Pascal Tutorial

PASCAL TUTORIAL

Simply Easy Learning by tutorialspoint.com

tutorialspoint.com

ABOUT THE TUTORIAL

Pascal Tutorial

Pascal is a procedural programming language, designed in 1968 and published in 1970 by Niklaus Wirth and named in honor of the French mathematician and philosopher Blaise Pascal.

Pascal runs on a variety of platforms, such as Windows, Mac OS, and various versions of UNIX/Linux.

This tutorial will give you great understanding of Pascal to proceed with Delphi and other related frameworks, etc.

Audience

This tutorial is designed for Software Professionals, who are willing to learn Pascal Programming Language in simple and easy steps. This tutorial will give you great understanding on Pascal Programming concepts, and after completing this tutorial, you will be at intermediate level of expertise from where you can take yourself to higher level of expertise.

Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of software basic concepts like what is source code, compiler, text editor and execution of programs, etc. If you already have understanding on any other computer programming language, then it will be an added advantage to proceed.

Compile/Execute Pascal Programs

If you are willing to learn the Pascal programming on a Linux machine but you do not have a setup for the same, then do not worry, the compileonline.com is available on a high end dedicated server giving you real programming experience with a comfort of single click compilation and execution. Yes! it is absolutely free and it's online.

Copyright & Disclaimer Notice

Table of Contents

Pascal Tutorial i
Audience i
Prerequisites i
Compile/Execute Pascal Programs i
Copyright & Disclaimer Notice i
Pascal Overview 1
Features of the Pascal Language? 1
Facts about Pascal 1
Why to use Pascal? 2
Environment 3
Installing Free Pascal on Linux 4
Installing Free Pascal on Mac 4
Installing Free Pascal on Windows 4
Text Editor 4
Program Structure 6
Pascal Hello World Example 7
Compile and Execute Pascal Program: 8
Basic Syntax 9
Functions/Procedures 9
Comments 9
Case Sensitivity 10
Pascal Statements 10
Reserved Words in Pascal 10
Character set and Identifiers in Pascal 10
Data Types 12
Pascal Data Types: 12
Type Declarations: 13
Integer Types 13
Constants 14
Enumerated types 14
Subrange Types 15
Variable Types 16
Basic Variables in Pascal 16
Variable Declaration in Pascal 17
Variable Initialization in Pascal 18
Enumerated Variables 19
Subrange Variables 19
Constants 21

Declaring Constants 21
Operators 23
Arithmetic Operators 23
Relational Operators 24
Boolean Operators 26
Bit Operators 27
..... 29
Operators Precedence in Pascal 29
Decision Making 31
Syntax: 34
Flow Diagram: 34
Example: 35
The if-then-else if-then-else Statement 35
Syntax: 35
Syntax: 37
Example: 37
Syntax: 38
Flow Diagram: 39
Example: 40
Syntax: 40
Flow Diagram: 41
Example: 41
Syntax: 42
Example: 42
Loops 43
while-do loop 44
Syntax: 44
Flow Diagram: 45
Example: 45
For-do LOOP 46
Syntax: 46
Example: 47
..... 47
Repeat-Until Loop 48
Syntax: 48
For example, ... 48
..... 48
Flow Diagram: 48
Example: 49

Example: 50
Loop Control Statements: 52
Syntax: 52
Flow Diagram: 52
Example: 53
Syntax: 54
Flow Diagram: 54
Example: 55
Syntax: 56
Flow Diagram: 56
Example: 57
Functions 58
Subprograms 58
Functions 58
Defining a Function: 59
Function Declarations: 60
Procedure 62
Defining a Procedure: 62
Procedure Declarations: 63
Calling a Procedure: 63
Recursive Subprograms 64
Arguments of a Subprogram: 66
Variable Scope 70
Local Variables 70
Global Variables 71
Strings 74
Examples 74
Pascal String Functions and Procedures 76
Boolean 79
Declaration of Boolean Data Types 79
Example: 80
Arrays 81
Declaring Arrays 81
Types of Array Subscript 82
Initializing Arrays 83
Accessing Array Elements 83
Pascal Arrays in Detail 84
Two-Dimensional Arrays: 85
Initializing Two-Dimensional Arrays: 85

Accessing Two-Dimensional Array Elements: 85
Declaring Dynamic Arrays 86
Declaring Packed Arrays 88
Pointers 90
What Are Pointers? 90
Printing a Memory Address in Pascal 91
NILL Pointers 92
Pascal Pointers in Detail: 93
Incrementing a Pointer 94
Decrementing a Pointer 94
Pointer Comparisons 95
Records 101
Defining a Record 101
Accessing Fields of a Record 102
Records as Subprogram Arguments 103
Pointers to Records 104
The With Statement 106
Variants 108
Declaring a Variant 108
Example: 109
Sets 110
Defining Set Types and Variables 110
Set Operators 111
Example: 112
File Handling 114
Creating and Writing to a File 115
Reading from a File 115
Files as Subprogram Parameter 116
Text Files 117
Appending to a File 118
File Handling Functions 118
Memory Management 124
Allocating Memory Dynamically 124
Resizing and Releasing Memory 126
Memory Management Functions 127
Units 131
Using Built-in Units 131
Creating and Using a Pascal Unit 132
Date Time 135

Getting the Current Date & Time: 135
Various Date & Time Functions: 136
Objects 142
Object Oriented Concepts: 142
Defining Pascal Objects 143
Visibility of the Object Members 146
Constructors and Destructors for Pascal Objects: 146
Inheritance for Pascal Objects: 148
Classes 152
Defining Pascal Classes: 152
Visibility of the Class Members 155
Constructors and Destructors for Pascal Classes: 156
Inheritance: 157
Interfaces: 160
Abstract Classes: 160
Static Keyword: 161

CHAPTER

1

Pascal Overview

This chapter describes the basic definition and concepts of Pascal.

Pascal is a general-purpose, high-level language that was originally developed by Niklaus Wirth in the early 1970s. It was developed for teaching programming as a systematic discipline and to develop reliable and efficient programs.

Pascal is Algol-based language and includes many constructs of Algol. Algol-60 is a subset of Pascal. Pascal offers several data types and programming structures. It is easy to understand and maintain the Pascal programs.

Pascal has grown in popularity in the teaching and academics arena for various reasons:

- Easy to learn.
- Structured language.
- It produces transparent, efficient and reliable programs.
- It can be compiled on a variety of computer platforms.

Features of the Pascal Language?

Pascal has the following features:

- Pascal is a strongly typed language.
- It offers extensive error checking.
- It offers several data types like arrays, records, files and sets.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object oriented programming.

Facts about Pascal

- The Pascal language was named for Blaise Pascal, French mathematician and pioneer in computer development.
- Niklaus Wirth completed development of the original Pascal programming language in 1970.

- Pascal is based on the block structured style of the Algol programming language.
- Pascal was developed as a language suitable for teaching programming as a systematic discipline, whose implementations could be both reliable and efficient.
- The ISO 7185 Pascal Standard was originally published in 1983.
- Pascal was the primary high-level language used for development in the Apple Lisa, and in the early years of the Mac.
- In 1986, Apple Computer released the first Object Pascal implementation, and in 1993, the Pascal Standards Committee published an Object-Oriented Extension to Pascal.

Why to use Pascal?

Pascal allows the programmers to define complex structured data types and build dynamic and recursive data structures, such as lists, trees and graphs. Pascal offers features like records, enumerations, subranges, dynamically allocated variables with associated pointers and sets.

Pascal allows nested procedure definitions to any level of depth. This truly provides a great programming environment for learning programming as a systematic discipline based on the fundamental concepts.

Among the most amazing implementations of Pascal are:

- Skype
- Total Commander
- TeX
- Macromedia Captivate
- Apple Lisa
- Various PC Games
- Embedded Systems

CHAPTER

2

Environment

This section describes the environmental setup for running Pascal

There are several Pascal compilers and interpreters available for general use.

Among these are:

- **Turbo Pascal:** provides an IDE and compiler for running Pascal programs on CP/M, CP/M-86, DOS, Windows and Macintosh.
- **Delphi:** provides compilers for running Object Pascal and generates native code for 32- and 64-bit Windows operating systems, as well as 32-bit Mac OS X and iOS. Embarcadero is planning to build support for the Linux and Android operating system.
- **Free Pascal:** it is a free compiler for running Pascal and Object Pascal programs. Free Pascal compiler is a 32- and 64-bit Turbo Pascal and Delphi compatible Pascal compiler for Linux, Windows, OS/2, FreeBSD, Mac OS X, DOS and several other platforms.
- **Turbo51:** it is a free Pascal compiler for the 8051 family of microcontrollers, with Turbo Pascal 7 syntax.
- **Oxygene:** it is an Object Pascal compiler for the .NET and Mono platforms.
- **GNU Pascal (GPC):** it is a Pascal compiler composed of a front end to GNU Compiler Collection.

We will be using Free Pascal in these tutorials. You can download Free Pascal for your operating system from the link: **Download Free Pascal**

Installing Free Pascal on Linux

The Linux distribution of Free Pascal comes in three forms:

- a tar.gz version, also available as separate files.
- a .rpm (Red Hat Package Manager) version.
- a .deb (Debian) version.

Installation code for the .rpm version:

```
rpm -i fpc-X.Y.Z-N.ARCH.rpm
```

Where X.Y.Z is the version number of the .rpm file, and ARCH is one of the supported architectures (i386, x86_64, etc.).

Installation code for the Debian version (like Ubuntu):

```
dpkg -i fpc-XXX.deb
```

Where XXX is the version number of the .deb file.

For details read: **Free Pascal Installation Guide**

Installing Free Pascal on Mac

If you use Mac OS X, the easiest way to use Free Pascal is to download the Xcode development environment from Apple's web site and follow the simple installation instructions. Once you have Xcode setup, you will be able to use the Free Pascal compiler.

Installing Free Pascal on Windows

For Windows, you will download the Windows installer, setup.exe. This is a usual installation program. You need to take the following steps for installation:

- Select a directory.
- Select parts of the package you want to install.
- Optionally choose to associate the .pp or .pas extensions with the Free Pascal IDE.

For details read: **Free Pascal Installation Guide**

Text Editor

This will be used to type your program. Examples of few editors include Windows Notepad, OS Edit command, Brief, Epsilon, EMACS, and vim or vi.

Name and version of text editor can vary on different operating systems. For example, Notepad will be used on Windows and vim or vi can be used on Windows as well as Linux or UNIX.

The files you create with your editor are called source files and contain program source code. The source files for Pascal programs are typically named with the extension **.pas**.

Before starting your programming, make sure you have one text editor in place and you have enough experience to write a computer program, save it in a file, compile it and finally execute it.

CHAPTER

3

Program Structure

This section describes basic Pascal program structure so that we can take it as a reference in upcoming chapters.

A Pascal program basically consists of the following parts:

- Program name
- Uses command
- Type declarations
- Constant declarations
- Variables declarations
- Functions declarations
- Procedures declarations
- Main program block
- Statements and Expressions within each block
- Comments

Every Pascal program generally have a heading statement, a declaration and an execution part strictly in that order. Following format shows the basic syntax for a Pascal program:

```
program {name of the program}
uses {comma delimited names of libraries you use}
const {global constant declaration block}
var {global variable declaration block}
function {function declarations, if any}
{ local variables }
begin
...
end;
procedure { procedure declarations, if any}
{ local variables }
begin
...
end;
begin { main program block starts}
...
end. { the end of main program block }
```

Pascal Hello World Example

Following is a simple Pascal code that would print the words "Hello, World!":

```
program HelloWorld;
uses crt;

(* Here the main program block starts *)
begin
  writeln('Hello, World!');
  readkey;
end.
```

Let us look various parts of the above program:

- The first line of the program **program HelloWorld;** indicates the name of the program.
- The second line of the program **uses crt;** is a preprocessor command, which tells the compiler to include the crt unit before going to actual compilation.
- The next lines enclosed within begin and end statements are the main program block. Every block in Pascal is enclosed within a **begin** statement and an **end** statement. However, the end statement indicating the end of the main program is followed by a full stop (.) instead of semicolon (;).
- The **begin** statement of the main program block is where the program execution begins.
- The lines within **(*...*)** will be ignored by the compiler and it has been put to add a **comment** in the program.
- The statement **writeln('Hello, World!');** uses the writeln function available in Pascal which causes the message "Hello, World!" to be displayed on the screen.
- The statement **readkey;** allows the display to pause until the user presses a key. It is part of the crt unit. A unit is like a library in Pascal.
- The last statement **end.** ends your program.

Compile and Execute Pascal Program:

- Open a text editor and add the above-mentioned code.
- Save the file as hello.pas
- Open a command prompt and go to the directory, where you saved the file.
- Type fpc hello.pas at command prompt and press enter to compile your code.
- If there are no errors in your code, the command prompt will take you to the next line and would generate **hello** executable file and **hello.o** object file.
- Now, type **hello** at command prompt to execute your program.
- You will be able to see "Hello World" printed on the screen and program waits till you press any key.

```
$ fpc hello.pas
Free Pascal Compiler version 2.6.0 [2011/12/23] for x86_64
Copyright (c) 1993-2011 by Florian Klaempfl and others
Target OS: Linux for x86-64
Compiling hello.pas
Linking hello
8 lines compiled, 0.1 sec

$ ./hello
Hello, World!
```

Make sure that free Pascal compiler **fpc** is in your path and that you are running it in the directory containing source file hello.pas.

CHAPTER

4

Basic Syntax

You have seen a basic structure of Pascal program, so it will be easy to understand other basic building blocks of the Pascal programming language.
This section shows the basic syntax of Pascal program.

Variables

A variable definition is put in a block beginning with a **var** keyword, followed by definitions of the variables as follows:

```
var
A_Variable, B_Variable ... : Variable_Type;
```

Pascal variables are declared outside the code-body of the function which means they are not declared within the **begin** and **end** pairs, but they are declared after the definition of the procedure/function and before the **begin** keyword. For global variables, they are defined after the program header.

Functions/Procedures

In Pascal, a **procedure** is set of instructions to be executed, with no return value and a **function** is a procedure with a return value. The definition of function/procedures will be as follows:

```
Function Func_Name(params...) : Return_Value;
Procedure Proc  Name(params...);
```

Comments

The multiline comments are enclosed within curly brackets and asterisks as {* ... *}. Pascal allows single-line comment enclosed within curly brackets { ... }.

```
{* This is a multi-line comments
   and it will span multiple lines. *}

{ This is a single line comment in pascal }
```

Case Sensitivity

Pascal is a case non-sensitive language, which means you can write your variables, functions and procedure in either case. Like variables A_Variable, a_variable and A_VARIABLE have same meaning in Pascal.

Pascal Statements

Pascal programs are made of statements. Each statement specifies a definite job of the program. These jobs could be declaration, assignment, reading data, writing data, taking logical decisions, transferring program flow control, etc.

For example:

```
readln (a, b, c);
s := (a + b + c)/2.0;
area := sqrt(s * (s - a)*(s-b)*(s-c));
writeln(area);
```

Reserved Words in Pascal

The statements in Pascal are designed with some specific Pascal words, which are called the reserved words. For example, the words, program, input, output, var, real, begin, readline, writeline and end are all reserved words. Following is a list of reserved words available in Pascal.

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

Character set and Identifiers in Pascal

The Pascal character set consists of:

- All upper case letters (A-Z)
- All lower case letters (a-z)
- All digits (0-9)
- Special symbols - + * / := , . ;. () [] = {} ` white space

The entities in a Pascal program like variables and constants, types, functions, procedures and records, etc., have a name or identifier. An identifier is a sequence of letters and digits, beginning with a letter. Special symbols and blanks must not be used in an identifier.

CHAPTER

5

Data Types

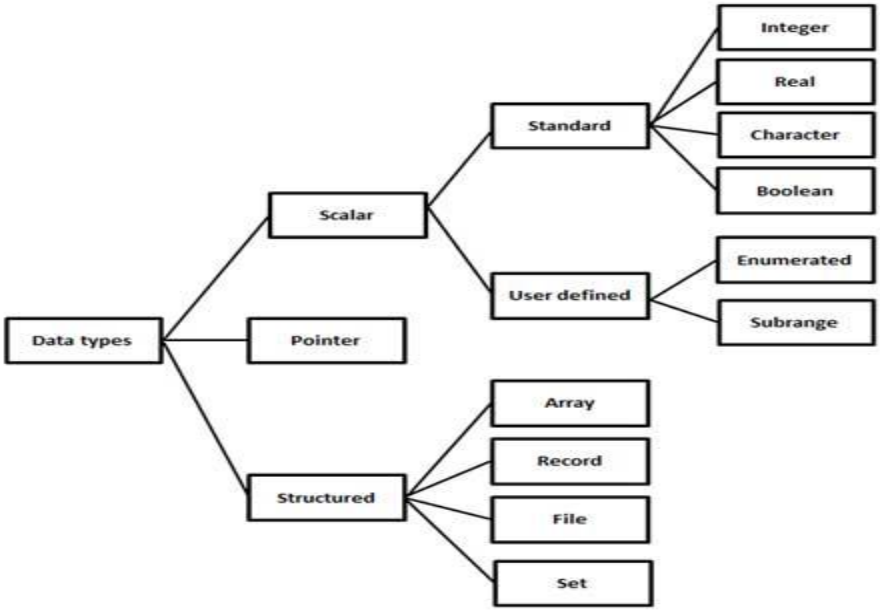
This section shows the data types used in a Pascal program.

Data types of an entity indicates the meaning, constraints, possible values, operations, functions and mode of storage associated with it.

Integer, real, Boolean and character types are referred as standard data types. Data types can be categorized as scalar, pointer and structured data types. Examples of scalar data types are integer, real, Boolean, character, subrange and enumerated. Structured data types are made of the scalar types; for example, arrays, records, files and sets. We will discuss the pointer data types later.

Pascal Data Types:

Pascal data types can be summarized as below in the following diagram:



Type Declarations:

The type declaration is used to declare the data type of an identifier. Syntax of type declaration is:

```
type-identifier-1, type-identfier-2 = type-specifier;
```

For example, the following declaration defines the variables days and age as integer type, yes and true as Boolean type, name and city as string type, fees and expenses as real type.

```
type
days, age = integer;
yes, true = boolean;
name, city = string;
```

Integer Types

Following table gives you details about standard integer types with its storage sizes and value ranges used in Object Pascal:

Type	**Minimum**	**Maximum**	**Format**
Integer	-2147483648	2147483647	signed 32-bit
Cardinal	0	4294967295	unsigned 32-bit
Shortint	-128	127	signed 8-bit
Smallint	-32768	32767	signed 16-bit
Longint	-2147483648	2147483647	signed 32-bit
Int64	-2^63	2^63 - 1	signed 64-bit
Byte	0	255	unsigned 8-bit
Word	0	65535	unsigned 16-bit
Longword	0	4294967295	unsigned 32-bit

Constants

Use of constants makes a program more readable and helps to keep special quantities at one place in the beginning of the program. Pascal allows *numerical, logical, string* and *character* constants. Constants can be declared in the declaration part of the program by specifying the **const** declaration.

Syntax of constant type declaration is follows:

```
const
Identifier = contant_value;
```

Following are some examples of constant declarations:

```
VELOCITY_LIGHT = 3.0E=10;
PIE = 3.141592;
NAME = 'Stuart Little';
CHOICE = yes;
OPERATOR = '+';
```

All constant declarations must be given before the variable declaration.

Enumerated types

Enumerated data types are user-defined data types. They allow values to be specified in a list. Only *assignment* operators and *relational* operators are permitted on enumerated data type. Enumerated data types can be declared as follows:

```
type
enum-identifier = (item1, item2, item3, ... )
```

Following are some examples of enumerated type declarations:

```
type
SUMMER = (April, May, June, July, September);
COLORS = (Red, Green, Blue, Yellow, Magenta, Cyan, Black, White);
TRANSPORT = (Bus, Train, Airplane, Ship);
```

The order in which the items are listed in the domain of an enumerated type defines the order of the items. For example, in the enumerated type SUMMER, April comes before May, May comes before June, and so on. The domain of enumerated type identifiers cannot consist of numeric or character constants.

Subrange Types

Subrange types allow a variable to assume values that lie within a certain range. For example, if the *age* of voters should lie between 18 to 100 years, a variable named age could be declared as:

```
var
age: 18 ... 100;
```

We will look at variable declaration in detail in the next section. You can also define a subrange type using the type declaration. Syntax for declaring a subrange type is as follows:

```
type
subrange-identifier = lower-limit ... upper-limit;
```

Following are some examples of subrange type declarations:

```
const
P = 18;
Q = 90;
type
Number = 1 ... 100;
Value = P ... Q;
```

Subrange types can be created from a subset of an already defined enumerated type, For example:

```
type
months = (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
Summer = Apr ... Aug;
Winter = Oct ... Dec;
```

CHAPTER

6

Variable Types

This section shows the variable types used in a Pascal program.

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in Pascal has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Pascal is not case-sensitive, so uppercase and lowercase letters mean same here. Based on the basic types explained in previous chapter, there will be following basic variable types:

Basic Variables in Pascal

Type	Description
Character	Typically a single octet (one byte). This is an integer type.
Integer	The most natural size of integer for the machine.
Real	A single-precision floating point value.
Boolean	Specifies true or false logical values. This is also an integer type.
Enumerated	Specifies a user-defined list.
Subrange	Represents variables, whose values lie within a range.
String	Stores an array of characters.

Pascal programming language also allows defining various other types of variables, which we will cover in subsequent chapters like Pointer, Array, Records, Sets, and Files, etc. For this chapter, let us study only basic variable types.

