Pascal Expert

Pascal Expert

---

by Peganza

Pascal Expert

Table of Contents

1 Introduction

Pascal Expert is a plug-in for Embarcadero's Delphi IDE (RAD Studio). Pascal Expert main task is to make a static code analysis. It only needs the source code, unlike other similar tools that perform an analysis of the running program. Pascal Expert will help you better understand your code and support you in producing code of higher quality, consistency, and reliability. It will point out possible issues and errors in your code.

Pascal Expert is a subset of our standalone static code analyzer Pascal Analyzer. Pascal Expert displays the same results as Pascal Analyzer, but integrated in the Delphi IDE, which makes it an ideal tool when working with code, as it lets you find problems earlier, and fix them at once.

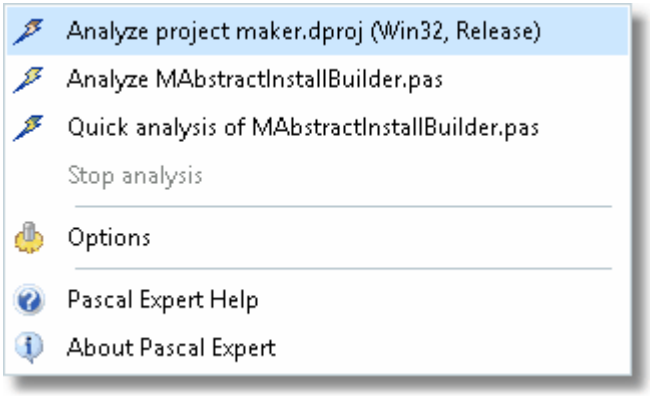
If you want to also buy the full-fledged Pascal Analyzer, you will find favorable pricing. Similarly, if you already use Pascal Analyzer, you will get a very large discount when buying Pascal Expert. See our web site for more details.

Pascal Expert and Pascal Analyzer quickly pay themselves back in easier maintenance, less errors and improved quality, not only during development, but also throughout the entire life cycle of your code.

Pascal Expert can be installed for these Delphi IDE versions:

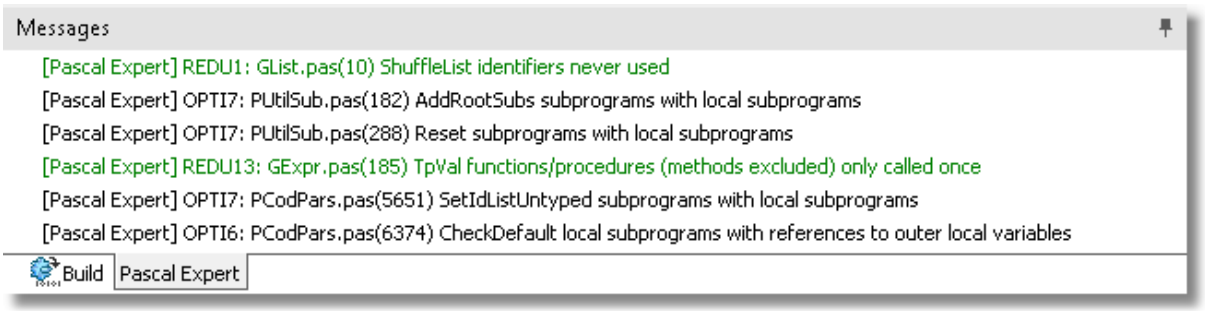
- Delphi 12 Athens
- Delphi 11 Alexandria
- Delphi 10.4 Sydney
- Delphi 10.3 Rio
- Delphi 10.2 Tokyo
- Delphi 10.1 Berlin
- Delphi 10 Seattle
- Delphi XE8
- Delphi XE7
- Delphi XE6
- Delphi XE5
- Delphi XE4
- Delphi XE3
- Delphi XE2
- Delphi XE
- Delphi 2010
- Delphi 2009
- Delphi 2007

Menu selections for Pascal Expert can be found in the Tools menu:



You can choose between parsing the entire project, or just the selected module. There is also a quick analysis option for the selected module, which is fast, but does not yield complete results.

Pascal Expert parses your source code in the same way as the compiler. The results are displayed as messages on a tab page in the Output window in the RAD Studio IDE. This tab page is normally located at the bottom of the window.



For each line Pascal Expert reports, it will display:

- optional "[Pascal Expert]" label to the very left of each line
- optional prefix, referring to the original report/section identifier in Pascal Analyzer, like **REDU1**
- unit and line number, like **GList.pas(10)**
- description of the issue

Double-click on a line to open the location in the editor. Or press **F1** to bring up the relevant topic from the help system.

If you want to copy the results to Windows clipboard, just press **Ctrl+A** (with the output tab focused) to select all messages, and then **Ctrl+C** to copy to your target.

The message display can be customized. You can select font and background/foreground colors for each category. See Options.

To analyze a particular set of source code with Pascal Expert, just make sure that a project is active. Then select Analyze project from the Pascal Expert menu, and Pascal

Expert will analyze the currently selected project and active configuration. If you want to analyze only the current editor module, select Analyze module. For a quick analysis of the current module, select Quick analysis of module.

Pascal Expert will do its work in the background, so it will be possible to continue working on other tasks. To interrupt the process, for any reason, just click Stop from the Pascal Expert menu. Results will be output to the tab page, as described above.

There is a special installation program for Pascal Expert. Use it to install Pascal Expert for the versions of Delphi that you wish. Rerun the installation if you want to add or remove Pascal Expert from a Delphi version.

You must restart Delphi after installing Pascal Expert. Pascal Expert will be available as a menu item to the right of "Tools", or as the first submenu below "Tools" in the main menu.

For licensing details, support plans etc., see our web site for more details.

Pascal Expert is a subset of Pascal Analyzer, our standalone code analysis product. Pascal Analyzer, released in 2001, is now at version 9. The two products share a lot of common code and features. This means also, that when an update is made for Pascal Analyzer, an update is also made for Pascal Expert, and vice versa.

Special thanks to

- **Borland**, for giving us Delphi, the most productive programming environment ever
- **Embarcadero/CodeGear**, for continuing Borland's work
- **Inno Setup** (https://jrsoftware.org/isinfo.php), a great utility to create powerful installation programs, we use it for all our applications

See also:

Known limitations
Main menu

This documentation was last updated May 8, 2025.

Web site: https://peganza.com
Email: support@peganza.com

2 Known limitations

There are situations that Pascal Expert currently cannot handle very well. Some of these limitations, but certainly not all, are:

1. Objects that are created through a class reference cannot always be resolved. The reason for this is that the actual class used is determined at runtime.

Example:

```
type
  TMyClassRef = class of TMyClass;

  TMyClass = class
  ..
    procedure Method; virtual;
  end;

  TMyDerivedClass = class(TMyClass)
  ..
    procedure Method; override;
  end;

..

procedure Proc(C : TMyClassRef);
begin
  C := TMyClassRef.Create; // TMyClass or TMyDerivedClass?
  C.Method; // TMyClass.Method or TMyDerivedClass.Method?
  ..
end;
```

2. Methods that are marked as abstract in a base class and used in that class, cannot be resolved:

Example:

```
type
  TBase = class
  ..
    procedure Main;
    procedure Proc; virtual; abstract;
  end;

  TDesc = class(TBase)
  ..
    procedure Proc; overload;
  end;

  TAnotherDesc = class(TBase)
  ..
    procedure Proc; overload;
  end;

  procedure TBase.Main;
  begin
    ..
    Proc; // which one, TDesc.Proc or TAnotherDesc.Proc?
  end;
```

The actual usage of Proc is determined at runtime.

3. Assert calls are not excluded from the parsing process, unlike in Delphi, regardless if the $C- setting is active or not. This means that identifiers used in the Assert procedure call, will be registered, and appear in the reports.

Example:

```
procedure MyProc(P : Pointer);
begin
  Assert(P <> nil);
  ..
end;
```

The parameter P will be registered and appear in the reports. When compiled by Delphi, this code line will be stripped out if $C- is defined.

See also:

Introduction
Main menu

3 Installation folders

Pascal Expert is installed with its special installation program. Depending on the Delphi IDE you select to support, different files are installed:

C:\Program Files (x86)\Peganza\Pascal Expert

This folder contains executable files, help files etc. Pascal Expert is 32-bits only, because Delphi IDE's are all 32-bits at this time.

C:\Documents and Settings\<acc>\Application Data\Peganza\Pascal Expert

This is where Pascal Expert INI files are stored. The INI file contains settings for your Pascal Expert installation.

Please note that a separate INI file is kept for each Delphi version, like PEX12.INI for Delphi 12, PEXXE8.INI for Delphi XE8 etc.

In this folder, you will also find error log files, if an unexpected error occurs. You should send those files to support@peganza.com together with a description about how the error occurred.

C:\Documents and Settings\<acc>\My Documents\Peganza\Pascal Expert

This is where you can save INI files in the Options dialog.

C:\Documents and Settings\<acc>\My Documents\Pascal Expert\Samples

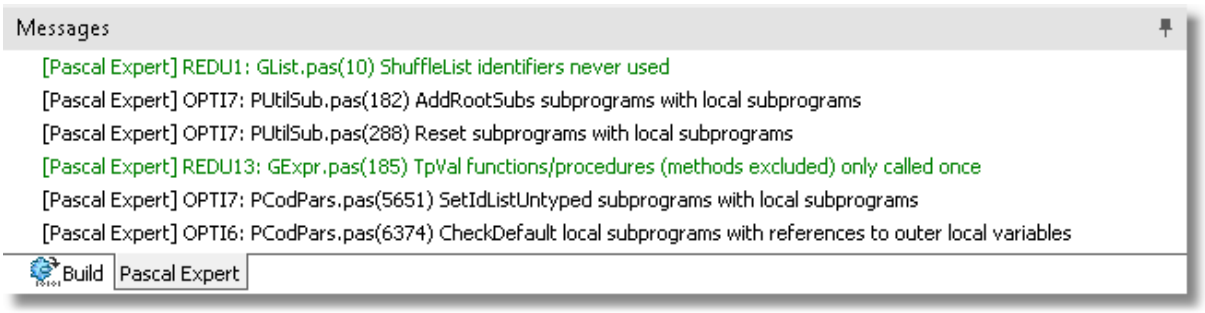
This folder contains sample source code files (*.dpr, *.pas) that illustrate various selected report sections.
Load Samples.dpr into the IDE, and do an analysis.

See also:

Introduction
Known limitations
Main menu

4 Reports

Pascal Expert outputs results in a special tab page of the Output window in the IDE:



There are four types of reports:

Alerts
Reductions
Optimizations
Conventions

You can customize the appearance of messages for the different report types, in the settings dialog.

For example, you can select to prefix each report section with an abbreviation, like "WARN12".
This uniquely identifies the report section.

