Pascal Programming

Pascal is an influential computer programming language named after the mathematician Blaise Pascal. It was invented by Niklaus Wirth in 1968 as a research project into the nascent field of compiler theory.

Contents

Alternative resources

•http://www.taoyue.com/tutorials/pascal/index.html
•http://wiki.freepascal.org/
•http://chadcrabtree.com/pascal_tutorial/paslist.html

Source
http://en.wikibooks.org

Beginning

Here is a basic program that does absolutely nothing:

```
program first;
begin
   (*comment*)
end.
```

Here is a line by line description of what the different lines do.

1. `program first;` - is the "program header". Program headers are for dealing with multiple programs and "units". For most dialects this is optional.
2. `begin` - tells the compiler where the program begins. In more complex programs this statement might be preceded by statements that declare variables, set up functions and handle other preparation before the main program begins.
3. `(*comment*) - is a comment that is ignored by the compiler. The (* and *) tell the compiler to ignore what's in between them, almost all dialects of pascal also use braces ({ and }) for the same purpose. In the original Pascal, these can't be embedded like this: (*(*example*)*). However, in almost all dialects that use braces also use precedence, so this works: {(*example*)}. In some dialects such as Free Pascal's default mode comments can be embedded.`
4. `end.` - (notice the period) tells the compiler to stop compiling. In fact, anything after that is completely ignored. As we shall see, there may be many `begin` and `end` statements in a program, but the `end statement that terminates the program is the only statement that ends in a period.`

While programs that do absolutely nothing may compile, they aren't generally that exciting. It would be much better to have a program that introduced itself to the world. These programs are very popular, especially when introducing a new language, and if you've ever learned another programming language you may be familiar with such "Hello World" examples.

```
program HelloWorld;
begin (*This is where the program begins*)
   writeln('Hello world!');
end. {This is where the program ends.}
```

Now instead of doing nothing our program contains one line, namely "writeln('Hello World!');". The command writeln prints its argument to the terminal. In the current case its argument, 'Hello World!' is a string. If you are familiar with other programming languages it would be useful to note that in Pascal strings and characters are contained between single quotes and not double quotes. The last but important thing to notice is the ; at the end of the writeln statement. Almost all statements in Pascal should end with a semicolon. This is because the compiler doesn't care about the white space you put, including line breaks. So our program would have been just as functional as

```
program HelloWorld;
begin (*This is where the program begins*)
                                    writeln('Hello world!');



end. {This is where the program ends.}
```

or

```
program HelloWorld;begin (*This is where the program begins*)writeln('Hello
world!');end.{This is where the program ends.}
```

To the compiler they are exactly the same. But the compiler still needs to know where one statement ends and the next begins. For this reason Pascal uses the semicolon to mark where lines end. The observant reader will notice that we only said that "almost all" lines need to end in a semi-colon. We shall point out the exceptions as they arise, there are three Exceptions in this program. The first exceptions is the line "begin", and the second is the line end. (since this terminates the program it ends in a period instead of a semicolon.). The last exception is that the line that comes before an "end" doesn't need to end in a semicolon. So really in this program the line "writeln('Hello World!');" didn't actually need a semicolon despite all the discussion about it. On the other hand, there is no harm in including the semicolon. It also makes the life easier later if you decide to add a line after the "writeln" command, because then you no don't need to remember to add it later.

Unlike many programming languages the Pascal language is not case sensitive! For example consider the following program:

```
program HelloWorld;
begin (*This is where the program begins*)
   writeln('Hello world!');
   WriteLn('It is nice to meet you.')
end. {This is where the program ends.}
```

Here both "writeln" and "WriteLn" refer to the same function. We could have also changed any of the other keywords in the program to as we liked. One of the benefits of ignoring white space and case is that it allows us to structure the program in which ever way is most readable to us.

As a final comment, depending on your operating system and how you start any of these programs you have compiled, you may find that the window closes the instant the program is finished, which for these programs is almost instantly. One useful trick to avoid this is to ask the program to wait for input from the user. This can be done with the "readln" command. So our simple HelloWorld program becomes:

```
program HelloWorld;
begin (*This is where the program begins*)
   writeln('Hello world!');
   readln;                      {This reads a line of input from the keyborad}
                                {So now the program waits for us to press enter
:)}
end. {This is where the program ends.}
```

Variables and Constants

Naming

> ... All names of things that are defined are identifiers. An identifier consists of 255 significant characters (letters, digits and the underscore character), from which the first must be an alphanumeric character, or an underscore (_) ...

