Table of Contents

ALGORITHM

An algorithm (pronounced AL-go-rith-um) is a procedure or formula for solving a problem. The word derives from the name of the mathematician, Mohammed ibn-Musa al-Khwarizmi, who was part of the royal court in Baghdad and who lived from about 780 to 850. Al-Khwarizmi's work is the likely source for the word *algebra* as well.

In mathematics and computer science, an algorithm usually means a small procedure that solves a recurrent problem. To make a computer do anything, you have to write a computer program.

A computer program can be viewed as an elaborate algorithm. To write a computer program, you have to tell the computer, step by step, exactly what you want it to do. The computer then "executes" the program, following each step mechanically, to accomplish the end goal.

When you are telling the computer *what* to do, you also get to choose *how* it's going to do it. That's where **computer algorithms** come in. The algorithm is the basic technique used to get the job done.

Let's follow an example to help get an understanding of the algorithm concept. Let's say that you have a friend arriving at the airport, and your friend needs to get from the airport to your house.

Here are four different algorithms that you might give your friend for getting to your home:

- **The taxi algorithm**:
  - Go to the taxi stand.
  - Get in a taxi.
  - Give the driver my address.
- **The call-me algorithm**:
  - When your plane arrives, call my cell phone.
  - Meet me outside baggage claim.
- **The rent-a-car algorithm**:
  - Take the shuttle to the rental car place.
  - Rent a car.
  - Follow the directions to get to my house.
- **The bus algorithm**:
  - Outside baggage claim, catch bus number 70.
  - Transfer to bus 14 on Main Street.
  - Get off on Elm street.
  - Walk two blocks north to my house.

All four of these algorithms accomplish exactly the same goal, but each algorithm does it in completely different way. Each algorithm also has a different cost and a different travel time. Taking a taxi, for example, is probably the fastest way, but also the most expensive.

Taking the bus is definitely less expensive, but a whole lot slower. You choose the algorithm based on the circumstances.

In computer programming, there are often many different ways -- algorithms -- to accomplish any given task. Each algorithm has advantages and disadvantages in different situations. **Sorting** is one place where a lot of research has been done, because computers spend a lot of time sorting lists. Here are five different algorithms that are used in sorting:

- Bin sort
- Merge sort
- Bubble sort
- Shell sort
- Quicksort

If you have a million integer values between 1 and 10 and you need to sort them, the **bin sort** is the right algorithm to use. If you have a million book titles, the **quicksort** might be the best algorithm. By knowing the strengths and weaknesses of the different algorithms, you pick the best one for the task at hand.

Expressing algorithms

Algorithms can be expressed in many kinds of notation, including natural languages, *pseudocode*, flowcharts, and programming languages. Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms.

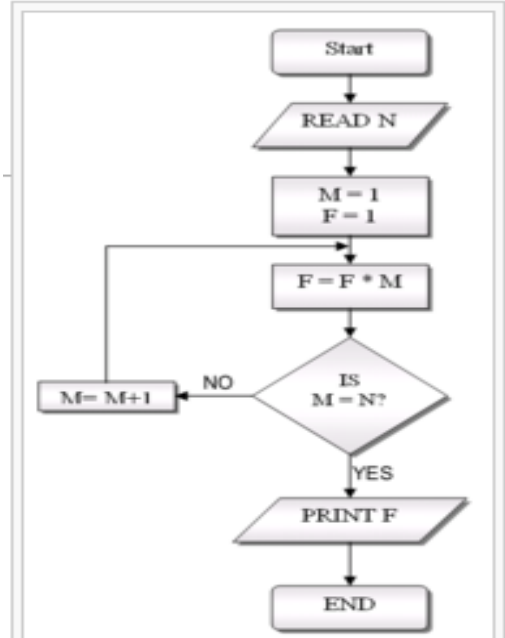
Pseudocode and flowcharts are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular implementation language. Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

A **flowchart** is a common type of chart, that represents an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. Flowcharts are used in analyzing, designing, documenting or managing a process or program in various fields.

Symbols

A typical flowchart from older Computer Science textbooks may have the following kinds of symbols:

- *Start* and *end* symbols represented as **circles, ovals** or **rounded rectangles**, usually containing the word "Start" or "End", or another phrase signaling the start or end of a process, such as "submit enquiry" or "receive product".



A simple flowchart for computing factorial N (N!)

- **Arrows** showing what's called "flow of control" in computer science. An arrow coming from one symbol and ending at another symbol represents that control passes to the symbol the arrow points to.
- Processing steps represented as **rectangles**. Examples: "Add 1 to X"; "replace identified part"; "save changes" or similar.
- Input/Output represented as a **parallelogram**. Examples: Get X from the user; display X.
- Conditional or decision Represented as a **diamond (rhombus).** These typically contain a Yes/No question or True/False test.

  This symbol is unique in that it has two arrows coming out of it, usually from the bottom point and right point, one corresponding to Yes or True, and one corresponding to No or False. The arrows should always be labeled. More than two arrows can be used, but this is normally a clear indicator that a complex decision is being taken, in which case it may need to be broken-down further, or replaced with the "pre-defined process" symbol. A number of other symbols that have less universal currency, such as:

- A Document represented as a rectangle with a **wavy base**;
- A Manual input represented by **parallelogram**, with the top irregularly sloping up from left to right. An example would be to signify data-entry from a form;

- A Manual operation represented by a **trapezoid** with the longest parallel side at the top, to represent an operation or adjustment to process that can only be made manually.
- A Data File represented by a **cylinder**.

Flowcharts may contain other symbols, such as connectors, usually represented as circles, to represent converging paths in the flowchart. Circles will have more than one arrow coming into them but only one going out.

Some flowcharts may just have an arrow point to another arrow instead. These are useful to represent an iterative process (what in Computer Science is called a loop). A loop may, for example, consist of a connector where control first enters, processing steps, a conditional with one arrow exiting the loop, and one going back to the connector. Off-page connectors are often used to signify a connection to a (part of another) process held on another sheet or screen. It is important to remember to keep these connections logical in order. All processes should flow from top to bottom and left to right.

PROGRAM STRUCTURE IN PASCAL PROGRAMMING

The Pascal programming language was created by Niklaus Wirth in 1970. It was named after Blaise Pascal, a famous French Mathematician. It was made as a language to teach programming and to be reliable and efficient. Pascal has since become more than just an academic language and is now used commercially.

Before you start learning Pascal, you will need a Pascal compiler. This tutorial uses the **Turbo Pascal for Windows** Compiler.

Program Structure

In a program, you must always obey the rules of the language, in our case, the Pascal language. A natural language has its own grammar rules, spelling and sentence construction. The Pascal programming language is a high level language that has its own syntax rules and grammar rules.

A program structure in Pascal basically consist of three parts, that are:

1. Program title (it is optional)
2. Declaration section
3. Main program

Below is a simple example of a small program. (you can type the program in a text file, save the text file as filename `.pas` and open it with Turbo Pascal for windows. The `.pas` extension is required.).

example:

```
Program exam_Program1;          {program title}
uses wincrt;                    {declaration section}
  Begin                         {start the main program,  no semicolon}
     Write('Hello. Prepare to learn PASCAL!!');
     readln;
End.                            {end of the main program}
```

The Pascal programming language has several important words in it. These are called **keywords** or **reserved word**. A program in Pascal starts with the keyword '**Program**' following the name (title) of the program. There are various restrictions on how to write this title. The title (name) of program is optional. You can omit the program title but it lets you identify what the program does quickly and easily.