See also:

Introduction
Installation folders
Known limitations
Main menu

4.1 Alerts

The Alert report area contains sections from several Pascal Analyzer reports:

- Strong Warnings (STWA1-STWA12)
- Warnings (WARN1-WARN62)
- Memory (MEMO1-MEMO7)
- Control Warnings (COWA1-COWA8)

STWA1-Property access in read/write methods
STWA2-Ambiguous unit references
STWA3-Subprogram calls itself unconditionally
STWA4-Index error
STWA5-Possible bad pointer usage
STWA6-Possible bad typecast (for objects: consider using "as")
STWA7-For-loop with possible bad condition
STWA8-Bad parameter usage
STWA9-Generic interface has GUID
STWA10-Interface lacks GUID
STWA11-Duplicated GUID
STWA12-Equal if-then and if-else statements

WARN1-Interfaced identifiers that are used, but not outside of unit
WARN2-Interfaced class identifiers that are public/published, but not used outside of unit
WARN3-Variables that are referenced, but never set
WARN4-Variables that are referenced, but possibly never set (ref/set by unknown subprograms)
WARN5-Variables that are set, but never referenced

WARN6-Variables that are set, but possibly never referenced (ref/set by unknown subprograms)
WARN7-Local variables that are referenced before they are set
WARN8-Local variables that may be referenced by unknown subprogram before they are set
WARN9-Var parameters that are used, but never set
WARN10-Var parameters that are used, but possibly never set (ref/set by unknown subprograms

WARN11-Value parameters that are set
WARN12-Value parameters that are possibly set (ref/set by unknown subprogram)
WARN13-Interfaces passed as parameters without "const" directive
WARN14-Variables with absolute directive
WARN15-Constructors/destructors without calls to inherited

WARN16-Destructors without override directive
WARN17-Classes with more than one destructor
WARN18-Function result not set
WARN19-Recursive subprograms
WARN20-Dangerous Exit-statements

WARN21-Dangerous Raise
WARN22-Dangerous Label-locations inside for-loops
WARN23-Dangerous Label-locations inside repeat/while-loops
WARN24-Possible bad object creation
WARN25-Bad thread-local variables

WARN26-Instance created of class with abstract methods
WARN27-Empty code blocks and short-circuited statements
WARN28-Empty case labels
WARN29-Short-circuited for-statements
WARN30-Short-circuited if/case-statements

WARN31-Short-circuited on-statements
WARN32-Short-circuited repeat-statements
WARN33-Short-circuited while-statements
WARN34-Empty except-block
WARN35-Empty finally-block

WARN36-Forward directive in interface
WARN37-Empty subprogram parameter list
WARN38-Ambiguous references in with-blocks
WARN39-Classes without overrides of abstract methods
WARN40-Local for-loop variables read after loop

WARN41-Local for-loop variables possibly read after loop
WARN42-For-loop variables not used in loop
WARN43-Non-public constructors/destructors
WARN44-Functions called as procedures
WARN45-Mismatch property read/write specifiers

WARN46-Local variables that are set but not later used
WARN47-Duplicate lines
WARN48-Duplicate class types in except-block
WARN49-Redeclared identifiers from System unit
WARN50-Identifier with same name as keyword/directive

WARN51-Out parameter is read before set
WARN52-Possible bad assignment
WARN53-Mixing interface variables and objects
WARN54-Set before passed as out parameter
WARN55-Redeclares ancestor member, or method in helped class/record

WARN56-Parameter to FreeAndNil is not an object
WARN57-Enumerated constant missing in case structure
WARN58-Mixed operator precedence levels
WARN59-Explicit float comparison
WARN60-Condition evaluates to constant value
WARN61-Assigned to itself
WARN62-Possible orphan event handler
WARN63-Mismatch 32/64-bits

MEMO1-Local objects with unprotected calls to Free
MEMO2-Non-local objects with unprotected calls to Free
MEMO3-Objects created in try-structure
MEMO4-Unbalanced Create/Free
MEMO5-Local objects that are created more than once without being freed in-between

MEMO6-Local objects that are referenced before being created
MEMO7-Local objects that are referenced after being freed

COWA1-Controls that overlap visually
COWA2-Labels with Caption-property that does not end in ":"
COWA3-Conflicting accelerators
COWA4-Labels (or static texts) that have accelerators but FocusControl is not set□

COWA5-Conflicting shortcuts

COWA6-Buttons/menu items with OnClick-event that is unassigned
COWA7-Menu items that have HelpContext=0
COWA8-Hint is not activated

See also:

Reports
Conventions
Optimizations
Reductions
Conventions

4.1.1 STWA1-Property access in read/write methods

Property access in read/write methods (STWA1)

This section reports locations where properties are referenced in read/write methods, like for example:

```
..
property MyProp : Integer read GetMyProp write SetMyProp;
..

function TMyClass.GetMyProp : Integer;
begin
  Result := MyProp; // error, correct is: Result := FMyProp;
end;

procedure TMyClass.SetMyProp(Value : Integer);
begin
  MyProp := Value; // error, correct is: FMyProp := Value;
end;

```

These sorts of errors can cause infinite recursion.

See also:

Alerts

4.1.2 STWA2-Ambiguous unit references

Ambiguous unit references (STWA2)

This sections lists identifiers with ambiguous unit references.
Consider this example:

```
program MyProg;

uses
  A, B;

begin
  writeln('Value='+TheValue);
end.

unit A;

interface

const
  TheValue = 'Hello';

implementation

end.

unit B;

interface

const
  TheValue = 'Goodbye';

implementation

end.
```

What will be the output from the program? In this case, it will be "Goodbye", because the last unit listed in the uses clause will have precedence.

The reference to TheValue is ambiguous or unclear, so it will be listed in this report section. Consider what happens if originally only unit "A" was listed in the uses clause. Then the output would be "Hello". If then maybe another programmer without any sense of danger will add "B" to the uses clause, the output will be changed.

You should prefix the reference, like "B.TheValue", to avoid any uncertainty.

See also:

Alerts

4.1.3 STWA3-Subprogram calls itself unconditionally

Subprogram calls itself unconditionally (STWA3)

This sections lists subprograms that call themselves unconditionally. This will lead to infinite recursion and stack failure at runtime if the subprogram is called:

Consider this example:

```
1  procedure Proc;
2  begin
3    ..
4    Proc;
5    ..
6  end.
7
```

See also:

Alerts

4.1.4 STWA4-Index error

Index error (STWA4)

This sections lists locations in your code with an index error.

Example:

```
1  procedure Proc;
2  var
3    X : Integer;
4    Arr : array[0..1] of Integer;
5  begin
6    X := 555;
7    ..
8    Arr[X-2] := 0; // index error
9  end;
```

If the code had been instead "Arr[553]" (an explicit value), the compiler would have halted on this line. But for a variable, it does not.

See also:

Alerts

4.1.5 STWA5-Possible bad pointer usage

Possible bad pointer usage (STWA5)

This section lists locations in your code where a pointer possibly is misused. For example a pointer that has been set to nil and further down in the code is dereferenced.

Example:

```
type
  TMyClass = class
    procedure MyProc;
  end;

procedure TMyClass.MyProc;
begin
  ..
end;

procedure Proc;
var
  Obj : TMyClass;
begin
  Obj := TMyClass.Create;
  ..
  Obj := nil;
  ..
  Obj.MyProc; // error, Obj is nil here
end;
```

See also:

Alerts

4.1.6 STWA6-Possible bad typecast (for objects: consider using "as")

Possible bad typecast (for objects: consider using "as") (STWA6)

This section lists locations in your code with a possibly bad typecast. If you use the "as" operator, an exception will instead be raised. Otherwise there may be access violations and errors in a totally different code location, which is not very easy to track down.

Example:

```
type
  TFruit = class
    ..
  end;

  TAnimal = class
    ..
  end;

procedure Proc;
var
  Monkey : TAnimal;
  Banana : TFruit;
begin
  ..
  Monkey := TAnimal(Banana); // bad typecast, a fruit cannot be typecast to an animal
end;
```

In the example above, the last line could better be written (although still faulty!) as

Monkey := Banana as TAnimal;

This should result in an exception. But this is still preferable; instead of letting the code proceed resulting maybe in access violations later in a totally unrelated part of the code.

Also situations where a "bigger" type is typecast to a "smaller", will trigger a warning. For example "Ch := Char(I)" where Ch is of type **Char** and I is of type **Integer**. This may of course be totally valid if you make sure that I is not too big.

When a smaller variable is typecast to a Pointer, there will also be a warning. For example "Pointer(I)" when I is an Integer, and Pointer is 64-bits.

See also:

Alerts

4.1.7 STWA7-For-loop with possible bad condition

For-loop with possible bad condition (STWA7)

This section lists locations in your code where for loop has any of these conditions:

```
..

for I := 0 to SL.Count do // SL.Count-1 intended?
begin
..
end;

for I := 1 to SL.Count-1 do // SL.Count intended?
begin
end;

..

```

See also:

Alerts

4.1.8 STWA8-Bad parameter usage

Bad parameter usage (same identifier used for different parameter) (STWA8)

This section lists locations in your code where a call to a subprogram is made with bad parameters. The situation occurs when the called subprogram has an "out" parameter plus at least one another parameter. The identifier passed is used for both these parameters. Because an "out"-parameter is cleared in the called subprogram this will give unexpected results for reference-counted variables like strings and dynamic arrays.

```
..

procedure Proc(const Value : string; out ReturnValue : string);
begin
  ReturnValue := '555'+Value;
end;

procedure Caller;
var
  S : string;
begin
  S := '111';
  Proc(S, S); // S will have '555' upon return, not '555111' which you would expect
end;

..
```

See also:

Alerts

4.1.9 STWA9-Generic interface has GUID

Generic interface has GUID (STWA9)

This section lists generic interface types that declare a GUID:

```
..

type
  IMyInterface<t> = interface
  ['{C9BC756B-6B30-4C44-B237-552AFBA5697C}']
    ..
  end;

..
```

The problem with this is that all generic types created from this interface, like IMyInterface<Integer> and IMyInterface<string> will share the same GUID. This will cause type casting to malfunction.

See also:

Alerts

4.1.10 STWA10-Interface lacks GUID

Interface lacks GUID (STWA10)

This section lists interface types that lack a GUID.

See also:

Alerts

4.1.11 STWA11-Duplicated GUID

Duplicated GUID (STWA11)

This section lists code locations with duplicated GUID.

See also:

Alerts

4.1.12 STWA12-Equal if-then and if-else statements

Equal if-then and if-else statements (STWA12)

This section reports if-structure with identical then- and else-statements.

If the statements only differ in case for literal strings, like "hello world" in if-then and "Hello World" in else-then, they are considered to be different, and thus does not trigger a warning.

See also:

Alerts

4.1.13 WARN1-Interfaced identifiers that are used, but not outside of unit

Interfaced identifiers that are used, but not outside of unit (WARN1)

This section lists all identifiers that are declared in the *interface* section of a unit, and that are used in the unit, but not outside the unit. You should declare these identifiers in the *implementation* section of the unit instead.

Restrictions:
Interfaced identifiers that are not used at all are not listed. These identifiers are already listed in the "Identifiers never used" section among the Reductions.

Recommendation:
Declare these identifiers in the *implementation* section of the unit, to avoid unnecessary exposure.

See also:

Alerts

4.1.14 WARN2-Interfaced class identifiers that are public/published, but not used outside of unit

Interfaced class identifiers that are public/published, but not used outside of unit (WARN2)

This section lists all identifiers that are members of a class, and are declared with the *public/published* directive, but not used outside of the unit.

Recommendation:

Declare these identifiers with the *private/protected* directive instead.

