Learn Pascal Programming Tutorial Lesson 1 - Introduction to Pascal

About Pascal

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.

What you will need

Before you start learning Pascal, you will need a Pascal compiler. This tutorial uses the Free Pascal Compiler. You can find a list of other Pascal compilers at TheFreeCountry's Pascal compiler list.

Your first program

The first thing to do is to either open your IDE if your compiler comes with one or open a text editor.

We always start a program by typing its name. Type *program* and the name of the program next to it. We will call our first program "Hello" because it is going to print the words "Hello world" on the screen.

```
program Hello;
```

Next we will type *begin* and *end.* We are going to type the main body of the program between these 2 keywords. Remember to put the full stop after the *end*.

```
program Hello;

begin
end.
```

The *Write* command prints words on the screen.

```
program Hello;

begin
   Write('Hello world');
end.
```

You will see that the "Hello world" is between single quotes. This is because it is what is called a string. All strings must be like this. The semi-colon at the end of the line is a statement separator. You must always remember to put it at the end of the line.

The *Readln* command will now be used to wait for the user to press enter before ending the program.

```
program Hello;

begin
   Write('Hello world');
   Readln;
end.
```

You must now save your program as hello.pas.

Compiling

Our first program is now ready to be compiled. When you compile a program, the compiler reads your source code and turns it into an executable file. If you are using an IDE then pressing CTRL+F9 is usually used to compile and run the program. If you are compiling from the command line with Free Pascal then enter the following:

```
fpc hello.pas
```

If you get any errors when you compile it then you must go over this lesson again to find out where you made them. IDE users will find that their programs compile and run at the same time. Command line users must type the name of the program in at the command prompt to run it.

You should see the words "Hello world" when you run your program and pressing enter will exit the program. Congratulations! You have just made your first Pascal program.

More commands

Writeln is just like *Write* except that it moves the cursor onto the next line after it has printed the words. Here is a program that will print "Hello" and then "world" on the next line:

```
program Hello;

begin
   Writeln('Hello');
   Write('world');
   Readln;
end.
```

If you want to skip a line then just use *Writeln* by itself without any brackets.

Using commands from units

The commands that are built into your Pascal compiler are very basic and we will need a few more. Units can be included in a program to give you access to more

commands. The crt unit is one of the most useful. The *ClrScr* command in the crt unit clears the screen. Here is how you use it:

```
program Hello;

uses
   crt;

begin
   ClrScr;
   Write('Hello world');
   Readln;
end.
```

Comments

Comments are things that are used to explain what parts of a program do. Comments are ignored by the compiler and are only there for the people who use the source code. Comments must be put between curly brackets. You should always have a comment at the top of your program to say what it does as well as comments for any code that is difficult to understand. Here is an example of how to comment the program we just made:

```
{This program will clear the screen, print "Hello world" and wait for the
user to press enter.}

program Hello;

uses
   crt;

begin
   ClrScr;{Clears the screen}
   Write('Hello world');{Prints "Hello world"}
   Readln;{Waits for the user to press enter}
end.
```

Learn Pascal Programming Tutorial Lesson 2 - Colors, Coordinates, Windows and Sound

Colors

To change the color of the text printed on the screen we use the *TextColor* command.

```
program Colors;

uses
   crt;

begin
   TextColor(Red);
   Writeln('Hello');
   TextColor(White);
   Writeln('world');
end.
```

The *TextBackground* command changes the color of the background of text. If you want to change the whole screen to a certain color then you must use *ClrScr*.

```
program Colors;

uses
   crt;

begin
   TextBackground(Red);
   Writeln('Hello');
   TextColor(White);
   ClrScr;
end.
```

Screen coordinates

You can put the cursor anywhere on the screen using the *GoToXY* command. In DOS, the screen is 80 characters wide and 25 characters high. The height and width varies on other platforms. You may remember graphs from Maths which have a X and a Y axis. Screen coordinates work in a similar way. Here is an example of how to move the cursor to the 10th column in the 5th row.

```
program Coordinates;

uses
   crt;

begin
   GoToXY(10,5);
```

```
   Writeln('Hello');
end.
```

Windows

Windows let you define a part of the screen that your output will be confined to. If you create a window and clear the screen it will only clear what is in the window. The *Window* command has 4 parameters which are the top left coordinates and the bottom right coordinates.

```
program Coordinates;

uses
   crt;

begin
   Window(1,1,10,5);
   TextBackground(Blue);
   ClrScr;
end.
```

Using window(1,1,80,25) will set the window back to the normal size.