After the program title comes declaration section. There are some keywords can be declared in this section, eg. **uses, const, var, type** and **label**. (we will discuss later).

In this example, this part consists the keyword 'Uses', that allows your program to use extra commands, such as **wincrt**. CRT stands for Cathode Ray Tube - ie. the screen, and we use wincrt as we work with pascal for windows.

After declaration section, the main program always starts with the reserved word **'Begin'**. This indicates the beginning of the main part of your program. After this comes your program code. The end of the program is indicated by the keyword 'end.'. Note the **full stop (.)** after the word 'end' (this is required though).

Program codes in the main program of the example above are :

```
Write('Hello. Prepare to learn PASCAL!!');
readln;
```

The code is only to display the message :

Hello. Prepare to learn PASCAL!!

So, to display any message on the screen, you should use **'write'** (or **'writeln'**). The **'readln'** statement, here is used as to 'stop' the program and wait until the user presses enter. If the 'readln' statement is missing in this program, then the message is displayed on the screen without giving any chance for the user to read it and obviously halts! Try running this program with and without the 'readln' statement and notice the difference.

Comments

The messages in between the braces { } are called **comments or in-line documentation**. In the example above {program title}, {declaration section}, {start the main program}, {end of the main program} are the 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, that use the braces { comment } or (* comment*).

Punctuation

Another important thing which must be noticed is the **semi-colon (;).** The semicolon is used after each statement in the program, except those that you will learn later. However, in the example above, there isn't a semicolon after a 'begin' statement. This is because the flow of the program has just started and must not be stopped by a ';'.

In Pascal program, you *must* have a semicolon following:

- the program heading
- each constant definition
- each variable declaration
- each type definition
- almost all statements

Indentation

Now, look at this next example:

```
Program exam_program2;  uses wincrt;  begin
Write('Hello. Prepare to learn PASCAL!!');Readln;End.
```

This program is same as `Program exam_program1`. The only difference is: **neatness** and **friendliness**. It is much better for it to look like the previous one.

This first program (`Program exam_program1`) is commonly referred to in programming, as **'indented'**. Indentation is a must in writing programs as it aids in the way the code is written i.e. neater. Indentation also helps with debugging and code presentation. In general, indent each block. Skip a line between blocks.

IDENTIFIERS

Identifiers are names that allow you to reference stored values, such as variables and constants. Also, every program and unit must be named by an identifier.

example:

In the example above, `exam_program1`, `exam_program2`, `exam_program3` are identifier, because they are the name of program.

Pascal has some rules for identifiers:

- Must begin with a letter from the English alphabet.

- Can be followed by alphanumeric characters (alphabetic characters and numerals) and possibly the underscore (_).
- May not contain certain special characters, many of which have special meanings in Pascal.
  ~ ! @ # $ % ^ & * ( ) + ` - = { } [ ] : " ; ' < > ? , . / |
- Pascal is *not* case sensitive! `MyProgram`, `MYPROGRAM`, and `mYpRoGrAm` are equivalent. But for readability purposes, it is a good idea to use meaningful capitalization

Different implementations of Pascal differ in their rules on special characters. Note that the underscore character (_) is usually allowed.

Several identifiers are *reserved* in Pascal as syntactical elements. You are not allowed to use these for your identifiers. These include but are not limited to:

and	array	begin	case	const	div
do	downto	else	end	file	For
forward	function	goto	if	in	label
mod	nil	not	of	or	Packed
procedure	program	record	repeat	set	then
to	type	until	var	while	with

Identifiers can be any length, but some Pascal compilers will only look at the first several characters. One usually does not push the rules with extremely long identifiers or loads of special characters, since it makes the program harder to type for the programmer. Also, since most programmers work with many different languages, each with different rules about special characters and case-sensitivity, it is usually best to stick with alphanumeric characters and the underscore character.

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 as a part of declaration section of the program:

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

For example, let's define some constants of various data types: strings, characters, integers, reals, and Booleans. These data types will be further explained in the next section.

Example:

```
Program exam_const;
uses wincrt;
const
        Name = 'Pascal';
         FirstLetter = 'a';
        Year = 2010;
        pi = 3.1415926535897932;
        Usingwincrt = TRUE;
Begin
    Write('Hello, My Name is ‘);Write(FirstLetter);
    writeln(Name);
    Write('This year  is ‘);Writeln(Year);
    Write('Do you know what is pi number? That is  ‘);Write(pi);
End.
```

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,

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

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

VARIABLES AND DATA TYPES

Variables are non-constant terms so that they are used in the program for storing values. Variables are similar to constants, but their values can be changed as the program runs. Variables must first be declared in the declaration section using keyword ‘var’ before they can be used. The **'var'** statement, is used to introduce any suitable **variables** which will be used later in the program. The syntax is:

```
var
  IdentifierList1 : DataType1;
  IdentifierList2 : DataType2;
  IdentifierList3 : DataType3;
  ...
```

IdentifierList is a series of **identifiers**, separated by commas (`,`). All identifiers in the list are declared as being of the same data type.

The basic data types in Pascal include: **integer, real, char** and **Boolean**

- The **`integer`** data type can contain integers from -32768 to 32767. This is the signed range that can be stored in a 16-bit word, and is a legacy of the era when 16-bit CPUs were common. For backward compatibility purposes, a 32-bit signed integer is a `longint` and can hold a much greater range of values.
- The **`real`** data type has a range from 3.4×10^{-38} to 3.4×10^{38}, in addition to the same range on the negative side. Real values are stored inside the computer similarly to scientific notation, with a mantissa and exponent, with some complications. In Pascal, you can express real values in your code in either fixed-point notation or in scientific notation, with the character `E` separating the mantissa from the exponent. Thus, `452.13` is the same as `4.5213e2`
- The **`char`** data type holds characters. Be sure to enclose them in single quotes, like so: `'a' 'B' '+'`
- The **`Boolean`** data type can have only two values: **`TRUE`** and **`FALSE`**

An example of declaring several variables is:

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

ASSIGNMENT AND OPERATIONS

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 unlike other languages, whose assignment operator is just an equals sign, 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 shown in the table below:

Operator	Operation	Operands	Result
+	Addition or unary positive	real or integer	real or integer
-	Subtraction or unary negative	real or integer	real or integer
*	Multiplication	real or integer	real or integer
/	Real division	real or integer	real
div	Integer division	integer	integer
Mod	Modulus (remainder division)	integer	integer

Notice that:

- **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 div 2 = 3` (and `7 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:

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

When the following block of statements executes,

```
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:

```
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:

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

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.

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.

Let see the following program, that contain some operations between two numbers.

Example:

```
Program exam_variable;
uses wincrt;
const Num1=10;
        Num2=25;
Var
     Sum : Integer;
     Mul, division :real;
Begin
   Sum := Num1 + Num2;                    {addition}
   Write('The Sum is = ');  Writeln(Sum);
   Mul := Num1 * Num2;                    {multiplication}
   Write('The product is = ');  Writeln(Mul);
   division := Num1 / Num2;               {division}
   Write('The division is = ');  Writeln(division);
End.
```

In this example, the declaration part consists the keyword **'uses', 'const'** and '**var'**. The terms 'Num1', 'Num2' are the **identifier** for the const. 'Sum', 'Mul' and 'division' are the **identifier** for the variables which store any numbers. In the example above, these variables are assigned to as **integers** and **real** respectively. The term 'integer' means any whole number, i.e. a number which is not a decimal number but a positive or negative number.