See also:

Alerts

4.1.15 WARN3-Variables that are referenced, but never set

Variables that are referenced, but never set (WARN3)

This section lists all declared and referenced variables that never are set. Possibly this is an error, but the reason could also be that the variable is set in code that is not seen by the parser.

Restrictions:
Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, and are changed whenever the other variable changes.

Recommendation:
Examine why these variables are referenced, but never set. False warnings may be generated in some cases for null-terminated strings, where the actual pointer (PChar) is not set, but when the contents of the buffer pointed to is indeed changed.

See also:

Alerts

4.1.16 WARN4-Variables that are referenced, but possibly never set (ref/set by unknown subprograms)

Variables that are referenced, but possibly never set (ref/set by unknown subprograms) (WARN4)

This section lists all variables that are declared and referenced but never set. They are referenced in unknown fashion, and the parser is unable to determine whether they are set or just referenced in these locations.

Restrictions:
Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, and are changed whenever the other variable changes.

Recommendation:
Examine why these variables are referenced, but never set. False warnings may be generated in some cases for null-terminated strings, where the actual pointer (PChar) is not set, but when the contents of the buffer pointed to is indeed changed.

See also:

Alerts

4.1.17 WARN5-Variables that are set, but never referenced

Variables that are set, but never referenced (WARN5)

This is a list of all variables that are set but never referenced. Either these variables are unnecessary or something is missing in the code, because it is meaningless to set a variable and then never reference, or use it.

Restrictions:
Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, and are changed whenever the other variable changes.

Recommendation:
Examine why these variables are set, but never referenced.

See also:

Alerts

4.1.18 WARN6-Variables that are set, but possibly never referenced (ref/set by unknown subprograms)

Variables that are set, but possibly never referenced (ref/set by unknown subprograms) (WARN6)

This is a list of all variables that are set but never referenced. The variables are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. They are either unnecessary or something is missing in the code, because it is meaningless to set a variable and then never reference, or use it.

Restrictions:
Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, they are changed whenever the other variable changes.

Recommendation:
Examine why these variables are set, but never referenced. Also, try to make more source code available to PAL.

See also:

Alerts

4.1.19 WARN7-Local variables that are referenced before they are set

Local variables that are referenced before they are set (WARN7)

This is a list of all local variables that are referenced before they are set. Probably this is an error, because the values of these identifiers are undefined before they are set. An exception is long strings that are not examined, because they are implicitly initialized upon creation.

Restrictions:
Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, they are changed whenever the other variable changes.

Recommendation:
Examine why these variables are referenced before they are set.

Example:

```
1  procedure MyProc;
2  var
3    I : integer;
4  begin
5    if I = 55 then // !! I is undefined
6    begin
7      ..
8    end;
9
10   I := 55;
11   ..
12 end;
```

Pascal Analyzer also examines local subprograms that are called. Consider this scenario:

Example:

```
1  procedure Proc;
2  var
3    I : integer;
4
5    procedure InnerProc;
6    begin
7      if Condition then
8        if I < 5 then
9      ..
10   end;
11
12 begin
13   InnerProc;
14   ..
15   I := 55;
16 end;
```

This code triggers a warning, because the local variable I is first referenced by InnerProc. The call to InnerProc occurs before I is set in the main body of Proc. Even if I is only referenced when Condition evaluates to True (in InnerProc), this must happen at some occasion, otherwise that check would be pointless.

A usual situation which triggers this warning is when an non-initialized variable is passed as a parameter to a function. The function signature declares the parameter as a var-parameter. Changing the parameter to an out-parameter (if possible), avoids this warning.

See also:

Alerts

4.1.20 WARN8-Local variables that may be referenced by unknown subprogram before they are set

Local variables that may be referenced by unknown subprogram before they are set (WARN8)

This is a list of all local variables that are referenced before they are set. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. Probably this is an error because the values of these identifiers are undefined before they are set. An exception is long strings that are not examined, because they are implicitly initialized to empty strings when created.

Restrictions:
Variables marked with the *absolute* directive are not examined. These identifiers shadow another variable in memory, they are changed whenever the other variable changes.

Example:

```
1  procedure MyProc;
2  var
3    I : Integer;
4  begin
5    UnknownProc(I);
6
7    if I = 55 then // !! I may be undefined
8    begin
9    ..
10   end;
11
12   I := 55;
13   ..
14 end;
```

See also:

Alerts

4.1.21 WARN9-Var parameters that are used, but never set

Var parameters that are used, but never set (WARN9)

This is a list of all *var* parameters that are used but never set in the subprogram they belong to. Although this is not an error, it may be an indication that something is wrong with your code. Otherwise, you may omit the *var* keyword, or change it to a *const* parameter.

Example:

```
1  procedure MyProc(var I : Integer);
2  begin
3    ..
4    if I = 5 then // !! I is not set
5    begin
6    ..
7    end;
8    ..
9  end;
```

Restrictions:
Parameters to event handlers are not reported.

See also:

Alerts

4.1.22 WARN10-Var parameters that are used, but possibly never set (ref/set by unknown subprograms

Var parameters that are used, but possibly never set (ref/set by unknown subprograms (WARN10)

This is a list of all *var* parameters that are used but never set in the subprogram they belong to. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. Although this is not an error, it may be an indication that something is wrong with your code. Otherwise, you may omit the *var* keyword, or change it to a *const* parameter.

Example:

```
procedure MyProc(var I : Integer);
begin
..
  UnknownProc(I);
..
  if I = 5 then // !! I may not be set
  begin
  ..
  end;
..
end;
```

Restrictions:
Parameters to event handlers are not reported.

See also:

Alerts

4.1.23 WARN11-Value parameters that are set

Value parameters that are set (WARN11)

This is a list of all value parameters that are set in the subprogram they belong to. Although this is permitted by the compiler, it may not be what you intended. If you want to really change the variable, use the *var* directive instead.

Example:

```
procedure MyProc(I : Integer);
begin
..
  I := 5; // !! I is changed
..
end;
```

See also:

Alerts

4.1.24 WARN12-Value parameters that are possibly set (ref/set by unknown subprogram)

Value parameters that are possibly set (ref/set by unknown subprogram) (WARN12)

This is a list of all value parameters that are set in the subprogram they belong to. They

are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. Although this is permitted by the compiler, it may not be what you intended. If your intention is to really change the variable, use the *var* directive instead.

Example:

```
procedure MyProc(I : Integer);
begin
..
  UnknownProc(I); // !! I may be changed
..
end;
```

See also:

Alerts

4.1.25 WARN13-Interfaces passed as parameters without "const" directive

Interfaces passed as parameters without "const" directive (WARN13)

This is a list of all parameters that are of interface type and passed without "const" directive.

See also:

Alerts

4.1.26 WARN14-Variables with absolute directive

Variables with absolute directive (WARN14)

This is a list of all variables that are declared with the *absolute* directive keyword. You should watch these variables carefully, since they may potentially overwrite memory.

Example:

```
procedure MyProc;
var
  I : Byte;
  K : Integer absolute I;
begin
..
  K := MaxInt; // !! I is overwritten
end;
```

Recommendation:

Examine absolute variables carefully, and make sure that they do not overwrite memory.

See also:

Alerts

4.1.27 WARN15-Constructors/destructors without calls to inherited

Constructors/destructors without calls to inherited (WARN15)

This is a list of all constructors and destructors that never call their inherited constructor/destructor. This call is often required. so that the object can be correctly created or destroyed. For a class descending directly from *TObject*, the inherited call in the constructor is not needed, since the constructor in *TObject* does not actually do anything. There is no guarantee though that the constructor will be empty in future versions. If the constructor/destructor does not call inherited itself, but calls another constructor/destructor that calls inherited, there will be no warning.

Recommendation:
Call the inherited constructor as the first statement in the constructor, and as the last statement in the destructor.

See also:

Alerts

4.1.28 WARN16-Destructors without override directive

Destructors without override directive (WARN16)

This is a list of all destructors that miss the *override* directive keyword. Normally this directive must be set, or a call to the *Free* method would never be successful. This is because *Free* calls the destructor.

Limitation:
Old-style objects are never reported, because in this case the *override* keyword is not allowed.

See also:

Alerts

4.1.29 WARN17-Classes with more than one destructor

Classes with more than one destructor (WARN17)

This is a list of all classes that have more than one destructor declared. To declare more than one destructor is usually pointless and should be avoided.

See also:

Alerts

4.1.30 WARN18-Function result not set

Function result not set (WARN18)

This is a list of all functions where the result value is not always set. It may be set for some but not all possible code paths. Although this is acceptable for the compiler, it implies an error in the code. Maybe the function could be implemented as a procedure instead, if the result value is not needed.

Functions that return long strings are not examined. Those strings are zero-initialized by the function.

Recommendation:
Check these functions and examine if they should be implemented as procedures instead.

See also:

Alerts

4.1.31 WARN19-Recursive subprograms

Recursive subprograms (WARN19)

This is a list of all subprograms (procedures and functions) that are recursive (call themselves). Recursive subprograms are difficult to implement, and should be given extra attention.

Recommendation:
Check these subprograms and make sure that they cannot fall into infinite recursion.

See also:

Alerts

4.1.32 WARN20-Dangerous Exit-statements

Dangerous Exit-statements (WARN20)

This is a list of all locations with dangerous *Exit*-statements. These *Exit*-statements may

leave a whole block of code that is never executed (dead code). Every unconditional (not within an if-statement) *Exit*-statement is considered dangerous in this respect. Exit-statements within except-blocks are considered as safe, however.

There are situations when a developer inserts Exit-commands just for testing purposes, for example to quit a function without executing a block of code. This report section catches those locations where the Exit-commands have not been removed.

Example:

```
1  procedure MyProc;
2  begin
3  ..
4    Exit;
5
6    Proc(X); // !! never executed
7  ..
8  end;
```

See also:

Alerts

4.1.33 WARN21-Dangerous Raise

Dangerous Raise (WARN21)

This is a list of all locations with dangerous raise commands. These raise-commands may leave a whole block of code that is never executed (dead code). Every unconditional (not within an if-statement) raise-command is considered dangerous in this respect. Raise-commands within except-blocks are considered as safe, however.

There are situations when a developer inserts raise-commands just for testing purposes, for example to quit a function without executing a block of code. This report section catches those locations where raise-commands have not been removed.

Example:

```
1  procedure MyProc;
2  begin
3  ..
4    raise Exception.Create('Leave here');
5
6    Proc(X); // !! never executed
7  ..
8  end;
```

See also:

Alerts

4.1.34 WARN22-Dangerous Label-locations inside for-loops

Dangerous Label-locations inside for-loops (WARN22)

This is a list of all locations with dangerous *goto*-labels. These labels are located inside *for* -loops. In the case of a *for*-loop, this is especially dangerous, since the loop variable will have an undefined value.

Example:

```
1  procedure MyProc;
2  label
3    MyLabel;
4
5  begin
6  ..
7    for I := 0 to NumItems-1 do
8    begin
9    ..
10 MyLabel:
11   ..
12   end;
13   ..
14   if Cond then
15     goto MyLabel; // !! dangerous
16   ..
17 end;
```

See also:

Alerts

4.1.35 WARN23-Dangerous Label-locations inside repeat/while-loops

Dangerous Label-locations inside repeat/while-loops (WARN23)

This is a list of all locations with dangerous *goto*-labels. These labels are located inside *repeat*/*while*-loops. If the loop counter is considered, this may work just fine, but these labels should be given extra attention.

