Object pascal programming pdf

Inside a larger routine (function, procedure, method) you can determine the routine of assistants. The local routine is free to access (read and write) all parent settings, and all local parent variables that have been announced above it. It's very powerful. This often allows you to divide long routines into a couple of small ones without much effort (since you don't have to bypass all the necessary information in the settings). Be careful not to overuse this feature - if many nested features use (and even change) the same parent variable, the code can get difficult to follow. These two examples are equivalent: SumOfSquares (const N: Integer): Integer; Square feature (Const Value: Integer): Integer; Start The Result: Value and Value The end; var I: Integrator; Start Result: 0; For I :' 0 to N do Result : 'Result and square (I); The end; Another version where we allow the local usual area to access me directly: the procedure SumOfSquares (const N: Integer): Integer; var I: Integrator; Square: Integer Start The Result: Me and I; The end; Start Result: 0; For I :' 0 to N doing Result :' Result and square; The end; Local treatments can go anywhere deep - meaning you can identify local routines within another local routine. So you can go wild (but, Please don't get too wild, or the code will get unreadable:). They allow you to call the function indirectly $mode, to the variable. multiply (Const A, B: Integer): Integer; Start Result: End; TMyFunction type - function (const A, B: Integer): Integer; ProcessTheList (const F: TMyFunction): Inte Ger; var I: Integrator; Start Result : No 1; for I:' 2 to 10 Do Result: F (Result, I); End; var SomeFunction: TMyFunction; Start SomeFunction : @Add; Writeln ('1 No 2 No 3 ... SomeFunction :» @Multiply; Writeln ('1 No 2 No 3 ... The end. Method: Announce with the object at the end. $mode Delphi uses SysUtils; TMyMethod type - procedure (const A: Integer) of the object; TMyClass - CurrentValue class: Integer; Supplement (Const A: Integer); Multiplication procedure (Const A: Integer); ProcessTheList (Const M: TMyMethod); The end; TMyClass.Add (Const A: Integer); Start CurrentValue: Current Valyu and A; The end; TMyClass.Multiply (Const A: Integer); Start CurrentValue : CurrentValue - A; The end; TMyClass.ProcessTheList (Const M: TMyMethod); var I: Integrator; Start CurrentValue : No 1; For I : 2 to 10 do M(I); The end; var C: C :' TMyClass.Create; C.CurrentValue :' 1; C.ProcessTheList (@C.Add); Writeln ('1 No 2 x 3 ... ... ', C.CurrentValue); C.CurrentValue : 1; C.ProcessTheList (@C.Multiply); Writeln ('1 No 2 - 3 ... Finally FreeAndNil (C) end; The end. Note that you cannot undergo global procedures/functions as methods. They are incompatible. If you have to provide a callback object but don't want to create a bogus class instance, you can transfer class methods as methods. TMyMethod type - function (const A, B: Integer): Object integra; TMyClass - Adding (const A, B: Integer): Integer Multiply (const A, B: Integer): End of Integer; Var M: TMyMethod; Start M:' @TMyClass (zero). Add; M: @TMyClass (zero). Multiply; The end; Unfortunately, you need to write ugly @TMyClass (zero). Add, not @TMyClass.Add. A (possibly) localized routine: declare with the nested at the end, and make sure to use $modeswitch nestedproquars for the code. They go hand in hand with local (invested) routines. A powerful feature of any modern language. Determining something (usually a class) can be parameterized by a different type. The most typical example is when you need to create a container (list, dictionary, tree, graph...): You can identify a T type list and then specialize on it to instantly get a list of integers, a list of lines, a TMyRecord list, and so on. The generics in Pascal are realized in the same way as generics in the NHS. This means that they are expanded during specialization, a bit like macros (but much safer than macros; for example, identifiers are decided during a general definition rather than specialization, so you can't enter any unexpected behavior when specializing in general). In fact, this means that they are very fast (can be optimized for each particular type) and work with types of any size. You can use a primitive type (integer, float) as well as a recording, as well as a class when specializing in general. $mode Delphi uses SysUtils; TMyCalculator - Class Value: T; Add (const A: T); The end; TMyCalculator.Add (const A: T); Beginning Value : Value - A; The end; TMyFloatCalculator - TMyCalculator; var FloatCalc: TMyFloatCalculator; StringCalc: TMyStringCalculator; Start FloatCalc:' TMyFloatCalculator.Create; Try FloatCalc.Add (3.14); FloatCalc.Add(1); Writeln ('FloatCalc:', FloatCalc.Value:1:2); finally FreeAndNil (FloatCalc) end; StringCalc : TMyStringCalculator.Create; Try StringCalc.Add ('something'); StringCalc.Add ('more'); Writeln ('StringCalc: ', StringCalc.Value); finally FreeAndNil (StringCalc) end; The end. The general information is not limited to classes, you can also have common functions and procedures: $mode delphi uses SysUtils; Min!t.T'gt; B: T): T; начать, если < b= then= result= :=A else= result= :=B; end;= begin= writeln('min= (1,= 0):= ',=> <Integer>A (1, 0)); Writeln('Мин (3.14, 5): ',</Integer> </T> </string> </Single> </T> </T> 5):1:2); Writeln ('Min ('a', "b'): ', Min<string>('a', 'b')); The end. Methods (and global functions and procedures) with the same name are allowed if they have different parameters. During compilation, the compiler determines which one you want to use, knowing the parameters you're