The variables 'Num1', 'Num2', 'Sum', 'Mul' and 'division' are terms which are not reserved words, but can be used as identifier for constant and variables in the program to store data in them. They could be changed more than once. Moreover, we could have used 'number1', 'number2' and 'totalsum' (note that there must be no spaces within the variables), instead of 'Num1', 'Num2' and 'Sum', respectively. It is much better to shorten the variables than writing long words, such as 'variable_number1'.

In the program above, both of the two types of 'write' are used. These are 'write' and 'writeln'. Both has the same function, except that the 'write' function, does not proceed to the following line when writing a statement. When using these two terms, any message that will be typed in between the brackets and the inverted commas '(' ')', is displayed on the screen. However, if a variable is used instead of a message, without using the inverted commas, it will display the

stored variable in the memory, on the screen. For example, statement writeln(Sum) will not display 'Sum' on the screen, but the stored number of variable `sum` in the memory

DATA TYPES

In Pascal there are several predefined data types, which can be devided into three groups: ordinal types, real types and strings.

Ordinal type

Ordinal types are based on the concept of order or sequence. Not only can you compare two values to see which is higher, but you can also ask for the value following or preceding a given value or compute the lowest or highest possible value.

Integer

The three most inportant predefined ordinal types are Integer, Boolean and Char (character). However, there a number of other related types that have the same meaning but adifferent internal representation and range of values. The following Table 1. lists the ordinal data types used for representing numbers.

Type	Size	Range
Byte	1 byte	0 s/d +255
Shortint	1 byte	-128 s/d +127
integer	2 bytes	-32768 s/d 32767
Word	2 bytes	0 s/d 65535
Longint	4 bytes	-2147483648 s/d 2147483647

Boolean

The **`Boolean`** data type can have only two values: **`TRUE`** and **`FALSE`**.

Char

Character can be represented with their symbolic notation, as in ‘k’, or with a numeric notation, as in #78. The latter can also be expressed using the Chr function, as in Chr(78). the opposite conversion can be done with Ord function.

It is generally better to use the symbolic notation when indicating letters, digits, or symbols. When referring to special characters, instead, you’ll generally use the numeric notation. The following list includes some of the most commonly used special characters:

Special character	Numeric notation
tabulator	#9
New line	#10
Carriage return (enter key)	#13

Real Type

Real types represent floating-point numbers in various formats. The smallest storage size is given by Single numbers, which are implemented with a 4-byte value. The Table 2 below shows real data types.

Types	Size	Range
real	6 bytes	2.9×10^{-39} s/d 1.7×10^{38}
single	4 bytes	1.5×10^{-45} s/d 3.4×10^{38}
double	8 bytes	5.0×10^{-324} s/d 1.7×10^{308}
extended	10 bytes	3.4×10^{-4932} s/d 1.1×10^{4932}
comp	8 bytes	-9.2×10^{18} s/d 9.2×10^{18}

String

You can access a specific character in a string if you put the number of the position of that character in square brackets behind a string.

```
program Strings;

var
   s: String;
   c: Char;

begin
   s := 'Hello';
   c := s[1];{c = 'H'}
end.
```

You can get the length of a string using the *Length* command.

```
program Strings;

var
   s: String;
   l: Integer;

begin
   s ;= 'Hello';
   l := Length(s);{l = 5}
end.
```

To find the position of a string within a string use the *Pos* command.
Parameters:
1: String to find
2: String to look in

```
program Strings;

var
   s: String;
   p: Integer;

begin
   s := 'Hello world';
   p := Pos('world',s);
end.
```

The *Delete* command removes characters from a string.
Parameters:
1: String to delete characters from
2: Position to start deleting from
3: Amount of characters to delete

```
program Strings;

var
   s: String;

begin
   s ;= 'Hello';
   Delete(s,1,1);{s = 'ello'}
end.
```

The *Copy* command is like the square brackets but can access more than just one character.
Parameters:
1: String to copy characters from
2: Position to copy from
3: Amount of characters to copy

```
program Strings;

var
   s, t: String;

begin
   s ;= 'Hello';
   t := Copy(s,1,3);{t = 'Hel'}
end.
```

Insert will insert characters into a string at a certain position.
Parameters:
1: String that will be inserted into the other string
2: String that will have characters inserted into it
3: Position to insert characters

```
program Strings;

var
   s: String;

begin
   s := 'Hlo';
   Insert('el',s,2);
end.
```

The *ParamStr* command will give you the command-line parameters that were passed to a program. *ParamCount* will tell you how many parameters were passed to the program. Parameter 0 is always the program's name and from 1 upwards are the parameters that have been typed by the user.

```
program Strings;

var
   s: String;
   i: Integer;

begin
   s := ParamStr(0);
   i := ParamCount;
end.
```

INPUT

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.

Statement(s)	a	b	c	d
`read (a);` `read (b);`	45	97		
`readln (a);` `read (b);`	45	1		
`read (a, b, c, d);`	45	97	3	1
`readln (a, b);` `readln (c, d);`	45	97	1	2

When reading in integers, all spaces are skipped until a numeral is found. Then all subsequent numberals 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 `.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.

OUTPUT

For writing data to the screen, there are also two statements, one of which you've seen already in last chapter's programming assignment:

```
write (Argument_List);
writeln (Argument_List);
```

The `writeln` statement skips to the next line when done.

You can use strings in the argument list, either constants or literal values. If you want to display an apostrophe within a string, use two consecutive apostrophes. Displaying two consecutive apostrophes would then requires you to use four. This use of a special sequence to refer to a special character is called *escaping*, and allows you to refer to any character even if there is no key for it on the keyboard.

Formatting Output

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

```
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:

```
write ('Hi':10, 5:4, 5673:2);
```

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

```
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:

```
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).

```
write (573549.56792:20:2);
```

would look like (with 11 spaces in front):

```
     573549.57
```

STANDARD FUNCTIONS

Pascal has several standard mathematical functions that you can utilize. For example, to find the value of `sin` of *pi* radians:

```
value := sin (3.1415926535897932);
```

Note that the `sin` function operates on angular measure stated in radians, as do all the trigonometric functions. If everything goes well, `value` should become 0.

Functions are called by using the function name followed by the argument(s) in parentheses. Standard Pascal functions include:

Function	Description	Argument type	Return type
abs	absolute value	real or integer	same as argument
arctan	arctan in radians	real or integer	real
cos	cosine of a radian measure	real or integer	real
exp	`e` to the given power	real or integer	real
ln	natural logarithm	real or integer	real
round	round to nearest integer	real	integer
sin	sin of a radian measure	real or integer	real
sqr	square (power 2)	real or integer	same as argument
sqrt	square root (power 1/2)	real or integer	real
trunc	truncate (round down)	real or integer	integer

For ordinal data types (integer or char), where the allowable values have a distinct predecessor and successor, you can use these functions:

Function	Description	Argument type	Return type
chr	character with given ASCII value	integer	char
ord	ordinal value	integer or char	integer
pred	predecessor	integer or char	same as argument type
succ	successor	integer or char	same as argument type

Real is not an ordinal data type! That's because it has no distinct successor or predecessor. What is the successor of `56.0`? Is it 56.1, 56.01, 56.001, 56.0001?

However, for an integer `56`, there is a distinct predecessor — `55` — and a distinct successor — `57`.

The same is true of characters:

```
'b'
Successor: 'c'
Predecessor: 'a'
```

Conversions

The *Str* command converts an integer to a string.

```
program Convert;

var
   s: String;
   i: Integer;

begin
   s ;= '123';
```

```
   Str(i,s);
end.
```

The *Val* command converts a string to an integer.