Sound

The *Sound* command makes a sound at the frequency you give it. It does not stop making a sound until the *NoSound* command is used. The *Delay* command pauses a program for the amount of milliseconds you tell it to. *Delay* is used between *Sound* and *NoSound* to make the sound last for a certain amount of time.

```
program Sounds;

uses
   crt;

begin
   Sound(1000);
   Delay(1000);
   NoSound;
end.
```

Learn Pascal Programming Tutorial Lesson 3 - Variables and Constants

What are variables?

Variables are names given to blocks of the computer's memory. The names are used to store values in these blocks of memory.

Variables can hold values which are either numbers, strings or Boolean. We already know what numbers are. Strings are made up of letters. Boolean variables can have one of two values, either True or False.

Using variables

You must always declare a variable before you use it. We use the *var* statement to do this. You must also choose what type of variable it is. Here is a table of the different variable types:

Byte	0 to 255
Word	0 to 65535
ShortInt	-128 to 127
Integer	-32768 to 32767
LongInt	-4228250000 to 4228249000
Real	floating point values
Char	1 character
String	up to 255 characters
Boolean	true or false

Here is an example of how to declare an integer variable named i:

```
program Variables;

var
   i: Integer;

begin
end.
```

To assign a value to a variable we use :=.

```
program Variables;

var
   i: Integer;

begin
```

```
   i := 5;
end.
```

You can create 2 or more variables of the same type if you seperate their names with commas. You can also create variables of a different type without the need for another *var* statemtent.

```
program Variables;

var
   i, j: Integer;
   s: String;

begin
end.
```

When you assign a value to a string variable, you must put it between single quotes. Boolean variables can only be assigned the values True and False.

```
program Variables;

var
   i: Integer;
   s: String;
   b: Boolean;

begin
   i := -3;
   s := 'Hello';
   b := True;
end.
```

Calculations with variables

Variables can be used in calculations. For example you could assign the value to a variable and then add the number 1 to it. Here is a table of the operators that can be used:

+	Add
-	Subtract
*	Multiply
/	Floating Point Divide
div	Integer Divide
mod	Remainder of Integer Division

The following example shows a few calculations that can be done:

```
program Variables;

var
   Num1, Num2, Ans: Integer;
```

```
begin
   Ans := 1 + 1;
   Num1 := 5;
   Ans := Num1 + 3;
   Num2 := 2;
   Ans := Num1 - Num2;
   Ans := Ans * Num1;
end.
```

Strings hold characters. Characters include the the letters of the alphabet as well as special characters and even numbers. It is important to understand that integer numbers and string numbers are different things. You can add strings together as well. All that happens is it joins the 2 strings. If you add the strings '1' and '1' you will get '11' and not 2.

```
program Variables;

var
   s: String;

begin
   s := '1' + '1';
end.
```

You can read vales from the keyboard into variables using *Readln* and *ReadKey*. ReadKey is from the crt unit and only reads 1 character. You will see that *ReadKey* works differently to *Readln*

```
program Variables;

var
   i: Integer;
   s: String;
   c: Char;

begin
   Readln(i);
   Readln(s);
   c := ReadKey;
end.
```

Printing variables on the screen is just as easy. If you want to print variables and text with the same *Writeln* then seperate them with commas.

```
program Variables;

var
   i: Integer;
   s: String;
begin
   i := 24;
   s := 'Hello';
   Writeln(i);
   Writeln(s,' world');
end.
```