Example:

```
procedure MyProc;
label
  MyLabel;

begin
..
  while I < NumItems-1 do
  begin
  ..
MyLabel:
  ..
  end;
  ..
  if Cond then
    goto MyLabel; // !! dangerous
  ..
end;
```

See also:

Alerts

4.1.36 WARN24-Possible bad object creation

Possible bad object creation (WARN24)

This is a list of all locations in the code where an object possibly is created in a bad fashion.

Example:

```
procedure Proc;
var
  Bmp : TBitmap;

begin
  ..
  Bmp.Create // !! Bmp := TBitmap.Create
  ..
end;
```

This is an error!

Example:

```
procedure Proc2;
begin
  TNode.Create(Parent);
end;
```

PAL reports this as an error, since the reference to the new object is not assigned to a

variable. It could possibly be a mistake. However, in a situation where the object is inserted into a list managed by "Parent", it is not a mistake. This is the case for the common TTreeView control.

Limitation:
Old-style objects are never reported, because in this case the *override* keyword is not allowed.

See also:

Alerts

4.1.37 WARN25-Bad thread-local variables

Bad-thread local variable (WARN25)

This is a list of all thread-local variables (declared with the "threadvar" keyword) with bad declarations. Reference-counted variables (such as long strings, dynamic arrays, or interfaces) are not thread-safe and should not be declared with "threadvar". Also, do not create pointer- or procedural-type thread variables.

Example:

```
threadvar
  S : string;           // !! these are all bad thread-local variables
  P : Pointer;
  R : array of Integer;
  X : IMyInterface;
  W : TProcedure;
```

See also:

Alerts

4.1.38 WARN26-Instance created of class with abstract methods

Instance created of class with abstract methods (WARN26)

This is a list of all locations where instances of classes with abstract methods are created. Such classes should serve as ancestor classes only.

Example:

```
type
  TAbstractClass = class
    procedure AbstractMethod; virtual; abstract;
  ..
  end;
..
var
  Obj : TAbstractClass;

begin
  Obj := TAbstractClass.Create; // triggers a warning
..
end;
```

See also:

Alerts

4.1.39 WARN27-Empty code blocks and short-circuited statements

Empty code blocks and short-circuited statements (WARN27)

This is a list of all empty code blocks and short-circuited statements. Short-circuited statements are of these kinds:

Example:

```
if x then;

for I := 0 to 5 do;

while x do;
```

These statements may be mistakes.

See also:

Alerts

4.1.40 WARN28-Empty case labels

Empty case labels (WARN28)

Example:

```
case X of
  1 :;
  2 : begin
      end;
end;
```

The first case-branch is empty, which may be a mistake.

See also:

Alerts

4.1.41 WARN29-Short-circuited for-statements

Short-circuited for-statements (WARN29)

Example:

```
for I := 1 to 5 do;
```

See also:

Alerts

4.1.42 WARN30-Short-circuited if/case-statements

Short-circuited if/case-statements (WARN30)

Example:

```
if X then;
```

Also short-circuited else-branches are reported, both in if- and case-statements.

See also:

Alerts

4.1.43 WARN31-Short-circuited on-statements

Short-circuited on-statements (WARN31)

Example:

```
on Exception do;
```

See also:

Alerts

4.1.44 WARN32-Short-circuited repeat-statements

Short-circuited repeat-statements (WARN32)

Example:

```
1 | repeat until X;
```

See also:

Alerts

4.1.45 WARN33-Short-circuited while-statements

Short-circuited while-statements (WARN33)

Example:

```
1 | while X do;
```

See also:

Alerts

4.1.46 WARN34-Empty except-block

Empty except-block (WARN34)

Example:

```
1 | try
2 |   DoSomething;
3 | except
4 | end;
```

See also:

Alerts

4.1.47 WARN35-Empty finally-block

Empty finally-block (WARN35)

Example:

```
try
  DoSomething;
finally
end;
```

See also:

Alerts

4.1.48 WARN36-Forward directive in interface

Forward directive in interface (WARN36)

Even if a forward directive is allowed by at least some versions of the Pascal/Delphi compiler, they are unnecessary and should be avoided.

See also:

Alerts

4.1.49 WARN37-Empty subprogram parameter list

Empty subprogram parameter list (WARN37)

Somewhat surprisingly, this code is accepted by at least some versions of the Pascal/Delphi compiler:

Example:

```
procedure Proc();
begin
  ..
end;
```

See also:

Alerts

4.1.50 WARN38-Ambiguous references in with-blocks

Ambiguous references in with-blocks (WARN38)

This section reports locations where a valid references to an identifier inside a with-block could be mixed up with another identifier declared in the same scope. It is not an error, but just means that you should check that the code does what you intended.

Example:

```
var
  Title : string;

type
  TMyRec = record
    Title : string;
  end;

var
  Rec : TMyRec;

begin
..
  with Rec do
  begin
  ..
    Title := ´Hello’;
  end;
end;
```

The record field referenced in the with-block could be mixed up with the global *Title*. Maybe the programmer instead intended to set the global *Title* identifier.

See also:

Alerts

4.1.51 WARN39-Classes without overrides of abstract methods

Classes without overrides of abstract methods (WARN39)

This section lists classes that do not override abstract methods in ancestor classes. If a method is declared abstract in an ancestor class, it must be overridden in descendant classes. Otherwise, calling the method for the descendant class will result in a runtime error.

See also:

Alerts

4.1.52 WARN40-Local for-loop variables read after loop

Local for-loop variables read after loop (WARN40)

This section lists for-loop variables that are read in code after the loop. Their values are undefined, and thus it is not recommended to use their values.

See also:

Alerts

4.1.53 WARN41-Local for-loop variables possibly read after loop

Local for-loop variables possibly read after loop (WARN41)

This section lists for-loop variables that *possibly* are read in code after the loop. Their values are undefined, and thus it is not recommended to use their values.

See also:

Alerts

4.1.54 WARN42-For-loop variables not used in loop

For-loop variables not used in loop (WARN42)

When a for-loop variable is *not* used in the loop, it may be a coding error.

See also:

Alerts

4.1.55 WARN43-Non-public constructors/destructors

Non-public constructors/destructors (WARN43)

This section lists constructors/destructors that are non-public.

See also:

Alerts

4.1.56 WARN44-Functions called as procedures

Functions called as procedures (WARN44)

This section lists locations in the source code where functions are called as procedures, that is without using the result value. Maybe this is a coding error, and the function should really be called as a function instead.

See also:

Alerts

4.1.57 WARN45-Mismatch property read/write specifiers

Mismatch property read/write specifiers (WARN45)

This section lists property declarations with mismatch between read/write specifiers, like

```
1 | property IntValue2 : Integer read FIntValue2 write FIntValue3;
```

This is probably a coding error, and the function should really be called as a function instead.

See also:

Alerts

4.1.58 WARN46-Local variables that are set but not later used

Local variables that are set but not later used (WARN46)

This section lists local variables that are set but not later used further down in the code, like

```
1  procedure MyLocal;
2  var
3    X, Z : Integer;
4  begin
5    X := 555;
6    Z := 33;
7    DoIt(X);
8
9    X := 333; // not read below this...
10
11   if Z > 888 then
12     DoIt(Z);
13 end;
```

See also:

Alerts

4.1.59 WARN47-Duplicate lines

Duplicate lines (WARN47)

This section lists locations in the source code where a line is duplicated, that is when two lines immediately following each other, have the same content.

```
..
begin
  AddProc(nil);
  AddProc(nil);
end;
..
```

The check is done without case-sensitivity, so ...

```
..
begin
  AddProc(nil);
  AddPRoc(nil);
end;
..
```

.. are considered to be duplicate.

But for differences within string literals, the lines below..

```
..
case AChar of
 'ä': Result := 'ae';
 'Ä': Result := 'Ae';
end;
..
```

.. in the case-structure are considered NOT to be duplicate.

See also:

Alerts

4.1.60 WARN48-Duplicate class types in except-block

Duplicate class types in except-block (WARN48)

This section lists locations in the source code where an except-block contains more than

one handler for the same class type.
Like in this code:

```
..
try
  ....
except
  on E:ExceptionClass1 do
    ExceptionHandler1;
  on E:ExceptionClass1 do
    ExceptionHandler2;
end;
..
```

See also:

Alerts

4.1.61 WARN49-Redeclared identifiers from System unit

Redeclared identifiers from System unit (WARN49)

This section lists identifiers that use the same name as an identifier from the System.pas unit for the compiler target.
Although allowed, at least it is a source for confusion when maintaining the code.

```
var
  Pos : Integer; // conflicts with System function Pos()
..
function BlockRead : boolean; // conflicts with System procedure BlockRead()
..
```

See also:

Alerts

4.1.62 WARN50-Identifier with same name as keyword/directive

Identifier with same name as keyword/directive (WARN50)

This section reports identifier with names that conflict with keywords/directives.
Although allowed, is a source for confusion when maintaining the code, and sharing it with others.

See also:

Alerts

4.1.63 WARN51-Out parameter is read before set, or never set

Out parameter is read before set, or never set (WARN51)

This section reports parameters marked with the "out" directive that are read before set in the function/procedure, or never set.
An "out" parameter is just a placeholder for a return value. The function should not assume that its initial value has any meaning.

See also:

Alerts

4.1.64 WARN52-Possible bad assignment

Possible bad assignment (WARN52)

This section reports occurrences of assignments to smaller from bigger, resulting in data loss.
It will also report situations where for example UInt32 is assigned to Int32, where the range of the types do not fully overlap.

See also:

Alerts

4.1.65 WARN53-Mixing interface variables and objects

Mixing interface variables and objects (WARN53)

This section reports locations in your code with assignments between objects and interface variables. Normally, unless you really know what you are doing, it is a bad idea to mix interfaces and objects. The reason is that the reference counting mechanism of interfaces can be disturbed, leading to access violations and/or memory leaks.

Example:

```
type
  IIntf = interface
  end;

  TIntf = class(TInterfacedObject, IIntf)
  end;

procedure Proc;
var
  Obj : IIntf;
  X : TIntf;
begin
  Obj := TIntf.Create;
  X := TIntf(Obj); // not OK, mixes objects and interfaces
  ..
end;
```

See also:

Alerts

4.1.66 WARN54-Set before passed as out parameter

Set before passed as out parameter (WARN54)

This section reports locations in your code where a variable is set and then passed as an "out" parameter to a function.

Because the "out" parameter will be set in the called function without being read first, it is at least pointless to set it before it is passed. It may also indicate some misunderstanding about the code.

It is recommended to check if it is meaningful to set the variable before passing it. If not, remove the assignment, or else modify the signature of the called function from "out" to "var".

Example:

```
procedure Proc(out Param : Integer);
begin
  ..
end;

procedure Test;
var
  I : Integer;
begin
  ..
  I := 555; // this is meaningless!
  Proc(I);
  ..
end;
```

See also:

Alerts

4.1.67 WARN55-Redeclares ancestor member, or method in helped class/record

Redeclares ancestor member, or method in helped class/record (WARN55)

This section lists class fields or methods that redeclare ancestor members with the same name. This may lead to confusion about which member is actually referenced.