```
program Convert;

var
   s: String;
   i: Integer;

begin
   i ;= 123;
   Val(s,i,i);
end.
```

Int Will give you the number before the comma in a real number.

```
program Convert;

var
   r: Real;

begin
   r := Int(3.14);
end.
```

Frac will give you the number after the comma in a real number.

```
program Convert;

var
   r: Real;

begin
   r := Frac(3.14);
end.
```

Round will round off a real number to the nearest integer.

```
program Convert;

var
   i: Integer;

begin
   i := Round(3.14);
end.
```

Trunc will give you the number before the comma of a real number as an integer.

```
program Convert;

var
   i: Integer;
```

```
begin
   i := Trunc(3.14);
end.
```

Computers use the numbers 0 to 255(1 byte) to represent characters internally and these are called ASCII characters. The *Ord* command will convert a character to number and the *Chr* command will convert a number to a character. Using a # in front of a number will also convert it to a character.

```
program Convert;

var
   b: Byte;
   c: Char;

begin
   c := 'a';
   b := Ord(c);
   c := Chr(b);
   c := #123;
end.
```

The *UpCase* command changes a character from a lowercase letter to and uppercase letter.

```
program Convert;

var
   c: Char;

begin
   c := 'a';
   c := UpCase(c);
end.
```

There is no lowercase command but you can do it by adding 32 to the ordinal value of an uppercase letter and then changing it back to a character.

The *Random* command will give you a random number from 0 to the number you give it - 1. The *Random* command generates the same random numbers every time you run a program so the *Randomize* command is used to make them more random by using the system clock.

```
program Rand;

var
   i: Integer;

begin
   Randomize;
   i := Random(101);
end.
```

MAKING DECISIONS

Most programs need to make decisions. There are several statements available in the Pascal language for this, that are **IF THEN**, **IF THEN ELSE** and **CASE OF**.

IF THEN

The format for the **IF THEN** statement is,

```
if  condition_is_true  then
        execute_this_program_statement;
```

The condition (ie, A < 5) is evaluated to see if it's true. When the condition is true, the program statement will be executed. If the condition is not true, then the statement following the keyword **then** will be ignored.

To create the condition we need **RELATIONAL OPERATORS**. The **RELATIONAL OPERATORS**, listed below, allow the programmer to test various variables against other variables or values.

```
=   Equal to
>   Greater than
<   Less than
<>  Not equal to
<=  Less than or equal to
>=  Greater than or equal to
```

Example :

```
program exam_IF;
var     number, guess : integer;
begin
   number := 2;
   writeln('Guess a number between 1 and 10');
   readln( guess );
   if number = guess then  writeln('You guessed correctly. Good on you!');
   if number <> guess  then  writeln('Sorry, you guessed wrong.')
end.
```

Executing more than one statement as part of an IF

To execute more than one statement when the **condition is true**, the statements are grouped using the ***begin*** and ***end*** keywords. Whether a semi-colon follows the *end* keyword depends upon what comes after it. When followed by another *end* or *end.* then it no semi-colon.

Example:

```
program IF_GROUP1;
var     number, guess : integer;
begin
   number := 2;
   writeln('Guess a number between 1 and 10');
   readln( guess );
   if number = guess then
      begin
            writeln('Lucky you. It was the correct answer.');
            writeln('You are just too smart.')
      end; { We put a semi-colon after the end keyword because it followed by
             another if statement}

   if number <> guess then   writeln('Sorry, you guessed wrong.')
end.
```

Example:

```
program IF_GROUP2;
var     number, guess : integer;
begin
    number := 2;
    writeln('Guess a number between 1 and 10');
    readln( guess );
    if number = guess then
    begin
      writeln('Lucky you. It was the correct answer.');
      writeln('You are just too smart.')
    end { there is no semi-colon after end keyword because it followed by end.}
end.
```

IF THEN ELSE

The **IF** statement can also include an **ELSE** statement, which specifies the statement (or block or group of statements) to be executed when the condition associated with the **IF** statement is false.

Example:

Rewriting the previous program using an IF THEN ELSE statement, yields:

```
program IF_ELSE_DEMO;
var     number, guess : integer;
begin
   number := 2;
   writeln('Guess a number between 1 and 10');
   readln( guess );
    if number = guess  then
       writeln('You guessed correctly. Good on you!')
    else
       writeln('Sorry, you guessed wrong.')
end.
```

If you want to execute more than one statement when a condition is true (or false) then use the **begin** and **end** keywords to make group blocks of code together.

Example:
Consider the following portion of code,

```
if number = guess  then
begin
   writeln('You guessed correctly. Good on you!');
   writeln('It may have been a lucky guess though')
end      {no semi-colon if followed by an else }
else
begin
   writeln('Sorry, you guessed wrong.');
   writeln('Better luck next time')
end;     {semi-colon depends on next keyword }
```

NESTED IF

A nested if statement is an if statement within another if statement. The syntax of a nested if statement is:

```
If (this happens) then         {if 1}
    If (this happens) then     {if 2}
        (do this) etc...
    Else (do this)             {if 2}
  Else (do this) etc...        {if 1}
```

Example:

```
program Decisions;
uses wincrt;
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');
end.
```

Logical Operators and Boolean Expressions

The logical operators are expressions which return a false or true result over a conditional expression. Such operators consist of simple logical operators, such as '**Not**' or '**And**'. They should be used between two conditional expressions ;

for example:

```
    If (x = 0) AND (a = 2) then...
```

There are three types of logical operators, each of which are concerned with conditional expressions. These are: **AND,OR,NOT.**

AND yields TRUE only if both values are TRUE:

Expression 1	Expression 2	AND (result)
true	true	true
false	true	false
true	false	false
false	false	false

Example:

```
Program exam_And;
Uses Crt;
Var n1, n2 : string;

Begin
 Writeln('Enter two numbers: (''0'' & ''0'' to exit)');
 Repeat
  Write('No.1: ');
  Readln(n1);
  Write('No.2: ');
  Readln(n2);
  If (n1 = '0') AND (n2 = '0') then Halt(0);
 Until (n1 = '0') AND (n2 = '0');

End.
```

OR yields TRUE if at least one value is TRUE:

Expression 1	Expression 2	OR (result)
true	true	true
false	true	true
true	false	true
false	false	false

example:

```
Program exam_OR;
Uses Crt;
Var n1, n2 : String;


Begin
 Writeln('Enter two numbers: (''1'' & ''2'' to exit)');
 Repeat
```

```
   Write('No.1: ');  Readln(n1);
   Write('No.2: ');  Readln(n2);
   If (n1 = '1') OR (n2 = '2') then Halt;
 Until (n1 = '1') OR (n2 = '2');

End.
```

Not is almost different from the two logical gates. It only accepts one input and is well-known as the 'inverter'. If for example the result of two conditional expressions is true, the '**not**' operator would invert the result to false! So, the of the logical operator, '**not**', is to output the inverse of the input.

The simple truth table for the *not* operator is:

Input	Output
true	false
false	true

example:

```
Program exam_Not;
Uses Crt;
Var n1 : String;

