# Introduction to Free Pascal

Andry Maykol Pinto, Paulo Portugal e José Faria
amgp@fe.up.pt

# Sumário / Outline


1. **Free Pascal**
2. **Program Structure**
   - *Comments*
   - *Constants*
   - *Variables*
   - *Operators*
     - *Boolean expressions*
     - *Relational expressions*
     - *Assignment*
     - *Arithmetic operator*
   - *Data Types*
     - *Standard data types*
     - *User-defined data types*
     - *Example of enumeration*
     - *Example of sub-range*
     - *Structured data types*
     - *Example of array*
     - *Example of multidimensional array*
     - *Example of dynamic array*
     - *Example of record*
   - *Statements*
     - *With*
     - *IF/ELSE*
     - *CASE*
     - *FOR*
     - *WHILE*
   - *Procedure*
   - *Function*
   - *Object*

# Free Pascal

## Free Pascal

# History

- A programming language intended for scientific computing.
- Pascal is based on the block structured style of the Algol programming language.
- Pascal was the primary high-level language used for development in the Apple Lisa, and in the early years of the Mac.
- The ISO 7185 Pascal Standard was originally published in 1983.
- In 1993, the Pascal Standards Committee published an Object-Oriented Extension to Pascal.

**Free Pascal:** it is a free compiler for running Pascal and Object Pascal programs.

# Program Structure

A typical program structure is made of:

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

# Program Structure

```
1  program {name of the program}
2  uses {comma delimited names of libraries you use}
3  const {global constant declaration block}
4  var {global variable declaration block}
5
6  function {function declarations, if any}
7  { local variables }
8  begin
9  ...
10 end;
11
12 procedure { procedure declarations, if any}
13 { local variables }
14 begin
15 ...
16 end;
17
18 begin { main program block starts}
19 ...
20 end. { the end of main program block }
```

code/FPProgramExample.txt

# Program Structure

**Identifiers** are names that allow you to reference stored values, such as **variables** and **constants**.

Every program and unit must be named by an identifier.

- Several identifiers are reserved in Pascal as syntactical elements.
- Must begin with a letter from the English alphabet.
- Followed by alphanumeric characters (alphabetic characters and numerals) and possibly the underscore (_).

# Program Structure

Program identifier consists of the program header, followed possibly by a "uses" clause, and a block.

A block is made of declarations of labels, constants, types, variables and functions or procedures.

# Program Structure

uses command serves to identify all units that are needed by the program.

Careful with the order in which the units appear since it will determine in which order they are initialized.

# Program Structure

The elements of a program must be in the correct order, by e.g.:

```
1 program HelloWorld;
2 uses crt;
3
4 (* Here the main program block starts *)
5 begin
6    writeln('Hello, World!');
7    readkey;
8 end.
```

code/FPProgramExample2.txt

- **uses crt** is a preprocessor command, which tells the compiler to include the crt unit before going to actual compilation.
- **begin** statement is where the program execution begins.
- **end.** statement ends your program.

# Comments

Comments are portions of the code which do not compile or execute.

- Pascal comments start with a **(*** and end with a ***)**.
- Most other modern compilers support brace comments, such as { **Comment** }

Whitespace (spaces, tabs, and end-of-lines) are ignored by the Pascal compiler unless they are inside a literal string.

Pascal is a **case non-sensitive language**. You can write your variables, functions and procedure in either case.

# Constants

Constant is an entity that remains unchanged during program execution..

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

Constants are assigned at the beginning of the program.

```
const
   Nome= 'Informatica Industrial';
   CaracterInit= 'a';
   Ano = 2020;
   pi = 3.1415926535897932;
   isGood= TRUE;
```

# Variables

var keyword starts a block where it will be defined a set of variables.

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

Variables are declared outside the code-body: they **are not declared within** the begin and end pairs.

Variables that are used only in one block should be declared in that block's declaration part.

# Variables

Variables are similar to constants, but their **values can be changed** as the program runs.