Also reported is when a helper class/record redeclares a method that exists in the helped class/record. The helper method will take precedence.

Example:

```
type
  TAncestor = class
  private
    FObj : TObject;
    procedure Proc;
  end;

  TDescendant = class(TAncestor)
  protected
    FObj : TObject; // redeclares!
    procedure Proc; // redeclares!
  end;
```

See also:

Alerts

4.1.68 WARN56-Parameter to FreeAndNil is not an object

Parameter to FreeAndNil is not an object (WARN56)

This section reports locations in your code where FreeAndNil takes a parameter which is not an object, for example an interface variable. This may lead to access violations. Unlike Free, the compiler will not complain.

Example:

```
type
  IMyInterface = interface
    ..
  end;

  TMyClass = class(TInterfacedObject, IMyInterface)
  end;

  procedure Proc;
  var
    Intf : IMyInterface;
    Obj : TMyClass;
  begin
    Intf := TMyClass.Create;
    FreeAndNil(Intf); // not OK!

    Obj := TMyClass.Create;
    FreeAndNil(Obj);  // OK
  end;
```

Note, that starting with Delphi 10.4 it is much harder to produce this warning, because FreeAndNil only accepts parameter based on TObject.
It is still however possible to produce the error, for example with code like:

FreeAndNil(TObject(IAnyInterface))

See also:

Alerts

4.1.69 WARN57-Enumerated constant missing in case structure

Enumerated constant missing in case structure (WARN57)

This section lists locations in your code where a case statement does not list all possible values of an enumerated type. This is probably most often as intended, but it may also point out an error in the code.

Example:

```
type
  TChessPiece = (cpPawn, cpKnight, cpBishop, cpRook,
    cpQueen, cpKing);

procedure Move(Piece : TChessPiece;
  FromSquare, ToSquare : TSquare);
begin
  case Piece of
    cpPawn : ..;
    cpKnight : ..;
    cpBishop : ..;
    cpRook : ..;
    cpQueen : ..;
  end;

  ..
end;
```

In the code above, **cpKing** is missing from the case structure, and will trigger a warning.

If you want to suppress warnings for a case-structure, just use PALOFF on the same line as the "case" keyword.

See also:

Alerts

4.1.70 WARN58-Mixed operator precedence levels

Mixed operator precedence levels (WARN58)

This section lists locations in your code where operators of different levels are mixed. Operators are in Object Pascal evaluated from left to right, unless parentheses are used. Operators of level 1 are evaluated before operators of level 2 etc.

Level 1: @, not
Level 2: *, /, div, mod, and, shl, shr, as
Level 3: +, -, or, xor
Level 4: =, <>, <, >, <=, >=, in, is

Example:

```
X := A + B * 5;
// evaluated as X := A + (B * 5)

B := BoolA and BoolB or C;
// evaluated as B := (BoolA and BoolB) or C
```

Mixing operators is perfectly valid but you will find that your code is clearer and easier to understand if you insert parentheses. Then you do not have to think about operator precedence.

See also:

Alerts

4.1.71 WARN59-Explicit float comparison

Explicit float comparison (WARN59)

This section lists locations in your code where floating point numbers are directly compared. It is considered not secure to compare floating numbers directly. Instead use functions in Delphi's System.Math unit, like IsZero and SameValue.

Example:

```
1  procedure CalculateProfit;
2  var
3    Yield, Income : Double;
4  begin
5    Yield := GetYield;
6    Income := GetIncome;
7
8    if Yield = Income then // not secure!
9    begin
10     ..
11   end;
12 end;
```

In the example above, use instead SameValue function from System.Math unit.

See also:

Alerts

4.1.72 WARN60-Condition evaluates to constant value

Condition evaluates to constant value (WARN60)

This section lists locations in your code where a condition evaluates to a constant value.

Example:

```
procedure MyProc;
var
  X : Integer;
begin
  X := 0;

  ..

  if X+5 = 15 then // if-condition has always the same value ...
  begin
  end;
end;

```

See also:

Alerts

4.1.73 WARN61-Assigned to itself

Assigned to itself (WARN61)

This section lists locations in your code where a variable has been assigned to itself. Even if this assignment is harmless, it makes no sense. It may indicate other problems with the code, so you should check the surrounding code.

See also:

Alerts

4.1.74 WARN62-Possible orphan event handler

Possible orphan event handler (WARN62)

This section lists class procedures in your code that look like event handlers. But they are not connected to any control in the corresponding DFM-file.

See also:

Alerts

4.1.75 WARN63-Mismatch 32/64-bits

Mismatch 32/64-bits (WARN63)

This section reports locations where 32-bits (or smaller) variables are passed as 64-bits parameters (or vice versa).

In many cases this is totally harmless, but consider the case where a 32-bits pointer is passed to a function that expects a 64-bits pointer.
Also if there is a mismatch when assigning values, it will be reported.

See also:

Alerts

4.1.76 MEMO1-Local objects with unprotected calls to Free

Local objects with unprotected calls to Free (MEMO1)

This section reports locations where calls to Free (and FreeAndNil or Release) are not done in try-finally blocks. Failure to wrap a try-finally block around a memory deallocation could result in a memory leak. The report does not list locations in FormDestroy and FormClose events, because these are normally called when a form is destroyed. Neither does it report calls to Free from a finalization block. Also an object that is freed in a try-except block is not reported.

See also:

Alerts

4.1.77 MEMO2-Non-local objects with unprotected calls to Free

Non-local objects with unprotected calls to Free (MEMO2)

Like the previous section, but for non-local objects.

See also:

Alerts

4.1.78 MEMO3-Objects created in try-structure

Objects created in try-structure (MEMO3)

This section lists lists locations where an object is created inside a try-structure, like:

```
1  try
2    Obj := TMyObject.Create;
3    ..
4  finally
5    Obj.Free;
6  end;
```

Here, Obj should be created before the "try", otherwise Obj.Free will be called even if the object fails to create, possibly causing a runtime error.

See also:

Alerts

4.1.79 MEMO4-Unbalanced Create/Free

Unbalanced Create/Free (MEMO4)

This section reports objects that are not created and freed the same number of times. This can indicate an error, like in the following example:

```
1  procedure LocalProc;
2  var
3    Obj : TMyClass;
4  begin
5    Obj := TmyClass.Create;
6    Obj.DoSomething;
7  end;
```

Here, the locally declared object Obj is never freed, so this code will cause a memory leak.

See also:

Alerts

4.1.80 MEMO5-Local objects that are created more than once without being freed in-between

Local objects that are created more than once without being freed in-between (MEMO5)

This section reports objects that are created more than once (in a row) without being freed in-between.
This leads to memory leakage, like in the following example:

```
procedure LocalProc;
var
  Obj : TMyClass;
begin
  Obj := TMyClass.Create;

  try
    Obj := TMyClass.Create;

    try
    ..
    finally
      FreeAndNil(Obj);
    end;

  ..
  finally
    FreeAndNil(Obj);
  end;
end;
```

Here, the locally declared object Obj is only freed once, which causes a memory leak.

See also:

Alerts

4.1.81 MEMO6-Local objects that are referenced before being created

Local objects that are referenced before being created (MEMO6)

This section reports objects that are referenced before being created.
This leads to an exception, like in the following example:

```
procedure LocalProc;
var
  Obj : TMyClass;
begin
  Obj.Field := 55; // !! gives an AV
end;
```

Objects that PAL cannot determine have been created at all, are not reported, only those cases where the object has been created further down in the code. Otherwise there should be many false positives.

See also:

Alerts

4.1.82 MEMO7-Local objects that are referenced after being freed

Local objects that are referenced after being freed (MEMO7)

This section reports objects that are freed but referenced further down in the code. This leads to an exception, like in the following example:

```
procedure LocalProc;
var
  Obj : TMyClass;
begin
  Obj := TMyClass.Create;

  try
  ..
  finally
    FreeAndNil(Obj);
  end;

  Obj.Field := 55; // !! Obj is already freed
end;
```

See also:

Alerts

4.1.83 COWA1-Controls that overlap visually

Controls that overlap visually (COWA1)

This is a list of controls that overlap each other visually, possibly hiding each other.

See also:

Alerts

4.1.84 COWA2-Labels with Caption-property that does not end in ":"

Labels with Caption-property that does not end in ":" (COWA2)

Labels (TLabels) above or to the left of other controls usually end their caption with the char ":". This lists all labels that not confirm to this. Of course, this does not apply to labels that are standalone and just used for display purposes. PAL cannot know the purpose of a label and reports all labels with missing ":".

A false warning is generated for captions that are so long that they span over more than one line in the DFM file.

See also:

Alerts

4.1.85 COWA3-Conflicting accelerators

Conflicting accelerators (COWA3)

This is a list of all controls with conflicting accelerators in the Caption property. Some types of controls are not reported even if they share the same accelerators, because they do not conflict. Those are menu items on different sub menus, and controls that reside on different TTabSheet pages of a TPageControl control.

See also:

Alerts

4.1.86 COWA4-Labels (or static texts) that have accelerators but FocusControl is not set

Labels (or static texts) that have accelerators but FocusControl is not set (COWA4)

Labels and static texts cannot receive focus. When an accelerator key is pressed, focus is given to the control specified by the FocusControl property. It is an error to omit the FocusControl property in this case.

See also:

Alerts

4.1.87 COWA5-Conflicting shortcuts

Conflicting shortcuts (COWA5)

This is a list of all menu items with conflicting shortcuts (key combination) (property ShortCut).

See also:

Alerts

4.1.88 COWA6-Buttons/menu items with OnClick-event that is unassigned

Buttons/menu items with OnClick-event that is unassigned (COWA6)

This is a list of all buttons and menu items with an unassigned OnClick-event. Normally there should be an action on OnClick for these controls, so it indicates an error in the code. Warnings are not created when the property Action is set. Buttons with ModalResult set will be excluded from the list.

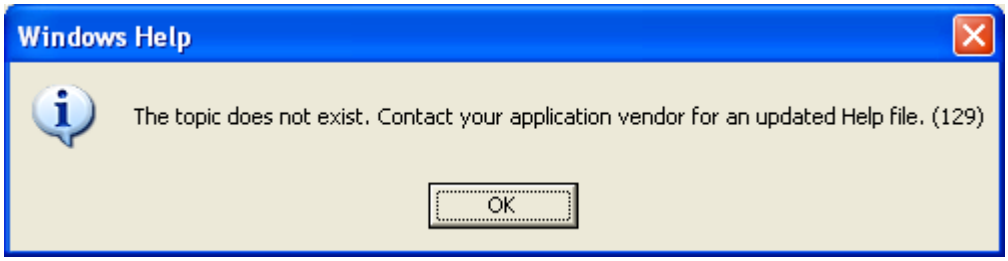
See also:

Alerts

4.1.89 COWA7-Menu items that have HelpContext=0

Menu items that have HelpContext=0 (COWA7)

This is a list of all menu items with a HelpContext property value of 0. Probably the menu item has not been assigned a topic in the help file. Failing to assign a topic could trigger this messagebox when the help system is invoked:



You can also use the Missing Property Report to generate this information. Add a "TMenuItem;HelpContext" item to the check list.