Begin
 Writeln('Enter two numbers: (any number except 0 to exit)');
 Repeat
  Write('No.1: ');  Readln(n1);
  If not(n1 = '0') then Halt;
 Until not(n1 = '0');
End.
```

The Boolean Expressions

The boolean expressions are the terms '**true**' and '**false**'. These are simply similar to 1's (for true) and 0's(for false). They describe an expression whether it is false or true. The variable types over boolean expressions is the '**boolean**' type. Example:

Var bool : **Boolean**;

Example Program:

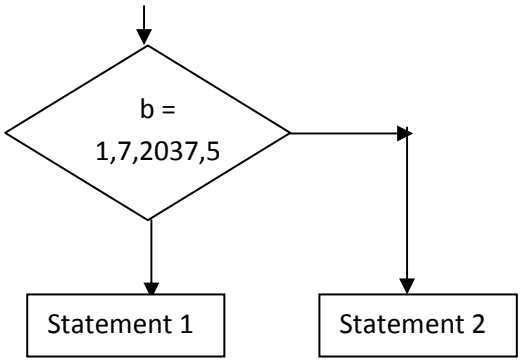
```
Program exam_Boolean;
Var quit : Boolean;
    a    : String;

Begin
 Repeat
  Write('Type ''exit'' to quit:');
  Readln(a);
  If a = 'exit' then quit := True else quit := False;
  If quit = True then Halt;
 Until quit = True;
```

```
End.
```

CASE OF

In some cases the '**case statement**' is preferred to the **if statement** because it reduces some unnecessary code but the same meaning is retained. The case statement is very similar to the if statement, except in that the it does not accept literal conditional expressions (i.e.: strings) but it allows single character conditional expressions.

Suppose you wanted to branch one way if `b` is 1, 7, 2037, or 5; and another way if otherwise.



You could do it by:

```
if (b = 1) or (b = 7) or (b = 2037) or (b = 5) then
  Statement1
else
  Statement2;
```

But in this case, it would be simpler to list the numbers for which you want Statement1 to execute. You would do this with a `case` statement:

```
case b of
  1,7,2037,5: Statement1;
  otherwise   Statement2
end;
```

The general form of the case statement is:

```
case selector of
   List1:    Statement1;
   List2:    Statement2;
   ...
   Listn:    Statementn;
   otherwise Statement
end;
```

The `otherwise` part is optional. When available, it differs from compiler to compiler. In many compilers, you use the word **`else`** instead of `otherwise`.

selector is any variable of an ordinal data type (**integer or char ONLY**). You may not use reals!

Example:

Consider the following code portion written using if else statements,

```
if operator = '*' then result := number1 * number2
   else if operator = '/' then result := number1 / number2
       else if operator = '+' then result := number1 + number2
           else if operator = '-' then result := number1 - number2
               else  invalid_operator = 1;
```

Rewriting this using case statements,

```
case  operator  of
      '*' : result:= number1 * number2;
      '/' : result:= number1 / number2;
      '+' : result:= number1 + number2;
      '-' : result:= number1 - number2;
otherwise    invalid_operator := 1
end;
```

The value of *operator* is compared against each of the values specified. If a match occurs, then the program statement(s) associated with that match are executed.

If *operator* does not match, it is compared against the next value. The purpose of the *otherwise* clause ensures that appropiate action is taken when *operator* does not match against any of the specified cases.

```
Program exam_case;
Uses Crt;
Label Return;  {used respectively with the goto statement; beware of it}

Var SEL : Integer;
    YN : Char;

Begin
 Return: Clrscr;
 Writeln('[1].PLAY GAME');
 WRITELN('[2].LOAD GAME');
 WRITELN('[3].MULTIPLAYER');
 WRITELN('[4].EXIT GAME');
 Writeln('note: Do note press anything except');
 Writeln('numbers; otherwise an error occurs!');
```

```
 Readln(SEL);
 If SEL = 1 then
  Begin
   Writeln('Are you able to create a game');
   Writeln('of yourself using pascal??');
   Goto Return;
  End;
 If SEL = 2 then
  Begin
   Writeln('Ahhh... no saved games');
   Goto Return;
  End;
 If SEL = 3 then
  Begin
   Writeln('networking or 2 players?');
   Goto Return;
  End;
 If SEL = 4 then
  Begin
   Writeln('Exit?');
   YN := Readkey;
   If YN = 'y' then
    Begin
     Writeln('Nooooooooooooo...');
     Halt; {EXIT PROGRAM}
    End;
   If YN = 'n' then
    Goto Return;
  End;
End.
```

The program below is written using the case statement and the output is almost the same

```
Program exam_case1;
Uses Crt;
Label Return; {use of the goto statement is not recommended..avoid it}
Var SEL : Integer;
    YN  : Char;

Begin
 Return:Clrscr;
 Writeln('[1].PLAY GAME');
 WRITELN('[2].LOAD GAME');
 WRITELN('[3].MULTIPLAYER');
 WRITELN('[4].EXIT GAME');
 Writeln('note: Do note press anything except');
 Writeln('numbers; otherwise an error occurs!');
 Readln(SEL);
 Case SEL of
  1 : Begin
       Writeln('Are you able to create');
       Writeln('a game of yourself using pascal??');
       Goto Return;
      End;
  2 : Begin
       Writeln('Ahhh... no saved games');
```

```
        Goto Return;
       End;
  3 : Begin
        Writeln('networking or 2 players?');
        Goto Return;
       End;
  4 : Begin
        Writeln('Exit?');
        YN := Readkey;
        Case YN of {a sort of a nested case statement}
         'y' : Begin
            Writeln('Noooooooooooo...');
            Halt;
            End;
         'n' : Goto Return;
        End;{End Case2}
       End;{Close Conditional Expression 4}
 End;  {End Case1}
End.
```

Another example:

```
program exam_case2;
uses
   wincrt;
var
   Choice: Char;

begin
   Writeln('Which on of these do you like?');
   Writeln('a - Apple:');
   Writeln('b - Banana:');
   Writeln('c - Carrot:');
   Choice := ReadKey;
   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;
end.
```

LOOPS

Looping means repeating a statement or compound statement over and over until some condition is met.

Loops are used when you want to repeat code a lot of times. For example, if you wanted to print "Hello" on the screen 10 times you would need 10 *Writeln* commands. You could do the same thing by putting 1 *Writeln* command inside a loop which repeats itself 10 times.

There are three types of loops:

- **fixed repetition** - only repeats a fixed number of times
- **pretest** - tests a Boolean expression, then goes into the loop if TRUE
- **posttest** - executes the loop, then tests the Boolean expression

FOR..DO

The most common loop in Pascal is the FOR loop. The statement inside the for block is executed a number of times depending on the control condition. In Pascal, the fixed repetition loop is the for loop. The general form is:

```
for index := StartingLow to EndingHigh do
   statement;
```

The index variable must be of an ordinal data type. You can use the index in calculations within the body of the loop, but you should not change the value of the index. An example of using the index is:

```
sum := 0;
for count := 1 to 100 do
  sum := sum + count;
```

If you want to have more than 1 command inside a loop then you must put them between a *begin* and an *end*.

```
for index := StartingLow to EndingHigh do
    begin
      statement_1;
      statement_2;
      ...
      statement_n;
    end;
```

Example:

```
program exam_Loops;
uses wincrt;

var
   i: Integer;

begin
   for i := 1 to 10 do
      begin
            Write('Hello');
            Writeln(' world');
      end
end.
```

In the for-to-do loop, the starting value MUST be lower than the ending value, or the loop will never execute! If you want to count down, you should use the for-downto-do loop:

```
for index := StartingHigh downto EndingLow do
   statement;
```

In Pascal, the for loop can only count in increments (steps) of 1.

The following program illustrates the for..do loop.

```
program  CELCIUS_TABLE;
var      celcius : integer; farenhiet : real;
begin
         writeln('Degree''s Celcius   Degree''s Farenhiet');
         for  celcius := 1  to  20  do
         begin
             farenhiet := ( 9 / 5 ) * celcius + 32;
             writeln( celcius:8, '      ',farenhiet:16:2 )
         end
end.
```

EXERCISE
What is the resultant output when this program is run.