- Variables are **not initialized with zero** by default;
- It is a better practice to initialize variables in a program.

```
1 var
2   age: integer = 15;
3   taxrate: real = 0.5;
4   grade: char = 'A';
5   name: string = 'John Smith';
```

”

# Variable scope

Initialized variables are initialized when they enter scope:

- **Global** will hold their value throughout the lifetime of your program and can be accessed by any function;
- **Local** are initialized each time the procedure/program is entered and, can be used only by statements that are inside that procedure/program.

# Example of variables and constants

Example about using variables and constants:

```
1  program Greetings;
2  const
3  message = ' Welcome to the world of Pascal ';
4
5  type
6  name = string;
7  var
8  firstname, surname: name;
9
10 begin
11    writeln('Please enter your first name: ');
12    readln(firstname);
13
14    writeln('Please enter your surname: ');
15    readln(surname);
16
17    writeln;
18    writeln(message, ' ', firstname, ' ', surname);
19 end.
```

code/EPProgramExampleVariables.txt

# Boolean expressions

Boolean expressions :

- AND operator: (A AND B)
- OR operator: (A OR B)
- NOT operator: (NOT A)
- XOR operator: (A XOR B)

# Relational expressions

## Relational expressions :

```
variable1 OPERATOR variable2
```

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

# Assignment operator

Assignment operator :

```
variable_name := expression;
```

"

```
{ Declaration of variables}
var
   some_int  : integer;
   some_real : real;

...

some_int  := 375;
some_real := 12.023
```

"

# Arithmetic operator

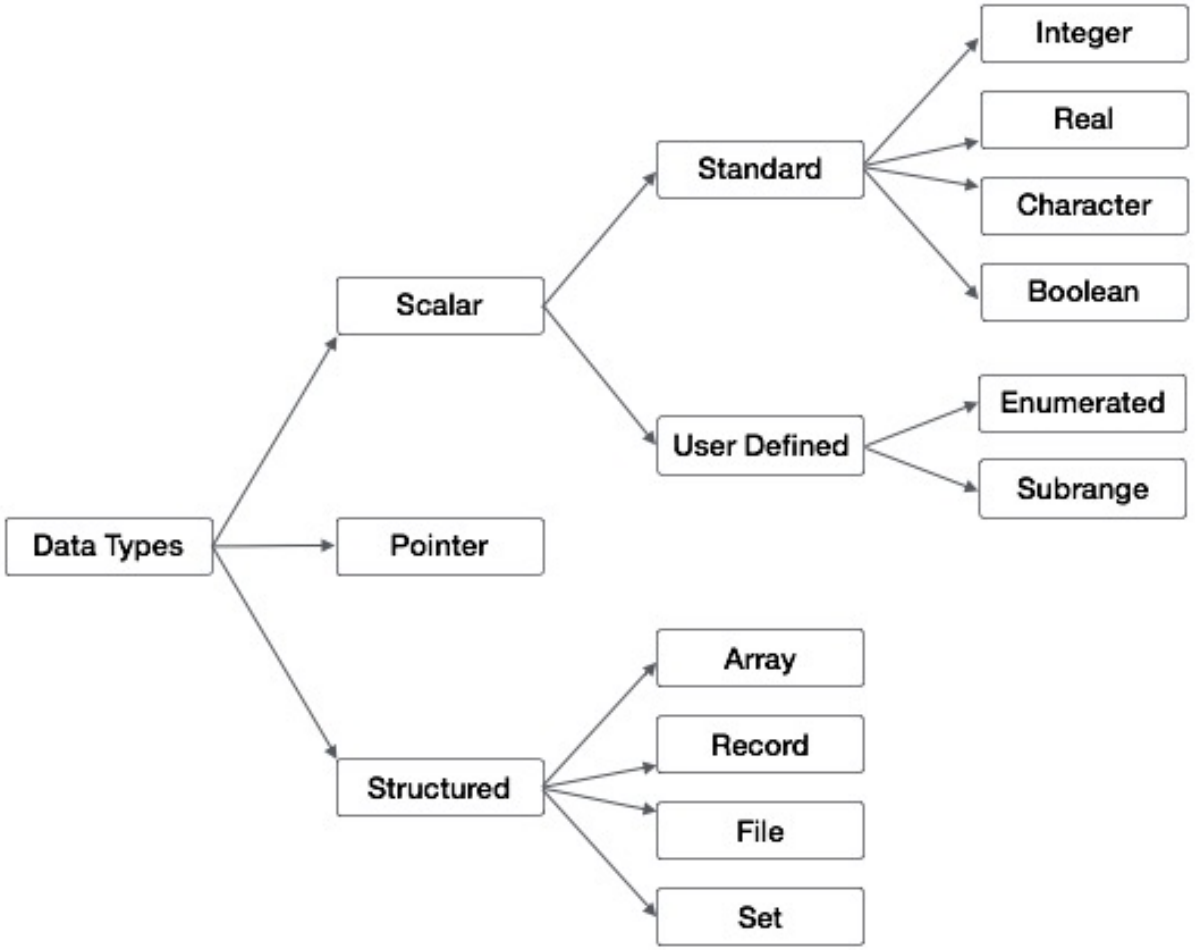
Arithmetic operator :