Variable Declaration in Pascal

All variables must be declared before we use them in Pascal program. All variable declarations are followed by the *var* keyword. A declaration specifies a list of variables, followed by a colon (:) and the type. Syntax of variable declaration is:

```
var
variable_list : type;
```

Here, type must be a valid Pascal data type including character, integer, real, boolean, or any user-defined data type, etc., and variable_list may consist of one or more identifier names separated by commas. Some valid variable declarations are shown here:

```
var
age, weekdays : integer;
taxrate, net_income: real;
choice, isready: boolean;
initials, grade: char;
name, surname : string;
```

In the previous tutorial, we have discussed that Pascal allows declaring a type. A type can be identified by a name or identifier. This type can be used to define variables of that type. For example:

```
type
days, age = integer;
yes, true = boolean;
fees, expenses = real;
```

Now, the types so defined can be used in variable declarations:

```
var
weekdays, holidays : days;
choice: yes;
student_name, emp_name : name;
capital: city;
cost: expenses;
```

Please note the difference between type declaration and var declaration. Type declaration indicates the category or class of the types such as integer, real, etc., whereas the variable specification indicates the type of values a variable may take. You can compare type declaration in Pascal with typedef in C. Most importantly, the variable name refers to the memory location where the value of the variable is going to be stored. This is not so with the type declaration.

Variable Initialization in Pascal

Variables are assigned a value with a colon and the equal sign, followed by a constant expression. The general form of assigning a value is:

```
variable_name := value;
```

By default, variables in Pascal are not initialized with zero. They may contain rubbish values. So it is a better practice to initialize variables in a program. Variables can be initialized (assigned an initial value) in their declaration. The initialization is followed by the **var** keyword and the syntax of initialization is as follows:

```
var
variable_name : type = value;
```

Some examples are:

```
age: integer = 15;
taxrate: real = 0.5;
grade: char = 'A';
name: string = 'John Smith';
```

Let us look at an example, which makes use of various types of variables discussed so far:

```
program Greetings;
const
message = ' Welcome to the world of Pascal ';
type
name = string;
var
firstname, surname: name;
begin
   writeln('Please enter your first name: ');
   readln(firstname);
   writeln('Please enter your surname: ');
   readln(surname);
   writeln;
   writeln(message, ' ', firstname, ' ', surname);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Please enter your first name:
John
Please enter your surname:
Smith
Welcome to the world of Pascal John Smith
```

Enumerated Variables

You have seen how to use simple variable types like integer, real and boolean. Now, let's see variables of enumerated type, which can be defined as:

```
var
var1, var2, ...  : enum-identifier;
```

When you have declared an enumerated type, you can declare variables of that type. For example:

```
type
months = (January, February, March, April, May, June, July, August,
September, October, November, December);
Var
m: months;
...
M := January;
```

The following example illustrates the concept:

```
program exEnumeration;
type
beverage = (coffee, tea, milk, water, coke, limejuice);
var
drink:beverage;
begin
   writeln('Which drink do you want?');
   writeln('You have ', sizeof(drink), ' choices');
end.
```

When the above code is compiled and executed, it produces the following result:

```
Which drink do you want?
You have 4 choices
```

Subrange Variables

Subrange variables are declared as:

```
var
subrange-name : lowerlim ... uperlim;
```

Examples of subrange variables are:

```
var
marks: 1 ... 100;
grade: 'A' ... 'E';
age: 1 ... 25;
```

The following program illustrates the concept:

```
program exSubrange;
var
marks: 1 .. 100;
grade: 'A' .. 'E';
begin
   writeln( 'Enter your marks(1 - 100): ');
   readln(marks);
   writeln( 'Enter your grade(A - E): ');
   readln(grade);
   writeln('Marks: ' , marks, ' Grade: ', grade);
end.
```

When the above code is compiled and executed, it produces the following result:

```
var
weekdays, holidays : days;
choice: yes;
student_name, emp_name : name;
capital: city;
cost: expenses;
```

CHAPTER

7

Constants

This section shows the constants used in a Pascal program.

A constant is an entity that remains unchanged during program execution. Pascal allows only constants of the following types to be declared:

- Ordinal types
- Set types
- Pointer types (but the only allowed value is Nil).
- Real types
- Char
- String

Declaring Constants

Syntax for declaring constants is as follows:

```
const
identifier = constant_value;
```

The following table provides examples of some valid constant declarations:

Constant Type	**Examples**
Ordinal(Integer)type constant	valid_age = 21;
Set type constant	Vowels = set of (A,E,I,O,U);
Pointer type constant	P = NIL;
Real type constant	e=2.7182818; velocity_light = 3.0E+10;
Character type constant	Operator = '+';

String type constant	president = 'Johnny Depp';

The following example illustrates the concept:

```
program const_circle (input,output);
const
PI = 3.141592654;
var
r, d, c : real;   {variable declaration: radius, dia, circumference}
begin
   writeln('Enter the radius of the circle');
   readln(r);
   d := 2 * r;
   c :=  PI * d;
   writeln('The circumference of the circle is ',c:7:2);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Enter the radius of the circle
23
The circumference of the circle is 144.51
```

Observe the formatting in the output statement of the program. The variable c is to be formatted with total number of digits 7 and 2 digits after the decimal sign. Pascal allows such output formatting with the numerical variables.

CHAPTER

8

Operators

This section shows the operators used in a Pascal program.

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. Pascal allows the following types of operators:

- Arithmetic operators
- Relational operators
- Boolean operators
- Bit operators
- Set operators
- String operators

Let us discuss the arithmetic, relational, Boolean and bit operators one by one.

Arithmetic Operators

Following table shows all the arithmetic operators supported by Pascal. Assume variable **A** holds 10 and variable **B** holds 20, then:

Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
div	Divides numerator by denominator	B div A will give 2
mod	Modulus Operator AND remainder after an integer division	B mod A will give 0

The following example illustrates the arithmetic operators:

```
program calculator;
var
a,b,c : integer;
d: real;
begin
   a:=21;
   b:=10;
   c := a + b;
   writeln(' Line 1 - Value of c is ', c );
   c := a - b;
   writeln('Line 2 - Value of c is ', c );
   c := a * b;
   writeln('Line 3 - Value of c is ', c );
   d := a / b;
   writeln('Line 4 - Value of d is ', d:3:2 );
   c := a mod b;
   writeln('Line 5 - Value of c is ' , c );
   c := a div b;
   writeln('Line 6 - Value of c is ', c );
end.
```

Please note that Pascal is very strongly typed programming language, so it would give an error if you try to store the results of a division in an integer type variable. When the above code is compiled and executed, it produces the following result:

```
Line 1 - Value of c is 31
Line 2 - Value of c is 11
Line 3 - Value of c is 210
Line 4 - Value of d is 2.10
Line 5 - Value of c is 1
Line 6 - Value of c is 2
```

Relational Operators

Following table shows all the relational operators supported by Pascal. Assume variable **A** holds 10 and variable **B** holds 20, then:

Operator	Description	Example
=	Checks if the values of two operands are equal or not, if yes, then condition becomes true.	(A = B) is not true.
<>	Checks if the values of two operands are equal or not, if values are not equal, then condition becomes true.	(A <> B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes, then condition becomes true.	(A > B) is not true.

<	Checks if the value of left operand is less than the value of right operand, if yes, then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes, then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes, then condition becomes true.	(A <= B) is true.

Try the following example to understand all the relational operators available in Pascal programming language:

```
program showRelations;
var
a, b: integer;
begin
   a := 21;
   b := 10;
   if a = b then
      writeln('Line 1 - a is equal to b' )
   else
      writeln('Line 1 - a is not equal to b' );
   if  a < b then
      writeln('Line 2 - a is less than b' )
   else
      writeln('Line 2 - a is not less than b' );
   if  a > b then
      writeln('Line 3 - a is greater than b' )
   else
      writeln('Line 3 - a is greater than b' );

   (* Lets change value of a and b *)
   a := 5;
   b := 20;
```

```
if  a <= b then
      writeln('Line 4 - a is either less than or equal to b' );
   if ( b >= a ) then
      writeln('Line 5 - b is either greater than  or equal to ' );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b
Line 4 - a is either less than or equal to b
Line 5 - b is either greater than or equal to b
```

Boolean Operators

Following table shows all the Boolean operators supported by Pascal language. All these operators work on Boolean operands and produce Boolean results. Assume variable **A** holds true and variable **B** holds false, then:

Operator	Description	Example
and	Called Boolean AND operator. If both the operands are true, then condition becomes true.	A and B) is false.
and then	It is similar to the AND operator, however, it guarantees the order in which the compiler evaluates the logical expression. Left to right and the right operands are evaluated only when necessary.	(A and then B) is false.
or	Called Boolean OR Operator. If any of the two operands is true, then condition becomes true.	(A or B) is true.
or else	It is similar to Boolean OR, however, it guarantees the order in which the compiler evaluates the logical expression. Left to right and the right operands are evaluated only when necessary.	(A or else B) is true.
<=	Called Boolean NOT Operator. Used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	not (A and B) is true.

The following example illustrates the concept:

```
program beLogical;
var
a, b: boolean;
begin
   a := true;
   b := false;

   if (a and b) then
      writeln('Line 1 - Condition is true' )
   else
      writeln('Line 1 - Condition is not true');
   if  (a or b) then
      writeln('Line 2 - Condition is true' );

   (* lets change the value of a and b *)
   a := false;
   b := true;
   if  (a and b) then
      writeln('Line 3 - Condition is true' )
   else
      writeln('Line 3 - Condition is not true' );
   if not (a and b) then
   writeln('Line 4 - Condition is true' );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Line 1 - Condition is not true
Line 2 - Condition is true
Line 3 - Condition is not true
Line 4 - Condition is true
```

Bit Operators

Bitwise operators work on bits and perform bit-by-bit operation. All these operators work on integer operands and produce integer results. The truth table for bitwise and (&), bitwise or (|), and bitwise not (~) are as follows:

p	**q**	**p & q**	**p ! q**	**~p**	**~q**
0	0	0	0	1	1
0	1	0	1	1	0
1	0	1	1	0	0
1	1	0	1	0	1

Assume if A = 60; and B = 13; now in binary format they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A^B = 0011 0001

~A = 1100 0011

The Bitwise operators supported by Pascal are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	**Description**	**Example**
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12, which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61, which is 0011 1101

!	Binary OR Operator copies a bit if it exists in either operand.	(A ! B) will give 61, which is 0011 1101
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -60, which is 1100 0011
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240, which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15, which is 0000 1111

Please note that different implementations of Pascal differ in bitwise operators. Free Pascal, the compiler we used here, however, supports the following bitwise operators:

Operators	**Operations**
not	Bitwise NOT
and	Bitwise AND
or	Bitwise OR
xor	Bitwise exclusive OR
shl	Bitwise shift left
shr	Bitwise shift right
<<	Bitwise shift left
>>	Bitwise shift right

The following example illustrates the concept:

```
program beBitwise;
var
a, b, c: integer;
begin
   a := 60;           (* 60 = 0011 1100 *)
   b := 13;           (* 13 = 0000 1101 *)
   c := 0;

   c := a and b;      (* 12 = 0000 1100 *)
   writeln('Line 1 - Value of c is  ', c );

   c := a or b;       (* 61 = 0011 1101 *)
   writeln('Line 2 - Value of c is  ', c );

   c := not a;        (* -61 = 1100 0011 *)
   writeln('Line 3 - Value of c is  ', c );

   c := a << 2;     (* 240 = 1111 0000 *)
   writeln('Line 4 - Value of c is  ', c );

   c := a >> 2;     (* 15 = 0000 1111 *)
   writeln('Line 5 - Value of c is  ', c );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Line 1 - Value of c is 12
Line 2 - Value of c is 61
Line 3 - Value of c is -61
Line 4 - Value of c is 240
Line 5 - Value of c is 15
```

Operators Precedence in Pascal

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator.

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Operators	**Precedence**
~, not,	Highest
*, /, div, mod, and, &	
|, !, +, -, or,	

=, <>, <, <=, >, >=, in	
or else, and then	Lowest

Try the following example to understand the operator precedence available in Pascal:

```
program opPrecedence;
var
a, b, c, d : integer;
e: real;
begin
   a := 20;
   b := 10;
   c := 15;
   d := 5;
   e := (a + b) * c / d;      (* ( 30 * 15 ) / 5 *)
   writeln('Value of (a + b) * c / d is : ',  e:3:1 );

   e := ((a + b) * c) / d;    (* (30 * 15 ) / 5  *)
   writeln('Value of ((a + b) * c) / d is  : ' ,  e:3:1 );

   e := (a + b) * (c / d);   (*  (30) * (15/5)  *)
   writeln('Value of (a + b) * (c / d) is  : ',  e:3:1);

   e := a + (b * c) / d;     (*  20 + (150/5)  *)
   writeln('Value of a + (b * c) / d is  : ' ,  e:3:1 );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Value of (a + b) * c / d is : 90.0
Value of ((a + b) * c) / d is  : 90.0
Value of (a + b) * (c / d) is  : 90.0
Value of a + (b * c) / d is  : 50.0
```

CHAPTER

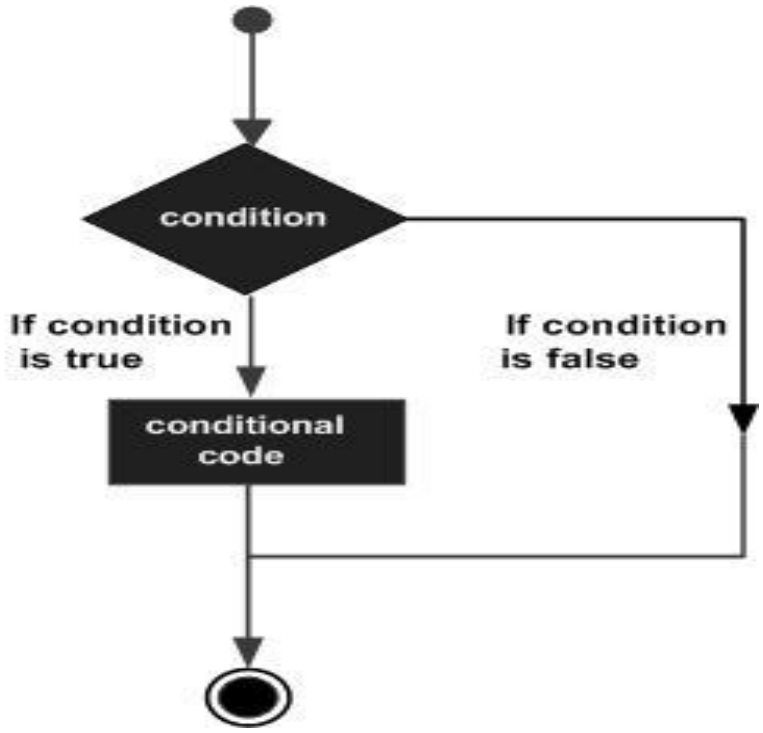
9

Decision Making

This section shows the decision making structure found in Pascal:

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



Pascal programming language provides the following types of decision making statements. Click the following links to check their details.

Statement	Description
if - then statement	An **if - then statement** consists of a boolean expression followed by one or more statements.
If-then-else statement	An **if - then statement** can be followed by an optional **else statement**, which executes when the boolean expression is false.
nested if statements	You can use one **if** or **else if** statement inside another **if** or **else if** statement(s).
case statement	A **case** statement allows a variable to be tested for equality against a list of values.
case - else statement	It is similar to the **if-then-else** statement. Here, an **else** term follows the **case statement**.
nested case statements	You can use one **case** statement inside another **case** statement(s).

if-then Statement

The **if-then** statement is the simplest form of control statement, frequently used in decision making and changing the control flow of the program execution.

Syntax

Syntax for **if-then** statement is:

```
if condition then S
```

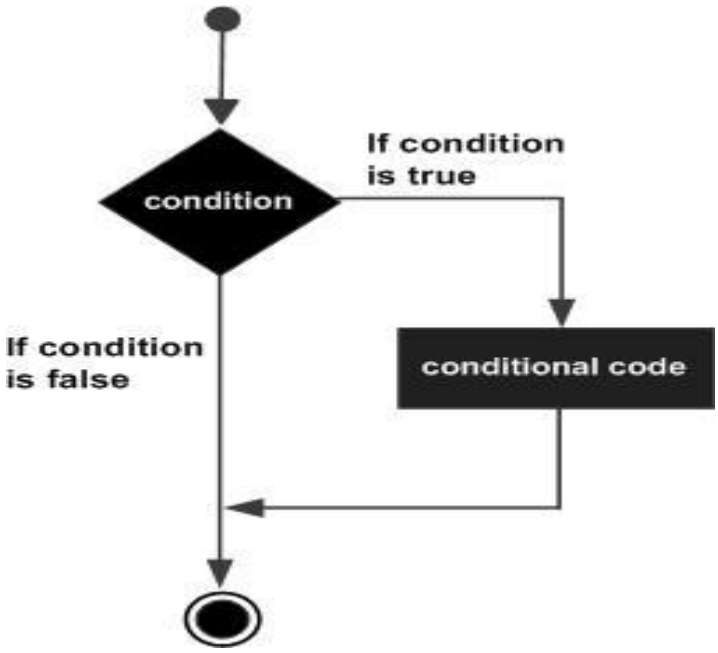
Where **condition** is a Boolean or relational condition and S is a simple or compound statement. Example of an if-then statement is:

```
if (a <= 20) then
   c:= c+1;
```

If the boolean expression **condition** evaluates to true, then the block of code inside the if statement will be executed. If boolean expression evaluates to false, then the first set of code after the end of the if statement (after the closing end;) will be executed.

Pascal assumes any non-zero and non-nil values as true, and if it is either zero or nil, then it is assumed as false value.

Flow Diagram:



Example:

Let us try a complete example that would illustrate the concept:

```
program ifChecking;
var
{ local variable declaration }
   a:integer;
begin
   a:= 10;
  (* check the boolean condition using if statement *)
   if( a < 20 ) then
      (* if condition is true then print the following *)
      writeln('a is less than 20 ' );
   writeln('value of a is : ', a);
end.
```

When the above code is compiled and executed, it produces the following result:

```
a is less than 20
value of a is : 10
```

if-then-else Statement

An **if-then** statement can be followed by an optional **else** statement, which executes when the Boolean expression is **false**.

Syntax:

Syntax for the if-then-else statement is:

```
if condition then S1 else S2;
```

Where, **S1** and **S2** are different statements. **Please note that the statement S1 is not followed by a semicolon**. In the if-then-else statements, when the test condition is true, the statement S1 is executed and S2 is skipped; when the test condition is false, then S1 is bypassed and statement S2 is executed.

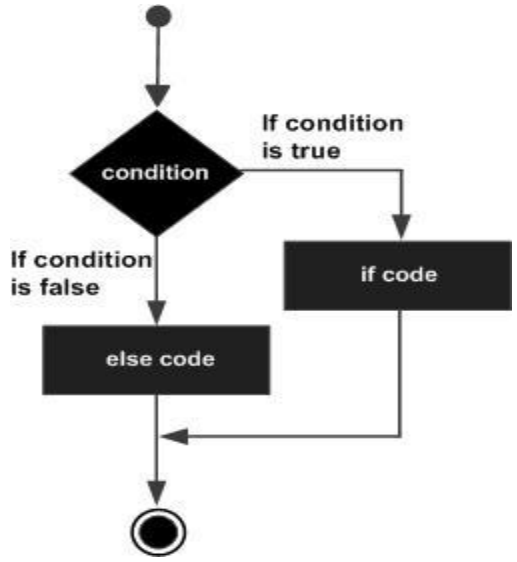
For example,

```
if color = red then
   writeln('You have chosen a red car')
else
   writeln('Please choose a color for your car');
```

If the boolean expression **condition** evaluates to true, then the if-then block of code will be executed, otherwise the else block of code will be executed.

Pascal assumes any non-zero and non-nil values as true, and if it is either zero or nil, then it is assumed as false value.

Flow Diagram:



Example:

Let us try a complete example that would illustrate the concept:

```
program ifelseChecking;
var
   { local variable definition }
   a : integer;
begin
   a := 100;
   (* check the boolean condition *)
   if( a < 20 ) then
      (* if condition is true then print the following *)
      writeln('a is less than 20' )
   else
      (* if condition is false then print the following *)
       writeln('a is not less than 20' );
      writeln('value of a is : ', a);
end.
```

When the above code is compiled and executed, it produces the following result:

```
a is not less than 20
value of a is : 100
```

The if-then-else if-then-else Statement

An if-then statement can be followed by an optional else if-then-else statement, which is very useful to test various conditions using single if-then-else if statement.

When using if-then, else if-then, else statements there are few points to keep in mind.

- An if-then statement can have zero or one else's and it must come after any else if's.
- An if-then statement can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.
- No semicolon (;) is given before the last else keyword, but all statements can be compound statements.

Syntax:

The syntax of an if-then-else if-then-else statement in Pascal programming language is:

```
if(boolean_expression 1)then
   S1 (* Executes when the boolean expression 1 is true *)
else if( boolean_expression 2) then
   S2 (* Executes when the boolean expression 2 is true *)
else if( boolean_expression 3) then
   S3 (* Executes when the boolean expression 3 is true *)
else
   S4; ( * executes when the none of the above condition is true *)
```

Example:

The following example illustrates the concept:

```
program ifelse_ifelseChecking;
var
   { local variable definition }
   a : integer;
begin
   a := 100;
  (* check the boolean condition *)
   if (a = 10)  then
      (* if condition is true then print the following *)
      writeln('Value of a is 10' )
   else if ( a = 20 ) then
      (* if else if condition is true *)
      writeln('Value of a is 20' )
   else if( a = 30 ) then
      (* if else if condition is true  *)
      writeln('Value of a is 30' )
   else
      (* if none of the conditions is true *)
      writeln('None of the values is matching' );
    writeln('Exact value of a is: ', a );
end.
```

The following example illustrates the concept:

```
None of the values is matching
Exact value of a is: 100
```

Nested if-then Statements

It is always legal in Pascal programming to nest **if-else** statements, which means you can use one **if** or **else if** statement inside another **if** or **else if** statement(s). Pascal allows nesting to any level, however, if depends on Pascal implementation on a particular system.