Van Canneyt, Michaël. "Free Pascal: Reference guide.". Retrieved 2008-08-17.

Pascal's identifiers are case-insensitive, meaning that for everything except characters and strings its case doesn't matter (`THING`, `thing`, `ThInG`, and etc are the same).

Constants

```
program Useless;
const
    zilch = 0;
begin
    (*some code*)
    if zilch = zilch then
    (*more code*)
end.
```

Constants (unlike variables) can't be changed while the program is running. This assumption allows the compiler to simply replace `zilch` instead of wasting time to analyze it or wasting memory. As you may expect, constants must be definable at compile-time. However, Pascal was also designed so that it could be compiled in 1 pass from top to bottom(the reason being to make compiling fast and simple). Simply put, constants must be definable from the code before it, otherwise the compiler gives an error. For example, this would not compile:

```
program Fail;
const
    zilch = c-c;
    c = 1;
begin
end.
```

but this will:

```
program Succeed;
const
    c = 1;
    zilch = c-c;
begin
end.
```

It should be noted that in standard Pascal, constants must be defined before anything else (most compilers wont enforce it though). Constants may also be typed but must be in this form: `identifier: type = value;`.

Variables

Variables must first be declared and typed under the `var` like so:

```
var
    number: integer;
```

You can also define multiple variables with the same type like this:

```
var
    number,othernumber: integer;
```

Declaration tells the compiler to pick a place in memory for the variable. The type tells compiler how the variable is formatted. A variable's value can be changed at run time like this: `variable := value`. There are more types but, we will concentrate on these for now:

type	definition
integer	a whole number from -32768 to 32767
real	a floating point number from 1E-38 to 1E+38
boolean	either true or false
char	a character in a character set (almost always ASCII)

Input and Output

Output

Before you learn more about Pascal it is important to know how to actually do something. For now, we will use console input and output. Here is an example of how output is done:

```
program output;
begin
  writeln(1);
  write('1');
end.
```

`writeln()` displays what ever is in the parentheses and prints a "new line character", making newly displayed characters start on the next line.

`;` is simply grammatical; it just separates the two statements.

`write()` displays what ever is in the parentheses without printing a "new line character".

Do note that characters and strings must be placed within single quotation marks. You may have noticed that `write` and `writeln` work for more than one type. You may also wonder how `write(1)` and `write('1')` could do the same thing yet, `1` and `'1'` be represented in completely different ways. Like wise, how `49` and `'1'` may be represented the same way yet, `write(49)` and `write('1')` don't do the same thing. The answer is that compiler always knows the type and thus how it's formatted.

```
readln;
```

end. </source> As you notice, the parentheses are not required when no parameters are present.

Boolean Expressions and Control Flow

Conditional Statements

If..Then..Else

The simplest control structure is the if..then..else statement.

```
if var1 = 0 then
    writeln('var1 is 0!')  (*No semicolon before an 'Else' keyword*)
else if var1 = 1 then
begin
    writeln('var1 is 1!');
    (*More code*)
end  (*No semicolon before an 'Else' keyword*)
else if var1 = 2 then
begin
    writeln('var1 is 2!');
    (*More code*)
end  (*No semicolon before an 'Else' keyword*)
else
begin
    writeln('var1 is not 0, 1, or 2!');
    (*More code*)
end;  (*Semicolon used to indicate the end of the If-then-else*)
```

Case

```
Case var1 of
  1 : writeln('var1 is 1!');
  2 : Begin
    writeln('var1 is 2!');
    (*More code*)
  End;
  Else writeln('var1 is neither 1 nor 2!'); (*'else' can be used instead of
the 'otherwise' keyword*)
end;
```

Loops

For

```
For var1 := 0 to 12 do  (*For conditional with only one statement:*)
  writeln('var1 is ', var1);

For var2 := 12 downto 0 do Begin  (*For conditional with multiple
statements:*)
  writeln('var2 is ', var2);
  (*More code*)
End;
```