- $+$ Addition (A + B)
- $-$ Subtraction (A - B)
- $*$ Multiplication (A * B)
- / Real division (A / B)
- *div* Integer division (A div B)
- *mod* Modulus (remainder division) (A mod B)

The operators "div" and "mod" only work on integers while other operations work on both reals and integers.

# Data Types

**Data types** can be summarized:



# Data Types

Type declaration is used to declare the data type of an identifier.

```
type
  Identifier1 , Identifier2 = typeSpecifier;
```

Example:

```
type
   days , age = integer;
   name, city = string;
   fees , expenses = real;
```

# Standard data types

**Standard data types**:

- char;
- boolean;
- string;
- integer;
- real.

Other integer types: Cardinal, Smallint, Longint, Shortint, Byte, ...

# Enumeration

**User-defined data types**:

**Enumerated** data types have values that can be specified in a list:

```
type
  enumIdentifier = (item1, item2, item3, ...);
```

The order that items are listed in the domain defines the order of the items.

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

# Enumeration

Example of enumerated data types:

```
1  program exEnumeration;
2  type
3    beverage = (coffee, tea, milk, water, coke,
       limejuice);
4
5  var
6    drink:beverage;
7
8  begin
9     writeln('Which drink do you want?');
10    drink := limejuice;
11
12    writeln('You can drink ', drink);
13 end.
```

# Sub-range

**Sub-range** data types make a variable to assume values that lie within a certain range:

```
type
   Identifier = lowerLimit ... upperLimit;
```

The order that items are listed in the domain defines the order of the items.

```
type
   marks = 1 ... 100;
   grade = 'A' ... 'E';
   age = 1 ... 25;
```

# Example of sub-range

Example of using sub-range:

```
1  program exSubrange;
2  type
3    tmarks =  1 .. 100;
4    tgrade =  'A' .. 'E';
5
6  var
7    mark: tmarks ;
8    grad: tgrade ;
9
10 begin
11    writeln( 'Enter your marks(1 - 100): ');
12    readln(mark);
13
14    writeln( 'Enter your grade(A - E): ');
15    readln(grad);
16
17    writeln('Marks: ' , mark, ' Grade: ', grad);
18 end.
```

code/PP1ProgramExampleSubRange.txt

# Structured data types

**Array** can store a fixed-size sequential collection of elements of the same type.

```
type
  Identifier = array[indexRange] of elementType;
```

- Array consists of contiguous memory locations.
- The first element is in the lowest address while the last element is in the highest address.

# Structured data types

The declaration of **arrays** in Free Pascal:

```
type
    { Array starting in 1 index and ending on 25 }
    vector_A = array [ 1..25] of real;
    { Array starting in 0 index and ending on 10 }
    vector_B = array [ 0..10] of real;
    { Array starting in -10 index and ending on 50 }
    temperature = array [-10 .. 50] of real;
    { Array from enumerated }
    color = ( red, black, blue, silver, beige);
    car_color = array of [0..3] of color;

var
    velocity: vector_A;
    position: vector_B;
    day_temp, night_temp : temperature;
    car_body: car_color;
```

