds Text</td>
<td>'New York', 'Evan'</td>
</tr>
<tr>
<td>Integer</td>
<td>Holds whole numbers</td>
<td>3, 6, 1024, -90, 0</td>
</tr>
<tr>
<td>Real</td>
<td>Holds Decimal Numbers</td>
<td>3.14, 503.2, -3.55, 0.0, 35.997E+11 (Here, the symbol E stands for 'times 10 to the power of')</td>
</tr>
<tr>
<td>Boolean</td>
<td>Holds True or False</td>
<td>TRUE, FALSE</td>
</tr>
<tr>
<td>Character</td>
<td>Holds a single character</td>
<td>'A', 'E', '{', '}', '=', '$'</td>
</tr>
</tbody>
</table>

Strings: When declaring a string variable, you usually indicate its maximum length (1-255). For example, to create a variable called City that can hold up to 25 characters, you would type: city: String[25]; If you do not include a maximum size, the string is given a default maximum size = 255.

Variable Ranges

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Minimum Value</th>
<th>Maximum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>-32,768</td>
<td>32,767</td>
</tr>
<tr>
<td>LongInt</td>
<td>-2,147,483,648</td>
<td>2,147,487,647</td>
</tr>
<tr>
<td>ShortInt</td>
<td>-128</td>
<td>128</td>
</tr>
<tr>
<td>Real</td>
<td>2.9 x 10 E-39</td>
<td>1.7 x 10 E+38</td>
</tr>
</tbody>
</table>
Rules for Identifier Names

All identifier names, including program names and variable names must follow the following rules:

- Must begin with a letter or underscore (_)
- Can only contain letters, numbers, or underscore (_)
- Cannot contain any blank spaces

Examples of Illegal Identifiers: 2Length, AB*3, Len gth.

An example of declaring several variables is:

```pascal
var
  age, year, grade : integer;
  circumference : real;
  LetterGrade : char;
  DidYouFail : Boolean;
```
CONSTANTS

Constants are referenced by identifiers, and can be assigned one value at the beginning of the program. The value stored in a constant cannot be changed.

Constants are defined in the constant section of the program:

```pascal
const
  Identifier1 = value;
  Identifier2 = value;
  Identifier3 = value;
```

For example, let's define some constants of various data types: strings, characters, integers, reals, and Boolean.

```pascal
const
  Name = 'Tao Yue';
  FirstLetter = 'a';
  Year = 1997;
  pi = 3.1415926535897932;
  UsingNCSAMosaic = TRUE;
```

Note that in Pascal, characters are enclosed in single quotes, or apostrophes ('')!

Constants are useful for defining a value which is used throughout your program but may change in the future. Instead of changing every instance of the value, you can change just the constant definition.

Typed constants force a constant to be of a particular data type. For example,

```pascal
const
  a : real = 12;
```

would yield an identifier `a` which contains a real value `12.0` instead of the integer value `12`. 
BASIC INPUT AND OUTPUT

As you probably know, nearly all computer programs take input from the user.

Pascal has two major functions for taking input from the user. These are:-

1. **read**

   **Syntax:**
   
   ```pascal
   read (variable);
   ```

   **Explanation:**
   This reads all the characters typed, until the user presses enter, into the variable.
   If the variable is of type integer, and the user types in string characters, then an error will occur. If the variable is a string of defined length then read will only take the first X characters from the line and put them into the string, where X is the size of the string. Read does not move the cursor to the next line after input.

2. **readln**

   **Syntax:**
   
   ```pascal
   readln (variable);
   ```

   **Explanation:**
   This is exactly the same as read except for the fact that it moves the cursor to the next line after the user presses enter.

The output commands in pascal are very similar in syntax to the input commands.

1. **write**

   **Syntax:**
   
   ```pascal
   write (variable);
   write (variable:f)
   write (real variable:f:d);
   ```

   **Explanation:**
   The write command displays a string of characters on the screen. When a field width is included,
the writing is right aligned within the field width e.g. `write ('Hello':10);` will produce the following output...

```
00000Hello
```

(0=space)

Notice that 'Hello' is right-aligned within the field of ten characters, the remaining spaces coming before 'Hello'.