Syntax:

The syntax for a nested if statement is as follows:

```
if( boolean_expression 1) then
   if(boolean_expression 2)then S1
else
   S2;
```

You can nest else if-then-else in the similar way as you have nested if-then statement. Please note that, the nested **if-then-else** constructs gives rise to some ambiguity as to which else statement pairs with which if statement. *The rule is that the else keyword matches the first if keyword (searching backwards) not already matched by an else keyword.*

The above syntax is equivalent to:

```
if( boolean_expression 1) then
begin
   if(boolean_expression 2)then
      S1
   else
      S2;
end;
```

It is not equivalent to

```
if ( boolean_expression 1) then
begin
   if exp2 then
      S1
end;
   else
      S2;
```

Therefore, if the situation demands the later construct, then you must put **begin** and **end** keywords at the right place.

Example:

```
program nested_ifelseChecking;
var
   { local variable definition }
   a, b : integer;
begin
   a := 100;
   b:= 200;
    (* check the boolean condition *)
   if (a = 100) then
      (* if condition is true then check the following *)
      if ( b = 200 ) then
         (* if nested if condition is true  then print the following *)
          writeln('Value of a is 100 and value of b is 200' );

   writeln('Exact value of a is: ', a );
   writeln('Exact value of b is: ', b );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

Case Statement

You have observed that **if-then-else** statements enable us to implement multiple decisions in a program. This can also be achieved using the **case** statement in simpler way.

Syntax:

The syntax of the case statement is:

```
case (expression) of
   L1 : S1;
   L2: S2;
   ...
   ...
   Ln: Sn;
end;
```

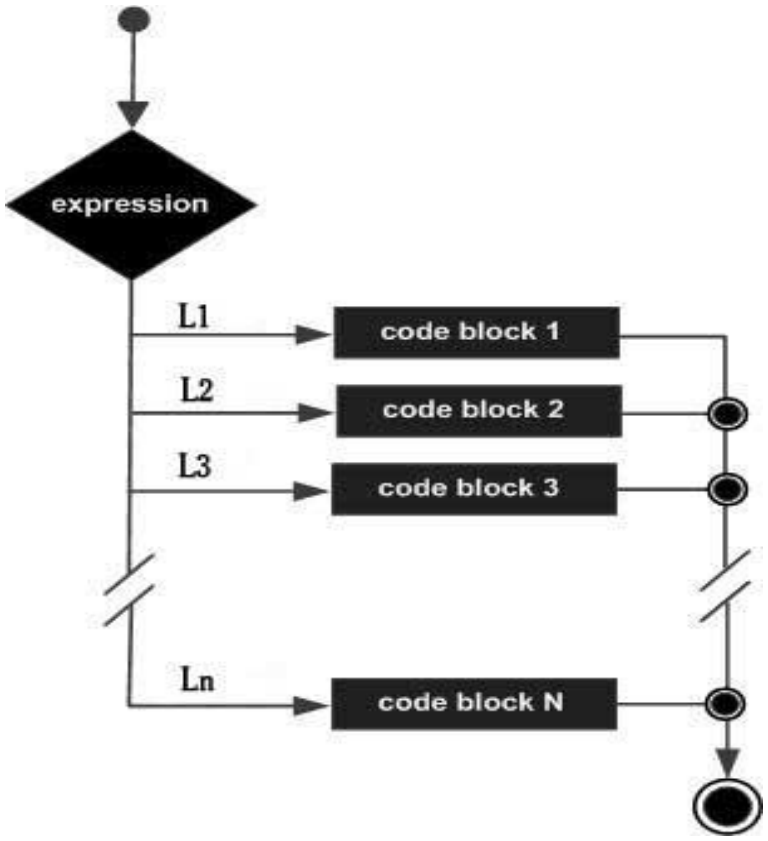
Where, **L1**, **L2**... are case labels or input values, which could be integers, characters, boolean or enumerated data items. **S1**, **S2**, ... are Pascal statements, each of these statements may have one or more than one case label associated with it. The expression is called the **case selector** or the **case index**. The case index may assume values that correspond to the case labels.

The case statement must always have an **end** statement associated with it.

The following rules apply to a case statement:

- The expression used in a case statement must have an integral or enumerated type or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a case. Each case is followed by the value to be compared to and a colon.
- The case label for a case must be the same data type as the expression in the case statement, and it must be a constant or a literal.
- The compiler will evaluate the case expression. If one of the case label's value matches the value of the expression, the statement that follows this label is executed. After that, the program continues after the final end.
- If none of the case label matches the expression value, the statement list after the else or otherwise keyword is executed. This can be an empty statement list. If no else part is present and no case constant matches the expression value, program flow continues after the final end.
- The case statements can be compound statements (i.e., a Begin ... End block).

Flow Diagram:



Example:

The following example illustrates the concept:

```
program checkCase;
var
   grade: char;
begin
   grade := 'A';

   case (grade) of
      'A' : writeln('Excellent!' );
      'B', 'C': writeln('Well done' );
      'D' : writeln('You passed' );
      'F' : writeln('Better try again' );
   end;
   writeln('Your grade is  ', grade );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Excellent!
Your grade is A
```

Case Else Statement

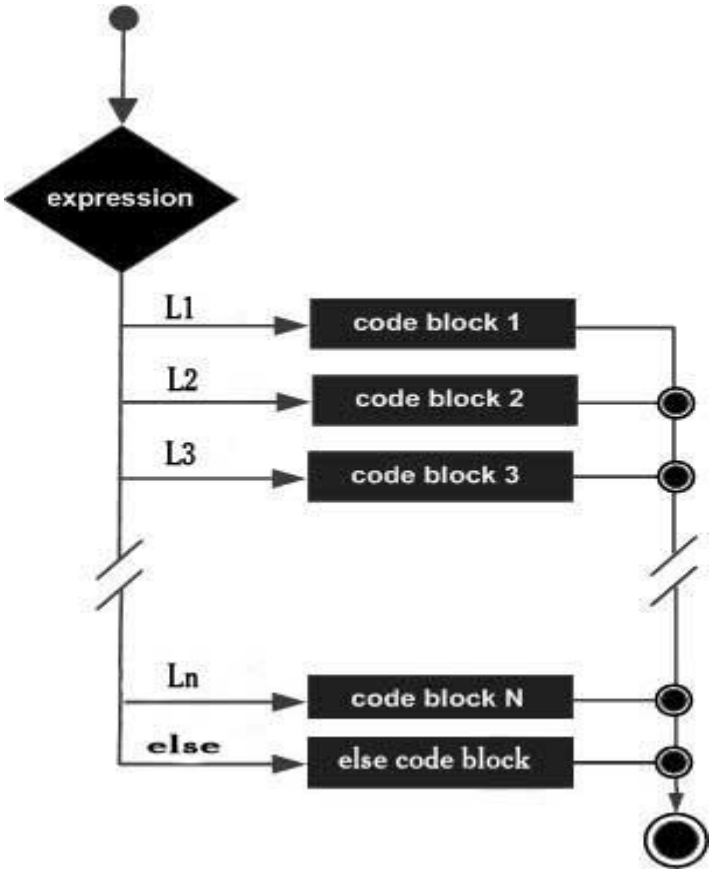
The **case-else** statement uses an **else** term after the **case** labels, just like an **if-then-else** construct.

Syntax:

The syntax for the case-else statement is:

```
case (expression) of
   L1 : S1;
   L2 : S2;
   ...
   ...
   Ln: Sn;
else
   Sm;
end;
```

Flow Diagram:



Example:

The following example illustrates the concept:

```
program checkCase;
var
   grade: char;
begin
   grade := 'F';
   case (grade) of
      'A' : writeln('Excellent!' );
      'B', 'C': writeln('Well done' );
      'D' : writeln('You passed' );
   else
      writeln('You really did not study right!' );
    end;
   writeln('Your grade is  ', grade );
end.
```

When the above code is compiled and executed, it produces the following result:

```
You really did not study right!
Your grade is F
```

Nested Case Statements

It is possible to have a **case statement** as part of the statement sequence of an outer **case statement**. Even if the **case constants** of the inner and outer case contain common values, no conflicts will arise.

Syntax:

The syntax for a nested case statement is as follows:

```
case (ch1) of
   'A': begin
          writeln('This A is part of outer case' );
          case(ch2) of
            'A': writeln('This A is part of inner case' );
            'B': (* case code *)
                                ...
                    end; {end of inner case}
      end; (* end of case 'A' of outer statement *)
   'B': (* case code *)
   'C': (* case code *)
   ...
end; {end of outer case}
```

Example:

The following program illustrates the concept.

```
program checknestedCase;
var
   a, b: integer;
begin
   a := 100;
   b := 200;
   case (a) of
      100: begin
             writeln('This  is part of outer statement' );
             case (b) of
               200: writeln('This  is part of inner statement' );
             end;
           end;
   end;
   writeln('Exact value of a is : ', a );
   writeln('Exact value of b is : ', b );
end.
```

When the above code is compiled and executed, it produces the following result:

```
This is part of outer switch
This is part of inner switch
Exact value of a is: 100
Exact value of b is: 200
```

CHAPTER

10

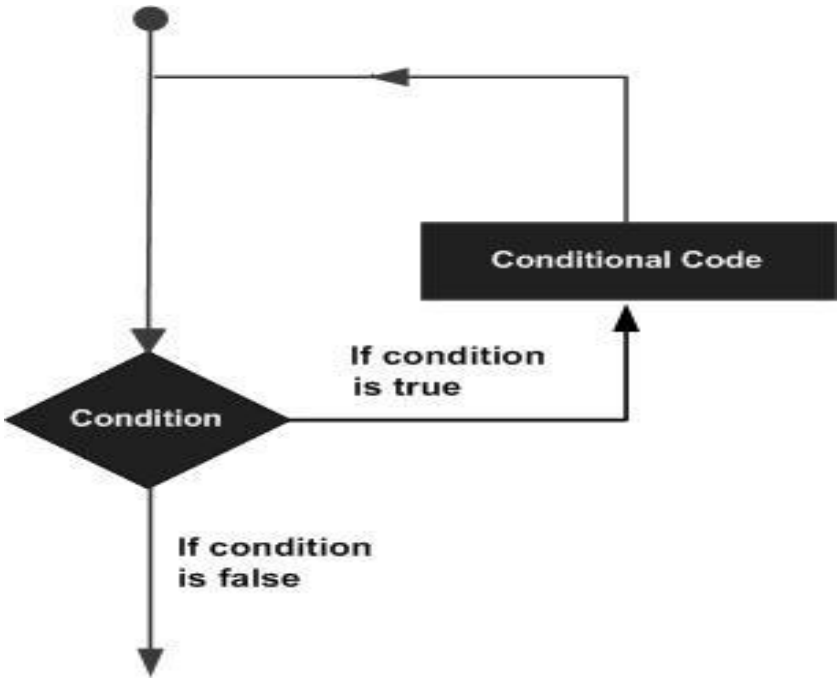
Loops

This section shows loop statements used in Pascal :

There may be a situation, when you need to execute a block of code several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general form of a loop statement in most of the programming languages:



Pascal programming language provides the following types of loop constructs to handle looping requirements. Click the following links to check their details.

Loop Type	Description
while-do loop	Repeats a statement or group of statements until a given condition is true. It tests the condition before executing the loop body.
for-do loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
repeat-until loop	Like a while statement, except that it tests the condition at the end of the loop body.
nested loops	You can use one or more loop inside any another while, for or repeat until loop.

while-do loop

A **while-do** loop statement in Pascal allows repetitive computations till some test condition is satisfied. In other words, it repeatedly executes a target statement as long as a given condition is true.

Syntax:

The syntax of a while-do loop is:

```
while (condition) do S;
```

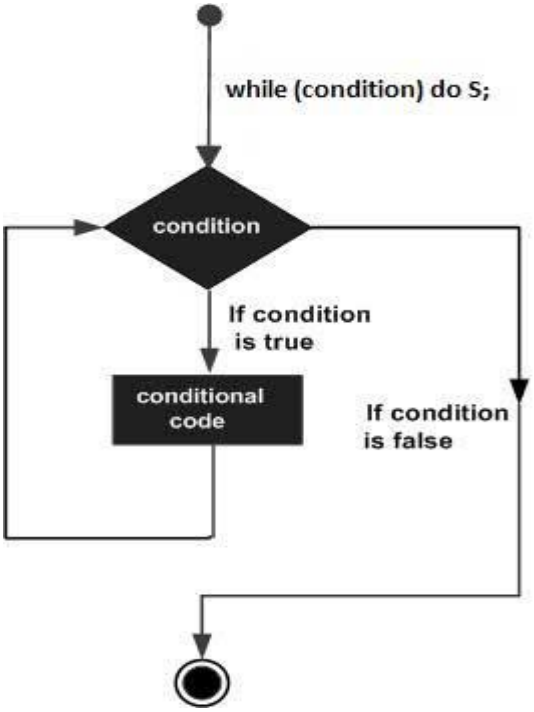
Where **condition** is a Boolean or relational expression, whose value would be true or false and S is a simple statement or group of statements within BEGIN ... END block.

For example,

```
while number>0 do
begin
   sum := sum + number;
   number := number - 2;
end;
```

When the condition becomes false, program control passes to the line immediately following the loop.

Flow Diagram:



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
program whileLoop;
var
   a: integer;
begin
   a := 10;
   while  a < 20  do
   begin
      writeln('value of a: ', a);
      a := a + 1;
   end;
end.
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

For-do LOOP

A **for-do** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

The syntax for the for-do loop in Pascal is as follows:

```
for < variable-name > := < initial_value > to [down to] < final_value > do
   S;
```

Where, the variable-name specifies a variable of ordinal type, called control variable or index variable; initial_value and final_value values are values that the control variable can take; and S is the body of the for-do loop that could be a simple statement or a group of statements.

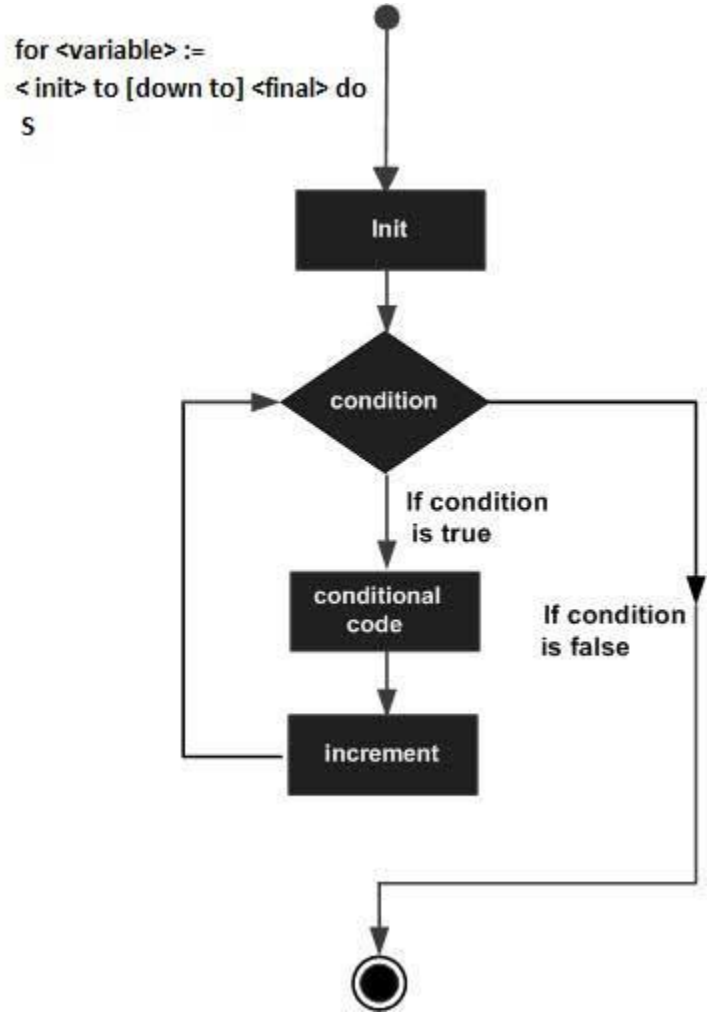
For example:

```
for i:= 1 to 10 do writeln(i);
```

Here is the flow of control in a for-do loop:

- The initial step is executed first, and only once. This step allows you to declare and initialize any loop control variables.
- Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for-do loop.
- After the body of the for-do loop executes, the value of the variable in increased or decreased.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for-do loop terminates.

Flow Diagram



Example:

```
program forLoop;
var
   a: integer;
begin
   for a := 10  to 20 do
   begin
      writeln('value of a: ', a);
   end;
end.
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
value of a: 20
```

Repeat-Until Loop

Unlike for and while loops, which test the loop condition at the top of the loop, the **repeat ... until** loop in Pascal checks its condition at the bottom of the loop.

A **repeat ... until** loop is similar to a while loop, except that a repeat ... until loop is guaranteed to execute at least one time.

Syntax:

```
repeat
   S1;
   S2;
   ...
   ...
   Sn;
until condition;
```

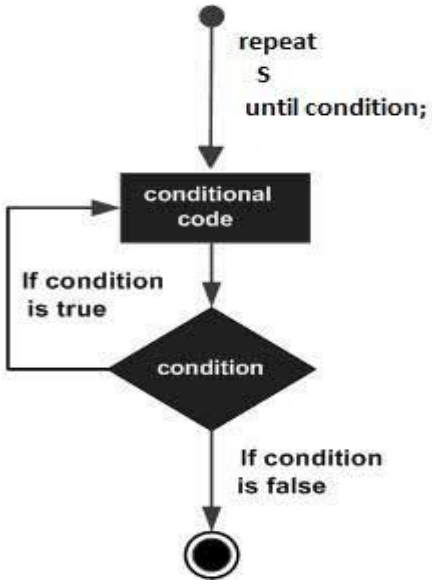
For example,

```
repeat
   sum := sum + number;
   number := number - 2;
until number = 0;
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to repeat and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

Flow Diagram:



Example:

```
program repeatUntilLoop;
var
   a: integer;
begin
   a := 10;
   (* repeat until loop execution *)
   repeat
      writeln('value of a: ', a);
      a := a + 1
   until a = 20;
end.
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Nested Loops

Pascal allows using one loop inside another loop. Following section shows few examples to illustrate the concept.

The syntax for a **nested for-do loop** statement in Pascal is as follows:

```
for variable1:=initial_value1 to [downto] final_value1 do
begin
   for variable2:=initial_value2 to [downto] final_value2 do
   begin
      statement(s);
   end;
end;
```

The syntax for a **nested while-do loop** statement in Pascal is as follows:

```
while(condition1)do
begin
   while(condition2) do
   begin
      statement(s);
   end;
   statement(s);
end;
```

The syntax for a **nested repeat ... until loop** Pascal is as follows:

```
repeat
   statement(s);
   repeat
      statement(s);
   until(condition2);
until(condition1);
```

A final note on loop nesting is that you can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

Example:

The following program uses a nested for loop to find the prime numbers from 2 to 50:

```
program nestedPrime;
var
   i, j:integer;
begin
   for i := 2 to 50 do
   begin
      for j := 2 to i do
         if (i mod j)=0  then
            break; {* if factor found, not prime *}
      if(j = i) then
         writeln(i , ' is prime' );
   end;
end.
```

When the above code is compiled and executed, it produces the following result:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
```

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Pascal supports the following control statements. Click the following links to check their details.