# Example of array

Example of using arrays:

```
1  program exArrays;
2  var
3     n: array [1..10] of integer;    (* n is an array of 10 integers *)
4     i, j: integer;
5
6  begin
7     (* initialize elements of array n to 0 *)
8     for i := 1 to 10 do
9         n[ i ] := i + 100;    (* set element at location i to i + 100 *)
10     (* output each array element's value *)
11
12    for j:= 1 to 10 do
13       writeln('Element[', j, '] = ', n[j] );
14 end.
```

code/FPProgramExampleArray.txt

# Structured data types

**Multidimensional array** supports arrays in multiple dimensions.

```
type
  identifier = array [indexRange1,
    indexRange2] of datatype;
```

”

# Example of multidimensional array

Example of using multidimensional arrays:

```
1  program arrayAddressOrderDemo;
2  var
3    i: integer;
4    f: array[0..1, 0..3] of boolean;
5  begin
6    for i := 0 to 7 do
7    begin
8      f[0, i] := true;
9    end;
10 end.
```

code/FPProgramExampleMultidimensionalArray.txt

# Structured data types

The declaration of **dynamic arrays** in Free Pascal:

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

"

You must declare the size with the **setlength** function before using the array, e.g., "*setlength*(*a*, 100);".

# Example of dynamic array

Example of a two dimensional dynamic array:

```
1  program exDynarray;
2  var
3     a: array of array of integer; (* a 2 dimensional array *)
4     i, j : integer;
5
6  begin
7     setlength(a,5,5);
8     for i:=0 to 4 do
9        for j:=0 to 4 do
10          a[i,j]:= i * j;
11
12    for i:=0 to 4 do
13    begin
14       for j:= 0 to 4 do
15       write(a[i,j]:2,' ');
16     writeln;
17    end;
18 end.
```

code/FP/ProgramExampleDynamicArrays.txt

# Structured data types

Record type are used to hold several data items (can be of different data types).

```
type
identifier = record
    field-1: type1;
    field-2: type2;
    ...
    field-n: typeN;
end;
```

”

# Structured data types

Members of a record must be initialized in the same order as they are declared.

```
type
 complex= record
      R,I: real;
 end;

const
      C1: complex = ( R:3; I: 1783.5 );
```

# Structured data types

## Example of using record:

```
1  program exRecords;
2  type
3  Books = record
4     title: packed array [1..50] of char;
5     author: packed array [1..50] of char;
6     subject: packed array [1..100] of char;
7     book_id: longint;
8  end;
9
10 var
11    Book1 : Books; (* Declare Book1 and Book2 of type Books *)
12
13 begin
14    (* book 1 specification *)
15    Book1.title   := 'C Programming';
16    Book1.author  := 'Nuha Ali ';
17    Book1.subject := 'C Programming Tutorial';
18    Book1.book_id := 6495407;
19
20    (* print Book1 info *)
21    writeln ('Book 1 title : ', Book1.title);
22    writeln('Book 1 author : ', Book1.author);
23    writeln;
24 end.
```

# With statement

**With statement** is an alternative way of accessing the members of a record-type instance.

```
With identifier do
begin
    member1:= ...
    member2:= ...
    member3:= ...
    ...
end;
```

”

# With statement

Example of using **with** statement:

```
(* UC1 specification *)
   UC1.title:= 'Informatica Industrial'
   UC1.year := 2020;
   UC1.class:= 'MG1';
   UC1.aval := 'continuous';

(* UC1 specification *)
With UC1 do
begin
   title:= 'Informatica Industrial'
   year := 2020;
   class:= 'MG1';
   aval := 'continuous';
end;
```

code/statementwith2.txt

# IF/ELSE statement

**IF/ELSE statement** evaluates an Boolean expression.

- If expression is True, the statement executes.
- If expression is False, then the code placed on "ELSE" statement is executed.

```
1  if BooleanExpression then
2  begin
3    Statement1;
4    Statement2;
5  end
6  else
7  begin
8    Statement3;
9    Statement4;
10 end;
```

# CASE statement

**CASE statement** is similar to the if-then-else statement but tests for equality against a list of values an Boolean expression.