While

```
While var1 = 0 do (*While the condition is True, perform the following code*)
  writeln('woo, an infinite loop!');

While var2 = 0 do Begin (*While-do with multiple statements:*)
  writeln('aww, no infinite loop!');
  var2 := 1;
End;
```

Repeat..Until

```
Repeat (*Repeat the following until the condition is True:*)
  writeln('var1 might be 0!');
  (*More code - Begin..End is not required between Repeat..Until*)
  (*Semicolon needed before the 'Until' keyword*)
Until var1 = 0;
```

Syntax and functions

Syntax

As you've seen in the earlier chapter, Pascal programs have a standard structure which looks like the following:

```
Program program_name;

{ Global variables }
Var
 A_Variable: Variable_Type;

{ Other functions/procedures }

Procedure SayHello;

 { Local variables }
 Var
  T : String;

 Begin
  { Redundant code to illustrate the use of local variables in a procedure }
  T := 'Hello';
  Writeln(T);
 End;

{ Main function }

Begin
 { Do something }
 SayHello;
End.
```

A program has a *program header*, followed by *global variable definitions*, *procedure or function definitions* and finally the *main function*.

Variables

As you may already figure, variable definitions are put in a block beginning with a `var` keyword, followed by definitions of the variables you wish to define. This block has no explicit end marker.

As you can see from the example, unlike C/C++, Pascal variables are declared outside the code-body of the function (i.e. they are not declared within the `begin` and `end` pairs), but are instead declared after the definition of the procedure/function and before the `begin` keyword. For global variables, they are defined after the `program` header.

A declaration in the `var` block has the following syntax:

```
A_Variable,Another_Variable ... : Variable_Type;
```

Declarations may occur on multiple lines, as in the following:

```
a,b : integer;
c : integer;
d,e : string;
f : real;
g : extended;
h,i,j,k : byte;
l,m,n,o : byte;
```

Basic numeric types include: `longint` (32-bit, hardware dependent), `integer` (16-bit, hardware dependent), `shortint` (8-bit). Their unsigned counterparts are cardinal (available only in some versions of Pascal), `word`, `byte`. Decimal numbers are supported using the `real` type, and the `extended` type (available only in some versions of Pascal)

Other types include the char (for holding characters), the string (as its name suggests).

(For now, arrays, pointers, types and records will be covered in a later chapter)

Functions/Procedures

Before we begin, let us first clarify the key difference between functions and procedures. A procedure is set of instructions to be executed, with no return value. A function is a procedure **with a return value**. For readers familiar with C/C++, a procedure is simply a function with a void return value, as in void proc_sayhello().

The definition of function/procedures is thus as such:

```
Function Func_Name(params...) : Return_Value;
Procedure Proc_Name(params...);
```

The function/procedure definition is usually followed by the local variables and the body. However, to provide prototypes, simply add a **forward** keyword behind the definition instead of the local variables and the body. Of course, the whole function must be defined somewhere else in the program. The following example illustrates the use of this:

```
Function Add(A, B : Integer): Integer; Forward;

Function Bad(A, B, C : Integer) : Integer;

Begin
 Bad := Add(Add(A,B),C);
End;

Function Add(A, B : Integer) : Integer;
Begin
 Add := A + B;
End;
```

In this example, Add is first defined as a function taking two integer variables and returning an integer, but it is defined as a forward definition (prototype), and thus no body is written. Later, we see that Add is defined with a body. Note that the two definitions of Add must be congruent with each other, or the compiler will complain.

From the above example, we can also gather that in Pascal, a function's return value is given by the value of the variable with the function's name (or by the variable named `result`), when the function returns. As you can see in the Bad function, an undefined variable named "Bad" has been assigned a value. That is the return value for the Bad function. Similarly, in Add, the variable named "Add" has been assigned a value, which is its return value.