Control Statements	Description
break statement	Terminates the **loop** or **case** statement and transfers execution to the statement immediately following the loop or case statement.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
goto statement	Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

break statement

The **break** statement in Pascal has the following two usages:

1. When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the **case** statement (covered in the next chapter).

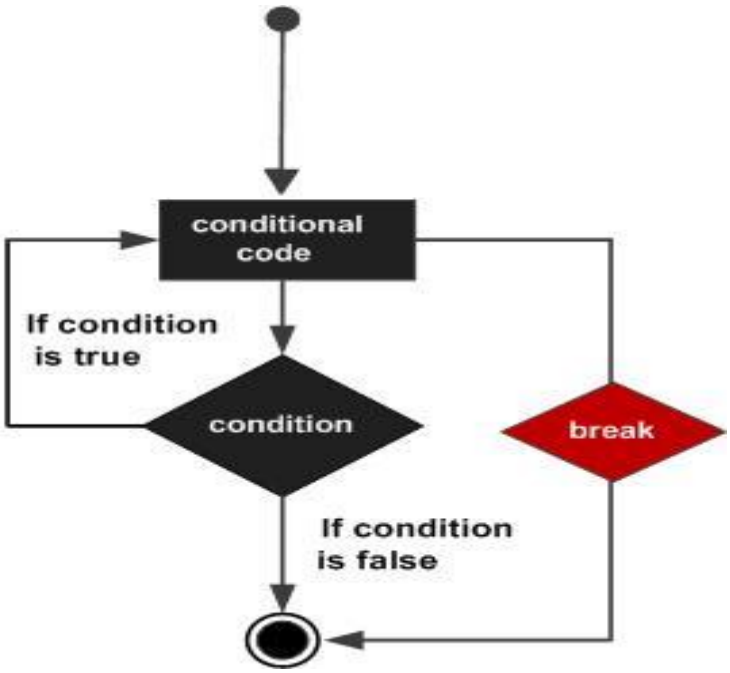
If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

The syntax for a **break** statement in Pascal is as follows:

```
break;
```

Flow Diagram:



Example:

```
program exBreak;
var
   a: integer;
begin
   a := 10;
  (* while loop execution *)
   while  a < 20 do
   begin
      writeln('value of a: ', a);
      a:=a +1;
      if( a > 15) then
         (* terminate the loop using break statement *)
          break;
    end;
end
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

continue statement

The **continue** statement in Pascal works somewhat like the **break** statement. Instead of forcing termination, however, **continue** forces the next iteration of the loop to take place, skipping any code in between.

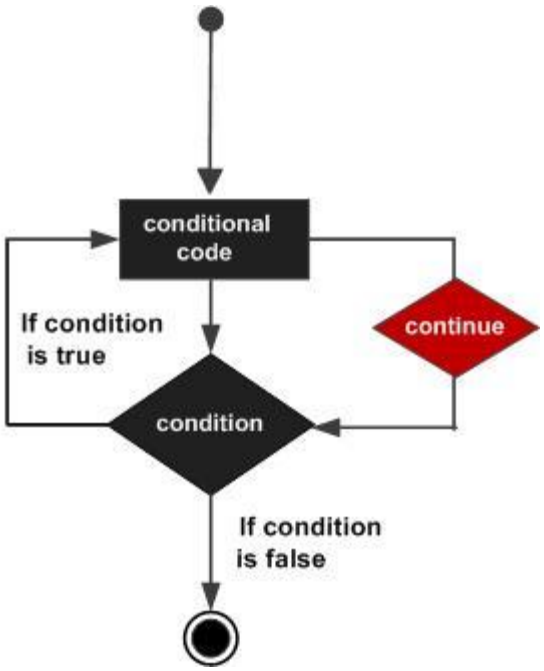
For the **for-do** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while-do** and **repeat...until** loops, **continue** statement causes the program control to pass to the conditional tests.

Syntax:

The syntax for a continue statement in Pascal is as follows:

```
continue;
```

Flow Diagram:



Example:

```
program exContinue;
var
   a: integer;
begin
   a := 10;
   (* repeat until loop execution *)
   repeat
      if( a = 15) then
      begin
         (* skip the iteration *)
         a := a + 1;
         continue;
      end;
      writeln('value of a: ', a);
      a := a+1;
   until ( a = 20 );
end.
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

goto statement

A **goto** statement in Pascal provides an unconditional jump from the goto to a labeled statement in the same function.

NOTE: Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten so that it doesn't need the goto.

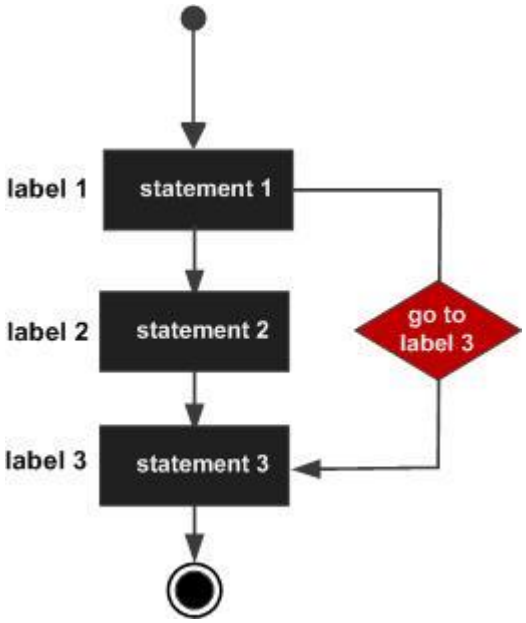
Syntax:

The syntax for a **goto** statement in Pascal is as follows:

```
goto label;
  ...
  ...
label: statement;
```

Here, label must be an unsigned integer label, whose value can be from 1 to 9999.

Flow Diagram:



Example:

The following program illustrates the concept.

```
program exGoto;
label 1;
var
   a : integer;
begin
   a := 10;
   (* repeat until loop execution *)
   1: repeat
      if( a = 15) then
      begin
         (* skip the iteration *)
         a := a + 1;
         goto 1;
      end;
      writeln('value of a: ', a);
      a:= a +1;
   until a = 20;
end.
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

CHAPTER

11

Functions

This section shows the general form of functions used in Pascal:

Subprograms

A subprogram is a program unit/module that performs a particular task. These subprograms are combined to form larger programs. This is basically called the 'Modular design.' A subprogram can be invoked by a subprogram/program, which is called the calling program.

Pascal provides two kinds of subprograms:

- **Functions**: these subprograms return a single value.
- **Procedures**: these subprograms do not return a value directly.

Functions

A **function** is a group of statements that together perform a task. Every Pascal program has at least one function, which is the program itself, and all the most trivial programs can define additional functions.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

Pascal standard library provides numerous built-in functions that your program can call. For example, function **AppendStr()** appends two strings, function **New()** dynamically allocates memory to variables and many more functions.

Defining a Function:

In Pascal, a **function** is defined using the function keyword. The general form of a function definition is as follows:

```
function name(argument(s): type1; argument(s): type2; ...): function_type;
local declarations;
begin
  ...
  < statements >
  ...
  name:= expression;
end;
```

A function definition in Pascal consists of a function **header**, local **declarations** and a function **body**. The function header consists of the keyword function and a **name** given to the function. Here are all the parts of a function:

- **Arguments:** The argument(s) establish the linkage between the calling program and the function identifiers and also called the formal parameters. A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of parameters of a function. Use of such formal parameters is optional. These parameters may have standard data type, user-defined data type or subrange data type.
  The formal parameters list appearing in the function statement could be simple or subscripted variables, arrays or structured variables, or subprograms.
- **Return-Type:** All functions must return a value, so all functions must be assigned a type. The **function-type** is the data type of the value the function returns. It may be standard, user-defined scalar or subrange type but it cannot be structured type.
- **Local declarations:** Local declarations refer to the declarations for labels, constants, variables, functions and procedures, which are application to the body of function only.
- **Function Body:** The function body contains a collection of statements that define what the function does. It should always be enclosed between the reserved words begin and end. It is the part of a function where all computations are done. There must be an assignment statement of the type - **name := expression;** in the function body that assigns a value to the function name. This value is returned as and when the function is executed. The last statement in the body must be an end statement.

Following is an example showing how to define a function in pascal:

```
(* function returning the max between two numbers *)
function max(num1, num2: integer): integer;
var
  (* local variable declaration *)
  result: integer;
begin
  if (num1 > num2) then
    result := num1
  else
    result := num2;
  max := result;
end;
```

Function Declarations:

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
function name(argument(s): type1; argument(s): type2; ...): function_type;
```

For the above-defined function max(), following is the function declaration:

```
function max(num1, num2: integer): integer;
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Calling a Function:

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, program control is transferred to the called function. A called function performs defined task, and when its return statement is executed or when it last end statement is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. Following is a simple example to show the usage:

```
program exFunction;
var
   a, b, ret : integer;

(*function definition *)
function max(num1, num2: integer): integer;
var
   (* local variable declaration *)
   result: integer;
begin
   if (num1 > num2) then
      result := num1
   else
      result := num2;
   max := result;
end;
begin
   a := 100;
   b := 200;
  (* calling a function to get max value *)
   ret := max(a, b);
   writeln( 'Max value is : ', ret );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Max value is : 200
```

CHAPTER

12

Procedure

This section explains procedure concepts used in Pascal:

Procedures are subprograms that, instead of returning a single value, allow to obtain a group of results.

Defining a Procedure:

In Pascal, a procedure is defined using the **procedure** keyword. The general form of a procedure definition is as follows:

```
procedure name(argument(s): type1, argument(s): type 2, ... );
   < local declarations >
begin
   < procedure body >
end;
```

A procedure **definition** in Pascal consists of a **header**, local **declarations** and a **body** of the procedure. The procedure header consists of the keyword **procedure** and a name given to the procedure. Here are all the parts of a procedure:

- **Arguments:** The argument(s) establish the linkage between the calling program and the procedure identifiers and also called the formal parameters. Rules for arguments in procedures are same as that for the functions.
- **Local declarations:** Local declarations refer to the declarations for labels, constants, variables, functions and procedures, which are applicable to the body of the procedure only.
- **Procedure Body:** The procedure body contains a collection of statements that define what the procedure does. It should always be enclosed between the reserved words begin and end. It is the part of a procedure where all computations are done.

Following is the source code for a procedure called *findMin()*. This procedure takes 4 parameters x, y, z and m and stores the minimum among the first three variables in the

variable named m. The variable m is passed by **reference** (we will discuss passing arguments by reference a little later):

```
procedure findMin(x, y, z: integer; var m: integer);
(* Finds the minimum of the 3 values *)
begin
   if x < y then
      m := x
   else
      m := y;
   if z <m then
      m := z;
end; { end of procedure findMin }
```

Procedure Declarations:

A procedure **declaration** tells the compiler about a procedure name and how to call the procedure. The actual body of the procedure can be defined separately.

A procedure declaration has the following syntax:

```
procedure name(argument(s): type1, argument(s): type 2, ... );
```

Please note that the **name of the procedure is not associated with any type**. For the above defined procedure *findMin()*, following is the declaration:

```
procedure findMin(x, y, z: integer; var m: integer);
```

Calling a Procedure:

While creating a procedure, you give a definition of what the procedure has to do. To use the procedure, you will have to call that procedure to perform the defined task. When a program calls a procedure, program control is transferred to the called procedure. A called procedure performs the defined task, and when its last end statement is reached, it returns the control back to the calling program.

To call a procedure, you simply need to pass the required parameters along with the procedure name as shown below:

```
program exProcedure;
var
   a, b, c,  min: integer;
procedure findMin(x, y, z: integer; var m: integer);
(* Finds the minimum of the 3 values *)
begin
   if x < y then
      m:= x
   else
      m:= y;
   if z < m then
      m:= z;
end; { end of procedure findMin }
begin
   writeln(' Enter three numbers: ');
   readln( a, b, c);
   findMin(a, b, c, min); (* Procedure call *)
   writeln(' Minimum: ', min);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Enter three numbers:
89 45 67
Minimum: 45
```

Recursive Subprograms

We have seen that a program or subprogram may call another subprogram. When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion.

To illustrate the concept, let us calculate the factorial of a number. Factorial of a number n is defined as:

```
n! = n*(n-1)!
   = n*(n-1)*(n-2)!
     ...
   = n*(n-1)*(n-2)*(n-3)... 1
```

The following program calculates the factorial of a given number by calling itself recursively.

```
program exRecursion;
var
   num, f: integer;
function fact(x: integer): integer; (* calculates factorial of x - x! *)
begin
   if x=0 then
      fact := 1
   else
      fact := x * fact(x-1); (* recursive call *)
end; { end of function fact}
begin
   writeln(' Enter a number: ');
   readln(num);
   f := fact(num);
   writeln(' Factorial ', num, ' is: ' , f);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Enter a number:
5
Factorial 5 is: 120
```

Following is another example, which generates the **Fibonacci Series** for a given number using a **recursive** function:

```
program recursiveFibonacci;
var
   i: integer;
function fibonacci(n: integer): integer;
begin
   if n=1 then
      fibonacci := 0
   else if n=2 then
      fibonacci := 1
   else
      fibonacci := fibonacci(n-1) + fibonacci(n-2);
end;
begin
   for i:= 1 to 10 do
   write(fibonacci (i), '  ');
end.
```

When the above code is compiled and executed, it produces the following result:

```
0 1 1 2 3 5 8 13 21 34
```

Arguments of a Subprogram:

If a subprogram (**function or procedure**) is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the subprogram.

The formal parameters behave like other local variables inside the subprogram and are created upon entry into the subprogram and destroyed upon exit.

While calling a subprogram, there are two ways that arguments can be passed to the subprogram:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the subprogram. In this case, changes made to the parameter inside the subprogram have no effect on the argument.
Call by reference	This method copies the address of an argument into the formal parameter. Inside the subprogram, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, Pascal uses **call by value** to pass arguments. In general, this means that code within a subprogram cannot alter the arguments used to call the subprogram. The example program we used in the chapter 'Pascal - Functions' called the function named max() using **call by value**.

Whereas, the example program provided here (*exProcedure*) calls the procedure findMin() using **call by reference**.

Call by Value

The **call by value** method of passing arguments to a subprogram copies the actual value of an argument into the formal parameter of the subprogram. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, Pascal uses **call by value** method to pass arguments. In general, this means that code within a subprogram cannot alter the arguments used to call the subprogram. Consider the procedure swap() definition as follows.

```
procedure swap(x, y: integer);
var
   temp: integer;
begin
   temp := x;
   x:= y;
   y := temp;
end;
```

Now, let us call the procedure swap() by passing actual values as in the following example:

```
program exCallbyValue;
var
   a, b : integer;
(*procedure definition *)
procedure swap(x, y: integer);
var
   temp: integer;
begin
   temp := x;
   x:= y;
   y := temp;
end;
begin
   a := 100;
   b := 200;
   writeln('Before swap, value of a : ', a );
   writeln('Before swap, value of b : ', b );
(* calling the procedure swap  by value   *)
   swap(a, b);
   writeln('After swap, value of a : ', a );
   writeln('After swap, value of b : ', b );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

The program shows that **there is no change in the values** though they had been changed inside the subprogram.

Call by Reference

The **call by reference** method of passing arguments to a subprogram copies the address of an argument into the formal parameter. Inside the subprogram, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.

In order to pass the arguments by reference, Pascal allows to define **variable parameters**. This is done by preceding the formal parameters by the keyword **var**. Let us take the example of the procedure *swap()* that swaps the values in two variables and reflect the change in the calling subprogram.

```
procedure swap(var x, y: integer);
var
   temp: integer;
begin
   temp := x;
   x:= y;
   y := temp;
end;
```

Next, let us call the procedure *swap()* by passing **values by reference** as in the following example:

```
program exCallbyRef;
var
   a, b : integer;
(*procedure definition *)
procedure swap(var x, y: integer);
var
   temp: integer;
begin
   temp := x;
   x:= y;
   y := temp;
end;

begin
   a := 100;
   b := 200;
   writeln('Before swap, value of a : ', a );
   writeln('Before swap, value of b : ', b );
   (* calling the procedure swap  by value   *)
   swap(a, b);
   writeln('After swap, value of a : ', a );
   writeln('After swap, value of b : ', b );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Before swap, value of a : 100
Before swap, value of b : 200
After swap, value of a : 200
After swap, value of b : 100
```

Which shows that now the procedure swap() **has changed the values in the calling program.**

CHAPTER

13

Variable Scope

This section explains the scope of variables in Pascal programming:

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable cannot be accessed. There are three places, where variables can be declared in Pascal programming language:

- Inside a subprogram or a block which is called local variables
- Outside of all subprograms which is called global variables
- In the definition of subprogram parameters which is called formal parameters

Let us explain what are local and global variables and formal parameters.

Local Variables

Variables that are declared inside a subprogram or block are called local variables. They can be used only by statements that are inside that subprogram or block of code. Local variables are not known to subprograms outside their own. Following is the example using local variables. Here, all the variables a, b and c are local to program named exLocal.

```
program exLocal;
var
   a, b, c: integer;
begin
  (* actual initialization *)
   a := 10;
   b := 20;
   c := a + b;
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
end.
```

When the above code is compiled and executed, it produces the following result:

```
value of a = 10 b = 20 c = 30
```

Now, let us extend the program little more, let us create a procedure named display, which will have its own set of variables a, b and c and display their values, right from the program exLocal.

```
program exLocal;
var
   a, b, c: integer;
procedure display;
var
   a, b, c: integer;
begin
   (* local variables *)
   a := 10;
   b := 20;
   c := a + b;
   writeln('Winthin the procedure display');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
end;
begin
   a:= 100;
   b:= 200;
   c:= a + b;
   writeln('Winthin the program exlocal');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
   display();
end.
```

When the above code is compiled and executed, it produces the following result:

```
Within the program exlocal
value of a = 100 b = 200 c = 300
Within the procedure display
value of a = 10 b = 20 c = 30
```

Global Variables

Global variables are defined outside of a function, usually on top of the program. The global variables will hold their value throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A **global** variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is an example using **global** and **local** variables:

```
program exGlobal;
var
   a, b, c: integer;
procedure display;
var
   x, y, z: integer;
begin
   (* local variables *)
   x := 10;
   y := 20;
   z := x + y;
  (*global variables *)
   a := 30;
   b:= 40;
   c:= a + b;
   writeln('Winthin the procedure display');
   writeln(' Displaying the global variables a, b, and c');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
   writeln('Displaying the local variables x, y, and z');
   writeln('value of x = ', x , ' y =  ',  y, ' and z = ', z);
end;
begin
   a:= 100;
   b:= 200;
   c:= 300;
   writeln('Winthin the program exlocal');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
   display();
end.
```

When the above code is compiled and executed, it produces the following result:

```
Within the program exlocal
value of a = 100 b = 200 c = 300
Within the procedure display
Displaying the global variables a, b, and c
value of a = 30 b = 40 c = 70
Displaying the local variables x, y, and z
value of x = 10 y = 20 z = 30
```

Please note that the procedure display has access to the variables a, b and c, which are global variables with respect to display as well as its own local variables. A program can have same name for local and global variables but value of local variable inside a function will take preference.

Let us change the previous example a little, now the local variables for the procedure display has same names as *a*, *b*, *c*:

```
program exGlobal;
var
   a, b, c: integer;
procedure display;
var
   a, b, c: integer;
begin
   (* local variables *)
   a := 10;
   b := 20;
   c := a + b;
   writeln('Winthin the procedure display');
   writeln(' Displaying the global variables a, b, and c');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
   writeln('Displaying the local variables a, b, and c');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
end;
begin
   a:= 100;
   b:= 200;
   c:= 300;
   writeln('Winthin the program exlocal');
   writeln('value of a = ', a , ' b =  ',  b, ' and c = ', c);
   display();
end.
```

When the above code is compiled and executed, it produces the following result:

```
Within the program exlocal
value of a = 100 b = 200 c = 300
Within the procedure display
Displaying the global variables a, b, and c
value of a = 10 b = 20 c = 30
Displaying the local variables a, b, and c
value of a = 10 b = 20 c = 30
```

CHAPTER

14

Strings

This section shows the concept of Strings:

The string in Pascal is actually a sequence of characters with an optional size specification. The characters could be numeric, letters, blank, special characters or a combination of all. Extended Pascal provides numerous types of string objects depending upon the system and implementation. We will discuss more common types of strings used in programs.

You can define a string in many ways:

- **Character arrays:** This is a character string (or string for short) is a sequence of zero or more byte-sized characters enclosed in single quotes.
- **String variables:** The variable of String type, as defined in Turbo Pascal.
- **Short strings:** The variable of String type with size specification.
- **Null terminated strings:** The variable of pchar type.
- **AnsiStrings:** AnsiStrings are strings that have no length limit.

Pascal provides only one string operator, string concatenation operator (+).

Examples

The following program prints first four kinds of strings. We will use AnsiStrings in the next example.

```
program exString;
var
   greetings: string;
   name: packed array [1..10] of char;
   organisation: string[10];
   message: pchar;
begin
```

When the above code is compiled and executed, it produces the following result:

```
Please Enter your Name
John Smith
Please Enter the name of your Organisation
Infotech
Hello John Smith from Infotech
```

Following example makes use of few more functions, let's see:

```
program exString;
uses sysutils;
var
   str1, str2, str3 : ansistring;
   str4: string;
   len: integer;
begin
   str1 := 'Hello ';
   str2 := 'There!';
  (* copy str1 into str3 *)
   str3 := str1;
   writeln('appendstr( str3, str1) :  ', str3 );
  (* concatenates str1 and str2 *)
   appendstr( str1, str2);
   writeln( 'appendstr( str1, str2) ' , str1 );
   str4 := str1 + str2;
   writeln('Now str4 is: ', str4);

  (* total lenghth of str4 after concatenation  *)
   len := byte(str4[0]);
   writeln('Length of the final string str4: ', len);
end.
```

When the above code is compiled and executed, it produces the following result:

```
appendstr( str3, str1) : Hello
appendstr( str1, str2) : Hello There!
Now str4 is: Hello There! There!
Length of the final string str4: 18
```

Pascal String Functions and Procedures

Pascal supports a wide range of functions and procedures that manipulate strings. These subprograms vary implement-wise. Here, we are listing various string manipulating subprograms provided by Free Pascal:

S.N.	Function Name & Description
1	**function AnsiCompareStr(const S1: ; const S2:):Integer;** Compares two strings
2	**function AnsiCompareText(const S1: ; const S2:):Integer;** Compares two strings, case insensitive
3	**function AnsiExtractQuotedStr(var Src: PChar; Quote: Char):;** Removes quotes from string
4	**function AnsiLastChar(const S:):PChar;** Gets last character of string
5	**function AnsiLowerCase(const s:):** Converts string to all-lowercase
6	**function AnsiQuotedStr(const S: ; Quote: Char):;** Quotes a string
7	**function AnsiStrComp(S1: PChar; S2: PChar):Integer;** Compares strings case-sensitive
8	**function AnsiStrIComp(S1: PChar; S2: PChar):Integer;** Compares strings case-insensitive
9	**function AnsiStrLComp(S1: PChar; S2: PChar; MaxLen: Cardinal):Integer;** Compares L characters of strings case sensitive
10	**function AnsiStrLComp(S1: PChar; S2: PChar; MaxLen: Cardinal):Integer;** Compares L characters of strings case insensitive
11	**function AnsiStrLastChar(Str: PChar):PChar;** Gets last character of string

12	**function AnsiStrLower(Str: PChar):PChar;** Converts string to all-lowercase
13	**function AnsiStrUpper(Str: PChar):PChar;** Converts string to all-uppercase
14	**function AnsiUpperCase(const s:):;** Converts string to all-uppercase
15	**procedure AppendStr(var Dest: ; const S:);** Appends 2 strings
16	**procedure AssignStr(var P: PString; const S:);** Assigns value of strings on heap
17	**function CompareStr(const S1: ; const S2:):Integer; overload;** Compares two strings case sensitive
18	**function CompareText(const S1: ; const S2:):Integer;** Compares two strings case insensitive
19	**procedure DisposeStr(S: PString); overload;** Removes string from heap
20	**procedure DisposeStr(S: PShortString); overload;** Removes string from heap
21	**function IsValidIdent(const Ident:):Boolean;** Is string a valid pascal identifier
22	**function LastDelimiter(const Delimiters: ; const S:):Integer;** Last occurrence of character in a string
23	**function LeftStr(const S: ; Count: Integer):;** Gets first N characters of a string
24	**function LoadStr(Ident: Integer):;** Loads string from resources
25	**function LowerCase(const s:):; overload;**

	Converts string to all-lowercase
26	**function LowerCase(const V: variant):; overload;** Converts string to all-lowercase
27	**function NewStr(const S:):PString; overload;** Allocates new string on heap
28	**function RightStr(const S: ; Count: Integer):;** Gets last N characters of a string
29	**function StrAlloc(Size: Cardinal):PChar;** Allocates memory for string
30	**function StrBufSize(Str: PChar):SizeUInt;** Reserves memory for a string
31	**procedure StrDispose(Str: PChar);** Removes string from heap
32	**function StrPas(Str: PChar):;** Converts PChar to pascal string
33	**function StrPCopy(Dest: PChar; Source:):PChar;** Copies pascal string
34	**function StrPLCopy(Dest: PChar; Source: ; MaxLen: SizeUInt):PChar;** Copies N bytes of pascal string
35	**function UpperCase(const s:):;** Converts string to all-uppercase

CHAPTER

15

Boolean

This section shows Boolean Data Type with its Declaration:

Pascal provides data type Boolean that enables the programmers to define, store and manipulate logical entities, such as constants, variables, functions and expressions, etc.