Constants

Constants are like variables except that their values can't change. You assign a value to a constant when you create it. *const* is used instead of *var* when declaring a constant. Constants are used for values that do not change such as the value of pi.

```
program Variables;

const
   pi: Real = 3.14;

var
   c, d: Real;

begin
   d := 5;
   c := pi * d;
end.
```

Learn Pascal Programming Tutorial Lesson 4 - String Handling and Conversions

String Handling

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.
```

Conversions

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

```
program Convert;

var
   s: String;
   i: Integer;

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

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

```
program Convert;

var
   s: String;
   i: Integer;
   e: Integer;

begin
   s := '123';
   Val(s,i,e);
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.

Extras

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.
```

Learn Pascal Programming Tutorial Lesson 5 - Decisions

if then else

The *if* statement allows a program to make a decision based on a condition. The following example asks the user to enter a number and tells you if the number is greater than 5:

```
program Decisions;

var
   i: Integer;

begin
   Writeln('Enter a number');
   Readln(i);
   if i > 5 then
      Writeln('Greater than 5');
end.
```

Here is a table of the operators than can be used in conditions:

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

The above example only tells you if the number is greater than 5. If you want it to tell you that it is not greater than 5 then we use *else*. When you use else you must not put a semi-colon on the end of the command before it.

```
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 condition is True then the *then* part is chosen but if it is False then the *else* part is chosen. This is because the conditions such as i > 5 is a Boolean equation. You can even assign the result of a Boolean equation to a Boolean variable.

```
program Decisions;

var
   i: Integer;
   b: Boolean;

begin
   Writeln('Enter a number');
   Readln(i);
   b := i > 5;
end.
```

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.

```
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');
end.
```

If you want to put 2 or more commands for an *if* statement for both the then and the else parts you must use *begin* and *end;* to group them together. You will see that this *end* has a semi-colon after it instead of a full stop.

```
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;
end.
```

You can also use *if* statements inside other *if* statements.

```
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');
end.
```

Case

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

```
program Decisions;

uses
   crt;

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.
```

Learn Pascal Programming Tutorial Lesson 6 - Loops

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 3 types of loops which are the *for* loop, *while* loop and *repeat until* loop.

For loop

The for loop uses a loop counter variable, which it adds 1 to each time, to loop from a first number to a last number.

```
program Loops;

var
   i: Integer;

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

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

```
program Loops;

var
   i: Integer;

begin
   for i := 1 to 10 do
      begin
         Writeln('Hello');
         Writeln('This is loop ',i);
      end;
end.
```

While loop

The *while* loop repeats while a condition is true. The condition is tested at the top of the loop and not at any time while the loop is running as the name suggests. A while loop does not need a loop variable but if you want to use one then you must initialize its value before entering the loop.

```
program Loops;

var
```

```
   i: Integer;

begin
   i := 0;
   while i <= 10
      begin
         i := i + 1;
         Writeln('Hello');
      end;
end.
```

Repeat until loop

The *repeat until* loop is like the *while* loop except that it tests the condition at the bottom of the loop. It also doesn't have to have a *begin* and an *end* if it has more than one command inside it.

```
program Loops;

var
   i: Integer;

begin
   i := 0;
   repeat
      i := i + 1;
      Writeln('Hello');
   until i = 10;
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 Loops;

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.
```

Break and Continue

The *Break* command will exit a loop at any time. The following program will not print anything because it exits the loop before it gets there.

```
program Loops;

var
```

```
   i: Integer;

begin
   i := 0;
   repeat
      i := i + 1;
      Break;
      Writeln(i);
   until i = 10;
end.
```

The *Continue* command will jump back to the top of a loop. This example will also not print anything but unlike the *Break* example, it will count all the way to 10.

```
program Loops;

var
   i: Integer;

begin
   i := 0;
   repeat
      i := i + 1;
      Continue;
      Writeln(i);
   until i = 10;
end.
```

Learn Pascal Programming Tutorial Lesson 7 - Arrays

Arrays are variables that are made up of many variables of the same data type but have only one name. Here is a visual representation of an array with 5 elements:

1	value 1
2	value 2
3	value 3
4	value 4
5	value 5

Arrays are declared in almost the same way as normal variables are declared except that you have to say how many elements you want in the array.

```
program Arrays;

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

begin
end.
```

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. With 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 + 1 to 5 do
         if a[i] > a[j] then
         begin
            tmp := a[i];
            a[i] := a[j];
            a[j] := tmp;
         end;

   for i := 1 to 5 do
      writeln(i,': ',a[i]);
end.
```

2D 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.
```

You can get multi-dimensional arrays that have more than 2 dimensions but these are not used very often so you don't need to worry about them.