```
program  FOR_TEST ( output );
var      s, j, k, i, l : integer;
begin
         s := 0;
         for  j:= 1 to 5 do
         begin
             write( j );
             s := s + j
         end;
         writeln( s );
         for  k := 0 to 1 do write( k );
         for  i := 10 downto 1 do writeln( i );
         j := 3; k := 8; l := 2;
         for  i := j to k do  writeln( i + l )
end.
```

NESTED LOOPS
A for loop can occur within another, so that the inner loop (which contains a block of statements) is repeated by the outer loop.

RULES RELATED TO NESTED FOR LOOPS
1. Each loop must use a separate variable
2. The inner loop must begin and end entirely within the outer loop.

Exercise:

Determine the output of the following program,

```
program  NESTED_FOR_LOOPS;
var      line, column : integer;
begin
         writeln('LINE');
         for  line := 1 to 6 do
         begin
             write( line:2 );
             for  column := 1 to 4 do
             begin
                 write('COLUMN':10);   write(column:2)
             end;
             writeln
         end
end.
```

Programming Assignment

The factorial of an integer is the product of all integers up to and including that integer, except that the factorial of 0 is 1.

eg, 3! = 1 * 2 * 3 (answer=6)

Evaluate the factorial of an integer less than 20, for five numbers input successively via the keyboard.

WHILE..DO

The pretest loop has the following format:

```
while BooleanExpression do
   statement;
```

This type of loop is executed while the condition is true. The loop continues to execute until the Boolean expression becomes FALSE. In the body of the loop, you must somehow affect the Boolean expression by changing one of the variables used in it. Otherwise, an infinite loop will result:

```
a := 5;
while a < 6 do
  writeln (a);
```

Remedy this situation by changing the variable's value:

```
a := 5;
while a < 6 do
  begin
    writeln (a);
    a := a + 1
  end;
```

The WHILE ... DO loop is called a pretest loop because the condition is tested before the body of the loop executes. So if the condition starts out as FALSE, the body of the while loop never executes.

```
Program exam_while;
Uses wincrt;

Var Ch : Char;
Begin
 Writeln('Press ''q'' to exit...');
 Ch := Readkey;
 While Ch <> 'q' do
  Begin
   Writeln('I told you press ''q'' to exit!!');
   Ch := Readkey;
  End;
End.
```

REPEAT..UNTIL

The posttest loop has the following format:

```
repeat
   statement1;
   statement2
until BooleanExpression;
```

In a repeat loop, compound statements are built-in -- you don't need to use begin-end. Also, the loop continues until the Boolean expression is TRUE, whereas the while loop continues until the Boolean expression is FALSE.

This loop is called a posttest loop because the condition is tested after the body of the loop executes. The REPEAT loop is useful when you want the loop to execute at least once, no matter what the starting value of the Boolean expression is.

```
Program exam_repeat;
Uses wincrt;
Var YN : String;

Begin
 Writeln('Y(YES) or N(NO)?');
 Repeat {repeat the code for at least one time}
  YN := Readkey ;
  If YN = 'y' then Halt; {Halt - exit}
  If YN = 'n' then Writeln('Why not? Exiting...');
```

```
 Until (YN = 'y') OR (YN = 'n');
End.
```

If you want to use more than one condition for either the *while* or *repeat* loops then you have to put the conditions between brackets.

```
program exam_Loops;
uses wincrt;
var
   i: Integer;
   s: String;

begin
   i := 0;
   repeat
      i := i + 1;
      Write('Enter a number: ');
      Readln(s);
   until (i = 10) or (s = 0);
end.
```

ONE-DIMENSIONAL ARRAYS

Suppose you wanted to read in 5000 integers and do something with them. How would you store the integers?

You could use 5000 variables, lapsing into:

```
aa, ab, ac, ad, ... aaa, aab, ... aba, ...
```

But this would grow tedious (after declaring those variables, you have to read values into each of those variables).

An array contains several storage spaces, all the same type. You refer to each storage space with the array name and with a subscript. The type definition is:

```
type
   typename = array [enumerated_type] of another_data_type;
```

The data type can be anything, even another array. Any enumerated type will do. You can specify the enumerated type inside the brackets, or use a predefined enumerated type. In other words,

```
type
   enum_type = 1..50;
   arraytype = array [enum_type] of integer;
```

is equivalent to

```
type
   arraytype = array [1..50] of integer;
```

Arrays are useful if you want to store large quantities of data for later use in the program. They work especially well with **for loops**, because the index can be used as the subscript. To read in 50 numbers, assuming the following definitions:

```
type
    arraytype = array[1..50] of integer;

var
    myarray : arraytype;
```

use:

```
for count := 1 to 50 do
    read (myarray[count]);
```

We access each of the elements using the number of the elements behind it in square brackets.

```
program Arrays;

var
   a: array[1..5] of Integer;

begin
   a[1] := 12;
   a[2] := 23;
   a[3] := 34;
   a[4] := 45;
   a[5] := 56;
end.
```

It is a lot easier when you use a loop to access the values in an array. Here is an example of reading in 5 values into an array:

```
program Arrays;

var
   a: array[1..5] of Integer;
   i: Integer;

begin
   for i := 1 to 5 do
      Readln(a[i]);
end.
```

Sorting arrays

You will sometimes want to sort the values in an array in a certain order. To do this you can use a bubble sort. A bubble sort is only one of many ways to sort an array but it is the most popular. In a bubble sort the biggest numbers are moved to the end of the array.

You will need 2 loops. One to go through each number and another to point to the other number that is being compared. If the number is greater then it is swapped with the other one. You will need to use a temporary variable to store values while you are swapping them.

```
program Arrays;

var
   a: array[1..5] of Integer;
   i, j, tmp: Integer;

begin
   a[1] := 23;
   a[2] := 45;
   a[3] := 12;
   a[4] := 56;
   a[5] := 34;
   for i := 1 to 4 do
      for j := i to 5
         if a[j] > a[j + 1] then
            begin
               tmp := a[j];
               a[j] := a[j + 1];
               a[j + 1] := tmp;
            end
end.
```

TWO DIMENSIONAL ARRAYS

Arrays can have 2 dimensions instead of just one. In other words they can have rows and columns instead of just rows.

	1	**2**	**3**
1	1	2	3
2	4	5	6
3	7	8	9

Here is how to declare a 2D array:

```
program Arrays;

var
   a: array [1..3,1..3] of Integer;

begin
end.
```

To access the values of a 2d array you must use 2 numbers in the square brackets. 2D arrays also require 2 loops instead of just one.

```
program Arrays;

var
   r, c: Integer;
   a: array [1..3,1..3] of Integer;

begin
   for r := 1 to 3 do
```

```
      for c := 1 to 3 do
         Readln(a[r,c]);
end.
```

ENUMERATED DATA TYPES

Enumerated variables are defined by the programmer. It allows you to create your own data types, which consist of a set of symbols. You first create the set of symbols, and assign to them a new data type variable name.

You can declare your own ordinal data types. You do this in the `type` section of your program:

```
type
    datatypeidentifier = typespecification;
```

One way to do it is by creating an enumerated type. An enumerated type specification has the syntax:

```
(identifier1, identifier2, ... identifiern)
```

For example, if you wanted to declare the months of the year, you would do a type:

```
type
    MonthType = (January, February, March, April,
          May, June, July, August, September,
          October, November, December);
```

You can then declare a variable:

```
var
    Month : MonthType;
```

You can assign any enumerated value to the variable:

```
Month := January;
```

All the ordinal functions are valid on the enumerated type. `ord(January) = 0`, and `ord(December) = 11`.