See also:

Alerts

4.1.90 COWA8-Hint is not activated

Hint is not activated (COWA8)

This is a list of all controls where the Hint property is set, and both ShowHint and ParentShowHint properties are "false".

See also:

Alerts

4.2 Reductions

The Reductions report area contains:

- Code Reduction (REDU1-REDU24)

This report considers items that could be removed from the code, thus making it easier to maintain.

REDU1-Identifiers never used
REDU2-Local identifiers only used at a lower scope

REDU3-Local identifiers only used at a lower scope, but in more than one subprogram
REDU4-Local identifiers that are set and referenced once
REDU5-Local identifiers that possibly are set and referenced once

REDU6-Local identifiers that are set more than once without referencing in-between
REDU7-Local identifiers that possibly are set more than once without referencing in-between
REDU8-Class fields that are zero-initialized in constructor
REDU9-Class fields that possibly are zero-initialized in constructor
REDU10-Local long strings that are initialized to empty string

REDU11-Local long strings that possibly are initialized to empty strings
REDU12-Functions called only as procedures (result ignored)
REDU13-Functions/procedures (methods excluded) only called once
REDU14-Methods only called once from other method of the same class
REDU15-Unneeded boolean comparisons

REDU16-Boolean assignment can be shortened
REDU17-Fields only used in single method
REDU18-Consider using interface type
REDU19-Redundant parentheses
REDU20-Common subexpression, consider elimination

REDU21-Default parameter values that can be omitted
REDU22-Inconsistent conditions
REDU23-Typecasts that possibly can be omitted
REDU24-Local identifiers never used

See also:

Reports
Alerts
Optimizations
Conventions

4.2.1 REDU1-Identifiers never used

Identifiers never used (REDU1)

This is a list of all identifiers that are declared but never used. The Delphi compiler (from Delphi 2) also reports this if warnings ($W+) have been turned on during compilation. Most often, you can remove these identifiers. If you remove any identifier, make sure your code still compiles and works properly. A wise habit is to first comment out these declarations, and remove them entirely when you have validated that the code still compiles and works as intended. Also, note that if a subprogram is not used, does not necessarily indicate that it is not needed at all. If it is part of a general unit, the subprogram could very well be used in other applications.

Identifiers (parameters, local variables etc) related to subprograms that are not used, are not reported.

Constructors/destructors are not examined by this section. Also parameters to event handlers, or methods that are referenced in form files, are not reported as unused. The reason is to avoid unnecessary warnings.

Also unused methods of a class that are implemented through interfaces are not reported. In this case, the class must implement these methods.

Example:

```
procedure TMyForm.mnuOpen(Sender : TObject);
begin
  OpenFile;
end;
```

In this case, the parameter Sender is not reported as unused, since mnuOpen is an event handler.

A subset of this report is its own section. It is the REDU24 report section, which reports only local identifiers.

See also:

Reductions

4.2.2 REDU2-Local identifiers only used at a lower scope

Local identifiers only used at a lower scope (REDU2)

This is a list of all local identifiers that are only used at a lower scope, in nested subprograms. You can declare these identifiers in the local procedures/functions where they are actually used.

Example:

```
procedure Outer;
var
  I : Integer;

  procedure Inner;
  begin
    I := 55; // !! I is declared in Outer
    ..
  end;

begin
  ..
end;
```

See also:

Reductions

4.2.3 REDU3-Local identifiers only used at a lower scope, but in more than one subprogram

Local identifiers only used at a lower scope, but in more than one subprogram (REDU3)

This is a list of all local identifiers that are only used at a lower scope, in nested subprograms. You can probably declare these identifiers in the local procedures/functions where they are actually used, unless they should be shared by the nested subprograms.

Example:

```
1  procedure Outer;
2  var
3    I : Integer;
4
5    procedure Inner1;
6    begin
7      ..
8      I := 55; // !! I is declared in Outer
9      ..
10   end;
11
12   procedure Inner2;
13   begin
14     ..
15     I := 5;
16     ..
17   end;
18
19 begin
20   ..
21 end;
```

See also:

Reductions

4.2.4 REDU4-Local identifiers that are set and referenced once

Local identifiers that are set and referenced once (REDU4)

This is a list of all local identifiers that are set and referenced just once. It may be more efficient to skip these intermediate identifiers.

Restrictions:
Identifiers that are first set as a *var* parameter in a call to a subprogram, and afterwards referenced, are not reported. Also, when the identifier is referenced in a loop, it is not reported.

Example:

```
1  procedure MyProc;
2  var
3    I : Integer;
4
5  begin
6    I := Func(5);
7    ..
8
9    if I = 5 then // !! if Func(5) = 5 then
10   begin
11     ..
12   end;
13 end;
```

See also:

Reductions

4.2.5 REDU5-Local identifiers that possibly are set and referenced once

Local identifiers that possibly are set and referenced once (REDU5)

This is a list of all local identifiers that possibly are set and referenced just once. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. It may be more efficient to skip this intermediate identifier.

Restrictions:
Identifiers that are first set as a *var* parameter in a call to a subprogram, and afterwards referenced, are not reported. Also, when the identifier is referenced in a loop, it is not reported.

Example:

```
1  procedure MyProc;
2  var
3    I : Integer;
4
5  begin
6    I := Func(5);
7    ..
8    UnknownProc(I); // !! UnknownProc(Func(5))
9    ..
10 end;
```

See also:

Reductions

4.2.6 REDU6-Local identifiers that are set more than once without referencing in-between

Local identifiers that are set more than once without referencing in-between (REDU6)

This is a list of all local identifiers that are set (assigned) more than once without referencing in-between. You can probably remove all but the last assignment. It may of course also indicate a coding error.

Example:

```
1  procedure MyProc;
2  var
3    I : Integer;
4
5  begin
6    I := 5;
7    ..
8    I := 10; // !! I is set again
9    ..
10   if I = 10 then
11   begin
12     ..
13   end;
14 end;
```

See also:

Reductions

4.2.7 REDU7-Local identifiers that possibly are set more than once without referencing in-between

Local identifiers that possibly are set more than once without referencing in-between (REDU7)

This is a list of all local identifiers that are set (assigned) more than once without referencing in-between. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. You can probably delete all but the last assignment.

Example:

```
procedure MyProc;
var
  I : Integer;

begin
  I := 5;
  ..
  UnknownProc(I);
  ..
  I := 10; // !! I may already be set
  ..
  if I = 10 then
  begin
    ..
  end;
end;
```

See also:

Reductions

4.2.8 REDU8-Class fields that are zero-initialized in constructor

Class fields that are zero-initialized in constructor (REDU8)

This is a list of all class fields that are zero-initialized in constructor. Since class fields are automatically zero-initialized when the object is created, there is usually no need to include this code.

Example:

```
type
  TMyClass = class(TAncestorClass)
  private
    FInt : Integer;
    FStr : string;
    FPtr : Pointer;
  public
    constructor Create; override;
    ..
  end;

  constructor TMyClass.Create;
  begin
    inherited Create;
    FInt := 0; // !! unnecessary
    FStr := '';
    FPtr := nil;
  end;
```

See also:

Reductions

4.2.9 REDU9-Class fields that possibly are zero-initialized in constructor

Class fields that possibly are zero-initialized in constructor (REDU9)

This is a list of all class fields that possibly are zero-initialized in constructor. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. Since class fields are automatically zero-initialized when the object is created, there is usually no need to include this code.

Example:

```
type
  TMyClass = class(TAncestorClass)
  private
    FInt : Integer;
    FStr : string;
    FPtr : Pointer;
  public
    constructor Create; override;
  end;

  constructor TMyClass.Create;
  begin
    inherited Create;
    UnknownProc1(FInt); // !! possibly unnecessary
    UnknownProc2(FStr);
    UnknownProc3(FPtr);
  end;
```

See also:

Reductions

4.2.10 REDU10-Local long strings that are initialized to empty string

Local long strings that are initialized to empty string (REDU10)

(Not relevant for BP7 and D1)

This is a list of all local long strings that are initialized to empty strings. An unnecessary action, since long strings are automatically initialized as empty strings upon creation.

Example:

```
procedure MyProc;
var
  S : string;

begin
  S := ''; // !! unnecessary
  ..
end;
```

See also:

Reductions

4.2.11 REDU11-Local long strings that possibly are initialized to empty strings

Local long strings that possibly are initialized to empty strings (REDU11)

(Not relevant for BP7 and D1)

This is a list of all local long strings that are initialized to empty strings. They are referenced in unknown fashion, and the parser cannot determine whether they are set or just referenced in these locations. An unnecessary action, since long strings are automatically initialized as empty strings upon creation.

Example:

```
procedure MyProc;
var
  S : string;

begin
  UnknownProc(S); // !! possibly unnecessary
  ..
end;
```

See also:

Reductions

4.2.12 REDU12-Functions called only as procedures (result ignored)

Functions called only as procedures (result ignored) (REDU12)

These functions may possibly better be implemented as procedures, because the result is never used.

See also:

Reductions

4.2.13 REDU13-Functions/procedures (methods excluded) only called once

Functions/procedures (methods excluded) only called once (REDU13)

The code in these functions/procedures could possibly be included inline instead, avoiding an unnecessary call.

See also:

Reductions

4.2.14 REDU14-Methods only called once from other method of the same class

Methods only called once from other method of the same class (REDU14)

These methods are never called from the outside. The code in these methods could possibly be included inline instead, avoiding an unnecessary call.

See also:

Reductions

4.2.15 REDU15-Unneeded boolean comparisons

Unneeded boolean comparisons (REDU15)

This list contains locations with statements like

```
if bReady = true then
```

This could be shorter and better written as

```
if bReady then
```

See also:

Reductions

4.2.16 REDU16-Boolean assignment can be shortened

Boolean assignment can be shortened (REDU16)

This list contains locations with statements like

```
if X then
  Bool := True
else
  Bool := False;
```

This could be shorter and better written as

```
Bool := X;
```

See also:

Reductions

4.2.17 REDU17-Fields only used in single method

Fields only used in single method (REDU17)

This list contains class or record fields that are only used in a single method. They could probably better be declared as local variables.

See also:

Reductions

4.2.18 REDU18-Consider using interface type

Consider using interface type (REDU18)

This list contains objects which can be declared and implemented as an interface type, instead of as the class type implementing the interface. The advantage is that interface reference counting can be used so you will not have to explicitly free the object.

Example:

```
type
  IIntf = interface
    ..
  end;

  TIntf = class(TInterfacedObject, IIntf)
    ..
  end;

var
  Obj : TIntf; // consider using IIntf!
  ..
```

The list will not include objects that are not created. These objects are probably just assigned to another object.
Another condition that must be met is that the object is of a class that implements exactly one interface.

See also:

Reductions

4.2.19 REDU19-Redundant parentheses

Redundant parentheses (REDU19)

This section lists locations in your code where superfluous parentheses can be removed, simplifying the code.

Example:

```
1  procedure Proc;
2  begin
3    if (3+4) = 7 then ..;
4
5    if (((True))) then ..;
6
7    if (True) then ..;
8
9    if (3+4+(5)+(6)) = 11 then ..;
10 end;
11
```

See also:

Reductions

4.2.20 REDU20-Common subexpression, consider elimination

Common subexpression, consider elimination (REDU20)

This section lists locations in your code with repeated common subexpressions. Those may be candidates to put into temporary variables to simplify and optimize the code.