Learn Pascal Programming Tutorial Lesson 8 - Types, Records and Sets

Types

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. 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:

```
program Types;

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

begin
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.
```

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

```
program Types;

Type
   StudentRecord = Record
      Number: Integer;
      Name: String;
   end;

var
   Student: StudentRecord;

begin
   Student.Number := 12345;
```

```
   Student.Name := 'John Smith';
end.
```

The other type is a set. Sets are not very useful and anything you can do with a set can be done just as easily in another way. The following is an example of a set called Animal which has dog, cat and rabbit as the data it can store:

```
program Types;

Type
   Animal = set of (dog, cat, rabbit);

var
   MyPet: Animal;

begin
   MyPet := dog;
end.
```

You can't use *Readln* or *Writeln* on sets so the above way of using it is not very useful. You can create a range of values as a set such as 'a' to 'z'. This type of set can be used to test if a value is in that range.

```
program Types;

uses
   crt;

Type
   Alpha = 'a'..'z';

var
   Letter: set of Alpha;
   c: Char;

begin
   c := ReadKey;
   if c in [Letter] then
      Writeln('You entered a letter');
end.
```

Learn Pascal Programming Tutorial Lesson 9 - Procedures and Functions

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.
```

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.
```

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.
```

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.
```

Functions

Functions are like procedures except they return a value. The *function* keyword is used instead of *procedure* when declaring a function. To say what data type the return value must be you must use a colon and the name of the type after the function's name.

```
program 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.
```

Learn Pascal Programming Tutorial Lesson 10 - Text Files

You should by now know that a text file is a file with lines of text. When you want to access a file in Pascal you have to first create a file variable.

```
program Files;

var
   f: Text;

begin
end.
```

After the variable has been declared you must assign the file name to it.

```
program Files;

var
   f: Text;

begin
   Assign(f,'MyFile.txt');
end.
```

To create a new empty file we use the *Rewrite* command. This will overwrite any files that exist with the same name.

```
program Files;

var
   f: Text;

begin
   Assign(f,'MyFile.txt');
   Rewrite(f);
end.
```

The *Write* and *Writeln* commands work on files in the same way they work on the screen except that you must use an extra parameter to tell it to write to the file.

```
program Files;

var
   f: Text;

begin
   Assign(f,'MyFile.txt');
   Rewrite(f);
   Writeln(f,'A line of text');
end.
```

If you want to read from a file that already exists then you must use *Reset* instead of *Rewrite*. Use *Readln* to read lines of text from the file. You will also need a *while* loop that repeats until it comes to the end of the file.

```
program Files;

var
   f: Text;
   s: String;

begin
   Assign(f,'MyFile.txt');
   Reset(f);
   while not eof(f) do
      Readln(f,s);
end.
```

Append opens a file and lets you add more text at the end of the file.

```
program Files;

var
   f: Text;
   s: String;

begin
   Assign(f,'MyFile.txt');
   Append(f);
   Writeln('Some more text');
end.
```

No matter which one of the 3 access types you choose, you must still close a file when you are finished using it. If you don't close it then some of the text that was written to it might be lost.

```
program Files;

var
   f: Text;
   s: String;

begin
   Assign(f,'MyFile.txt');
   Append(f);
   Writeln('Some more text');
   Close(f);
end.
```

You can change a file's name with the *Rename* command and you can delete a file with the *Erase* command.

```
program Files;

var
   f: Text;

begin
```

```
   Assign(f,'MyFile.txt');
   Rename(f,'YourFile.txt');
   Erase(f);
   Close(f);
end.
```