```
type  civil_servant = ( clerk, police_officer, teacher, mayor );
        var   job, office : civil_servant;
```

The new data type created is *civil_servant*. It is a set of values, enclosed by the ( ) parenthesis. These set of values are the only ones which variables of type *civil_servant* can assume or be assigned.

The next line declares two working variables, *job* and *office*, to be of the new data type *civil_servant*.

The following assignments are valid,

```
job := mayor;
office := teacher;

if office = mayor then writeln('Hello mayor!');
```

The list of values or symbols between the parenthesis is an ordered set of values. The first symbol in the set has an ordinal value of zero, and each successive symbol has a value of one greater than its predecessor.

```
police_officer < teacher
```

evaluates as true, because *police_officer* occurs before *teacher* in the set.

MORE EXAMPLES ON ENUMERATED DATA TYPES

```
type beverage = ( coffee, tea, cola, soda, milk, water );
     color    = ( green, red, yellow, blue, black, white );
var  drink : beverage;
     chair : color;

drink := coffee;
chair := green;

if chair = yellow then drink := tea;
```

ADDITIONAL OPERATIONS WITH USER DEFINED VARIABLE TYPES

Consider the following code,

```
type Weekday = ( Monday, Tuesday, Wednesday, Thursday, Friday );
var  Workday : Weekday;
```

The first symbol of the set has the value of 0, and each symbol which follows is one greater. Pascal provides three additional operations which are performed on user defined variables. The three operations are,

```
ord( symbol )   returns the value of the symbol, thus ord(Tuesday)
                will give a value of 1

pred( symbol )  obtains the previous symbol, thus
                pred(Wednesday) will give Tuesday

succ( symbol )  obtains the next symbol, thus succ(Monday)
                gives Tuesday
```

Enumerated values can be used to set the limits of a *for* statement, or as a constant in a *case* statement, eg,

```
for Workday := Monday to Friday
     .........

case Workday of
      Monday : writeln('Mondays always get me down.');
      Friday : writeln('Get ready for partytime!')
```

```
end;
```

Enumerated type values cannot be input from the keyboard or outputted to the screen, so the following statements are illegal,

```
writeln( drink );
readln( chair );
```

A few restrictions apply, though: enumerated types are internal to a program -- they can neither be read from nor written to a text file. You must read data in and convert it to an enumerated type. Also, the identifier used in the type (such as January) cannot be used in another type.

One purpose of an enumerated type is to allow you, the programmer, to refer to meaningful names for data. In addition, enumerated types allow functions and procedures to be assured of a valid parameter, since only variables of the enumerated type can be passed in and the variable can only have one of the several enumerated values

---

SELF TEST ON ENUMERATED DATA TYPES
Whats wrong with?

```
type Day = (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday,
            Sunday);
var  Today : Day;

for Today := Sunday to Monday do
begin
   writeln( Today );
   Today := succ( Today )
end;
```

Whats wrong with

```
type COLOR = ( Red, Blue, Green, Yellow );
var  Green, Red : COLOR;
```

SUBRANGES

A subrange type is defined in terms of another ordinal data type.

Just as you can create your own set of pre-defined data types, you can also create a smaller subset or subrange of an existing set which has been previously defined. Each subrange consists of a defined lower and upper limit. The type specification is:

```
lowest_value .. highest_value
```

where *lowest_value* **<** *highest_value* and the two values are both in the range of another ordinal data type.

For example, you may want to declare the days of the week as well as the work week:

```
type
    DaysOfWeek = (Sunday, Monday, Tuesday, Wednesday,
                  Thursday, Friday, Saturday);
    DaysOfWorkWeek = Monday..Friday;
```

Consider the following,

```
        type DAY = (Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday);
             Weekday = Monday..Friday;     {subrange of DAY}
             Weekend = Saturday..Sunday;   {subrange of DAY}
             Hours   = 0..24;              {subrange of integers}
             Capitals= 'A'..'Z';           {subrange of characters)

        NOTE: You cannot have subranges of type real.
```

Which of the following are legal

```
        type  Gradepoints = 0.0..4.0;
              Numbers = integer;
              Alphabet = 'Z'..'A';
Answer:
Which of the following are legal....NONE ARE!
        Cannot have subranges of real type
        Cannot do this, must be  Numbers = 1..500;
        Cannot do this, must be  Alphabet = 'A'..'Z'   as 'A' comes before 'Z'
```

RECORDS

It is possible to create your own variable types using the *type* statement. The first type you can make is **records**. Records are 2 or more variables of different types in one. A record allows you to keep related data items in one structure. To declare a record, you'd use:

```
TYPE
   TypeName = record
      identifierlist1 : datatype1;
      ...
      identifierlistn : datatypen;
   end;
```

An example of how this could be used is for a student who has a student number and a name. Here is how you create a type:

```
Type
   Student = Record
      Number: Integer;
      Name: String;
   end;
```

After you have created the type you must declare a variable of that type to be able to use it.

```
program Types;

Type
   StudentRecord = Record
      Number: Integer;
```

```
      Name: String;
   end;

var
   Student: StudentRecord;

begin
end.
```

If you want information about a person, you may want to know name, age, city, state, and zip, then you must declare:

```
     type
         InfoType = record
             Name : string;
             Age : integer;
             City, State : String;
             Zip : integer;
         end;
      var person : InfoType
```

Each of the identifiers Name, Age, City, State, and Zip are referred to as fields. You access a field within a variable by:

```
     VariableIdentifier.FieldIdentifier
```

A period separates the variable and the field name.

To access the Number and Name parts of the record you must do the following:

```
program exam_record;
uses wincrt;
Type
   StudentRecord = Record
      Number: Integer;
      Name: String;
   end;

var
   Student: StudentRecord;

begin
Student.Number := 12345;
Student.Name := 'John Smith';
writeln('Number : ',Student.Number);
writeln('Name : ',Student.Name);
end.
```

There's a very useful statement for dealing with records. If you are going to be using one record variable for a long time and don't feel like typing the variable name over and over, you can strip off the variable name and use only field identifiers. You do this by:

```
     WITH RecordVariable DO
        BEGIN
```

```
    ...
END;
```

Example:

```
with student do
    begin
       Number := 12345;
       Name := 'John Smith';
       writeln('Number : ',Number);
       writeln('Name : ',Name);
    end;
```

PROCEDURES

Procedures are sub-programs that can be called from the main part of the program. Procedures are declared outside of the main program body using the *procedure* keyword. Procedures must also be given a unique name. Procedures have their own *begin* and *end*. Here is an example of how to make a procedure called Hello that prints "Hello" on the screen.

```
program Procedures;

procedure Hello;
begin
   Writeln('Hello');
end;

begin
end.
```

To use a procedure we must call it by using its name in the main body.