When writing real numbers, you must specify the field width and number of decimal places displayed, otherwise Pascal will write it to the screen in standard form (this is not good). A field width of zero will just write the real as if you had not specified a field width.

If you want to write a combination of things, separate these by a comma.

e.g. `write ('Hello ' , name , ', you weigh ' , weight:0:2 , ' kg.');`

The above will write something like this... Hello Joe Bob, you weigh 76.54 kg.

**Note:** The `write` command does not move the cursor to the next line after execution.

### writeln

**Syntax:**

```
writeln (variable);
writeln (variable:f)
writeln (real variable:f:d);
```

**f=field width d=number of decimal places**

**Explanation:**
The `writeln` command is exactly the same as the `write` command except for the fact that it moves the cursor to the next line after execution.
EXAMPLE PROGRAM

Program example;
{This is an example program for input and output}

uses Crt;
var

    name : string[30];

begin
    clrscr;            {This clears the screen}
    write ('What is your name? '); {Writes the question without
    writeln (name);    {take input from user }
    writeln ('Hello ', name); {Output Hello joebob}
    readln;            {waits for a key to be pressed}
end.
Pascal can do many mathematical operations. They are all relatively simple and easy to remember.

The first thing to note is that Pascal uses `:=` not `=` to assign a value to a variable. 

\[ x := 3; \]

Once you have declared a variable, you can store values in it. This is called assignment.

To assign a value to a variable, follow this syntax:

```
variable_name := expression;
```

Note that Pascal uses a colon followed by an equals sign, similarly to how it's done in most computer algebra systems.

The expression can either be a single value:

```
some_real := 385.385837;
```

or it can be an arithmetic sequence:

```
some_real := 37573.5 * 37593 + 385.8 / 367.1;
```

The arithmetic operators in Pascal are:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Operands</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition or unary positive</td>
<td>real or integer</td>
<td>real or integer</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction or unary negative</td>
<td>real or integer</td>
<td>real or integer</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>real or integer</td>
<td>real or integer</td>
</tr>
<tr>
<td>/</td>
<td>Real division</td>
<td>real or integer</td>
<td>real</td>
</tr>
<tr>
<td>div</td>
<td>Integer division</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>mod</td>
<td>Modulus (remainder division)</td>
<td>integer</td>
<td>integer</td>
</tr>
</tbody>
</table>

`div` and `mod` only work on integers. `/` works on both reals and integers but will always yield a real answer. The other operations work on both reals and integers. When mixing integers and reals, the result will always be a real since data loss would result otherwise. This is why Pascal uses two different operations for division and integer division. \( 7 / 2 = 3.5 \) (real), but \( 7 \text{ div } 2 = 3 \) (and \( 7 \text{ mod } 2 = 1 \) since that's the remainder).
Each variable can only be assigned a value that is of the same data type. Thus, you cannot assign a real value to an integer variable. However, certain data types will convert to a higher data type. This is most often done when assigning integer values to real variables. Suppose you had this variable declaration section:

```pascal
var
  some_int : integer;
  some_real : real;
```

When the following block of statements executes,

```pascal
some_int := 375;
some_real := some_int;
```

`some_real` will have a value of 375.0.

In Pascal, the minus sign can be used to make a value negative. The plus sign can also be used to make a value positive, but is typically left out since values default to positive.

Do not attempt to use two operators side by side, like in:

```pascal
some_real := 37.5 * -2;
```

This may make perfect sense to you, since you're trying to multiply by negative-2. However, Pascal will be confused — it won't know whether to multiply or subtract. You can avoid this by using parentheses to clarify:

```pascal
some_real := 37.5 * (-2);
```

**Operator Precedence**

The following precedence table applies to Real and Integer operators. Parentheses always have the *highest* priority, and operators of equal priority are always performed left to right.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>* / DIV MOD</td>
</tr>
<tr>
<td>Second</td>
<td>+ -</td>
</tr>
</tbody>
</table>

The computer follows an order of operations similar to the one that you follow when you do arithmetic. Multiplication and division (* / div mod) come before addition and subtraction (+ -), and parentheses always take precedence. So, for example, the value of: 3.5*(2+3) will be 17.5.