To find out if a file exists, you must first turn off error checking using the {$I-} compiler directive. After that you must *Reset* the file and if *IOResult* = 2 then the file was not found. If *IOResult* = 0 then the file was found but if it is any other value then the program must be ended with the *Halt* command. *IOResult* loses its value once it has been used so we also have to put that into another variable before using it. You must also use {$I+} to turn error checking back on.

```
program Files;

var
   f: Text;
   IOR: Integer;

begin
   Assign(f,'MyFile.txt');
{$I-}
   Reset(f);
{$I+}
   IOR := IOResult;
   if IOR = 2 then
      Writeln('File not found');
   else
      if IOR <> 0 then
         Halt;
   Close(f);
end.
```

Learn Pascal Programming Tutorial Lesson 11 - Data Files

Data files are different from text files in a few ways. Data files are random access which means you don't have to read through them line after line but instead access any part of the file at any time. Here is how you declare a data file:

```
program DataFiles;

var
   f: file of Byte;

begin
end.
```

We then use *Assign* in the same way as we do with a text file.

```
program DataFiles;

var
   f: file of Byte;

begin
   Assign(f,'MyFile.txt');
end.
```

You can use *Rewrite* to create a new file or overwrite an existing one. The difference between text files and data files when using *Rewrite* is that data files can be read and written to.

```
program DataFiles;

var
   f: file of Byte;

begin
   Assign(f,'MyFile.txt');
   Rewrite(f);
end.
```

Reset is the same as *Rewrite* except that it doesn't overwrite the file.

```
program DataFiles;

var
   f: file of Byte;

begin
   Assign(f,'MyFile.txt');
   Reset(f);
end.
```

When you write to a file using the *Write* command you must first put the value to be written to the file into a variable. Before you can write to or read from a data file you

must use the *Seek* command to find the right place to start writing. You must also remember that data files start from position 0 and not 1.

```
program DataFiles;

var
   f: file of Byte;
   b: Byte;

begin
   Assign(f,'MyFile.txt');
   Reset(f);
   b := 1;
   Seek(f,0);
   Write(f,b);
end.
```

The *Read* command is used to read from a data file.

```
program DataFiles;

var
   f: file of Byte;
   b: Byte;

begin
   Assign(f,'MyFile.txt');
   Reset(f);
   Seek(f,0);
   Read(f,b);
end.
```

You must close a data file when you are finished with it just like with text files.

```
program DataFiles;

var
   f: file of Byte;
   b: Byte;

begin
   Assign(f,'MyFile.txt');
   Reset(f);
   Seek(f,0);
   Read(f,b);
   Close(f);
end.
```

The *FileSize* command can be used with the *FilePos* command to find out when you have reached the end of the file. *FileSize* returns the actual number of records which means it starts at 1 and not 0. The *FilePos* command will tell at what position in the file you are.

```
program DataFiles;

var
   f: file of Byte;
   b: Byte;
```

```
begin
   Assign(f,'MyFile.txt');
   Reset(f);
   while FilePos(f) <> FileSize(f) do
      begin
         Read(f,b);
         Writeln(b);
      end;
   Close(f);
end.
```

The *Truncate* command will delete everything in the file from the current position.

```
program DataFiles;

var
   f: file of Byte;

begin
   Assign(f,'MyFile.txt');
   Reset(f);
   Seek(f,3);
   Truncate(f);
   Close(f);
end.
```

One of the most useful things about data files is that you can use records with them.

```
program DataFiles;

type
   StudentRecord = Record
   Number: Integer;
   Name: String;

var
   Student: StudentRecord;

begin
   Assign(f,'MyFile.txt');
   Reset(f);
   Student.Number := 12345;
   Student.Name := 'John Smith';
   Write(f,Student);
   Close(f);
end.
```

Learn Pascal Programming Tutorial Lesson 12 - Units

We already know that units, such as the crt unit, let you use more procedures and functions than the built-in ones. You can make your own units which have procedures and functions that you have made in them.