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

  { like ELSE condition }
        otherwise Statement
end;
```

”

# FOR statement

**FOR statement** is a loop that needs to execute a specific number of times.

```
{  Counting UP }
for index := StartingLow to EndingHigh do
begin
        statement;
end;

{  Counting DOWN }
for index :=StartingLow downto EndingHigh  do
begin
        statement;
end;
```

# WHILE statement

**WHILE statement** is a loop continues to execute until the Boolean expression becomes FALSE.

```
while BooleanExpression do
        statement;

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

”

# Procedure

Procedure is a subprogram that helps to reduce the amount of redundancy in a program. These subprograms do not return a value directly.

```
procedure name(argumentA: type1; argumentB:
  type2; ... );
const
   (* Constants *)

var
   (* Variables *)

begin
   (* Statements *)
end;
```

# Function

Function is a block of code that always returns a single value.

```
1  Function Func_Name(params...) : Return_Type;
2  const
3    (* Constants *)
4
5  var
6    (* Variables *)
7
8  begin
9    (* Statements *)
10
11   { Returns a value }
12   Func_Name := ...
13 end;
```

# Parameter of procedure or function

The parameter list allows variable values to be transferred from the main program to the procedure.

Value parameter procedure/function gets a copy (value) of the parameters that the calling statement passes. Any modifications to these parameters are purely local.

Variable parameter procedure/function gets a pointer (reference) to the variable that was passed. Any changes made to the parameter will propagate back.

Out parameter to pass values back to the calling routine: the variable is passed by reference. The initial value of the parameter on function entry is discarded.

Const parameter when the contents of the parameter will not be changed by the called routine.

# Object

Object is like a Record but it adds *procedures* and *functions*. starts a block where it will be defined a set of variables.

```
type
 oldentifier = object
   private
     field1 : fieldType;
     field2 : fieldType;
   ...
   public
     procedure proc1;
     function f1(): functionType;
 end;

var objectvar : oldentifier
```

# Object

Free Pascal is an object-oriented programming language.

- **Member Variables** are variables defined inside an object/class;
- **Member Functions** are functions or procedures defined inside an object/class;
- **Visibility of Members** describes their accessibility:
  - *public* are always accessible from everywhere;
  - *private* can only be accessed in the module (e.g., program or unit) that contains the object definition;
  - *protected* the members are also accessible to descendent types.

# Object

- **Instantiation** is the creation of a variable of that class/object type;
- **Constructor** is a special type of function (called automatically with the instantiation of an object/class);
- **Destructor** is a special type of function (called automatically when an instance of an object/class is deleted).

# Example of object

```
1  program exObjects;
2  {Declaration of an Object}
3  type
4     Rectangle = object
5     private
6        length, width: integer;
7
8     public
9        procedure setlength(l: integer);
10       function getlength(): integer;
11       procedure setwidth(w: integer);
12       function getwidth(): integer;
13       procedure draw;
14 end;
15
16 { Declaration of Instances of the Rectangle}
17 var
18    r1: Rectangle;
19
20 { Definition of the Function Members}
21 procedure Rectangle.setlength(l: integer);
22 begin
23    length := l;
24 end;
25 procedure Rectangle.setwidth(w: integer);
```

# Example of object

```
1  function Rectangle.getlength(): integer;
2  ...
3  function Rectangle.getwidth(): integer;
4  begin
5     getwidth := width;
6  end;
7
8  procedure Rectangle.draw;
9  var
10    i, j: integer;
11 begin
12    for i:= 1 to length do
13    begin
14      for j:= 1 to width do
15         write(' * ');
16      writeln;
17    end;
18 end;
19
20 begin
21    r1.setlength(3); r1.setwidth(7);
22    writeln('Draw a rectangle:', r1.getlength(), ' by ', r1.getwidth());
23    r1.draw;
24 end.
```

code/FPProgramExampleObject2.txt

# Reference

- https://www.freepascal.org/docs-html/ref
- https://www.tutorialspoint.com/pascal
- https://wiki.freepascal.org/

# Questions / Dúvidas ?

Thank you!

Andry Maykol Pinto
amgp@fe.up.pt