For example: To evaluate 4.0 + 6.0 * 3.0, you must follow the precedence table and perform the multiplication first, and the addition second. Hence, evaluating this expression will always give
an answer of 22.0. If you want to perform the addition first, you must include the 4.0 + 6.0 within parentheses: (4.0 + 6.0) * 3.0.

Pascal cannot perform standard arithmetic operations on Booleans. There is a special set of Boolean operations. Also, you should not perform arithmetic operations on characters.

---

**Mixed Mode**

If you use integers and reals together, integers are automatically converted to reals. For example, if you have the following expression:

```
Profit := (3.0 * 4.0)/2
```

the 2 is automatically converted to 2.0 and the result is equal to 6.0.

---

**Mathematical functions**

**SQR**

**Syntax:**

```
SQR(Real Variable)
```

**Explanation:**
SQR returns the square of the real variable that is passed to it, pretty simple really.

**Example:**

```
x := SQR(y);
```

This finds the square of y and puts the result in x.

---

**SQRT**

**Syntax:**

```
SQRT(Real Variable)
```

**Explanation:**
SQRT returns the square root of the real variable that is passed to it, pretty simple really.

**Example:**

```
x := SQRT(y);
```

This finds the square root of y and puts the result in x.
SIN

Syntax:

\[ \text{SIN(Real variable)} \]

Explantation:
SIN returns the sin of the number that is passed to it. Unfortunately this is in radians (stupid radians).
2\(\pi\) radians is equal to 360 degrees, so to convert from degrees to radians it is \(\text{degrees/180} \times \pi\), and from radians to degrees it is \(\text{radians/}\pi \times 180\).

Example:
\[ x := \text{SIN}(y); \]
This finds the sin of \(y\) (radians) and puts the value in \(x\).

COS

Syntax:

\[ \text{COS(Real variable)} \]

Explantation:
COS returns the cos of the number that is passed to it. This is also in radians. If you want to know how to convert radians into degrees and vice-versa then read the explanation of SIN.

Example:
\[ x := \text{COS}(y); \]
This finds the cos of \(y\) (radians) and puts the value in \(x\).

ARCTAN

Syntax:

\[ \text{ARCTAN(Real variable)} \]

Explantation:
ARCTAN returns the inverse tangent of the number that is passed to it. It returns the angle in radians.
Example:
\[ x := \text{ARCTAN}(y); \]
This finds the inverse tangent, in radians, of \( y \) and puts the value in \( x \).

**Finding TANGENT**

To find tangent just divide \( \sin(Y) \) by \( \cos(Y) \).
\[ e.g \quad x := \frac{\sin(y)}{\cos(y)}; \] finds the tangent of \( y \) and puts it in \( x \) (remember radians).

Functions are called by using the function name followed by the argument(s) in parentheses.

For example, to find the value of \( \sin \) of \( \pi \) radians:
\[ \text{value} := \sin(3.1415926535897932); \]

Standard Pascal functions include:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Argument type</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>absolute value</td>
<td>real or integer</td>
<td>same as argument</td>
</tr>
<tr>
<td>arctan</td>
<td>arctan in radians</td>
<td>real or integer</td>
<td>real</td>
</tr>
<tr>
<td>cos</td>
<td>cosine of a radian measure</td>
<td>real or integer</td>
<td>real</td>
</tr>
<tr>
<td>exp</td>
<td>( e ) to the given power</td>
<td>real or integer</td>
<td>real</td>
</tr>
<tr>
<td>ln</td>
<td>natural logarithm</td>
<td>real or integer</td>
<td>real</td>
</tr>
<tr>
<td>round</td>
<td>round to nearest integer</td>
<td>real</td>
<td>integer</td>
</tr>
<tr>
<td>sin</td>
<td>sin of a radian measure</td>
<td>real or integer</td>
<td>real</td>
</tr>
<tr>
<td>sqr</td>
<td>square (power 2)</td>
<td>real or integer</td>
<td>same as argument</td>
</tr>
<tr>
<td>sqrt</td>
<td>square root (power 1/2)</td>
<td>real or integer</td>
<td>real</td>
</tr>
<tr>
<td>trunc</td>
<td>truncate (round down)</td>
<td>real or integer</td>
<td>integer</td>
</tr>
</tbody>
</table>
INPUT AND OUTPUT 2

Input is what comes into the program. It can be from the keyboard, the mouse, a file on disk, a scanner, a joystick, etc.