Example:

```
procedure Proc;
var
  A, B, C, D, E : Integer;
begin
  ..
  E := A+B+C;
  ..
  D := A+B+C; // expression is repeated
  ..
end;
```

If any of the variables involved in the repeated expressions would have been modified, between the locations, there should not be any warning.

See also:

Reductions

4.2.21 REDU21-Default parameter values that can be omitted

Default parameter values that can be omitted (REDU21)

This list contains calls to functions or procedures that use default parameters, and where the parameter can be omitted at the call site. The reason is then that the value of the parameter passed is the same as the default parameter value.

Example:

```
procedure ProcWithDefaultParam(P : Pointer = nil);
begin
  ..
end;

procedure Proc;
var
  P : Pointer;
begin
  P := nil;
  ProcWithDefaultParam(P); // the parameter P is not needed here
end;
```

See also:

Reductions

4.2.22 REDU22-Inconsistent conditions

Inconsistent conditions (REDU22)

This section reports locations with inconsistent conditions. These are places where a condition check is repeated, even if the outcome will be the same as in the previous location.

Example:

```
procedure Proc;
var
  I : Integer;
begin
  if I = 1 then
  begin
    ..
  end
  else
    if I = 1 then // gives inconsistent condition warning
    begin
      ..
    end;
end;
```

See also:

Reductions

4.2.23 REDU23-Typecasts that possibly can be omitted

Typecasts that possibly can be omitted (REDU23)

This section reports locations with typecasts that possibly can be omitted. It is locations where the typecast casts the variable to the same type that it already has.

Example:

```
var
  I, K : Integer;
begin
  ..
  K := Integer(I);
  ..
end;
```

See also:

Reductions

4.2.24 REDU24-Local identifiers never used

Local identifiers never used (REDU24)

This a list of all local identifiers that are declared but never used. It is actually a subset of the REDU1 report section, which reports all identifiers, not only local.

See also:

Reductions

4.3 Conventions

The Conventions report area contains:

- Conventions (CONV1-CONV31)

This report contains several lists with identifiers that do not comply with conventions.

The choice of names for identifiers has a considerable influence on the ease of understanding and maintenance costs of your source code. Developers familiar with the coding standards can understand the code more easily if it follows general conventions.

CONV1-Ordinary types that do not start with "T"
CONV2-Exception types that do not start with "E"
CONV3-Pointer types that do not start with "P"
CONV4-Interface types that do not start with "I"
CONV5-Class fields that are not declared in the private section

CONV6-Class fields that are exposed by properties (read/write) but do not start with "F"
CONV7-Properties to method pointers that do not start with "On/Before/After"
CONV8-Functions that are exposed by properties (read) but do not start with "Get"
CONV9-Procedures that are exposed by properties (write) but do not start with "Set"
CONV10-Classes that have visible constructors with bad names

CONV11-Classes that have visible destructors with bad names
CONV12-Identifiers that have unsuitable names
CONV13-Multiple with-variables
CONV14-Property access methods that are not private/protected
CONV15-Hard to read identifier names

CONV16-Label usage
CONV17-Bad class visibility order
CONV18-Identifiers with numerals
CONV19-Local identifiers that "shadow" outer scope identifiers
CONV20-Local identifiers that "shadow" class members

CONV21-Class/member name collision
CONV22-Class fields that are not declared in the private/protected sections
CONV23-Class fields that do not start with "F"

CONV24-Value parameters that do not start with selected prefix
CONV25-Const parameters that do not start with selected prefix

CONV26-Out parameters that do not start with selected prefix
CONV27-Var parameters that do not start with selected prefix
CONV28-Old-style function result
CONV29-With statements
CONV30-Private can be changed to strict private
CONV31-Protected can be changed to strict protected
CONV32-Multiple statements on the same line

See also:

Reports
Alerts
Optimizations
Reductions

4.3.1 CONV1-Ordinary types that do not start with "T"

Ordinary types that do not start with "T" (CONV1)

This is a list of all ordinary types that do not start with the letter “T”. Exception, pointer and interface types are not included. As a convention, user-defined type names start with the letter “T”. A class that is a CoClass is an exception and is not reported. PAL assumes a CoClass when the name of the class starts with the letters “Co”. Furthermore, the class must have a class function with the name “Create”.

Also custom attributes inheriting from TCustomAttribute are not reported.

See also:

Conventions

4.3.2 CONV2-Exception types that do not start with "E"

Exception types that do not start with "E" (CONV2)

This is a list of all exception types that do not start with the letter “E”. As a convention, user-defined exception type names start with the letter “E”.

See also:

Conventions

4.3.3 CONV3-Pointer types that do not start with "P"

Pointer types that do not start with "P" (CONV3)

This is a list of all pointer types that do not start with the letter "P". As a convention, user-defined pointer type names start with the letter "P".

See also:

Conventions

4.3.4 CONV4-Interface types that do not start with "I"

Interface types that do not start with "I" (CONV4)

This is a list of all interface types that do not start with the letter "I". As a convention, user-defined interface type names start with the letter "I".

See also:

Conventions

4.3.5 CONV5-Class fields that are not declared in the private section

Class fields that are not declared in the private section (CONV5)

This is a list of all class fields that are not declared in the private section of a class.

See also:

Conventions

4.3.6 CONV6-Class fields that are exposed by properties (read/write) but do not start with "F"

Class fields that are exposed by properties (read/write) but do not start with "F" (CONV6)

This is a list of all class fields that are exposed by properties but do not start with the letter "F". As a convention, private class field names start with the letter "F".

This section is similar to CONV23, but that section reports all fields, not only those exposed by properties.

See also:

Conventions

4.3.7 CONV7-Properties to method pointers that do not start with "On/Before/After"

Properties to method pointers that do not start with "On/Before/After" (CONV7)

This is a list of all properties to method pointers that do not start with "On/Before/After".

See also:

Conventions

4.3.8 CONV8-Functions that are exposed by properties (read) but do not start with "Get"

Functions that are exposed by properties (read) but do not start with "Get" (CONV8)

This is a list of all functions that are exposed by properties read methods, but do not start with “Get”. As a convention, these functions (methods) should start with the letters “Get” (e g GetIndex, GetBitmap).

See also:

Conventions

4.3.9 CONV9-Procedures that are exposed by properties (write) but do not start with "Set"

Procedures that are exposed by properties (write) but do not start with "Set" (CONV9)

This is a list of all functions that are exposed by properties write methods, but do not start with “Set”. As a convention, these procedures (methods) should start with the letters “Set” (e.g. SetIndex, SetBitmap).

See also:

Conventions

4.3.10 CONV10-Classes that have visible constructors with bad names

Classes that have visible constructors with bad names (CONV10)

This is a list of all classes that have constructors with bad names. As a convention, constructor names start with the letters “Create”. For old-style objects (BP7), the constructor names start with the letters “Init”.

See also:

Conventions

4.3.11 CONV11-Classes that have visible destructors with bad names

Classes that have visible destructors with bad names (CONV11)

This is a list of all classes that have destructors with bad names. As a convention, destructor names start with the letters "Destroy". For
old-style objects (BP7), the destructor names start with the letters "Done".

See also:

Conventions

4.3.12 CONV12-Identifiers that have unsuitable names

Identifiers that have unsuitable names (CONV12)

This is a list of all identifiers with names that are the same as directives, e.g. "pascal", "dynamic", "index" and others. Even if the compiler allows this, it may lead to misunderstandings. For Delphi 1 and higher, the list also includes identifiers with identical names as identifiers from the System unit (like "Copy", "AllocMem").

See also:

Conventions

4.3.13 CONV13-Multiple with-variables

Multiple with-variables (CONV13)
This is a list of all locations in the source where multiple with-variables ("with A, B do") are used. It is often considered a bad coding habit to use multiple with-variables, since they make the source more difficult to understand.

See also:

Conventions

4.3.14 CONV14-Property access methods that are not private/protected

Property access methods that are not private/protected (CONV14)
This is a list of all property access methods that are not declared as private/protected. Property access methods are used with properties, e. g:

```
property MyProp : integer read GetMyProp write SetMyProp
```

where GetMyProp and SetMyProp are property access methods.

Those methods should not be directly callable from the outside, because all access should go through the associated property.

See also:

Conventions

4.3.15 CONV15-Hard to read identifier names

Hard to read identifier names (CONV15)

This is a list of all identfiers with hard to read names. A name is considered hard to read if it contains both the letter "O" and the number"0", or both the letter "l" and the number "1".

See also:

Conventions

4.3.16 CONV16-Label usage

Label usage (CONV16)

This list contains all labels that are used in the source code. Labels define jump-locations for a goto statement. Usage of labels and goto-statements is considered as a bad thing, which is most often not needed in modern object-oriented programming. There are situations though, when a label may be the right choice.

See also:

Conventions

4.3.17 CONV17-Bad class visibility order

Bad class visibility order (CONV17)

This list contains all class types with bad class visibility order in the declaration. Bad order is defined as when **private** sections appear after **public**/**protected** sections or when **protected** sections appear after **public** sections. The code is probably easier to understand and maintain if a good visibility order is used.

Classes that are derived from TForm are not reported. This is because these type of classes depend on a special order, starting with published identifiers.

See also:

Conventions

4.3.18 CONV18-Identifiers with numerals

Identifiers with numerals (CONV18)

This list contains all identifiers with names that contain numerals.

See also:

Conventions

4.3.19 CONV19-Local identifiers that "shadow" outer scope identifiers

Local identifiers that "shadow" outer scope identifiers (CONV19)

This list contains local identifiers that have the same name as outer scope identifiers in the same unit.

Example:

```
1  var
2    I : Integer;
3
4  procedure Proc;
5  var
6    I : Integer; // I shadows outer global I!
7  begin
8    ..
9  end;
```

Although this is allowed, it may lead to confusion and misunderstandings when maintaining the code.

See also:

Conventions

4.3.20 CONV20-Local identifiers that "shadow" class members

Local identifiers that "shadow" class members (CONV20)

This list contains local identifiers in methods that have the same name as a class member.

Example:

```
type
  TMyClass = class
  private
    Field : Integer;
    procedure M;
  end;
  ..
  procedure TMyClass.M;
  var
    Field : Integer;
  begin
    ..
  end;
```

Although this is allowed, it may lead to confusion and misunderstandings when maintaining the code.

See also:

Conventions

4.3.21 CONV21-Class/member name collision

Class/member name collision (CONV21)

This section reports situations where class and member names collide.

See also:

Conventions

4.3.22 CONV22-Class fields that are not declared in the private/protected sections

Class fields that are not declared in the private/protected sections (CONV22)

This section lists fields that are not declared in the private/protected sections.

See also:

Conventions

4.3.23 CONV23-Class fields that do not start with "F"

Class fields that do not start with "F" (CONV23)

This is a list of all class fields that do not start with the letter "F". As a convention, private class field names start with the letter "F".
Component fields in the DFM-file are not reported.

This section is similar to CONV6, but that section only reports fields exposed by properties.

See also:

Conventions

4.3.24 CONV24-Value parameters that do not start with selected prefix

Value parameters that do not start with selected prefix (CONV24)

This is a list of all value parameters that do not start with selected prefix. Set the selected prefix by clicking on the CONV24 list item on the Conventions tab page. Then click the Prefix button to enter the prefix.