Boolean values are basically integer type. Boolean type variables have two pre-defined possible values True and False. The expressions resolving to a Boolean value can also be assigned to a Boolean type.

Free Pascal also supports the ByteBool, WordBool and LongBool types. These are of type Byte, Word or Longint, respectively.

The value False is equivalent to 0 (zero) and any nonzero value is considered True when converting to a Boolean value. A Boolean value of True is converted to -1 in case it is assigned to a variable of type LongBool.

It should be noted that logical operators and, or and not are defined for Boolean data types.

Declaration of Boolean Data Types

A variable of Boolean type is declared using the var keyword.

```
var
boolean-identifier: boolean;
```

for example,

```
var
choice: boolean;
```

Example:

```
program exBoolean;
var
exit: boolean;
choice: char;
   begin
   writeln('Do you want to continue? ');
   writeln('Enter Y/y for yes, and N/n for no');
   readln(choice);
if(choice = 'n') then
   exit := true
else
  exit := false;
if (exit) then
  writeln(' Good Bye!')
else
  writeln('Please Continue');
readln;
end.
```

When the above code is compiled and executed, it produces the following result:

```
Do you want to continue?
Enter Y/y for yes, and N/n for no
N
Good Bye!
Y
Please Continue
```

CHAPTER

16

Arrays

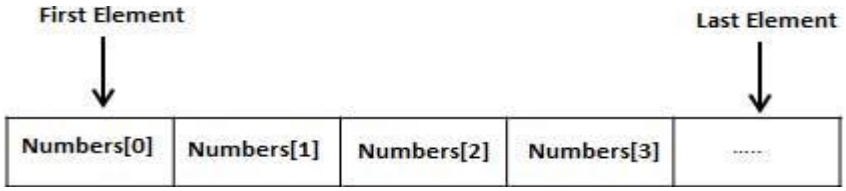
This section shows concept of Arrays:

Pascal programming language provides a data structure called the array, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number1, number2, ..., and number100, you declare one array variable such as numbers and use numbers[1], numbers[2], and ..., numbers[100] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

Please note that if you want a C style array starting from index 0, you just need to start the index from 0, instead of 1.



Declaring Arrays

To declare an array in Pascal, a programmer may either declare the type and then create variables of that array or directly declare the array variable.

The general form of type declaration of one-dimensional array is:

```
type
   array-identifier = array[index-type] of element-type;
```

Where,

- array-identifier indicates the name of the array type.
- index-type specifies the subscript of the array; it can be any scalar data type except real
- element-type specifies the types of values that are going to be stored.

For example,

```
type
   vector = array [ 1..25] of real;
var
   velocity: vector;
```

Now, velocity is a variable array of vector type, which is sufficient to hold up to 25 real numbers.

To start the array from 0 index, the declaration would be:

```
type
   vector = array [ 0..24] of real;
var
   velocity: vector;
```

Types of Array Subscript

In Pascal, an array subscript could be of any scalar type like, integer, Boolean, enumerated or subrange, except real. Array subscripts could have negative values too.

For example,

```
type
   temperature = array [-10 .. 50] of real;
var
   day_temp, night_temp: temperature;
```

Let us take up another example where the subscript is of character type:

```
type
   ch_array = array[char] of 1..26;
var
   alphabet: ch_array;
```

Subscript could be of enumerated type:

```
type
   color = ( red, black, blue, silver, beige);
   car_color = array of [color] of boolean;
var
   car_body: car_color;
```

Initializing Arrays

In Pascal, arrays are initialized through assignment, either by specifying a particular subscript or using a for-do loop.

For example:

```
type
   ch_array = array[char] of 1..26;
var
   alphabet: ch_array;
   c: char;
begin
   ...
   for c:= 'A' to 'Z' do
   alphabet[c] := ord[m];
   (* the ord() function returns the ordinal values *)
```

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
a: integer;
a: = alphabet['A'];
```

The above statement will take the first element from the array named alphabet and assign the value to the variable a.

Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays:

```
program exArrays;
var
   n: array [1..10] of integer;   (* n is an array of 10 integers *)
   i, j: integer;
begin
   (* initialize elements of array n to 0 *)
   for i := 1 to 10 do
      n[ i ] := i + 100;   (* set element at location i to i + 100 *)
    (* output each array element's value *)
   for j:= 1 to 10 do
      writeln('Element[', j, '] = ', n[j] );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
Element[10] = 110
```

Pascal Arrays in Detail

Arrays are important to Pascal and should need lots of more details. There are following few important concepts related to array which should be clear to a Pascal programmer:

Concept	Description
Multi-dimensional arrays	Pascal supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
Dynamic array	In this type of arrays, the initial length is zero. The actual length of the array must be set with the standard **SetLength** function.
Packed array	These arrays are bit-packed, i.e., each character or truth values are stored in consecutive bytes instead of using one storage unit, usually a word (4 bytes or more).
Passing arrays to subprograms	You can pass to a subprogram a pointer to an array by specifying the array's name without an index.

Multidimensional arrays

Pascal programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
type
   array-identifier = array [index-type1, index-type2, ...] of element-type;
var
   a1, a2, ... : array-identifier;
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

```
var
   threedim: array[1..5, 1..10, 1..4] of integer;
```

Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x, y you would write something as follows:

```
var
  arrayName: array[1..x, 1..y] of element-type;
```

Where element-type can be any valid Pascal data type and arrayName will be a valid Pascal identifier. A two-dimensional array can be visualized as a table, which will have x number of rows and y number of columns. A 2-dimensional array that contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form a[i][j], where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays:

Multidimensional arrays, like one-dimensional array, are initialized by through assignment, either by specifying a particular subscript or using a for-do loop.

For example,

```
var
  a: array [0..3, 0..3] of integer;
  i, j : integer;
begin
  for i:= 0 to 3 do
    for j:= 0 to 3 do
      a[i,j]:= i * j;
  ...
end;
```

Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
var
  val: integer;
  val := a[2, 3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two-dimensional array:

```
program ex2dimarray;
var
  a: array [0..3, 0..3] of integer;
  i,j : integer;
begin
  for i:=0 to 3 do
    for j:=0 to 3 do
      a[i,j]:= i * j;
  for i:=0 to 3 do
  begin
    for j:=0 to 3 do
      write(a[i,j]:2,' ');
    writeln;
  end;
end.
```

When the above code is compiled and executed, it produces the following result:

```
0 0 0 0
0 1 2 3
0 2 4 6
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Dynamic Arrays

In case of a dynamic array type, the initial length of the array is zero. The actual length of the array must be set with the standard SetLength function, which will allocate the necessary memory for storing the array elements.

Declaring Dynamic Arrays

For declaring dynamic arrays you do not mention the array range. For example:

```
type
  darray = array of integer;
var
  a: darray;
```

Before using the array, you must declare the size using the **setlength** function:

```
setlength(a,100);
```

Now, the array a has a valid array index range from 0 to 999: the array index is always zero-based.

The following example declares and uses a two-dimensional dynamic array:

```
program exDynarray;
var
   a: array of array of integer; (* a 2 dimensional array *)
   i, j : integer;
begin
   setlength(a,5,5);
   for i:=0 to 4 do
      for j:=0 to 4 do
         a[i,j]:= i * j;
   for i:=0 to 4 do
   begin
      for j:= 0 to 4 do
      write(a[i,j]:2,' ');
    writeln;
   end;
end.
```

When the above code is compiled and executed, it produces the following result:

```
0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
```

Packed Array

These arrays are bit-packed, i.e., each character or truth values are stored in consecutive bytes instead of using one storage unit, usually a word (4 bytes or more).

Normally, characters and Boolean values are stored in such a way that each character or truth value uses one storage unit like a word. This is called unpacked mode of data storage. Storage is fully utilized if characters are stored in consecutive bytes. This is called packed mode of data storage. Pascal allows the array data to be stored in packed mode.

Declaring Packed Arrays

Packed arrays are declared using the keywords **packed array** instead of array. For example:

```
type
   pArray: packed array[index-type1, index-type2, ...] of element-type;
var
   a: pArray;
```

The following example declares and uses a two-dimensional packed array:

```
program packedarray;
var
   a: packed array [0..3, 0..3] of integer;
   i, j : integer;
begin
   for i:=0 to 3 do
      for j:=0 to 3 do
         a[i,j]:= i * j;
   for i:=0 to 3 do
   begin
      for j:=0 to 3 do
         write(a[i,j]:2,' ');
      writeln;
   end;
end.
```

When the above code is compiled and executed, it produces the following result:

```
0 0 0 0
0 1 2 3
0 2 4 6
1 3 6 9
```

Passing Arrays as Subprogram Arguments

Pascal allows passing arrays as subprogram parameters. Following function will take an array as an argument and return average of the numbers passed through the array as follows:

```
program arrayToFunction;
const
   size = 5;
type
   a = array [1..size] of integer;
var
   balance:  a = (1000, 2, 3, 17, 50);
   average: real;
function avg( var arr: a) : real;
var
   i :1..size;
   sum: integer;
begin
   sum := 0;
   for i := 1 to size do
      sum := sum + arr[i];
   avg := sum / size;
end;
begin
   (*  Passing the array to the function  *)
   average := avg( balance ) ;
   (* output the returned value *)
   writeln( 'Average value is: ', average:7:2);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Average value is: 214.40
```

CHAPTER

17

Pointers

This section shows the concepts and usage of Pointers in Pascal:

Pointers in Pascal are easy and fun to learn. Some Pascal programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect Pascal programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using the name of the pointer variable, which denotes an address in memory.

What Are Pointers?

A pointer is a dynamic variable, whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type
   ptr-identifier = ^base-variable-type;
```

The pointer type is defined by prefixing the up-arrow of caret symbol (^) with the base type. The base-type defines the types of the data items. Once a pointer variable is defined to be of certain type, it can point data items of that type only. Once a pointer type has been defined, we can use the var declaration to declare pointer variables.

```
var
   p1, p2, ... : ptr-identifier;
```

Following are some valid pointer declarations:

```
type
  Rptr = ^real;
  Cptr = ^char;
  Bptr = ^ Boolean;
  Aptr = ^array[1..5] of real;
  date-ptr = ^ date;
    Date = record
      Day: 1..31;
      Month: 1..12;
      Year: 1900..3000;
    End;
var
  a, b : Rptr;
  d: date-ptr;
```

The pointer variables are dereferenced by using the same caret symbol (^). For example, the associated variable referred by a pointer rptr, is rptr^. It can be accessed as:

```
rptr^ := 234.56;
```

The following example will illustrate this concept:

```
program exPointers;
var
  number: integer;
  iptr: ^integer;
begin
  number := 100;
  writeln('Number is: ', number);
  iptr := @number;
  writeln('iptr points to a value: ', iptr^);
  iptr^ := 200;
  writeln('Number is: ', number);
  writeln('iptr points to a value: ', iptr^);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Number is: 100
iptr points to a value: 100
Number is: 200
iptr points to a value: 200
```

Printing a Memory Address in Pascal

In Pascal, we can assign the address of a variable to a pointer variable using the address operator (@). We use this pointer to manipulate and access the data item. However, if for

some reason, we need to work with the memory address itself, we need to store it in a word type variable.

Let us extend the above example to print the memory address stored in the pointer iptr:

```
program exPointers;
var
   number: integer;
   iptr: ^integer;
   y: ^word;
begin

   number := 100;
   writeln('Number is: ', number);
   iptr := @number;
   writeln('iptr points to a value: ', iptr^);
   iptr^ := 200;
   writeln('Number is: ', number);
   writeln('iptr points to a value: ', iptr^);
   y := addr(iptr);
   writeln(y^);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Number is: 100
iptr points to a value: 100
Number is: 200
iptr points to a value: 200
36864
```

NIL Pointers

It is always a good practice to assign a NIL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NIL points to nowhere. Consider the following program:

```
program exPointers;
var
   number: integer;
   iptr: ^integer;
   y: ^word;
begin
   iptr := nil;
   y := addr(iptr);
   writeln('the vaule of iptr is ', y^);
end.
```

When the above code is compiled and executed, it produces the following result:

```
The value of ptr is 0
```

To check for a nil pointer, you can use an if statement as follows:

```
if(ptr <> nill )then     (* succeeds if p is not null *)
if(ptr = nill)then    (* succeeds if p is null *)
```

Pascal Pointers in Detail:

Pointers have many but easy concepts and they are very important to Pascal programming. There are following few important pointer concepts, which should be clear to a Pascal programmer:

Concept	**Description**
Pascal - Pointer arithmetic	There are four arithmetic operators that can be used on pointers: increment, decrement, +, -
Pascal - Array of pointers	You can define arrays to hold a number of pointers.
Pascal - Pointer to pointer	Pascal allows you to have pointer on a pointer and so on.
Passing pointers to subprograms in Pascal	Passing an argument by reference or by address both enable the passed argument to be changed in the calling subprogram by the called subprogram.
Return pointer from subprograms in Pascal	Pascal allows a subprogram to return a pointer.

Pointer arithmetic

As explained in main chapter, Pascal pointer is an address, which is a numerical value stored in a word. Therefore, you can perform arithmetic operations on a pointer just as you can on a numeric value. There are four arithmetic operators that can be used on pointers: increment, decrement, +, and -.

To understand pointer arithmetic, let us consider that ptr is an integer pointer, which points to the address 1000. Assuming 32-bit integers, let us perform the increment operation on the pointer:

```
Inc(ptr);
```

Now, after the above operation, the **ptr** will point to the location 1004 because each time **ptr** is incremented, it will point to the next integer location, which is 4 bytes next to the current location. This operation will move the pointer to next memory location without impacting actual value at the memory location. If **ptr** points to a character, whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name, which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array:

```
program exPointers;
const MAX = 3;
var
   arr: array [1..MAX] of integer = (10, 100, 200);
   i: integer;
   iptr: ^integer;
   y: ^word;
begin
   (* let us have array address in pointer *)
   iptr := @arr[1];
   for  i := 1 to MAX do
   begin
      y:= addr(iptr);
      writeln('Address of arr[', i, '] = ' , y^ );
      writeln(' Value of arr[', i, '] = ' , iptr^ );
      (* move to the next location *)
      inc(iptr);
   end;
end.
```

When the above code is compiled and executed, it produces the following result:

```
Address of arr[1] = 32880
Value of arr[1] = 10
Address of arr[2] = 32882
Value of arr[2] = 100
Address of arr[3] = 32884
Value of arr[3] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

```
program exPointers;
const MAX = 3;
var
  arr: array [1..MAX] of integer = (10, 100, 200);
  i: integer;
  iptr: ^integer;
  y: ^word;
begin

  (* let us have array address in pointer *)
  iptr := @arr[MAX];
  for  i := MAX downto 1 do
  begin
    y:= addr(iptr);
    writeln('Address of arr[', i, '] = ' , y^ );
    writeln(' Value of arr[', i, '] = ' , iptr^ );

    (* move to the next location *)
    dec(iptr);
  end;
end.
```

When the above code is compiled and executed, it produces the following result:

```
Address of arr[3] = 32884
Value of arrr[3] = 200
Address of arr[2] = 32882
Value of arr[2] = 100
Address of arr[1] = 32880
Value of arr[1] = 10
```

Pointer Comparisons

Pointers may be compared by using relational operators, such as =, <, and >. If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.

The following program modifies the previous example one by incrementing the variable pointer so long as the address to which it points is either less than or equal to the address of the last element of the array, which is @arr[MAX]:

```
program exPointers;
const MAX = 3;
var
   arr: array [1..MAX] of integer = (10, 100, 200);
   i: integer;
   iptr: ^integer;
   y: ^word;
begin
   i:=1;
   (* let us have array address in pointer *)
   iptr := @arr[1];
   while (iptr <= @arr[MAX]) do
   begin
      y:= addr(iptr);
      writeln('Address of arr[', i, '] = ' , y^ );
      writeln(' Value of arr[', i, '] = ' , iptr^ );
      (* move to the next location *)
      inc(iptr);
      i := i+1;
  end;
end.
```

When the above code is compiled and executed, it produces the following result:

```
Address of arr[1] = 32880
Value of arr[1] = 10
Address of arr[2] = 32882
Value of arr[2] = 100
Address of arr[3] = 32884
Value of arr[3] = 200
```

Array of Pointers

Pascal allows defining an array of pointers. There may be a situation, when we want to maintain an array, which can store pointers to integers or characters or any other data type available. Following is the declaration of an array of pointers to an integer:

```
type
   iptr = ^integer;
var
   parray: array [1..MAX] of iptr;
```

This declares *parray* as an array of MAX integer pointers. Thus, each element in *parray*, now holds a pointer to an integer value. Following example makes use of three integers, which will be stored in an array of pointers as follows:

```
program exPointers;
const MAX = 3;
type
  iptr = ^integer;
var
  arr: array [1..MAX] of integer = (10, 100, 200);
  i: integer;
  parray: array[1..MAX] of iptr;
begin
  (* let us assign the addresses to parray *)
  for i:= 1 to MAX do
  parray[i] := @arr[i];
  (* let us print the values using the pointer array *)
  for i:=1 to MAX do
    writeln(' Value of arr[', i, '] = ' , parray[i]^ );
end.
```

You can also use an array of pointers to string variables to store a list of strings as follows:

```
program exPointers;
const
  MAX = 4;
type
  sptr = ^ string;
var
  i: integer;
  names: array [1..4] of string = ('Zara Ali', 'Hina Ali',
                                   'Nuha Ali','Sara Ali') ;
  parray: array[1..MAX] of sptr;
begin
  for i := 1 to MAX do
    parray[i] := @names[i];
  for i:= 1 to MAX do
    writeln('Value of names[', i, '] = ' , parray[i]^ );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Value of names[1] = Zara Ali
Value of names[2] = Hina Ali
Value of names[3] = Nuha Ali
Value of names[4] = Sara Ali
```

Pointer to Pointer

A pointer to a pointer is a form of multiple indirection or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

A variable that is a pointer to a pointer must be declared as such. For example,

```
type
   iptr = ^integer;
   pointerptr = ^ iptr;
```

Following example would illustrate the concept as well as display the addresses:

```
Enter the radius of the circle
program exPointertoPointers;
type
   iptr = ^integer;
   pointerptr = ^ iptr;
var
   num: integer;
   ptr: iptr;
   pptr: pointerptr;
   x, y : ^word;
begin
   num := 3000;
   (* take the address of var *)
   ptr := @num;
   (* take the address of ptr using address of operator @ *)
   pptr := @ptr;
  (* let us see the value and the adresses *)
   x:= addr(ptr);
   y := addr(pptr);
   writeln('Value of num = ', num );
   writeln('Value available at ptr^ = ', ptr^ );
   writeln('Value available at pptr^^ = ', pptr^^);
   writeln('Address at ptr = ', x^);
   writeln('Address at pptr = ', y^);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Value of num = 3000
Value available at ptr^ = 3000
Value available at pptr^^ = 3000
Address at ptr = 36864
Address at pptr = 36880
```

Passing Pointers to Subprograms

Pointer variables may be passed as parameters in function and procedure arguments. Pointer variables can be passed on both as value and variable parameters; however, when passed as variable parameters, the subprogram might inadvertently alter the value of the pointer which will lead to strange results.