To make a unit you need to create new Pascal file which we will call MyUnit.pas. The first line of the file should start with the *unit* keyword followed by the unit's name. The unit's name and the unit's file name must be exactly the same.

```
unit MyUnit;
```

The next line is the interface keyword. After this you must put the names of the procedures that will be made available to the program that will use your unit. For this example we will be making a function called *NewReadln* which is like *Readln* but it lets you limit the amount of characters that can be entered.

```
unit MyUnit;

interface

function NewReadln(Max: Integer): String;
```

The next line is *implementation*. This is where you will type the full code for the procedures and functions. We will also need to use the crt unit to make *NewReadln*. We end the unit just like a normal program with the *end* keyword.

```
unit MyUnit;

interface

function NewReadln(Max: Integer): String;

implementation

function NewReadln(Max: Integer): String;
var
   s: String;
   c: Char;
begin
   s := '';
   repeat
      c := ReadKey;
      if (c = #8){#8 = BACKSPACE} and (s <> '') then
         begin
            Write(#8+' '+#8);
            delete(s,length(s),1);
         end;
      if (c <> #8) and (c <> #13){#13 = ENTER} and (length(s) < Max) then
         begin
            Write(c);
            s := s + c;
         end;
```

```
   until c = #13;
   NewReadln := s;
end;

end.
```

Once you have saved the unit you must compile it. Now we must make the program that uses the unit that we have just made. This time we will type MyUnit in the *uses* section and then use the *NewReadln* function.

```
program MyProgram;

uses
   MyUnit;

var
   s: String;

begin
   s := NewReadln(10);
end.
```

Learn Pascal Programming Tutorial Lesson 13 - Pointers

What is a pointer?

A pointer is a type of variable that stores a memory address. Because it stores a memory address it is said to be pointing to it. There are 2 types of pointers which are typed and untyped. A typed pointer points to a variable such as an integer. An untyped pointer can point to any type of variable.

Declaring and using typed pointers

When you declare a typed pointer you must put a ^ in front of the variable type which you want it to point to. Here is an example of how to declare a pointer to an integer:

```
program Pointers;

var
   p: ^integer;

begin
end.
```

The @ sign can be used in front of a variable to get its memory address. This memory address can then be stored in a pointer because pointers store memory addresses. Here is an example of how to store the memory address of an integer in a pointer to an integer:

```
program Pointers;

var
   i: integer;
   p: ^integer;

begin
   p := @i;
end.
```

If you want to change the value stored at the memory address pointed at by a pointer you must first dereference the pointer variable using a ^ after the pointer name. Here is an example of how to change the value of an integer from 1 to 2 using a pointer:

```
program Pointers;

var
   i: integer;
   p: ^integer;

begin
   i := 1;
   p := @i;
   p^ := 2;
```

```
   writeln(i);
end.
```

You can allocate new memory to a typed pointer by using the *new* command. The *new* command has one parameter which is a pointer. The *new* command gets the memory that is the size of the variable type of the pointer and then sets the pointer to point to the memory address of it. When you are finished using the pointer you must use the *dispose* command to free the memory that was allocated to the pointer. Here is an example:

```
program Pointers;

var
   p: ^integer;

begin
   new(p);
   p^ := 3;
   writeln(p^);
   dispose(p);
end.
```

Declaring and using untyped pointers

When you declare an untyped pointer you must use the variable type called *pointer*.

```
program Pointers;

var
   p: pointer;

begin
end.
```

When you allocate memory to an untyped pointer you must use the *getmem* command instead of the *new* command and you must use *freemem* instead of *dispose*. *getmem* and *freemem* each have a second parameter which is the size in bytes of the amount of memory which must be allocated to the pointer. You can either use a number for the size or you can use the *sizeof* function to get the size of a specific variable type.

```
program Pointers;

var
   p: pointer;

begin
   getmem(p,sizeof(integer));
   freemem(p,sizeof(integer));
end.
```

Learn Pascal Programming Tutorial Lesson 14 - Linked Lists

What is a linked list

A linked list is like an array except that the amount of elements in a linked list can change unlike an array. A linked list uses pointers to point to the next or previous element.