Note that unlike C or other languages, assigning a return value to a function **does not** return from the function. Thus, the function will continue executing, as in the following example:

```
Function Weird(A : Integer) : Integer;

Var
 S : Integer;

Begin
 S := A/2;

 If S < 10 Then
  Weird := 1;

 S := S + 9;

 If S >= 10 Then
  Weird := 0;

 Weird := 2;
End;
```

If A happens to be 6, the function will not return the expected result of 1 or even 0. Instead, it would return a result of 2, because the function to execute continues even after the return value is set. In fact, as you would notice, the function would return 2 all the time because it runs all the way to the end, at which the return value is set to 2.

To mimic C style function returns, the exit statement must be used. The exit statement in Pascal, unlike C, exits from the current block of code (in this case, the function), and NOT from the program. The code would then look like this:

```
Function Weird(A : Integer) : Integer;

Var
 S : Integer;

Begin
 S := A/2;

 If S < 10 Then
 Begin
  Weird := 1;
  Exit;
 End;

 S := S + 9;

 If S >= 10 Then
 Begin
  Weird := 0;
  Exit;
 End;

 Weird := 2;
End;
```

Note that a third exit is not necessary at the end of the function since nothing else would be executed that could overwrite the function return.

Enumerations

Sometimes, programmers need a way of expressing ordinal values as names. A good example is Pascal's boolean type, which is essentially an enumeration type that holds the values False and True. We can make our own with the following.

```
type
  TMonth = ( Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec );
```

A variable declared as an enumeration type such as TMonth is restricted to using only those values. It is possible to explicitly typecast a record variable to an ordinal type, but this is impractical most of the time.

```
var
  month: TMonth;
begin
  month := Mar;
end;
```

Extensions and optimizations

C-style enumerations

Some compilers, such as Free Pascal, offer an enumeration syntax in the style of C's enums. That is, ordinal values may be associated with the constants declared in an enumeration.

```
type
  TMonth = ( Jan = 0, Feb, Mar, Apr = 45, May );
```

Note that, just like in C, ordinal values assigned to an enumeration's constants must be in order from least to greatest, otherwise the compiler will generate an error.

Sets

With enumerations, there comes the opportunity to use sets. Sets store multiple values of an enumeration. They act as a set of switches, so to speak. I suppose an example should give a clearer meaning to this.

```
type
  Skills = ( Cooking, Cleaning, Driving, Videogames, Eating );

var
  Slob: set of Skills = [Videogames, Eating];
```

Here, we've made a variable Slob, which represents a set of the Skills enumeration. By default, set types are usually stored internally as LongInts if they have under 32 values. Anything above that, and they are stored as one byte per element. Modern compilers limit set types to up to 255 values maximum.

Putting set variables to use in code is as follows:

```
Slob := Slob + [Driving]; // Append a value to our existing Slob variable

if Videogames in Slob then // if Slob has the Videogames value...
  writeln('Is he a level 45 dungeon master?');
```

Arrays

Lists, or Arrays are types in Pascal that can be used to store multiple data using one variable.

Static Size Arrays

These lists are defined in syntax of:

```
type MyArray1 = array[0..100] of integer;

type MyArray2 = array[0..5] of array[0..10] of char;
 {or}
type MyArray2 = array[0..5,0..10] of char;
```

Examples of this would be having a list of different types or records.

```
type my_list_of_names = array[0..7] of string;

var the_list: my_list_of_names;

begin
  the_list[5] := 'John';
  the_list[0] := 'Newbie';
end;
```

This would create an array of ['Newbie','','','','','John','',''']. We can access these variables the same way as we have set them:

```
begin
  writeln('Name 0: ', the_list[0]);
  writeln('Name 5: ', the_list[5]);
end;
```

The following example shows an array with indices of chars.

```
var
  a: array['A'..'Z'] of integer;
  s: string;
  i: byte;

begin
  readln(s); {reads the string}
  for i := 1 to length(s) do {executes the code for each letter in the string}
    if upcase(s[i]) in['A'..'Z'] then
      inc( a[upcase(s[i])] );
    {counts the number the times a letter is counted. the case doesn't count}
  writeln('The number of times the letter A appears in the string is
',a['A']);
  {returns 5 if the string is 'Abracadabra'}
end.
```

Dynamic Arrays

Lists also can contain unlimited number of items, depending on the amount of computer memory. This is acheived by dynamic arrays, which appeared with **Delphi** (not available with **Turbo Pascal**). To declare a dynamic array, simply declare an array without bounds :

```
var a: array of <element_type>;
```

Then, define the size at any moment with the function *SetLength*

```
SetLength( a, <New_Length> );
```

The indices of the array will from 0 to length(a)-1.