The following program illustrates passing pointer to a function:

```
program exPointertoFunctions;
type
   iptr = ^integer;
var
   i: integer;
   ptr: iptr;
function getNumber(p: iptr): integer;
var
   num: integer;
begin
   num:=100;
   p:= @num;
   getNumber:=p^;
   end;
begin
   i := getNumber(ptr);
   writeln(' Here the pointer brings the value ', i);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Here the pointer brings the value: 100
```

Return Pointer from Subprograms

A function can return a pointer as its result. The following program illustrates returning pointer from a function:

```
program exPointersFromFunctions;
type
   ptr = ^integer;
var
   i: integer;
   iptr: ptr;
function getValue(var num: integer): ptr;
begin
   getValue:= @num;
end;
begin
   i := 100;
   iptr := getValue(i);
   writeln('Value deferenced: ', iptr^);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Value dereferenced: 100
```

CHAPTER

18

Records

This section shows the concepts under Records:

Pascal arrays allow you to define type of variables that can hold several data items of the same kind, but a record is another user-defined data type available in Pascal which allows you to combine data items of different kinds.

Records consist of different fields. Suppose you want to keep track of your books in a library, you might want to track the following attributes about each book:

- Title
- Author
- Subject
- Book ID

Defining a Record

To define a record type, you may use the type declaration statement. The record type is defined as:

```
type
record-name = record
   field-1: field-type1;
   field-2: field-type2;
   ...
   field-n: field-typen;
end;
```

Here is the way you would declare the Book record:

```
type
Books = record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: integer;
end;
```

The record variables are defined in the usual way as:

```
var
  r1, r2, ... : record-name;
```

Alternatively, you can directly define a record type variable as:

```
var
Books : record
  title: packed array [1..50] of char;
  author: packed array [1..50] of char;
  subject: packed array [1..100] of char;
  book_id: integer;
end;
```

Accessing Fields of a Record

To access any field of a record, we use the member access operator (.). The member access operator is coded as a period between the record variable name and the field that we wish to access. Following is the example to explain usage of structure:

```
program exRecords;
type
Books = record
  title: packed array [1..50] of char;
  author: packed array [1..50] of char;
  subject: packed array [1..100] of char;
  book_id: longint;
end;
var
  Book1, Book2: Books; (* Declare Book1 and Book2 of type Books *)
begin
  (* book 1 specification *)
  Book1.title  := 'C Programming';
  Book1.author := 'Nuha Ali ';
  Book1.subject := 'C Programming Tutorial';
  Book1.book_id := 6495407;
```

```
   (* book 2 specification *)
   Book2.title := 'Telecom Billing';
   Book2.author := 'Zara Ali';
   Book2.subject := 'Telecom Billing Tutorial';
   Book2.book_id := 6495700;

   (* print Book1 info *)
   writeln ('Book 1 title : ', Book1.title);
   writeln('Book 1 author : ', Book1.author);
   writeln( 'Book 1 subject : ', Book1.subject);
   writeln( 'Book 1 book_id : ', Book1.book_id);
   writeln;

   (* print Book2 info *)
   writeln ('Book 2 title : ', Book2.title);
   writeln('Book 2 author : ', Book2.author);
   writeln( 'Book 2 subject : ', Book2.subject);
   writeln( 'Book 2 book_id : ', Book2.book_id);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407

Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Records as Subprogram Arguments

You can pass a record as a subprogram argument in very similar way as you pass any other variable or pointer. You would access the record fields in the similar way as you have accessed in the above example:

```
program exRecords;
type
Books = record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: longint;
end;
var
   Book1, Book2: Books; (* Declare Book1 and Book2 of type Books *)
```

```
(* procedure declaration *)
procedure printBook( var book: Books );
begin
   (* print Book info *)
   writeln ('Book  title : ', book.title);
   writeln('Book  author : ', book.author);
   writeln( 'Book  subject : ', book.subject);
   writeln( 'Book book_id : ', book.book_id);
end;

begin
   (* book 1 specification *)
   Book1.title  := 'C Programming';
   Book1.author := 'Nuha Ali ';
   Book1.subject := 'C Programming Tutorial';
   Book1.book_id := 6495407;

  (* book 2 specification *)
   Book2.title := 'Telecom Billing';
   Book2.author := 'Zara Ali';
   Book2.subject := 'Telecom Billing Tutorial';
   Book2.book_id := 6495700;

  (* print Book1 info *)
   printbook(Book1);
   writeln;

   (* print Book2 info *)
   printbook(Book2);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407

Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Pointers to Records

You can define pointers to records in very similar way as you define pointer to any other variable as follows:

```
type
record-ptr = ^ record-name;
record-name = record
   field-1: field-type1;
   field-2: field-type2;
   ...
   field-n: field-typen;
end;
```

Now, you can store the address of a record type variable in the above-defined pointer variable. To declare a variable of the created pointer type, you use the var keyword:

```
var
   r1, r2, ... : record-ptr;
```

Before using these pointers, you must create storage for a record-name type variable, which will be manipulated by these pointers.

```
new(r1);
new(r2);
```

To access the members of a record using a pointer to that record, you must use the ^. operator as follows:

```
r1^.feild1 := value1;
r1^.feild2 := value2;
...
r1^fieldn := valuen;
```

Finally, don't forget to dispose the used storage, when it is no longer in use:

```
dispose(r1);
dispose(r2);
```

Let us re-write the first example using a pointer to the Books record. Hope this will be easy for you to understand the concept:

```
program exRecords;
type
BooksPtr = ^ Books;
Books = record
   title: packed array [1..50] of char;
   author: packed array [1..50] of char;
   subject: packed array [1..100] of char;
   book_id: longint;
end;
var
  (* Declare Book1 and Book2 of pointer type that refers to Book type *)
   Book1, Book2: BooksPtr;
```

```
begin
   new(Book1);
   new(book2);
  (* book 1 specification *)
   Book1^.title  := 'C Programming';
   Book1^.author := 'Nuha Ali ';
   Book1^.subject := 'C Programming Tutorial';
   Book1^.book_id := 6495407;

  (* book 2 specification *)
   Book2^.title := 'Telecom Billing';
   Book2^.author := 'Zara Ali';
   Book2^.subject := 'Telecom Billing Tutorial';
   Book2^.book_id := 6495700;

  (* print Book1 info *)
   writeln ('Book 1 title : ', Book1^.title);
   writeln('Book 1 author : ', Book1^.author);
   writeln( 'Book 1 subject : ', Book1^.subject);
   writeln( 'Book 1 book_id : ', Book1^.book_id);

   (* print Book2 info *)
   writeln ('Book 2 title : ', Book2^.title);
   writeln('Book 2 author : ', Book2^.author);
   writeln( 'Book 2 subject : ', Book2^.subject);
   writeln( 'Book 2 book_id : ', Book2^.book_id);
   dispose(Book1);
   dispose(Book2);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Book 1 title : C Programming
Book 1 author : Nuha Ali
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407

Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

The With Statement

We have discussed that the members of a record can be accessed using the member access operator (.). This way the name of the record variable has to be written every time. The **With** statement provides an alternative way to do that.

Look at the following code snippet taken from our first example:

```
(* book 1 specification *)
   Book1.title  := 'C Programming';
   Book1.author := 'Nuha Ali ';
   Book1.subject := 'C Programming Tutorial';
   Book1.book_id := 6495407;
```

The same assignment could be written using the **With** statement as:

```
(* book 1 specification *)
With Book1 do
begin
   title  := 'C Programming';
   author := 'Nuha Ali ';
   subject := 'C Programming Tutorial';
   book_id := 6495407;
end;
```

CHAPTER

19

Variants

This section shows a unique type of storage named variants supported by Pascal.

Pascal supports a unique type of storage named variants. You can assign any simple type of values in a variant variable. The type of a value stored in a variant is only determined at runtime. Almost any simple type can be assigned to variants: ordinal types, string types, int64 types. Structured types such as sets, records, arrays, files, objects and classes are not assignment-compatible with a variant. You can also assign a pointer to a variant. Free Pascal supports variants.

Declaring a Variant

You can declare variant type like any other types using the var keyword. The syntax for declaring a variant type is:

```
var
   v: variant;
```

Now, this variant variable v can be assigned to almost all simple types including the enumerated types and vice versa.

```
type
   color = (red, black, white);
var
   v : variant;
   i : integer;
   b : byte;
   w : word;
   en : color;
   as : ansistring;
   ws : widestring;
begin
   v := i;
   v := b;
   v := w;
   v := en;
   v := as;
```

Example:

The following example would illustrate the concept:

```
Program exVariant;
uses variants;
type
   color = (red, black, white);
var
   v : variant;
   i : integer;
   r: real;
   c : color;
   as : ansistring;

begin
   i := 100;
   v:= i;
   writeln('Variant as Integer: ', v);

   r:= 234.345;
   v:= r;
   writeln('Variant as real: ', v);
```

```
c := red;
   v := c;
   writeln('Variant as Enumerated data: ', v);

   as:= ' I am an AnsiString';
   v:= as;
   writeln('Variant as AnsiString: ', v);
 end.
```

When the above code is compiled and executed, it produces the following result:

```
Variant as Integer: 100
Variant as real: 234.345
Variant as Enumerated data: 0
Variant as AnsiString: I am an AnsiString
```

CHAPTER

20

Sets

This section explains sets, which is a collection of elements of same type.

A set is a collection of elements of same type. Pascal allows defining the set data type. The elements in a set are called its members. In mathematics, sets are represented by enclosing the members within *braces{}*. However, in Pascal, set elements are enclosed within square brackets [], which are referred as set constructor.

Defining Set Types and Variables

Pascal Set types are defined as

```
type
set-identifier = set of base type;
```

Variables of set type are defined as

```
var
s1, s2, ...: set-identifier;
```

or,

```
type
Days = (mon, tue, wed, thu, fri, sat, sun);
Letters = set of char;
DaySet = set of days;
Alphabets = set of 'A' .. 'Z';
studentAge = set of 13..20;
```

Set Operators

You can perform the following set operations on Pascal sets.

Operations	Descriptions
Union	This joins two sets and gives a new set with members from both sets.
Difference	Gets the difference of two sets and gives a new set with elements not common to either set.
Intersection	Gets the intersection of two sets and gives a new set with elements common to both sets.
Inclusion	A set P is included in set Q, if all items in P are also in Q but not vice versa.
Symmetric difference	Gets the symmetric difference of two sets and gives a set of elements, which are in either of the sets and not in their intersection.
In	It checks membership.

Following table shows all the set operators supported by Free Pascal. Assume that S1 and S2 are two character sets, such that:

```
S1 := ['a', 'b', 'c'];
S2 := ['c', 'd', 'e'];
```

Operator	Description	Example
+	Union of two sets	S1 + S2 will give a set ['a', 'b', 'c', 'd', 'e']
_	Difference of two sets	S1 - S2 will give a set ['a', 'b']
*	Intersection of two sets	S1 * S2 will give a set ['c']
><	Symmetric difference of two sets	S1 >< S2 will give a set ['a', 'b', 'd', 'e']
=	Gets the symmetric difference of two sets and gives a set of elements, which are in either of the sets and not in their intersection	S1 = S2 will give the boolean value False
<>	Checks equality of two sets	S1 <> S2 will give the boolean value True

<=	Contains (Checks if one set is a subset of the other)	S1 <= S2 will give the boolean value False
Include	Includes an element in the set; basically it is the Union of a set and an element of same base type	Include (S1, ['d']) will give a set ['a', 'b', 'c', 'd']
Exclude	Excludes an element from a set; basically it is the Difference of a set and an element of same base type	Exclude (S2, ['d']) will give a set ['c', 'e']
In	Checks set membership of an element in a set	['e'] in S2 gives the boolean value True

Example:

The following example illustrates the use of some of these operators:

```
program setColors;
type
color = (red, blue, yellow, green, white, black, orange);
colors = set of color;

procedure displayColors(c : colors);
const
names : array [color] of String[7]
  = ('red', 'blue', 'yellow', 'green', 'white', 'black', 'orange');
var
   cl : color;
   s : String;
begin
   s:= ' ';
   for cl:=red to orange do
      if cl in c then
      begin
         if (s<>' ') then s :=s +' , ';
         s:=s+names[cl];
      end;
   writeln('[',s,']');
end;

var
   c : colors;
begin
   c:= [red, blue, yellow, green, white, black, orange];
   displayColors(c);

   c:=[red, blue]+[yellow, green];
   displayColors(c);
```

```
   c:=[red, blue, yellow, green, white, black, orange] - [green, white];
   displayColors(c);

   c:= [red, blue, yellow, green, white, black, orange]*[green, white];
   displayColors(c);

   c:= [red, blue, yellow, green]><[yellow, green, white, black];
   displayColors(c);
end.
```

When the above code is compiled and executed, it produces the following result:

```
[ red , blue , yellow , green , white , black , orange]
[ red , blue , yellow , green]
[ red , blue , yellow , black , orange]
[ green , white]
[ red , blue , white , black]
```

CHAPTER

21

File Handling

This section shows File Handling in Pascal:

Pascal treats a file as a sequence of components, which must be of uniform type. A file's type is determined by the type of the components. File data type is defined as:

```
type
file-name = file of base-type;
```

Where, the base-type indicates the type of the components of the file. The base type could be anything like, integer, real, Boolean, enumerated, subrange, record, arrays and sets except another file type. Variables of a file type are created using the var declaration:

```
var
f1, f2,...: file-name;
```

Following are some examples of defining some file types and file variables:

```
type
   rfile = file of real;
   ifile = file of integer;
   bfile = file of boolean;
   datafile = file of record
   arrfile = file of array[1..4] of integer;
var
   marks: arrfile;
   studentdata: datafile;
   rainfalldata: rfile;
   tempdata: ifile;
   choices: bfile;
```

Creating and Writing to a File

Let us write a program that would create a data file for students' records. It would create a file named students.dat and write a student's data into it:

```
program DataFiles;
type
   StudentRecord = Record
      s_name: String;
      s_addr: String;
      s_batchcode: String;
   end;
var
   Student: StudentRecord;
   f: file of StudentRecord;
begin
   Assign(f,'students.dat');
   Rewrite(f);
   Student.s_name := 'John Smith';
   Student.s_addr := 'United States of America';
   Student.s_batchcode := 'Computer Science';
   Write(f,Student);
   Close(f);
end.
```

When compiled and run, the program would create a file named students.dat into the working directory. You can open the file using a text editor, like notepad, to look at John Smith's data.

Reading from a File

We have just created and written into a file named students.dat. Now, let us write a program that would read the student's data from the file:

```
program DataFiles;
type
   StudentRecord = Record
      s_name: String;
      s_addr: String;
      s_batchcode: String;
   end;
var
   Student: StudentRecord;
   f: file of StudentRecord;
begin
   assign(f, 'students.dat');
   reset(f);
   while not eof(f) do
   begin
      read(f,Student);
      writeln('Name: ',Student.s_name);
      writeln('Address: ',Student.s_addr);
      writeln('Batch Code: ', Student.s_batchcode);
   end;
   close(f);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Name: John Smith
Address: United States of America
Batch Code: Computer Science
```

Files as Subprogram Parameter

Pascal allows file variables to be used as parameters in standard and user-defined subprograms. The following example illustrates this concept. The program creates a file named rainfall.txt and stores some rainfall data. Next, it opens the file, reads the data and computes the average rainfall. Please note that, **if you use a file parameter with subprograms, it must be declared as a var parameter.**

```
program addFiledata;
const
   MAX = 4;
type
   raindata = file of real;
var
   rainfile: raindata;
   filename: string;
procedure writedata(var f: raindata);
var
   data: real;
   i: integer;
begin
   rewrite(f, sizeof(data));
   for i:=1 to MAX do
   begin
      writeln('Enter rainfall data: ');
      readln(data);
      write(f, data);
   end;
   close(f);
end;
procedure computeAverage(var x: raindata);
var
   d, sum: real;
   average: real;
begin
   reset(x);
   sum:= 0.0;
   while not eof(x) do
   begin
      read(x, d);
      sum := sum + d;
   end;
   average := sum/MAX;
   close(x);
   writeln('Average Rainfall: ', average:7:2);
end;
```

```
begin
   writeln('Enter the File Name: ');
   readln(filename);
   assign(rainfile, filename);
   writedata(rainfile);
   computeAverage(rainfile);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Enter the File Name:
rainfall.txt
Enter rainfall data:
34
Enter rainfall data:
45
Enter rainfall data:
56
Enter rainfall data:
78
Average Rainfall: 53.25
```

Text Files

A text file, in Pascal, consists of lines of characters where each line is terminated with an end-of-line marker. You can declare and define such files as:

```
type
file-name = text;
```

Difference between a normal file of characters and a text file is that a text file is divided into lines, each terminated by a special end-of-line marker, automatically inserted by the system. The following example creates and writes into a text file named contact.txt:

```
program exText;
var
   filename, data: string;
   myfile: text;
begin
   writeln('Enter the file name: ');
   readln(filename);
   assign(myfile, filename);
   rewrite(myfile);
   writeln(myfile, 'Note to Students: ');
   writeln(myfile, 'For details information on Pascal Programming');
   writeln(myfile, 'Contact: Tutorials Point');
   writeln('Completed writing');
   close(myfile);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Enter the file name:
contact.txt
Completed writing
```

Appending to a File

Appending to a file means writing to an existing file that already has some data without overwriting the file. The following program illustrates this:

```
program exAppendfile;
var
   myfile: text;
   info: string;
begin
   assign(myfile, 'contact.txt');
   append(myfile);
   writeln('Contact Details');
   writeln('webmaster@tutorialspoint.com');
   close(myfile);
   (* let us read from this file *)
   assign(myfile, 'contact.txt');
   reset(myfile);
   while not eof(myfile) do
   begin
      readln(myfile, info);
      writeln(info);
   end;
   close(myfile);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Contact Details
webmaster@tutorialspoint.com
Note to Students:
For details information on Pascal Programming
Contact: Tutorials Point
```

File Handling Functions

Free Pascal provides the following functions/procedures for file handling:

S.N.	Function Name & Description
1	**procedure Append(var t: Text);** Opens a file in append mode
2	**procedure Assign(out f: file; const Name:);** Assigns a name to a file
3	**procedure Assign(out f: file; p: PChar);** Assigns a name to a file
4	**procedure Assign(out f: file; c: Char);** Assign a name to a file
5	**procedure Assign(out f: TypedFile; const Name:);** Assigns a name to a file
6	**procedure Assign(out f: TypedFile; p: PChar);** Assigns a name to a file
7	**procedure Assign(out f: TypedFile; c: Char);** Assigns a name to a file
8	**procedure Assign(out t: Text; const s:);** Assigns a name to a file
9	**procedure Assign(out t: Text; p: PChar);** Assigns a name to a file
10	**procedure Assign(out t: Text; c: Char);** Assigns a name to a file
11	**procedure BlockRead(var f: file; var Buf; count: Int64; var Result: Int64);** Reads data from a file into memory
12	**procedure BlockRead(var f: file; var Buf; count: LongInt; var Result: LongInt);** Reads data from a file into memory
13	**procedure BlockRead(var f: file; var Buf; count: Cardinal; var Result: Cardinal);**

	Reads data from a file into memory
14	**procedure BlockRead(var f: file; var Buf; count: Word; var Result: Word);** Reads data from a file into memory
15	**procedure BlockRead(var f: file; var Buf; count: Word; var Result: Integer);** Reads data from a file into memory
16	**procedure BlockRead(var f: file; var Buf; count: Int64);** Reads data from a file into memory
17	**procedure BlockWrite(var f: file; const Buf; Count: Int64; var Result: Int64);** Writes data from memory to a file
18	**procedure BlockWrite(var f: file; const Buf; Count: LongInt; var Result: LongInt);** Writes data from memory to a file
19	**procedure BlockWrite(var f: file; const Buf; Count: Cardinal; var Result: Cardinal);** Writes data from memory to a file
20	**procedure BlockWrite(var f: file; const Buf; Count: Word; var Result: Word);** Writes data from memory to a file
21	**procedure BlockWrite(var f: file; const Buf; Count: Word; var Result: Integer);** Writes data from memory to a file
22	**procedure BlockWrite(var f: file; const Buf; Count: LongInt);** Writes data from memory to a file
23	**procedure Close(var f: file);** Closes a file
24	**procedure Close(var t: Text);** Closes a file
25	**function EOF(var f: file):Boolean;** Checks for end of file

26	**function EOF(var t: Text):Boolean;** Checks for end of file
27	**function EOF: Boolean;** Checks for end of file
28	**function EOLn(var t: Text):Boolean;** Checks for end of line
29	**function EOLn: Boolean;** Checks for end of line
30	**procedure Erase(var f: file);** Deletes file from disk
31	**procedure Erase(var t: Text);** Deletes file from disk
32	**function FilePos(var f: file):Int64;** Position in file
33	**function FileSize(var f: file):Int64;** Size of file
34	**procedure Flush(var t: Text);** Writes file buffers to disk
35	**function IOResult: Word;** Returns result of last file IO operation
36	**procedure Read(var F: Text; Args: Arguments);** Reads from file into variable
37	**procedure Read(Args: Arguments);** Reads from file into variable
38	**procedure ReadLn(var F: Text; Args: Arguments);** Reads from file into variable and goto next line
39	**procedure ReadLn(Args: Arguments);**

	Reads from file into variable and goto next line
40	**procedure Rename(var f: file; const s:);** Renames file on disk
41	**procedure Rename(var f: file; p: PChar);** Renames file on disk
42	**procedure Rename(var f: file; c: Char);** Renames file on disk
43	**procedure Rename(var t: Text; const s:);** Renames file on disk
44	**procedure Rename(var t: Text; p: PChar);** Renames file on disk
45	**procedure Rename(var t: Text; c: Char);** Renames file on disk
46	**procedure Reset(var f: file; l: LongInt);** Opens file for reading
47	**procedure Reset(var f: file);** Opens file for reading
48	**procedure Reset(var f: TypedFile);** Opens file for reading
49	**procedure Reset(var t: Text);** Opens file for reading
50	**procedure Rewrite(var f: file; l: LongInt);** Opens file for writing
51	**procedure Rewrite(var f: file);** Opens file for writing
52	**procedure Rewrite(var f: TypedFile);** Opens file for writing

53	**procedure Rewrite(var t: Text);** Opens file for writing
54	**procedure Seek(var f: file; Pos: Int64);** Sets file position
55	**function SeekEOF(var t: Text):Boolean;** Sets file position to end of file
56	**function SeekEOF: Boolean;** Sets file position to end of file
57	**function SeekEOLn(var t: Text):Boolean;** Sets file position to end of line
58	**function SeekEOLn: Boolean;** Sets file position to end of line
59	**procedure SetTextBuf(var f: Text; var Buf);** Sets size of file buffer
60	**procedure SetTextBuf(var f: Text; var Buf; Size: SizeInt);** Sets size of file buffer
61	**procedure Truncate(var F: file);** Truncates the file at position
62	**procedure Write(Args: Arguments);** Writes variable to file
63	**procedure Write(var F: Text; Args: Arguments);** Writes variable to file
64	**procedure Writeln(Args: Arguments);** Writes variable to file and appends newline
65	**procedure WriteLn(var F: Text; Args: Arguments);** Writes variable to file and appends newline

CHAPTER

22

Memory Management

This section explains dynamic memory management in Pascal.

Pascal programming language provides several functions for memory allocation and management.

Allocating Memory Dynamically

While doing programming, if you are aware about the size of an array, then it is easy and you can define it as an array. For example, to store a name of any person, it can go max 100 characters so you can define something as follows:

```
var
name: array[1..100] of char;
```

But now, let us consider a situation, where you have no idea about the length of the text you need to store, for example, you want to store a detailed description about a topic. Here, we need to define a pointer to string without defining how much memory is required.
Pascal provides a procedure **new** to create pointer variables.