We will not get into mouse input in detail, because that syntax differs from machine to machine. In addition, today's event-driven windowing operating systems usually handle mouse input for you.

The basic format for reading in data is:

```
read (Variable_List);
```

`Variable_List` is a series of variable identifiers separated by commas.

`read` treats input as a stream of characters, with lines separated by a special end-of-line character. `readln`, on the other hand, will skip to the next line after reading a value, by automatically moving past the next end-of-line character:

```
readln (Variable_List);
```

Suppose you had this input from the user, and `a`, `b`, `c`, and `d` were all integers.

```
45 97 3
1 2 3
```

Here are some sample `read` and `readln` statements, along with the values read into the appropriate variables.

<table>
<thead>
<tr>
<th>Statement(s)</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>read (a);</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read (b);</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>readln (a);</td>
<td></td>
<td>45</td>
<td>97</td>
<td></td>
</tr>
<tr>
<td>read (b);</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read (a, b, c, d);</td>
<td>45</td>
<td>97</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>readln (a, b);</td>
<td></td>
<td>45</td>
<td>97</td>
<td>1</td>
</tr>
<tr>
<td>readln (c, d);</td>
<td></td>
<td>45</td>
<td>97</td>
<td>2</td>
</tr>
</tbody>
</table>

When reading in integers, all spaces are skipped until a numeral is found. Then all subsequent numeral values are read, until a non-numeric character is reached (including, but not limited to, a space).

```
8352.38
```
When an integer is read from the above input, its value becomes 8352. If, immediately afterwards, you read in a character, the value would be ‘.’ since the read head stopped at the first alphanumeric character.

Suppose you tried to read in two integers. That would not work, because when the computer looks for data to fill the second variable, it sees the ‘.’ and stops since it couldn't find any data to read.

With real values, the computer also skips spaces and then reads as much as can be read. However, many Pascal compilers place one additional restriction: a real that has no whole part must begin with 0. So 0.678 is invalid, and the computer can't read in a real, but 0.678 is fine.

Make sure that all identifiers in the argument list refer to variables! Constants cannot be assigned a value, and neither can literal values.

**FORMATTING OUTPUT**

Formatting output is quite easy. For each identifier or literal value on the argument list, use:

\[
\text{Value : field_width}
\]

The output is right-justified in a field of the specified integer width. If the width is not long enough for the data, the width specification will be ignored and the data will be displayed in its entirety (except for real values — see below).

Suppose we had: \(\text{write ('Hi':10, 5:4, 5673:2);}\)

The output would be (that's eight spaces before the Hi and three spaces after):

\(\text{Hi 55673}\)

For real values, you can use the aforementioned syntax to display scientific notation in a specified field width, or you can convert to fixed decimal-point notation with:

\[
\text{Value : field_width : decimal_field_width}
\]

The field width is the total field width, including the decimal part. The whole number part is always displayed fully, so if you have not allocated enough space, it will be displayed anyway. However, if the number of decimal digits exceeds the specified decimal field width, the output will be displayed rounded to the specified number of places (though the variable itself is not changed).

\(\text{write (573549.56792:20:2);}\)

would look like (with 11 spaces in front):

\(573549.57\)
SAMPLE PROGRAM

(* Author: Alex Green
  Date: 9/14/2008
  Description: Find the sum and average of five numbers *)

program SumAverage;

const
  NumberOfIntegers = 5;

var
  num1, num2, num3, num4, num5 : integer;
  sum : integer;
  average : real;

begin  (* Main *)

  write ('Enter the first number: ');
  readln (num1);

  write ('Enter the second number: ');
  readln (num2);

  write ('Enter the third number: ');
  readln (num3);

  write ('Enter the fourth number: ');
  readln (num4);

  write ('Enter the fifth number: ');
  readln (num5);

  sum := num1 + num2 + num3 + num4 + num5;
  average := Sum / 5;

  writeln ('Number of integers = ', NumberOfIntegers);
  writeln;

  writeln ('Number1:', num1:8);
  writeln ('Number2:', num2:8);
  writeln ('Number3:', num3:8);
  writeln ('Number4:', num4:8);
  writeln ('Number5:', num5:8);
  writeln ('================');
  writeln ('Sum:', sum:12);
  writeln ('Average:', average:10:1);

end.
BOOLEAN EXPRESSIONS

Boolean expressions are used to compare two values and get a true-or-false answer:

\[ \text{value1 relational_operator value2} \]