You can get the current length with the *Length* function, as for String type. To go through the array, you can write for example :

```
var
  i: integer;
  a: array of integer;

begin
  randomize;
  setlength(a, 10); { a will contain 10 random numbers }
  for i := 0 to length(a)-1 do
    a[i] := random(10)+1;
end;
```

The memory is **freed automatically** when the array is not used anymore, but you can free it before by assigning **nil** value to the variable which is the same as defining a length of 0.

Strings

String is just an array of ASCII characters.

Definition

string type is defined like this:

```
type string = packed array [0..255] of char;
```

Consider this code:

```
program string_sample;
 uses crt;
 var s: string;
     i: longint;
 begin
  s:="This is a example of string";
  writeln(s[1]);
  writeln(ord(s[0]));
  readln;
 end.
```

The output:

```
T27
```

We can see that:

•Because strings are just an ASCII array, we can access to any character in string just like array (`s[1]` in the example above will return the character `T`)

•What is the purpose of `s[0]`? `s[0]` stores the length of the string *s* but the length is not stored as a number, it is stored as the ASCII character of the length. For example, if string *s* has the length of 65, `s[0]` will return the character *A*, because the ASCII number of *A* is 65. And the first character of the string is stored in `s[1]`. So a string can only store up to 255 characters. ***But don't use this method to retrieve the length of a string because there is a function that can do that (see below)***

Declare

Just like *longint* or *integer*:

```
var s: string;
```

In the above example, string *s* can store up to 255 characters.
But what if you want the string *s* to store up to 10 characters?

```
var s: string[10]; {OK, so now s can only store up to 10 chars}
```

Records

Often, it becomes necessary for programmers to want or need to store structured data, such as the attributes of a person, for example. In Pascal, we can declare a record type. This is a user-defined data type to store structured data as follows:

```
type
  TPerson = record
    name: String[16];
    age: Integer;
    gender: TGender; // TGender is assumed here to be an enumerated type that
holds the values Male and Female
  end;
```

Next, we can use TPerson as a type in our code in the following ways.

```
var
  p: TPerson;
  x: integer;
begin
  p.name := 'Bob';
  p.age := 37;
  p.gender := Male;
  x := p.age;
end;
```

The 'with' clause

As you can see in the above example, the calls to our record variable get a little tedious. Fortunately, Pascal offers us the 'with' clause. This will ease the way you use record (and Turbo Pascal-compatible object) variables.

```
with p do
begin
  name := 'Bob';
  age := 37;
  gender := Male;
end;
```

Pointers

Pointers are pointers or addresses to specific variables in the memory. Pointers allow the developer to make an alias or referencing of a specific variable. Professionally, Pointers are being used for lists since they require less memory although they are more intricate.

A sample pointer app:

```
program Pointers;

var
  number : integer;
  { ^ before the type shows that it's a pointer }
  numbers_pointer : ^integer;

begin
  { Set to 5 }
  number := 5;
  { Output }
  writeln('Number is ', number);

  { Assign the number's address, which is @number to numbers_pointer }
  numbers_pointer := @number;
  { To access the pointer's address, you've got to add a ^ after the pointer
variable's name: }
  numbers_pointer^ := 8;
  writeln('Pointed Content is: ', numbers_pointer^); { 8 }

  writeln('Number is: ', number); { Should be 8 }
end.
```

Pointers are introduced as lists, explained above. Simply, you've got to point to the next or the previous record.

Note that there are three ways pointers are notated:

- the "@" indicates the memory address of another type; it is a common way of initializing a pointer. - when "^" is placed before a name, you are asking for a pointer for a particular *type*, like a pointer to an integer or a char. Alternately you can use the generic "Pointer" type if you wish to use a pointer to reference many kinds of objects. - when "^" is placed after a name, you are asking for a *dereference* - for an existing pointer to return the variable it's referencing. So if you have a variable you wish to change, but have only a pointer to access it from, you use "variable^" to obtain the value.