```
program exMemory;
var
name: array[1..100] of char;
description: ^string;
begin
   name:= 'Zara Ali';
   new(description);
      if not assigned(description) then
         writeln(' Error - unable to allocate required memory')
      else
         description^ := 'Zara ali a DPS student in class 10th';
   writeln('Name = ', name );
   writeln('Description: ', description^ );
end.
```

When the above code is compiled and executed, it produces the following result:

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```

Now, if you need to define a pointer with specific number of bytes to be referred by it later, you should use the **getmem** function or the **getmem** procedure, which has the following syntax:

```
procedure Getmem(
   out p: pointer;
   Size: PtrUInt
);

function GetMem(
   size: PtrUInt
):pointer;
```

In the previous example, we declared a pointer to a string. A string has a maximum value of 255 bytes. If you really don't need that much space, or a larger space, in terms of bytes, getmem subprogram allows specifying that. Let us rewrite the previous example, using getmem:

```
program exMemory;
var
name: array[1..100] of char;
description: ^string;
begin
   name:= 'Zara Ali';
   description := getmem(200);
      if not assigned(description) then
         writeln(' Error - unable to allocate required memory')
      else
         description^ := 'Zara ali a DPS student in class 10th';
   writeln('Name = ', name );
   writeln('Description: ', description^ );
   freemem(description);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Name = Zara Ali
Description: Zara ali a DPS student in class 10th
```

So, you have complete control and you can pass any size value while allocating memory unlike arrays, where once you defined the size cannot be changed.

Resizing and Releasing Memory

When your program comes out, operating system automatically releases all the memory allocated by your program, but as a good practice when you are not in need of memory anymore, then you should release that memory.

Pascal provides the procedure **dispose** to free a dynamically created variable using the procedure **new.** If you have allocated memory using the **getmem** subprogram, then you need to use the subprogram **freemem** to free this memory. The *freemem* subprograms have the following syntax:

```
procedure Freemem(
   p: pointer;
  Size: PtrUInt
);

function Freemem(
   p: pointer
):PtrUInt;
```

Alternatively, you can increase or decrease the size of an allocated memory block by calling the function ReAllocMem. Let us check the above program once again and make use of ReAllocMem and freemem subprograms. Following is the syntax for ReAllocMem:

```
function ReAllocMem(
   var p: pointer;
   Size: PtrUInt
):pointer;
```

Following is an example which makes use of ReAllocMem and freemem subprograms:

```
program exMemory;
var
name: array[1..100] of char;
description: ^string;
desp: string;
begin
   name:= 'Zara Ali';
   desp := 'Zara ali a DPS student.';
   description := getmem(30);
   if not assigned(description) then
      writeln('Error - unable to allocate required memory')
   else
      description^ := desp;
(* Suppose you want to store bigger description *)
   description := reallocmem(description, 100);
   desp := desp + ' She is in class 10th.';
   description^:= desp;
   writeln('Name = ', name );
   writeln('Description: ', description^ );
   freemem(description);
end.
```

```
Name = Zara Ali
Description: Zara ali a DPS student. She is in class 10th
```

When the above code is compiled and executed, it produces the following result:

```
Name = Zara Ali
Description: Zara ali a DPS student. She is in class 10th
```

Memory Management Functions

Pascal provides a hoard of memory management functions that is used in implementing various data structures and implementing low-level programming in Pascal. Many of these functions are implementation dependent. Free Pascal provides the following functions and procedures for memory management:

S.N.	Function Name & Description
1	**function Addr(X: TAnytype):Pointer;** Returns address of variable
2	**function Assigned(P: Pointer):Boolean;** Checks if a pointer is valid
3	**function CompareByte(const buf1; const buf2; len: SizeInt):SizeInt;** Compares 2 memory buffers byte per byte
4	**function CompareChar(const buf1; const buf2; len: SizeInt):SizeInt;** Compares 2 memory buffers byte per byte
5	**function CompareDWord(const buf1; const buf2; len: SizeInt):SizeInt;** Compares 2 memory buffers byte per byte
6	**function CompareWord(const buf1; const buf2; len: SizeInt):SizeInt;** Compares 2 memory buffers byte per byte
7	**function Cseg: Word;** Returns code segment
8	**procedure Dispose(P: Pointer);** Frees dynamically allocated memory
9	**procedure Dispose(P: TypedPointer; Des: TProcedure);**

	Frees dynamically allocated memory
10	**function Dseg: Word;** Returns data segment
11	**procedure FillByte(var x; count: SizeInt; value: Byte);** Fills memory region with 8-bit pattern
12	**procedure FillChar(var x; count: SizeInt; Value: Byte|Boolean|Char);** Fills memory region with certain character
13	**procedure FillDWord(var x; count: SizeInt; value: DWord);** Fills memory region with 32-bit pattern
14	**procedure FillQWord(var x; count: SizeInt; value: QWord);** Fills memory region with 64-bit pattern
15	**procedure FillWord(var x; count: SizeInt; Value: Word);** Fills memory region with 16-bit pattern
16	**procedure Freemem(p: pointer; Size: PtrUInt);** Releases allocated memory
17	**procedure Freemem(p: pointer);** Releases allocated memory
18	**procedure Getmem(out p: pointer; Size: PtrUInt);** Allocates new memory
19	**procedure Getmem(out p: pointer);** Allocates new memory
20	**procedure GetMemoryManager(var MemMgr: TMemoryManager);** Returns current memory manager
21	**function High(Arg: TypeOrVariable):TOrdinal;** Returns highest index of open array or enumerate
22	**function IndexByte(const buf; len: SizeInt; b: Byte):SizeInt;** Finds byte-sized value in a memory range

23	**function IndexChar(const buf; len: SizeInt; b: Char):SizeInt;** Finds char-sized value in a memory range
24	**function IndexDWord(const buf; len: SizeInt; b: DWord):SizeInt;** Finds DWord-sized (32-bit) value in a memory range
25	**function IndexQWord(const buf; len: SizeInt; b: QWord):SizeInt;** Finds QWord-sized value in a memory range
26	**function Indexword(const buf; len: SizeInt; b: Word):SizeInt;** Finds word-sized value in a memory range
27	**function IsMemoryManagerSet: Boolean;** Is the memory manager set
28	**function Low(Arg: TypeOrVariable):TOrdinal;** Returns lowest index of open array or enumerated
29	**procedure Move(const source; var dest; count: SizeInt);** Moves data from one location in memory to another
30	**procedure MoveChar0(const buf1; var buf2; len: SizeInt);** Moves data till first zero character
31	**procedure New(var P: Pointer);** Dynamically allocates memory for variable
32	**procedure New(var P: Pointer; Cons: TProcedure);** Dynamically allocates memory for variable
33	**function Ofs(var X):LongInt;** Returns offset of variable
34	**function ptr(sel: LongInt; off: LongInt):farpointer;** Combines segment and offset to pointer
35	**function ReAllocMem(var p: pointer; Size: PtrUInt):pointer;** Resizes a memory block on the heap
36	**function Seg(var X):LongInt;**

	Returns segment
37	**procedure SetMemoryManager(const MemMgr: TMemoryManager);** Sets a memory manager
38	**function Sptr: Pointer;** Returns current stack pointer
39	**function Sseg: Word;** Returns stack segment register value

CHAPTER

23

Units

This section shows the units used in a Pascal program:

A Pascal program can consist of modules called units. A unit might consist of some code blocks, which in turn are made up of variables and type declarations, statements, procedures, etc. There are many built-in units in Pascal and Pascal allows programmers to define and write their own units to be used later in various programs.

Using Built-in Units

Both the built-in units and user-defined units are included in a program by the uses clause. We have already used the variants unit in Pascal - Variants tutorial. This tutorial explains creating and including user-defined units. However, let us first see how to include a built-in unit crt in your program:

```
program myprog;
uses crt;
```

The following example illustrates using the **crt** unit:

```
Program Calculate_Area (input, output);
uses crt;
var
   a, b, c, s, area: real;
begin
   textbackground(white); (* gives a white background *)
   clrscr; (*clears the screen *)
   textcolor(green); (* text color is green *)
   gotoxy(30, 4); (* takes the pointer to the 4th line and 30th column)
   writeln('This program calculates area of a triangle:');
   writeln('Area = area = sqrt(s(s-a)(s-b)(s-c))');
   writeln('S stands for semi-perimeter');
   writeln('a, b, c are sides of the triangle');
   writeln('Press any key when you are ready');
```

```
readkey;
  clrscr;
  gotoxy(20,3);
  write('Enter a: ');
  readln(a);
  gotoxy(20,5);
  write('Enter b:');
  readln(b);
  gotoxy(20, 7);
  write('Enter c: ');
  readln(c);

  s := (a + b + c)/2.0;
  area := sqrt(s * (s - a)*(s-b)*(s-c));
  gotoxy(20, 9);
  writeln('Area: ',area:10:3);
  readkey;
end.
```

It is the same program we used right at the beginning of the Pascal tutorial, compile and run it to find the effects of the change.

Creating and Using a Pascal Unit

To create a unit, you need to write the modules or subprograms you want to store in it and save it in a file with **.pas** extension. The first line of this file should start with the keyword unit followed by the name of the unit. For example:

```
unit calculateArea;
```

Following are three important steps in creating a Pascal unit:

- The name of the file and the name of the unit should be exactly same. So, our unit calculateArea will be saved in a file named calculateArea.pas.
- The next line should consist of a single keyword **interface**. After this line, you will write the declarations for all the functions and procedures that will come in this unit.
- Right after the function declarations, write the word **implementation**, which is again a keyword. After the line containing the keyword implementation, provide definition of all the subprograms.

The following program creates the unit named calculateArea:

```
unit CalculateArea;
interface
function RectangleArea( length, width: real): real;
function CircleArea(radius: real) : real;
function TriangleArea( side1, side2, side3: real): real;

implementation
function RectangleArea( length, width: real): real;
begin
   RectangleArea := length * width;
end;

function CircleArea(radius: real) : real;
const
   PI = 3.14159;
begin
   CircleArea := PI * radius * radius;
end;

function TriangleArea( side1, side2, side3: real): real;
var
   s, area: real;
begin
   s := (side1 + side2 + side3)/2.0;
   area := sqrt(s * (s - side1)*(s-side2)*(s-side3));
   TriangleArea := area;
end;
end.
```

Next, let us write a simple program that would use the unit we defined above:

```
program AreaCalculation;
uses CalculateArea,crt;

var
   l, w, r, a, b, c, area: real;
begin
   clrscr;
   l := 5.4;
   w := 4.7;
   area := RectangleArea(l, w);
   writeln('Area of Rectangle 5.4 x 4.7 is: ', area:7:3);
r:= 7.0;
   area:= CircleArea(r);
   writeln('Area of Circle with radius 7.0 is: ', area:7:3);
   a := 3.0;
   b:= 4.0;
   c:= 5.0;
   area:= TriangleArea(a, b, c);
   writeln('Area of Triangle 3.0 by 4.0 by 5.0 is: ', area:7:3);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Area of Rectangle 5.4 x 4.7 is: 25.380
Area of Circle with radius 7.0 is: 153.938
Area of Triangle 3.0 by 4.0 by 5.0 is: 6.000
```

CHAPTER

24

Date Time

This section shows various date and time functions:

Most of the softwares you write need implementing some form of date functions returning current date and time. Dates are so much part of everyday life that it becomes easy to work with them without thinking. Pascal also provides powerful tools for date arithmetic that makes manipulating dates easy. However, the actual name and workings of these functions are different for different compilers.

Getting the Current Date & Time:

Pascal's TimeToString function gives you the current time in a colon(:) delimited form. The following example shows how to get the current time:

```
program TimeDemo;
uses sysutils;
begin
   writeln ('Current time : ',TimeToStr(Time));
end.
```

When the above code was compiled and executed, it produced the following result:

```
Current time : 18:33:08
```

The **Date** function returns the current date in **TDateTime** format. The TDateTime is a double value, which needs some decoding and formatting. The following program demonstrates how to use it in your program to display the current date:

```
Program DateDemo;
uses sysutils;
var
   YY,MM,DD : Word;
begin
   writeln ('Date : ',Date);
   DeCodeDate (Date,YY,MM,DD);
   writeln (format ('Today is (DD/MM/YY): %d/%d/%d ',[dd,mm,yy]));
end.
```

When the above code was compiled and executed, it produced the following result:

```
Date: 4.111300000000000E+004
Today is (DD/MM/YY):23/7/2012
```

The Now function returns the current date and time:

```
Program DatenTimeDemo;
uses sysutils;
begin
   writeln ('Date and Time at the time of writing : ',DateTimeToStr(Now));
end.
```

When the above code was compiled and executed, it produced the following result:

```
Date and Time at the time of writing : 23/7/2012 18:51:
```

Free Pascal provides a simple time stamp structure named TTimeStamp, which has the following format:

```
type TTimeStamp = record
  Time: Integer;
  Date: Integer;
end;
```

Various Date & Time Functions:

Free Pascal provides the following date and time functions:

S.N.	Function Name & Description
1	**function DateTimeToFileDate(DateTime: TDateTime):LongInt;** Converts DateTime type to file date.

2	**function DateTimeToStr(DateTime: TDateTime):;** Constructs string representation of DateTime
3	**function DateTimeToStr(DateTime: TDateTime; const FormatSettings: TFormatSettings):;** Constructs string representation of DateTime
4	**procedure DateTimeToString(out Result: ;const FormatStr: ;const DateTime: TDateTime);** Constructs string representation of DateTime
5	**procedure DateTimeToString(out Result: ; const FormatStr: ; const DateTime: TDateTime; const FormatSettings: TFormatSettings);** Constructs string representation of DateTime
6	**procedure DateTimeToSystemTime(DateTime: TDateTime; out SystemTime: TSystemTime);** Converts DateTime to system time
7	**function DateTimeToTimeStamp(DateTime: TDateTime):TTimeStamp** Converts DateTime to timestamp
8	**function DateToStr(Date: TDateTime):;** Constructs string representation of date
9	**function DateToStr(Date: TDateTime; const FormatSettings: TFormatSettings):;** Constructs string representation of date
10	**function Date: TDateTime;** Gets current date
11	**function DayOfWeek(DateTime: TDateTime):Integer;** Gets day of week
12	**procedure DecodeDate(Date: TDateTime; out Year: Word; out Month: Word; out Day: Word);** Decodes DateTime to year month and day
13	**procedure DecodeTime(Time: TDateTime; out Hour: Word; out Minute: Word; out Second: Word; out MilliSecond: Word);** Decodes DateTime to hours, minutes and seconds

14	**function EncodeDate(Year: Word; Month: Word; Day: Word):TDateTime;** Encodes year, day and month to DateTime
15	**function EncodeTime(Hour: Word; Minute: Word; Second: Word; MilliSecond: Word):TDateTime;** Encodes hours, minutes and seconds to DateTime
16	**function FormatDateTime(const FormatStr: ; DateTime: TDateTime):;** Returns string representation of DateTime
17	**function FormatDateTime(const FormatStr: ; DateTime: TDateTime; const FormatSettings: TFormatSettings):;** Returns string representation of DateTime
18	**function IncMonth(const DateTime: TDateTime; NumberOfMonths: Integer = 1):TDateTime;** Adds 1 to month
19	**function IsLeapYear(Year: Word):Boolean;** Determines if year is leap year
20	**function MSecsToTimeStamp(MSecs: Comp):TTimeStamp;** Converts number of milliseconds to timestamp
21	**function Now: TDateTime;** Gets current date and time
22	**function StrToDateTime(const S:):TDateTime;** Converts string to DateTime
23	**function StrToDateTime(const s: ShortString; const FormatSettings: TFormatSettings):TDateTime;** Converts string to DateTime
24	**function StrToDateTime(const s: AnsiString; const FormatSettings: TFormatSettings):TDateTime;** Converts string to DateTime
25	**function StrToDate(const S: ShortString):TDateTime;** Converts string to date
26	**function StrToDate(const S: Ansistring):TDateTime;**

	Converts string to date
27	**function StrToDate(const S: ShortString; separator: Char):TDateTime;** Converts string to date
28	**function StrToDate(const S: AnsiString; separator: Char):TDateTime;** Converts string to date
29	**function StrToDate(const S: ShortString; const useformat: ; separator: Char):TDateTime;** Converts string to date
30	**function StrToDate(const S: AnsiString; const useformat: ; separator: Char):TDateTime;** Converts string to date
31	**function StrToDate(const S: PChar; Len: Integer; const useformat: ; separator: Char = #0):TDateTime;** Converts string to date
32	**function StrToTime(const S: Shortstring):TDateTime;** Converts string to time
33	**function StrToTime(const S: Ansistring):TDateTime;** Converts string to time
34	**function StrToTime(const S: ShortString; separator: Char):TDateTime;** Converts string to time
35	**function StrToTime(const S: AnsiString; separator: Char):TDateTime;** Converts string to time
36	**function StrToTime(const S: ; FormatSettings: TFormatSettings):TDateTime;** Converts string to time
37	**function StrToTime(const S: PChar; Len: Integer; separator: Char = #0):TDateTime;** Converts string to time
38	**function SystemTimeToDateTime(const SystemTime: TSystemTime):TDateTime;** Converts system time to datetime

39	**function TimeStampToDateTime(const TimeStamp: TTimeStamp):TDateTime;** Converts time stamp to DateTime
40	**function TimeStampToMSecs(const TimeStamp: TTimeStamp):comp;** Converts Timestamp to number of milliseconds
41	**function TimeToStr(Time: TDateTime):;** Returns string representation of Time
42	**function TimeToStr(Time: TDateTime; const FormatSettings: TFormatSettings):;** Returns string representation of Time
43	**function Time: TDateTime;** Gets current time

The following example illustrates the use of some of the above functions:

```
Program DatenTimeDemo;
uses sysutils;
var
year, month, day, hr, min, sec, ms: Word;
begin
   writeln ('Date and Time at the time of writing : ',DateTimeToStr(Now));
   writeln('Today is ',LongDayNames[DayOfWeek(Date)]);
   writeln;
   writeln('Details of Date: ');
   DecodeDate(Date,year,month,day);
   writeln (Format ('Day: %d',[day]));
   writeln (Format ('Month: %d',[month]));
   writeln (Format ('Year: %d',[year]));
   writeln;
   writeln('Details of Time: ');
   DecodeTime(Time,hr, min, sec, ms);
   writeln (format('Hour: %d:',[hr]));
   writeln (format('Minutes: %d:',[min]));
   writeln (format('Seconds: %d:',[sec]));
   writeln (format('Milliseconds: %d:',[hr]));
end.
```

When the above code was compiled and executed, it produced the following result:

```
Date and Time at the time of writing : 7/24/2012 8:26:
Today is Tuesday
Details of Date:
Day:24
Month:7
Year: 2012
Details of Time:
Hour: 8
Minutes: 26
Seconds: 21
Milliseconds: 8
```

CHAPTER

25

Objects

This section shows the concept of Objects under Object-Oriented Pascal:

We can imagine our universe made of different objects like sun, earth, moon, etc. Similarly, we can imagine our car made of different objects like wheel, steering, gear, etc. Same way, there are object-oriented programming concepts, which assume everything as an object and implement a software using different objects. In Pascal, there are two structural data types used to implement a real world object:

- Object types
- Class types

Object-Oriented Concepts:

Before we go in detail, let's define important Pascal terms related to Object-Oriented Pascal.

- **Object**: An Object is a special kind of record that contains fields like a record; however, unlike records, objects contain procedures and functions as part of the object. These procedures and functions are held as pointers to the methods associated with the object's type.
- **Class**: A Class is defined in almost the same way as an Object, but there is a difference in way they are created. The Class is allocated on the Heap of a program, whereas the Object is allocated on the Stack. It is a pointer to the object, not the object itself.
- **Instantiation of a class**: Instantiation means creating a variable of that class type. Since a class is just a pointer, when a variable of a class type is declared, there is memory allocated only for the pointer, not for the entire object. Only when it is instantiated using one of its constructors, memory is allocated for the object. Instances of a class are also called 'objects', but do not confuse them with Object Pascal Objects. In this tutorial, we will write 'Object' for Pascal Objects and 'object' for the conceptual object or class instance.
- **Member Variables**: These are the variables defined inside a Class or an Object.
- **Member Functions**: These are the functions or procedures defined inside a Class or an Object and are used to access object data.

- **Visibility of Members**: The members of an Object or Class are also called the fields. These fields have different visibilities. Visibility refers to accessibility of the members, i.e., exactly where these members will be accessible. Objects have three visibility levels: public, private and protected. Classes have five visibility types: public, private, strictly private, protected and published. We will discuss visibility in details.
- **Inheritance**: When a Class is defined by inheriting existing functionalities of a parent Class, then it is said to be inherited. Here, child class will inherit all or few member functions and variables of a parent class. Objects can also be inherited.
- **Parent Class**: A Class that is inherited by another Class. This is also called a base class or super class.
- **Child Class**: A class that inherits from another class. This is also called a subclass or derived class.
- **Polymorphism**: This is an object-oriented concept where same function can be used for different purposes. For example, function name will remain same but it may take different number of arguments and can do different tasks. Pascal classes implement polymorphism. Objects do not implement polymorphism.
- **Overloading**: It is a type of polymorphism in which some or all of operators have different implementations depending on the types of their arguments. Similarly, functions can also be overloaded with different implementation. Pascal classes implement overloading, but the Objects do not.
- **Data Abstraction**: Any representation of data in which the implementation details are hidden (abstracted).
- **Encapsulation**: Refers to a concept where we encapsulate all the data and member functions together to form an object.
- **Constructor**: Refers to a special type of function, which will be called automatically whenever there is an object formation from a class or an Object.
- **Destructor**: Refers to a special type of function, which will be called automatically whenever an Object or Class is deleted or goes out of scope.

Defining Pascal Objects

An object is declared using the type declaration. The general form of an object declaration is as follows:

```
type object-identifier = object
   private
   field1 : field-type;
   field2 : field-type;
   ...
   public
   procedure proc1;
   function f1(): function-type;
   end;
var objectvar : object-identifier;
```

Let us define a Rectangle Object that has two integer type data members - **length** and **width** and some member functions to manipulate these data members and a procedure to draw the rectangle.