```
program Procedures;

procedure Hello;
begin
   Writeln('Hello');
end;

begin
   Hello;
end.
```

To have an exact definition of a procedure, you should compare a program which includes a repeated section with another program avoiding the repeated sections by using a procedure, which is called several times:

```
Program exam_proc1;
Uses wincrt;
Var Counter : Integer;
Begin
 textcolor(green);
 GotoXy(10,5);
```

```
 For Counter := 1 to 10 do
  Begin {Step [1]}
   write(chr(196)); {Step [2]}
  End; {Step [3]}
 GotoXy(10,6);
 For Counter := 1 to 10 do
  Begin {Step [1]}
   write(chr(196)); {Step [2]}
  End; {Step [3]}
 GotoXy(10,7);
 For Counter := 1 to 10 do
  Begin {Step [1]}
   write(chr(196)); {Step [2]}
  End; {Step [3]}
 GotoXy(10,10);
 For Counter := 1 to 10 do
  Begin {Step [1]}
   write(chr(196)); {Step [2]}
  End; {Step [3]}
 Readkey;

End.
```

Now have a look at the next program which uses a procedure:

```
Program exam_proc2;
Uses wincrt;
Procedure DrawLine;
{This procedure helps me to
 avoid the repetition of steps [1]..[3]}
Var Counter : Integer;

Begin
 textcolor(green);
 For Counter := 1 to 10 do
  Begin {Step [1]}
   write(chr(196)); {Step [2]}
  End; {Step [3]}
End;
Begin
 GotoXy(10,5);
 DrawLine;
 GotoXy(10,6);
 DrawLine;
 GotoXy(10,7);
 DrawLine;
 GotoXy(10,10);
 DrawLine;
 Readkey;

End.
```

There are some differences between these two programs which are very important to note. These are :

- **Size of the program**

It is very important for a program to be small in size. The first program, say, its size is 1900 bytes, but the second one holds about 1350 bytes!

- **Neatness**

Adopting a neat style of writing for a program helps the programmer (and other future debuggers) to cater with future bugs. I think that the first program is cumbersome, whilst the other is not! What do you think??!

- **Repetitions**

Repetitions in a program can cause a hard time for a programmer. So procedures are an essential way to avoid repetitions in a program. They also enlarge the size of a program!

- **Debugging Efficiency**

When you are required to debug the program, bugs could be much more easier to find out as the program is sliced into smaller chunks. You may run the program and notice a mistake at a certain point and which is located in a particular procedure/function. It would be much more difficult to find a mistake in a program if it would be one whole piece of code. Do slice your program into smaller chunks, and this needs design of the whole problem in hand prior to coding.

Procedures must always be above where they are called from. Here is an example of a procedure that calls another procedure.

```
program Procedures;

procedure Hello;
begin
   Writeln('Hello');
end;

procedure HelloCall;
begin
   Hello;
end;

begin
   HelloCall;
end.
```

Global and Local variables

The variables we have been using so far have been global because they can be used at any time during the program. Local variables can only be used inside procedures but the memory they use is released when the procedure is not being used. Local variables are declared just underneath the procedure name declaration.

```
program Procedures;

procedure Print(s: String);
var
   i: Integer;
begin
   for i := 1 to 3 do
      Writeln(s);
end;
```

```
begin
   Print('Hello');
end.
```

Using Procedures with Parameters

Procedures can have parameters just like the other commands we have been using. Each parameter is given a name and type and is then used just like any other variable. If you want to use more than one parameter then they must be separated with semi-colons.

```
program Procedures;

procedure Print(s: String; i: Integer);
begin
   Writeln(s);
   Writeln(i);
end;

begin
   Print('Hello',3);
end.
```

Returning back to program exam_proc1, the gotoxy statement before the **DrawLine**; could be "kicked off" so that we can avoid the repetition of the **gotoxy**! We cannot build up another procedure for the **gotoxy**, but it should be done by adding parameters with the procedure. The new program is as follows:

```
Program exam_proc3;
Uses wincrt;
Procedure DrawLine(X : Integer; Y : Integer);
                  {the decleration of the variables in brackets are called
                   parameters or arguments}
Var Counter : Integer;    {this is called a local variable}
Begin
 GotoXy(X,Y); {use the parameters}
 textcolor(green);
 For Counter := 1 to 10 do
  Begin
   write(chr(196));
  End;
End;
Begin
 DrawLine(10,5);
 DrawLine(10,6);
 DrawLine(10,7);
 DrawLine(10,10);
 Readkey;

End.
```

Now, this program includes a procedure which uses parameters. Every time it is called, the parameters can be variable, so that the position of the line could be changed. This time, we have also eliminated the gotoxy statement before every DrawLine statement. The numbers in the brackets of the DrawLine are the parameters which state the position of the line. They also serve as a gotoxy statement.

When you apply parameters to a procedure, variables should be declared on there own, and must be separated by a semi-colon ";". They are put in between the brackets, following the procedure name. The variables (known as the parameters) should be used by the procedure/sub-program only.

The Variable Parameter

Parameters of procedures may be variable. In this case, data may flow through the variable in both ways. What I am trying to say is that you can pass data and get data through the procedure using a variable parameter. Here is a declaration of a variable parameter:

```
Procedure <PROCEDURE_NAME(Var Variable_Name : Type);>
```

Here is an example of how to use a variable parameter and what's its purpose:

```
Program VAR_PARAM_EXAMPLE;

  Procedure Square(Index : Integer; Var Result : Integer);
  Begin
    Result := Index * Index;
  End;

Var
  Res : Integer;

Begin
 Writeln('The square of 5 is: ');
 Square(5, Res);
 Writeln(Res);
End.
```

FUNCTIONS

The second type of sub-program is called a function. The only difference from the procedure is that the function return a value at the end.

Note that a procedure cannot return a value. A function start and end in a similar way to that of a procedure. If more than one value is required to be returned by a module, you should make use of the variable parameter.

A function can have parameters too. If you change the sub-program from procedure to a function, of the previous program, there will be no difference in the output of the program. Just make sure which one is best when you can to implement a module.

For example, if you don't need to return any values, a procedure is more best. However if a value should be returned after the module is executed, function should be used instead.

```
program exam_Functions;

function Add(i, j:Integer): Integer;
begin
end;

begin
end.
```

Assigning the value of a function to a variable make the variable equal to the return value. If you use a function in something like *Writeln* it will print the return value. To set the return value just make the name of the function equal to the value you want to return.

```
program Functions;

var
   Answer: Integer;

function Add(i, j:Integer): Integer;
begin
   Add := i + j;
end;

begin
   Answer := Add(1,2);
   Writeln(Add(1,2));
end.
```

You can exit a procedure or function at any time by using the *Exit* command.

```
program Procedures;

procedure GetName;
var
   Name: String;
begin
   Writeln('What is your name?');
   Readln(Name);
   if Name = '' then
      Exit;
   Writeln('Your name is ',Name);
end;

begin
   GetName;
end.
```

Example of a program using a function:

```
Program exam_function;
Uses wincrt;

Var SizeA, sizeB : Real;
    YN : Char;
    unitS : String[2];
```

```
Function PythagorasFunc(A:Real; B:Real) : Real;      {The pythagoras theorem}
Begin
 PythagorasFunc := SQRT(A*A + B*B);
     {Output: Assign the procedure name to the value.If you forget to assign
     the function to the value, you will get a trash value from the memory}
End;

Begin
 Repeat
  Writeln;
  Write('Enter the size of side A : ');
  Readln(sizeA);
  Write('Enter the size of side B : ');
  Readln(sizeB);
  Repeat
     Write('metres or centimetres? Enter : [m or cm] ');
     Readln(unitS);
  Until (unitS = 'm') or (unitS = 'cm');
  Writeln(PythagorasFunc(sizeA,sizeB),' ',unitS);
  Writeln;
  Write('Repeat? ');
  YN := Readkey;
 Until (YN in ['N','n']);
End.
```

REFERENCE:

http://pascalprogramming.byethost15.com
http://www.taoyue.com
http://www.geocities.com/SiliconValley/Horizon/5444/