Pascal pointers are often notated as "Pvar" where "var" or "Tvar" is the original.

One significant difference between C and Pascal programming is that C requires the use of pointers in more cases. When you call a function in C, there is no "var" keyword to indicate pass-by-reference; instead, C expects you to call the function with a pointer to the variable you want changed, and then dereference the pointer inside the function. Although the functionality is nearly the same, Pascal was originally designed to use pass-by-reference for subroutines, and pointers for complex data structures; later implementations added more generalized functionality for pointers.

Modern Pascal adds plain pointer type, named Pointer, which is compatible to C's `void*`. This pointer type is semantically compatible with any other pointers. Therefore, the following code snippet is valid:

```
var
  P: Pointer;
  PL: PLongWord;
  PB: PByte;
  PC: PChar;

begin
  New(PL);
  // Assuming little endian, the byte order would be: $41424300
  PL^ := $00434241;
  P := PL;
  // Now PB can read PI's value byte per byte
  PB := PByte(P);
  // Byte order cross check
  WriteLn(PB^,' ',(PB + 1)^,' ',(PB + 2)^,' ',(PB + 3)^);
  PC := PChar(P);
  // Guess what?
  WriteLn(PC);
  Dispose(PL);
end.
```

Another Modern Pascal enhancement regarding pointers is that you can treat it as array of infinite length. The byte order cross check above could be rewritten as:

```
WriteLn(PB[0],' ',PB[1],' ',PB[2],' ',PB[3]);
```

Be careful, this enhancement also brings bad things. For instance, what if you access index that's out of your program's memory region? That's why you better be sure to have range checking on whenever you use this feature (except for a guaranteed free of errors code).

Object oriented

Object Oriented Pascal allows the user to create applications with Classes and Types. This saves the developer time on developing programs that would be very flexible.

This is a sample program (tested with the FreePascal compiler) that will store a number 1 in private variable One, increase it by one and then print it.

```
program types;  // this is a simple program
 type MyType=class
       private
        One:Integer;
       public
        function Myget():integer;
        procedure Myset(val:integer);
        procedure Increase();
      end;

 function MyType.Myget():integer;
 begin
   Myget:=One;
 end;
 procedure MyType.Myset(val:integer);
 begin
   One:=val;
 end;
 procedure MyType.Increase();
 begin
   One:=One+1;
 end;

 var
   NumberClass:MyType;
 begin
   NumberClass:=MyType.Create;  // creating instance
   NumberClass.Myset(1);
   NumberClass.Increase();
   writeln('Result: ',NumberClass.Myget());
   NumberClass.Free;  // destroy instance
 end.
```

This example is very basic and would be pretty useless when used as OOP. Much more complicated examples can be found in Delphi and Lazarus which include a lot of Object Oriented programming.

Cheatsheet

Syntax cheat sheet

- `monospaced` denotes keywords and syntax
- **[ ]** denotes optional syntax
- **|** denotes multiple possible syntaxes
- **( )** denotes grouped syntax

Statements

syntax	definition	availability
`if` *condition* `then begin` *statement(s)* `end; if` *condition* `then` *statement*`;`	Conditional statement	standard
`while` *condition* `do begin` *statement(s)* `end; while` *condition* `do` *statement*`;`	`while` loop	standard
`repeat` *statement(s)* `until` *condition*`;`	`repeat` loop	standard
`with` *variable* `do begin` *statement(s)* `end; with` *variable* `do` *statement*`;`	Eases the use of a variable or pointer variable of a structured type by omitting the dot notation for the variable.	standard

Appendix

Noteworthy types

type	definition	size (in bytes)	availability
AnsiChar	one ANSI-standard textual character	1	
AnsiString	an array of ANSI-standard characters of indefinite length with an optional size constraint	1 * number of characters	
Boolean	true or false	1	standard
Byte	whole number ranging from 0 to 255	1	
Cardinal	synonym depending on processor type (16 bit=Word, 32 bit=LongWord, 64 bit=QWord)	varies (2, 4, 8)	
Char	one textual character (likely ASCII)	1	standard
Comp	a floating point number ranging 19-20 digits that is effectively a 64-bit integer	8	
Currency	a floating point number ranging 19-20 digits that is a fixed point data type	8	
Double	a floating point number ranging 15-16 digits	8	
DWord	whole number	4	