passing. By default, overload uses the FPC approach, which means that all methods in a given name space are equal, and hide other methods in name spaces with less priority. For example, if you define a class using Foo (Integer) and Foo (string) methods and it comes down from a Foo class (Float), then users in your new class won't be able to easily access the Foo (Float) method (they can still --- if they typecast the class of its ancestor type). To overcome this, use the keyword overload. You can use simple preprocessor directives for conditional compilation (code depending on the platform or some custom switches) to include one file in another, you can also use macros without parameters. Please note that macros with parameters are not allowed. In general, you should avoid using preprocessor things... if it's really justified. Pre-recycling takes place before parsing, which means you can break the normal syntax of Pascal's language. It's a powerful but also somewhat messy feature. ($mode Delphi) PreprocessorStuff unit; Foo interface ($ifdef) ($endif) const NewLine ($ifdef UNIX) #10 ($endif) ($ifdef MSWINDOWS) #13 10 ($endif) ; ($define MY_SYMBOL) ($ifdef MY_SYMBOL) Bar Procedure; ($endif) $define CallingConventionMacro : unknown ($ifdef UNIX) ($define CallingConventionMacro :' cdecl) ($endif) ($ifdef MSWINDOWS) ($define CallingConventionMacro :' stdcall) procedure $endif RealProcedureName; CallingConventionMacro; External some_external_library; ($include some_file.inc) ($I some_other_file.inc). Include files usually an extension of .inc, and are used for two purposes: the inclusion file may contain only other compiler directives that customize the source code. For example, you can create a myconfig.inc file with the content of $mode delphi ($modeswitch advancedrecords) ($ifndef VER3) $error This code can only be compiled using the FPC version of at least 3.x. ($endif) Now you can turn on this file using $I myconfig.inc in all your sources. Another common use is to divide a large unit into many files, while keeping one unit in terms of language rules. Don't overuse this technique - your first instinct is to split one unit into several units rather than split one unit into multiple files. However, this is a useful method. This avoids exploding the number of units while retaining your source files are short. For example, it might be better to have one unit with the normally used user interface. than to create one unit for each user interface management class, as the latter approach will make the typical position uses long (since typical user interface code will depend on multiple user interface classes). But placing all these UI classes in one myunit.pas file would make it a long file, a no-brainer, so dividing it into multiple to include files might make sense. This makes it easy to have a cross-platform interface with a platform implementation. Basically, you can do $ifdef UNIX ($I my_unix_implementation.inc) ($endif) ($ifdef MSWINDOWS) ($I my_windows_implementation.inc) ($endif) Sometimes it's better than writing long code with many UNIX $ifdef, $ifdef MSWINDOWS mixed with plain code (declarations, routine implementation). The code is more readable in this way. You can even use this method more aggressively using the FPC command line option to include some direction only for specific platforms. Then you can have many versions to include the file $I my platform_specific_implementation.inc and you just turn them on, allowing the compiler to find the right version. The recording is only a container for other variables. It's like a lot, a very simplistic class: no inheritance or virtual methods. It's like a structure in C-like languages. If you use the $modeswitch Advanced Records directive, records may have methods and visibility guidelines. In general, language functions that are available for class, and do not violate the simple predictable layout of memory recording, are then possible. ($mode Delphi) ($modeswitch advanced recordings) such as TMyRecord - recording of public I, Square: Integer; WriteInDescription procedure; The end; TMyRecord.WriteInDescription Start Writeln ('Square', I,' 'eat', square); The end; var A: TMyRecord array (0.9 euros; R: TMyRecord; I: Integrator; start for I:'0 to 9 start A'I. I: I am; A.I. Square: Me and I; The end; for R in do R.WriteInDescription; The end. In today's Pascal facility, your first instinct should be a design class, not a record - because the classes are packed with useful features like designers and legacy. But records are still very useful when you need a speed or a predictable memory layout: Records don't have a designer or a disruptor. You just identify the variable type of recording. It has uncertain content (memory debris) at the beginning (except for automatically controlled types such as strings; they are guaranteed to be initialized to be empty, and completed to release a number of links). So you have to be more careful when dealing with records, but this gives you some performance gains. The records arrays are well linear in memory, so they are cache friendly. Memory layout (size, upholstery between fields) is clearly defined in some situations: when requesting a C layout or using a packed recording. It's useful: to communicate with libraries written in programming languages, when they expose APIs based on records, read and write binary files to do dirty low-level tricks (such as unsafe typecasting one type to another, knowing about their memory presentation). Entries can also have body parts that work as unions in C-type languages. They