The following relational operators are used:

- `<` less than
- `>` greater than
- `=` equal to
- `<=` less than or equal to
- `>=` greater than or equal to
- `<>` not equal to

You can assign Boolean expressions to Boolean variables. Here we assign a true expression to `some_bool`:

\[ \text{some_bool := 3 < 5;} \]

Complex Boolean expressions are formed by using the Boolean operators:

- `not` negation (~)
- `and` conjunction (^)
- `or` disjunction (v)
- `xor` exclusive-or

**NOT** is a unary operator — it is applied to only one value and inverts it:

- `not true = false`
- `not false = true`

**AND** yields **TRUE** only if both values are **TRUE**:

- `FALSE and FALSE = FALSE`
- `FALSE and TRUE = FALSE`
- `TRUE and FALSE = FALSE`
- `TRUE and TRUE = TRUE`

**OR** yields **TRUE** if at least one value is **TRUE**:

- `TRUE or TRUE = TRUE`
- `TRUE or FALSE = TRUE`
• FALSE or TRUE = TRUE
• FALSE or FALSE = FALSE

**XOR** yields TRUE if one expression is TRUE and the other is FALSE. Thus:

• TRUE xor TRUE = FALSE
• TRUE xor FALSE = TRUE
• FALSE xor TRUE = TRUE
• FALSE xor FALSE = FALSE

When combining two Boolean expressions using relational and Boolean operators, be careful to use parentheses.

(3>5) or (650<1)

This is because the Boolean operators are higher on the order of operations than the relational operators:

1. not
2. * / div mod and
3. + - or
4. < > <= >= = <>

So 3 > 5 or 650 < 1 becomes evaluated as 3 > (5 or 650) < 1, which makes no sense, because the Boolean operator or only works on Boolean values, not on integers.

The Boolean operators (AND, OR, NOT, XOR) can be used on Boolean variables just as easily as they are used on Boolean expressions.

Whenever possible, don’t compare two real values with the equals sign. Small round-off errors may cause two equivalent expressions to differ.
Using the Selection Structure

The selection structure, also called the decision structure, is one of the three basic control structures used in programming. The other two are sequence, which has been covered previously, and the repetition structure, which will be covered in later. The selection structure is used when you would like the program to make a decision, or comparison. Dependent upon the outcome of the decision, one of two paths will be chosen (the true path or the false path). Note that the decision within the selection structure will also result in either true or false, which is called a boolean result.

Including the Selection Structure in Pseudocode

<table>
<thead>
<tr>
<th>Input</th>
<th>Process</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>score</td>
<td>1. Input score</td>
<td>result</td>
</tr>
<tr>
<td></td>
<td>2. if score &gt;= 50 then result = 'pass'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else result = 'fail'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>endif</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3. display result</td>
<td></td>
</tr>
</tbody>
</table>

Drawing a Flowchart of a Selection Structure

While the concept of a selection structure is quite simple, it is important to properly design the selection structure. The diamond symbol represents a decision and is called the selection/repetition symbol. Each diamond has one flowline entering the symbol, and two flowlines leaving the symbol. The two flowlines leaving the symbol represent the true and false paths and should be marked accordingly.
Coding the selection Structure

The **IF statement** allows you to branch based on the result of a Boolean operation. The one-way branch format is:

\[
\text{if BooleanExpression then} \\
\text{StatementIfTrue;}
\]

If the Boolean expression evaluates to true, the statement executes. Otherwise, it is skipped.