Single linked lists

There are 2 types of single linked lists which are called queues and stacks.

Queues

A queue is like standing in a queue at a shop. The first person that joins a queue is the first person to be served. You must always join at the back of a queue because if you join at the front the other people will be angry. This is called FIFO(First In First Out).

Item 1 --> Item 2 --> Item 3 --> (Until the last item)

Each item of a linked list is a record which has the data and a pointer to the next or previous item. Here is an example of how to declare the record for a queue and a pointer to a queue record as well as the variables needed:

```
program queue;

type
   pQueue = ^tqueue;
   tQueue = record
      data: integer;
      next: pQueue;
   end;

var
   head, last, cur: pQueue;

begin
end.
```

We will now make 3 procedures. The first procedure will add items to the list, the second will view the list and the third will free the memory used by the queue. Before we make the procedures lets first take a look at the main program.

```
begin
   head := nil; {Set head to nil because there are no items in the queue}
   add(1) {Add 1 to the queue using the add procedure};
   add(2);
   add(3);
```

```
   view; {View all items in the queue}
   destroy; {Free the memory used by the queue}
end.
```

The add procedure will take an integer as a parameter and add that integer to the end of the queue.

```
procedure add(i: integer);
begin
   new(cur); {Create new queue item}
   cur^.data := i; {Set the value of the queue item to i}
   cur^.next := nil; {Set the next item in the queue to nil because it
doesn't exist}
   if head = nil then {If there is no head of the queue then}
      head := cur {Current is the new head because it is the first item
being added to the list}
   else
      last^.next := cur; {Set the previous last item to current because it
is the new last item in the queue}
   last := cur; {Make the current item the last item in the queue}
end;
```

The view procedure uses a loop to display the data from the first item to the last item of the queue.

```
procedure view;
begin
   cur := head; {Set current to the beginning of the queue}
   while cur <> nil do {While there is a current item}
      begin
         writeln(cur^.data); {Display current item}
         cur := cur^.next; {Set current to the next item in the queue}
      end;
end;
```

The destroy procedure will free the memory that was used by the queue.

```
procedure destroy;
begin
   cur := head; {Set current to the beginning of the queue}
   while cur <> nil do {While there is a item in the queue}
      begin
         cur := cur^.next; {Store the next item in current}
         dispose(head); {Free memory used by head}
         head := cur; {Set the new head of the queue to the current item}
      end;
end;
```

Stacks

To understand a stack you must think about a stack of plates. You can add a plate to the top of the stack and take one off the top but you can't add or take away a plate from the bottom without all the plates falling. This is called LIFO(Last In First Out).

Item 1 <-- Item 2 <-- Item 3 <-- (Until the last item)

When you declare the record for a stack item you must use previous instead of next. Here is an example.

```
program stack;

type
   pStack = ^tStack;
   tStack = record
      data: integer;
      prev: pStack;
   end;

var
   last, cur: pStack;

begin
   last := nil;
   add(3);
   add(2);
   add(1);
   view;
   destroy;
end.
```

You will see that the numbers are added from 3 to 1 with a stack instead of 1 to 3. This is because things must come off the top of the stack instead of from the head of a queue.

The add procedure adds the item after the last item on the stack.

```
procedure add(i: integer);
begin
   new(cur); {Create new stack item}
   cur^.data := i; {Set item value to the parameter value}
   cur^.prev := last; {Set the previous item to the last item in the stack}
   last := cur; {Make the current item the new last item}
end;
```

The view and destroy procedures are almost the same as with a queue so they will not need to be explained.

```
procedure view;
begin
   cur := last;
   while cur <> nil do
      begin
         writeln(cur^.data);
         cur := cur^.prev;
      end;
end;

procedure destroy;
begin
   while last <> nil do
      begin
         cur := last^.prev;
         dispose(last);
         last := cur;
```

```
        end;
    end;
```