```
type
  Rectangle = object
  private
    length, width: integer;
  public
    constructor init;
    destructor done;
    procedure setlength(l: inteter);
    function getlength(): integer;
    procedure setwidth(w: integer);
    function getwidth(): integer;
    procedure draw;
end;
var
  r1: Rectangle;
  pr1: ^Rectangle;
```

After creating your objects, you will be able to call member functions related to that object. One member function will be able to process member variable of related object only.

Following example shows how to set lengths and widths for two rectangle objects and draw them by calling the member functions.

```
r1.setlength(3);
r1.setwidth(7);
writeln(' Draw a rectangle: ', r1.getlength(), ' by ' , r1.getwidth());
r1.draw;
new(pr1);
pr1^.setlength(5);
pr1^.setwidth(4);
writeln(' Draw a rectangle: ', pr1^.getlength(), ' by ' ,pr1^.getwidth());
pr1^.draw;
dispose(pr1);
```

Following is a complete example to show how to use objects in Pascal:

```
program exObjects;
type
  Rectangle = object
  private
    length, width: integer;
  public
    procedure setlength(l: integer);
    function getlength(): integer;
    procedure setwidth(w: integer);
    function getwidth(): integer;
    procedure draw;
end;
```

```
var
   r1: Rectangle;
   pr1: ^Rectangle;
procedure Rectangle.setlength(l: integer);
begin
   length := l;
end;

procedure Rectangle.setwidth(w: integer);
begin
   width :=w;
end;

function Rectangle.getlength(): integer;
begin
   getlength := length;
end;

function Rectangle.getwidth(): integer;
begin
   getwidth := width;
end;

procedure Rectangle.draw;
var
   i, j: integer;
begin
   for i:= 1 to length do
   begin
     for j:= 1 to width do
        write(' * ');
     writeln;
   end;
end;

begin
   r1.setlength(3);
   r1.setwidth(7);
   writeln('Draw a rectangle:', r1.getlength(), ' by ' , r1.getwidth());
   r1.draw;
   new(pr1);
   pr1^.setlength(5);
   pr1^.setwidth(4);
   writeln('Draw a rectangle:', pr1^.getlength(), ' by ' ,pr1^.getwidth());
   pr1^.draw;
   dispose(pr1);
end.
```

When the above code is compiled and executed, it produces the following result:

```
Draw a rectangle: 3 by 7
* * * * * * *
* * * * * * *
* * * * * * *
Draw a rectangle: 5 by 4
* * * *
* * * *
* * * *
* * * *
* * * *
```

Visibility of the Object Members

Visibility indicates the accessibility of the object members. Pascal object members have three types of visibility:

Visibility	**Accessibility**
Public	The members can be used by other units outside the program unit
Private	The members are only accessible in the current unit.
Protected	The members are available only to objects descended from the parent object.

By default, fields and methods of an object are public and are exported outside the current unit.

Constructors and Destructors for Pascal Objects:

Constructors are special type of methods, which are called automatically whenever an object is created. You create a constructor in Pascal just by declaring a method with a keyword constructor. Conventionally, the method name is Init, however, you can provide any valid identifier of your own. You can pass as many arguments as you like into the constructor function.

Destructors are methods that are called during the destruction of the object. The destructor methods destroy any memory allocation created by constructors.

Following example will provide a constructor and a destructor for the Rectangle class which will initialize length and width for the rectangle at the time of object creation and destroy it when it goes out of scope.

```
program exObjects;
type
  Rectangle = object
  private
    length, width: integer;
  public
    constructor init(l, w: integer);
    destructor done;
    procedure setlength(l: integer);
    function getlength(): integer;
    procedure setwidth(w: integer);
    function getwidth(): integer;
    procedure draw;
end;
var
  r1: Rectangle;
  pr1: ^Rectangle;
constructor Rectangle.init(l, w: integer);
begin
  length := l;
  width := w;
end;

destructor Rectangle.done;
begin
  writeln(' Desctructor Called');
end;

procedure Rectangle.setlength(l: integer);
begin
  length := l;
end;

procedure Rectangle.setwidth(w: integer);
begin
  width :=w;
end;

function Rectangle.getlength(): integer;
begin
  getlength := length;
end;

function Rectangle.getwidth(): integer;
begin
  getwidth := width;
end;

procedure Rectangle.draw;
```

```
begin
   r1.init(3, 7);
   writeln('Draw a rectangle:', r1.getlength(), ' by ' , r1.getwidth());
   r1.draw;
   new(pr1, init(5, 4));
   writeln('Draw a rectangle:', pr1^.getlength(), ' by ',pr1^.getwidth());
   pr1^.draw;
   pr1^.init(7, 9);
   writeln('Draw a rectangle:', pr1^.getlength(), ' by ' ,pr1^.getwidth());
   pr1^.draw;
   dispose(pr1);
   r1.done;
end.
```

When the above code is compiled and executed, it produces the following result:

```
Draw a rectangle: 3 by 7
* * * * * * *
* * * * * * *
* * * * * * *
Draw a rectangle: 5 by 4
* * * *
* * * *
* * * *
* * * *
* * * *
Draw a rectangle: 7 by 9
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * * *
Destructor Called
```

Inheritance for Pascal Objects:

Pascal objects can optionally inherit from a parent object. The following program illustrates inheritance in Pascal Objects. Let us create another object named **TableTop**, which is inheriting from the Rectangle object.

```
program exObjects;
type
   Rectangle = object
   private
      length, width: integer;
   public
      procedure setlength(l: integer);
      function getlength(): integer;
      procedure setwidth(w: integer);
      function getwidth(): integer;
      procedure draw;
end;
TableTop = object (Rectangle)
   private
     material: string;
   public
      function getmaterial(): string;
      procedure setmaterial( m: string);
      procedure displaydetails;
      procedure draw;
end;
var
   tt1: TableTop;

procedure Rectangle.setlength(l: integer);
begin
   length := l;
end;

procedure Rectangle.setwidth(w: integer);
begin
   width :=w;
end;

function Rectangle.getlength(): integer;
begin
   getlength := length;
end;
function Rectangle.getwidth():integer;
begin
   getwidth := width;
end;
procedure Rectangle.draw;
var
   i, j: integer;
begin
   for i:= 1 to length do
   begin
      for j:= 1 to width do
         write(' * ');
      writeln;
  end;
end;
```

```
function TableTop.getmaterial(): string;
begin
   getmaterial := material;
end;

procedure TableTop.setmaterial( m: string);
begin
   material := m;
end;

procedure TableTop.displaydetails;
begin
   writeln('Table Top: ', self.getlength(), ' by ' , self.getwidth());
   writeln('Material: ', self.getmaterial());
end;

procedure TableTop.draw();
var
   i, j: integer;
begin
   for i:= 1 to length do
   begin
      for j:= 1 to width do
         write(' * ');
   writeln;
   end;
   writeln('Material: ', material);
end;

begin
   tt1.setlength(3);
   tt1.setwidth(7);
   tt1.setmaterial('Wood');
   tt1.displaydetails();
   writeln;
   writeln('Calling the Draw method');
   tt1.draw();
end.
```

Following are the important points, which should be noted down:

- The object *Tabletop* has inherited all the members of the Rectangle object.
- There is a draw method in *TableTop* also. When the *draw* method is called using a *TableTop* object, TableTop's draw gets invoked.
- There is an implicit instance named **self** that refers to the current instance of the object.

When the above code is compiled and executed, it produces the following result:

```
Table Top: 3 by 7
Material: Wood

Calling the Draw Method
* * * * * * *
* * * * * * *
* * * * * * *
Material: Wood
```

CHAPTER

26

Classes

This section shows the classes under Pascal programming languages:

You have seen that Pascal Objects exhibit some characteristics of object-oriented paradigm. They implement encapsulation, data hiding and inheritance, but they also have limitations. For example, Pascal Objects do not take part in polymorphism. So, classes are widely used to implement proper object-oriented behavior in a program, especially the GUI-based software.

A Class is defined in almost the same way as an Object, but is a pointer to an Object rather than the Object itself. Technically, this means that the Class is allocated on the Heap of a program, whereas the Object is allocated on the Stack. In other words, when you declare a variable the object type, it will take up as much space on the stack as the size of the object, but when you declare a variable of the class type, it will always take the size of a pointer on the stack. The actual class data will be on the heap.

Defining Pascal Classes:

A class is declared in the same way as an object, using the type declaration. The general form of a class declaration is as follows:

```
type class-identifier = class
  private
    field1 : field-type;
    field2 : field-type;
      ...
  public
    constructor create();
    procedure proc1;
    function f1(): function-type;
end;
var classvar : class-identifier;
```

It's worth to note following important points:

- Class definitions should come under the type declaration part of the program only.
- A class is defined using the **class** keyword.
- Fields are data items that exist in each instance of the class.
- Methods are declared within the definition of a class.
- There is a predefined constructor called **Create** in the Root class. Every abstract class and every concrete class is a descendant of Root, so all classes have at least one constructor.
- There is a predefined destructor called **Destroy** in the Root class. Every abstract class and every concrete class is a descendant of Root, so, all classes have at least one destructor.

Let us define a Rectangle class that has two integer type data members - length and width and some member functions to manipulate these data members and a procedure to draw the rectangle.

```
type
  Rectangle = class
  private
    length, width: integer;
  public
    constructor create(l, w: integer);
    procedure setlength(l: integer);
    function getlength(): integer;
    procedure setwidth(w: integer);
    function getwidth(): integer;
    procedure draw;
end;
```

Let us write a complete program that would create an instance of a rectangle class and draw the rectangle. This is the same example we used while discussing Pascal Objects. You will find both programs are almost same, with the following exceptions:

- You will need to include the {$mode objfpc} directive for using the classes.
- You will need to include the {$m+} directive for using constructors.
- Class instantiation is different than object instantiation. Only declaring the variable does not create space for the instance, you will use the constructor create to allocate memory.

Here is the complete example:

```
{$mode objfpc} // directive to be used for defining classes
{$m+}                  // directive to be used for using constructor

program exClass;
type
  Rectangle = class
  private
    length, width: integer;
  public
    constructor create(l, w: integer);
    procedure setlength(l: integer);
    function getlength(): integer;
    procedure setwidth(w: integer);
    function getwidth(): integer;
    procedure draw;
end;
var
  r1: Rectangle;
constructor Rectangle.create(l, w: integer);
begin
  length := l;
  width := w;
end;

procedure Rectangle.setlength(l: integer);
begin
  length := l;
end;

procedure Rectangle.setwidth(w: integer);
begin
  width :=w;
end;
function Rectangle.getlength(): integer;
begin
  getlength := length;
end;

function Rectangle.getwidth(): integer;
begin
  getwidth := width;
end;
procedure Rectangle.draw;
var
  i, j: integer;
begin
  for i:= 1 to length do
  begin
    for j:= 1 to width do
      write(' * ');
    writeln;
  end;
end;
```

```
begin
  r1:= Rectangle.create(3, 7);
  writeln(' Darw Rectangle: ', r1.getlength(), ' by ' , r1.getwidth());
  r1.draw;
  r1.setlength(4);
  r1.setwidth(6);
  writeln(' Darw Rectangle: ', r1.getlength(), ' by ' , r1.getwidth());
  r1.draw;
end.
```

When the above code is compiled and executed, it produces the following result:

```
Darw Rectangle: 3 by 7
* * * * * * *
* * * * * * *
* * * * * * *
Darw Rectangle: 4 by 6
* * * * * *
* * * * * *
* * * * * *
* * * * * *
```

Visibility of the Class Members

Visibility indicates the accessibility of the class members. Pascal class members have five types of visibility:

Visibility	**Accessibility**
Public	These members are always accessible.
Private	These members can only be accessed in the module or unit that contains the class definition. They can be accessed from inside the class methods or from outside them.
Strict Private	These members can only be accessed from methods of the class itself. Other classes or descendent classes in the same unit cannot access them.
Protected	This is same as private, except, these members are accessible to descendent types, even if they are implemented in other modules.
Published	This is same as a Public, but the compiler generates type information that is needed for automatic streaming of these classes if the compiler is in the {$M+} state. Fields defined in a published section must be of class type.

Constructors and Destructors for Pascal Classes:

Constructors are special methods, which are called automatically whenever an object is created. So we take full advantage of this behavior by initializing many things through constructor functions.

Pascal provides a special function called create() to define a constructor. You can pass as many arguments as you like into the constructor function.

Following example will create one constructor for a class named Books and it will initialize price and title for the book at the time of object creation.

```
program classExample;

{$MODE OBJFPC} //directive to be used for creating classes
{$M+} //directive that allows class constructors and destructors
type
   Books = Class
   private
      title : String;
      price: real;
   public
      constructor Create(t : String; p: real); //default constructor
      procedure setTitle(t : String); //sets title for a book
      function getTitle() : String; //retrieves title
      procedure setPrice(p : real); //sets price for a book
      function getPrice() : real; //retrieves price
      procedure Display(); // display details of a book
end;
var
   physics, chemistry, maths: Books;

//default constructor
constructor Books.Create(t : String; p: real);
begin
   title := t;
   price := p;
end;

procedure Books.setTitle(t : String); //sets title for a book
begin
   title := t;
end;

function Books.getTitle() : String; //retrieves title
begin
   getTitle := title;
end;
procedure Books.setPrice(p : real); //sets price for a book
begin
   price := p;
end;
```

```
function Books.getPrice() : real; //retrieves price
begin
   getPrice:= price;
end;

procedure Books.Display();
begin
   writeln('Title: ', title);
   writeln('Price: ', price:5:2);
end;

begin
   physics := Books.Create('Physics for High School', 10);
   chemistry := Books.Create('Advanced Chemistry', 15);
   maths := Books.Create('Algebra', 7);
   physics.Display;
   chemistry.Display;
   maths.Display;
end.
```

When the above code is compiled and executed, it produces the following result:

```
Title: Physics for High School
Price: 10
Title: Advanced Chemistry
Price: 15
Title: Algebra
Price: 7
```

Like the implicit constructor named create, there is also an implicit destructor method destroy using which you can release all the resources used in the class.

Inheritance:

Pascal class definitions can optionally inherit from a parent class definition. The syntax is as follows:

```
type
childClas-identifier = class(baseClass-identifier)
< members >
end;
```

Following example provides a novels class, which inherits the Books class and adds more functionality based on the requirement.

```
program inheritanceExample;

{$MODE OBJFPC} //directive to be used for creating classes
{$M+} //directive that allows class constructors and destructors

type
  Books = Class
  protected
    title : String;
    price: real;
  public
    constructor Create(t : String; p: real); //default constructor
    procedure setTitle(t : String); //sets title for a book
    function getTitle() : String; //retrieves title
    procedure setPrice(p : real); //sets price for a book
    function getPrice() : real; //retrieves price
    procedure Display(); virtual; // display details of a book
end;
(* Creating a derived class *)

type
  Novels = Class(Books)
  private
    author: String;
  public
    constructor Create(t: String); overload;
    constructor Create(a: String; t: String; p: real); overload;
    procedure setAuthor(a: String); // sets author for a book
    function getAuthor(): String; // retrieves author name
    procedure Display(); override;
end;
var
  n1, n2: Novels;
//default constructor
constructor Books.Create(t : String; p: real);
begin
  title := t;
  price := p;
end;

procedure Books.setTitle(t : String); //sets title for a book
begin
  title := t;
end;

function Books.getTitle() : String; //retrieves title
begin
  getTitle := title;
end;
procedure Books.setPrice(p : real); //sets price for a book
begin
  price := p;
end;
```

```
function Books.getPrice() : real; //retrieves price
begin
   getPrice:= price;
end;

procedure Books.Display();
begin
   writeln('Title: ', title);
   writeln('Price: ', price);
end;

(* Now the derived class methods  *)
constructor Novels.Create(t: String);
begin
   inherited Create(t, 0.0);
   author:= ' ';
end;

constructor Novels.Create(a: String; t: String; p: real);
begin
   inherited Create(t, p);
   author:= a;
end;

procedure Novels.setAuthor(a : String); //sets author for a book
begin
   author := a;
end;

function Novels.getAuthor() : String; //retrieves author
begin
   getAuthor := author;
end;

procedure Novels.Display();
begin
   writeln('Title: ', title);
   writeln('Price: ', price:5:2);
   writeln('Author: ', author);
end;
begin
   n1 := Novels.Create('Gone with the Wind');
   n2 := Novels.Create('Ayn Rand','Atlas Shrugged', 467.75);
   n1.setAuthor('Margaret Mitchell');
   n1.setPrice(375.99);
   n1.Display;
   n2.Display;
end.
```

When the above code is compiled and executed, it produces the following result:

```
Title: Gone with the Wind
Price: 375.99
Author: Margaret Mitchell
Title: Atlas Shrugged
Price: 467.75
Author: Ayn Rand
```

It's worth to note following important points:

- The members of the Books class have protected visibility.
- The Novels class has two constructors, so the overload operator is used for function overloading.
- The Books.Display procedure has been declared virtual, so that the same method from the Novels class can override it.
- The Novels.Create constructor calls the base class constructor using the inherited keyword.

Interfaces:

Interfaces are defined to provide a common function name to the implementers. Different implementers can implement those interfaces according to their requirements. You can say, interfaces are skeletons, which are implemented by developers. Following is an example of interface:

```
type
   Mail = Interface
      Procedure SendMail;
      Procedure GetMail;
   end;
   Report = Class(TInterfacedObject,  Mail)
      Procedure SendMail;
      Procedure GetMail;
   end;
```

Please note that, when a class implements an interface, it should implement all methods of the interface. If a method of an interface is not implemented, then the compiler will give an error.

Abstract Classes:

An abstract class is one that cannot be instantiated, only inherited. An abstract class is specified by including the word symbol abstract in the class definition, like this:

```
type
   Shape = ABSTRACT CLASS (Root)
      Procedure draw; ABSTRACT;
      ...
   end;
```

When inheriting from an abstract class, all methods marked abstract in the parent's class declaration must be defined by the child; additionally, these methods must be defined with the same visibility.

Static Keyword:

Declaring class members or methods as static makes them accessible without needing an instantiation of the class. A member declared as static cannot be accessed with an instantiated class object (though a static method can). The following example illustrates the concept:

```
{$mode objfpc}
{$static on}
type
  myclass=class
    num : integer;static;
  end;
var
  n1, n2 : myclass;
begin
  n1:= myclass.create;
  n2:= myclass.create;
  n1.num := 12;
  writeln(n2.num);
  n2.num := 31;
  writeln(n1.num);
  writeln(myclass.num);
  myclass.num := myclass.num + 20;
  writeln(n1.num);
  writeln(n2.num);
 end.
```

When the above code is compiled and executed, it produces the following result:

```
12
31
31
51
51
```

Please note that:

- In Pascal, all labels must be declared before constant and variables declarations.
- The **if** and **goto** statements may be used in the compound statement to transfer control out of the compound statement, but it is illegal to transfer control into a compound statement.