**Example 1:** The following example asks the user to enter a number and tells you if the number is greater than 5:

```pascal
program Decisions;
var
  i: Integer;
begin
  writeln('Enter a number');
  readln(i);
  if i > 5 then
    writeln('Greater than 5');
  readln;
end.
```

The IF statement accepts only one statement. If you would like to branch to a compound statement (i.e., if you want to carry out more than one step if the condition is true), you must use a `begin-end` to enclose the statements:

\[
\text{if BooleanExpression then} \\
\text{begin} \\
\text{Statement1;} \\
\text{Statement2} \\
\text{end;}
\]
Example 2

program Decisions;
var
  i: Integer;
begin
  writeln('Enter a number');
  readln(i);

  if i > 0 then
    begin
      writeln('You entered ',i);
      writeln('It is a positive number');
    end;

  readln;
end.

There is also a two-way selection:

if BooleanExpression then
  StatementIfTrue
else
  StatementIfFalse;

Example 3

program Decisions;
var
  i: Integer;
begin
  writeln('Enter a number');
  readln(i);
  if i > 5 then
    writeln('Greater than 5')
  else
    writeln('Not greater than 5');
end.

If the Boolean expression evaluates to FALSE, the statement following the else will be performed.
Note:

- **Compound statements require a begin-end.**
- **You may not** use a semicolon after the statement preceding the else. That causes the computer to treat it as a one-way selection, leaving it to wonder where the else came from.

If you need multi-way selection, simply nest if statements:

```plaintext
if Condition1 then
  Statement1
else
  if Condition2 then
    Statement2
  else
    Statement3;
```

Be careful with nesting. Sometimes the computer won't do what you want it to do:

```plaintext
if Condition1 then
  if Condition2 then
    Statement2
  else
    Statement1;
```

The else is always matched with the most recent if, so the computer interprets the preceding block of code as:

```plaintext
if Condition1 then
  if Condition2 then
    Statement2
  else
    Statement1;
```

You can get by with a null statement:

```plaintext
if Condition1 then
  if Condition2 then
    Statement2
  else
    Statement1;
else
  if Condition2 then
    Statement2;
```

Or you could use a begin-end block. But the best way to clean up the code would be to rewrite the condition.

```plaintext
if not Condition1 then
  Statement1
else
  if Condition2 then
    Statement2;
```
This example illustrates where the not operator comes in very handy. If Condition1 had been a Boolean like: (not(a < b) or (c + 3 > 6)) and g, reversing the expression would be more difficult than NOTting it.

Also notice how important indentation is to convey the logic of program code to a human, but the compiler ignores the indentation.

Some additional Examples:

If you want to use more than 1 condition then you must put each condition in brackets. To join the conditions you can use either AND or OR. If you use AND then both conditions must be true but if you use OR then only 1 or both of the conditions must be true.

```pascal
program Decisions;
var
  i: Integer;
begin
  Writeln('Enter a number');
  Readln(i);
  if (i > 1) and (i < 100) then
    Writeln('The number is between 1 and 100');
  readln;
end.
```

You can also use if statements inside other if statements.

```pascal
program Decisions;
var
  i: Integer;
begin
  Writeln('Enter a number');
  Readln(i);
  if i > 0 then
    Writeln('Positive')
  else
    if i < 0 then
      Writeln('Negative')
    else
      Writeln('Zero');
  readln;
end.
```
Case

The case command is like an if statement but you can have many conditions with actions for each one.

program Decisions;
var
  Choice: Char;
begin
  Writeln('Which on of these do you like?');
  Writeln('a - Apple:');
  Writeln('b - Banana:');
  Writeln('c - Carrot:');
  readln(Choice);
  case Choice of
    'a': Writeln('You like apples');
    'b': Writeln('You like bananas');
    'c': Writeln('You like carrots');
  else;
    Writeln('You made an invalid choice');
  end;
  readln;
end.

The same problem using if statements

program Decisions;
var
  Choice: Char;
begin
  Writeln('Which on of these do you like?');
  Writeln('a - Apple:');
  Writeln('b - Banana:');
  Writeln('c - Carrot:');
  readln(Choice);

  if choice = 'a' then
  Writeln('You like apples')
  else
    if choice = 'b' then
      Writeln('You like bananas')
    else
      if choice = 'c' then
        Writeln('You like carrots');
      else
        Writeln('You made an invalid choice');
  readln;
end.