	ranging from 0 to 4,294,967,295		
Extended	a floating point number ranging 19-20 digits	10	
Int64	whole number ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8	
Integer	synonym depending on processor type (16 bit=SmallInt, 32 bit=LongInt, 64 bit=Int64)	varies (2, 4, 8)	standard
LongInt	whole number ranging from -2,147,483,640 to 2,147,483,647	4	
LongWord	whole number ranging from 0 to 4,294,967,295	4	
Pointer	an untyped pointer holding an address	32 bit=4, 64 bit=8	standard
PtrUInt	a pointer type implicitly convertable to an unsigned integer	32 bit=4, 64 bit=8	
QWord	whole number ranging from 0 to 18,446,744,073,709,551,615	8	Free Pascal
Real	a floating point number whose range is platform dependent	4 or 8	standard
ShortInt	whole number ranging from -128 to	1	

	127		
ShortString	an array of textual characters of up to 255 elements (likely ASCII) with an optional size constraint	1 * number of characters (max 255)	standard
Single	a floating point number ranging 7-8 digits	4	
SmallInt	whole number ranging from -32,768 to 32,767	4	
String	synonym for ShortString (or AnsiString with the $H preprocessor directive turned on)	1 * number of characters (max 255)	standard
UInt64	whole number ranging from 0 to 18,446,744,073,709,551,615	8	Free Pascal, Delphi 8 or later
WideChar	one UTF-8 textual character	2	
WideString	an array of UTF-8 characters of indefinite length with an optional size constraint	2 * number of characters	
Word	whole number ranging from 0 to 65,535	2	

Noteworthy preprocessor directives

directive	description	value(s)	example	availability

$COPERATORS	allows use of C-style operators	OFF or ON	`{$COPERATORS ON}`
			`i += 5;`
			`i -= 5;`
			`i *= 5;`
			`i /= 5;`
				Free Pascal

$DEFINE	defines a symbol for the preprocessor (if '$macro on', can have a value assigned)	symbol name (:= value if '$macro on')	`{$DEFINE Foo}` `{$DEFINE Bar := 5}`	standard
$H	implies whether the String type is a ShortString or AnsiString	- or +		Delphi, Free Pascal
$I	inserts a file's contents into the current source code	filename	`{$I hello.txt}`	standard
$IF	begins a preprocessor conditional statement	compile-time boolean expression	`{$IF DEFINED(DELPHI) OR DECLARED(Foo)}`	standard
$IFDEF	begins a preprocessor conditional	preprocessor symbol	`{$IFDEF`	standard

	statement depending if a preprocessor symbol is defined		`MSWINDOWS}`	
$IFNDEF	begins a preprocessor conditional statement depending if a preprocessor symbol is not defined	preprocessor symbol	`{$IFNDEF UNIX}`	standard
$IFOPT	begins a preprocessor conditional statement depending on the status of a preprocessor switch	compiler option	`{$IFOPT D+}`	standard
$INCLUDE	inserts a file's contents into the current source code	filename	`{$INCLUDE hello.txt}`	standard
$INLINE	allows inline functions and procedures	OFF or ON	`unit Foo; {$INLINE ON} interface function Give5: integer; inline; implementation function Give5: integer; begin`	Free Pascal

```
Give5 := 5;
  end;
end.
```

$MACRO	allows defined symbols to hold values	OFF or ON	`{$MACRO ON}` `{$DEFINE Foo := 7}`	Free Pascal
$MODE	sets the Pascal dialect	DELPHI, FPC, MACPAS, OBJFPC, TP	Free Pascal	
$R	embeds a resource file into the code	a file name	`{$R *.dfm}`	Delphi, Free Pascal

$STATIC	allows use of the 'static' keyword	OFF or ON	Free Pascal

```
unit Foo;
{$STATIC
ON}
{$MODE
OBJFPC}
interface
  type
    Bar =
class

function
Baz:
string;
static;
    end;
implementat
ion
  function
Bar.Baz:
string;
  begin

Result :=
'This
function is
not part of
a Bar
instance.';
  end;
end.
```