allow you to treat the same piece of memory as another type, depending on your needs. This allows you to improve memory efficiency in some cases. And it allows for dirtier, lower-level unsafe tricks:) In the old days, Turbo Pascal introduced another syntax to the class as functionality using a keyword object. It's a kind of mixture between the concept of recording and modern class. Old-style objects can be highlighted/released, and during this operation you can call them a designer/destructor. But they can also simply be advertised and used as records. A simple record or type of object is not a reference (pointer) to something, it's just data. This makes them convenient for small data where call distribution/free will be annoying. Old-style objects offer inheritance and virtual methods, albeit with little difference from modern classes. Be careful - bad things will happen if you try to use an object without calling it a constructor, and the object has virtual methods. In most cases, it is not advisable to use old-style objects. Modern classes provide a lot more functionality. And if necessary, recordings (including extended recordings) can be used for performance. These concepts are usually better than the idea of old-style objects. You can create a pointer for any other type. The TMyRecord pointer is billed as TMyRecord, and by convention is called PMyRecord. This is a traditional example of a linked list of record integrators: PMyRecord and TMyRecord; TMyRecord - record value: Integer; Next: PMyRecord; The end; Note that the definition is recursive (PMyRecord is defined by the TMyRecord type, while TMyRecord is defined by PMyRecord). It is allowed to determine the type of pointer to an un-determined type if it is resolved within the same type of block. You can highlight and free pointers using new/dispose methods, or (lower-level, non-type-safe) GetMem/FreeMem methods. You dereference pointer (to access things pointed at) you are an app operator. To do the reverse operation, which is to get a pointer to the existing variable, you attach it to the operator. There is also an atypical type of pointer, similar to the void in C-like languages. It is completely unsafe, and can be typecasted to any other type of pointer. that instance class is also actually a pointer, although it doesn't require any operators to use it. A related list using classes, of course, perhaps it would be just like this: the type of TMyClass and the value of the class: Integer; Next: TMyClass; The end; You can override override the value of many language operators, for example, to allow the addition and multiplication of custom types. Here's how it is: $mode Delphi uses StrUtils; Operator (const S: string; const A: Integer): line; Start The Result:' DupeString (S, A); The end; Start Writeln ('bla' No 10); The end. You can override operators

on classes too. Because new class instances are usually created within the operator function, the caller must remember to release the result. $mode Delphi uses SysUtils; TMyClass class - MyInt class: Integer; The end; Operator (Const C1, C2: TMyClass): TMyClass; Start the result: TMyClass.Create; Result.MyInt : C1. MyInt and C2. MyInt; The end; var C1, C2: TMyClass; Start C1 : TMyClass.Create; Try C1. MyInt : No 12; C2 : C1 and C1; Try Writeln ('12 No 12', C2. MyInt); Finally FreeAndNil (C2) end; Finally FreeAndNil (C1) end; The end. You can override the operators on the record too. This is usually easier than overloading them for classes, as the subscriber does not have to deal with memory management. $mode Delphi uses SysUtils; TMyRecord - MyInt: Integer The end; Operator (Const C1, C2: TMyRecord): TMyRecord; start Result.MyInt:' C1. MyInt and C2. MyInt; The end; var R1, R2: TMyRecord; start R1. MyInt : No 12; R2 : R1 and R1; Writeln ('12 No 12 ', R2. MyInt); The end. For entries, this tips use $modeswitch advancedrecords and override operators as a class operator inside the recording (for delphi mode available by design). This allows you to use common classes that depend on the existence of some operators (such as TFPGList, which depends on the equality of the operator available) with such records. Otherwise, the global definition of the operator (not inside the record) will not be found (because it is not available in the code that TFPGList implements), and you cannot specialize in a list like TFPGList'lt;TMyRecord. Note to TList from the Generics.Collections module you don't need to override operators. ($mode Delphi) ($modeswitch advancedrecords) uses SysUtils, FGL; TMyRecord - MyInt: Integer Class Operator (Const C1, C2: TMyRecord): TMyRecord; Class operator (Const C1, C2: TMyRecord): boolean; The end; TMyRecord class operator. (const C1, C2: TMyRecord): TMyRecord; start Result.MyInt:' C1. MyInt and C2. MyInt; The end; TMyRecord class. (const C1, C2: TMyRecord): boolean; Start The Result:' C1. MyInt and C2. MyInt; The end; TMyRecordList Type - TFPGList'lt;TMyRecord var R, ListItem: TMyRecord; L: TMyRecordList; Start L:' TMyRecordList.Create; Try R.MyInt :1; L.Add (R); R.Myint: 10; L.Add (R); R.Myint: 100; L.Add (R); R.Myint : 0; for ListItem in L-do R : Writeln ('1 Nos 10 and 100', R.MyInt); Finally FreeAndNil (L) end; The end. The end. </TMyRecord></TMyRecord> object pascal programming language. object pascal programming tutorial. delphi/object pascal programming language. pascal object oriented programming. object oriented pascal programming language. start programming using object pascal. free pascal object oriented programming. turbo pascal 5.5 object-oriented programming guide