See also:

Conventions

4.3.25 CONV25-Const parameters that do not start with selected prefix

Const parameters that do not start with selected prefix (CONV25)

This is a list of all value parameters that do not start with selected prefix. Set the selected prefix by clicking on the CONV25 list item on the Conventions tab page. Then click the Prefix button to enter the prefix.

See also:

Conventions

4.3.26 CONV26-Out parameters that do not start with selected prefix

Out parameters that do not start with selected prefix (CONV26)

This is a list of all value parameters that do not start with selected prefix. Set the selected prefix by clicking on the CONV26 list item on the Conventions tab page. Then click the Prefix button to enter the prefix.

See also:

Conventions

4.3.27 CONV27-Var parameters that do not start with selected prefix

Var parameters that do not start with selected prefix (CONV27)

This is a list of all value parameters that do not start with selected prefix. Set the selected prefix by clicking on the CONV27 list item on the Conventions tab page. Then click the Prefix button to enter the prefix.

See also:

Conventions

4.3.28 CONV28-Old-style function result

Old-style function result (CONV28)

This is a list of functions where instead of "Result", the function name is used as the result variable.

See also:

Conventions

4.3.29 CONV29-With statements

With statements (CONV29)

This is a list of locations where "with" is used.

See also:

Conventions

4.3.30 CONV30-Private can be changed to strict private

Private can be changed to strict private (CONV30)

This is a list of all class members that are private but can be changed to strict private.

See also:

Conventions

4.3.31 CONV31-Protected can be changed to strict protected

Protected can be changed to strict protected (CONV31)

This is a list of all class members that are protected but can be changed to strict protected.

See also:

Conventions

4.3.32 CONV32-Multiple statements on the same line

Multiple statements on the same line (CONV32)

This is a list of locations with more than one statement on the same line.

See also:

Conventions

4.4 Optimizations

The Optimizations report area contains:

- Optimizations (OPTI1-OPTI11)

This report pinpoints elements of the code that you can improve, resulting in better performance. With better performance, we here mean faster execution, not necessarily smaller code.

OPTI1-Missing "const" for unmodified string parameter
OPTI2-Missing "const" for unmodified record parameter
OPTI3-Missing "const" for unmodified array parameter
OPTI4-Array properties that are referenced/set within methods
OPTI5-Virtual methods (procedures/functions) that are not overridden

OPTI6-Local subprograms with references to outer local variables
OPTI7-Subprograms with local subprograms
OPTI8-Parameter is "var", can be changed to "out"
OPTI9-Inlined subprogs not inlined because not yet implemented
OPTI10-Managed local variable that can be declared inline
OPTI11-Managed local variable is inlined in loop

See also:

Reports
Alerts
Reductions
Conventions

4.4.1 OPTI1-Missing "const" for unmodified string parameter

Missing “const” for unmodified string parameter (OPTI1)

This is a list of all string parameters that you can declare with the *const* directive, resulting in better performance since the compiler can assume that the parameter will not be changed. For example, for a long string the reference count for the string does not need to be updated on entry and exit to the function. For other types of strings, like WideString, the string may have to be copied when passed to the function.

Example:

```
procedure MyProc(S : string);
begin
  ..
  if S = 'Foo' then // !! S is not changed
  begin
    ..
  end;
  ..
end;
```

In this case, the parameter S should have the *const* directive, since it is never changed in the procedure. The compiler can generate code that is more efficient.

No warning is given for methods that are marked with the "override" directive. This is because they must follow the parameter list that the overridden method has.

See also:

Optimizations

4.4.2 OPTI2-Missing "const" for unmodified record parameter

Missing “const” for unmodified record parameter (OPTI2)

This is a list of all record parameters that you can declare with the *const* directive, resulting in better performance since the compiler can assume that the parameter will not be changed. Generally, if the parameter is larger than 4 bytes, and it doesn't need to be altered in the subroutine, *const* is more efficient. Also, it is a good idea to use CONST on parameters that aren't intended to be altered, so the compiler can catch those errors for you.

Example:

```
type
  TRec = record
    X : Integer;
  end;

procedure MyProc(R : TRec);
begin
  ..
  if R.X = 5 then // !! R is not changed
  begin
    ..
  end;
  ..
end;
```

In this case, the parameter R should have the *const* directive, since it is never changed in the procedure. The compiler can generate code that is more efficient.

No warning is given for methods that are marked with the "override" directive. This is because they must follow the parameter list that the overridden method has.

See also:

Optimizations

4.4.3 OPTI3-Missing "const" for unmodified array parameter

Missing "const" for unmodified array parameter (OPTI3)

This is a list of all array parameters that you can declare with the *const* directive, resulting in better performance since the compiler can assume that the parameter will not be changed.

No warning is given for methods that are marked with the "override" directive. This is because they must follow the parameter list that the overridden method has.

See also:

Optimizations

4.4.4 OPTI4-Array properties that are referenced/set within methods

Array properties that are referenced/set within methods (OPTI4)

This is a list of all array properties that are referenced or set within methods of a class. Methods that include a reference to the property are listed.

For performance reasons it is faster to directly access the private array field. However, if the Get- or Set-method performs side effects, it makes sense to access the property.

For simple non-array properties, the compiler generates the same code for both access of the property or the field. Therefore, for normal properties there is no advantage in referencing the private field.

Example:

```
type
  TMyClass = class
  private
    FItems : array[0..10] of Integer;
    FPlain : Integer;
  protected
    function GetItem(Index : Integer) : Integer;
    procedure SetItem(Index, Value : Integer);
  public
    procedure MyProc;
    property Items[Index : Integer] : Integer read GetItem write SetItem;
    property Plain : Integer read FPlain write FPlain;
  end;
  ..
  function TMyClass.GetItem(Index : Integer) : Integer;
  begin
    Result := FItems[Index];
  end;

  procedure TMyClass.SetItem(Index, Value : Integer);
  begin
    FItems[Index] := Value;
  end;

  procedure TMyClass.MyProc;
  begin
    if Items[0] < 5 then  //!! ineffective, calls GetItem
      Items[0] := 5;      //!! ineffective, calls SetItem

    if FItems[0] < 5 then //!! this is faster
      FItems[0] := 5;

    Plain := 5;  // these two lines generate the same machine code
    FPlain := 5;
  end;
```

See also:

Optimizations

4.4.5 OPTI5-Virtual methods (procedures/functions) that are not overridden

Virtual methods (procedures/functions) that are not overridden (OPTI5)

This is a list of all methods that are declared as virtual, but that never are overridden. Since virtual methods have slightly worse performance than static methods, it is better to change these methods to static ones instead.

This section is also generated for multi-projects.

Recommendation:

Examine if these methods should really be overridden. If they belong to a base class, you should probably keep them virtual, so descendant classes can create their own implementations.

See also:

Optimizations

4.4.6 OPTI6-Local subprograms with references to outer local variables

Local subprograms with references to outer local variables (OPTI6)

This section shows nestled local procedures, with references to outer local variables. Those local variables require some special stack manipulation so that the variables of the outer routine can be seen by the inner routine. This results in a good bit of overhead.

See also:

Optimizations

4.4.7 OPTI7-Subprograms with local subprograms

Subprograms with local subprograms (OPTI7)

This section lists subprograms that themselves have local subprograms. Especially when these subprograms share local variables, it can have a negative effect on performance.

See also:

Optimizations

4.4.8 OPTI8-Parameter is "var", can be changed to "out"

Parameter is "var", can be changed to "out" (OPTI8)

This section lists parameters that are marked with the "var" directive, but that can be changed to "out".

Even if it may not improve performance, it improves the readability of the code and makes its intentions clearer.

See also:

Optimizations

4.4.9 OPTI9-Inlined subprograms not inlined because not yet implemented

Inlined subprograms not inlined because not yet implemented

This section lists calls to inlined subprograms, where the subprogram will not be inlined. The reason is that the subprogram has not been implemented yet. It is implemented further down in the same module. There are a number of other conditions that also must be fulfilled for the subprogram to be inlined.

```
..

type
  TMyClass = class
  private
     ..
     procedure Process;
     function GetValue : Integer; inline;
     procedure MoreProcessing;
     ..
  end;

  ..

  procedure TMyClass.Process;
  var
     Value : Integer;
  begin
     ..
     Value := GetValue; // this call cannot be inlined!
     ..
  end;

  function TMyClass.GetValue : Integer;
  begin
    ..
  end;

  procedure TMyClass.MoreProcessing;
  var
     Value : Integer;
  begin
     ..
     Value := GetValue; // this call can be inlined!
     ..
  end;

..
```

To make the subprogram inlined for this call, make sure that it is implemented higher up in the same module.

See also:

Optimizations

4.4.10 OPTI10-Managed local variable that can be declared inline

Managed local variables that can be declared inline

This section lists local managed variables that can benefit from being declared inline instead of in the main var-section. For example, if the variable is only needed in some section of the function, the initialization-finalization code only needs to be run under some circumstances.

Managed variables are strings, interfaces, dynamic arrays and records that contain managed fields.

Inline variable declarations were introduced in Delphi 10.3, so this report section is not relevant for older targets.

See also:

Optimizations

4.4.11 OPTI11-Managed local variable is inlined in loop

Managed local variables is inlined in loop

This section lists local variables that instead of being declared in the main var-section, are declared inline.
They are declared inside a loop, which decreases performance. The reason is that initialization-finalization of the variable has to be done for each loop iteration.

Managed variables are strings, interfaces, dynamic arrays and records that contain managed fields.

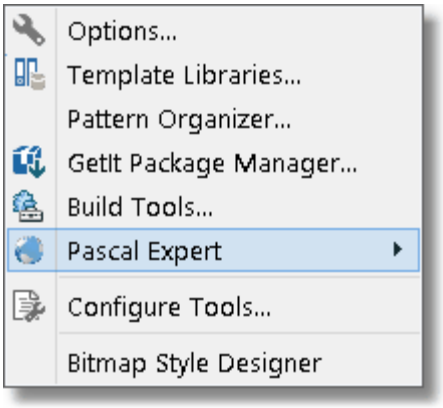
Inline variable declarations were introduced in Delphi 10.3, so this report section is not relevant for older targets.

See also:

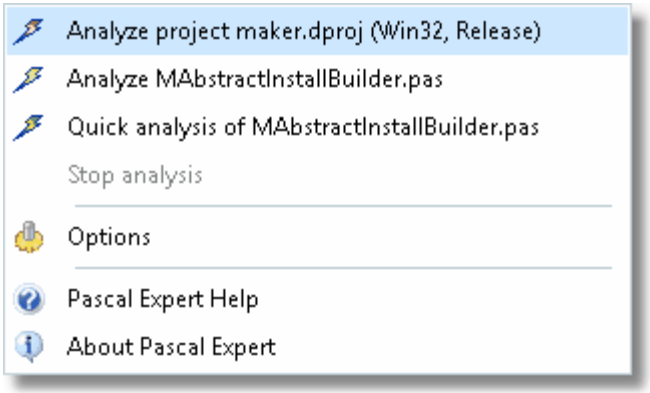
Optimizations

5 Menu items

The Pascal Expert menu is located in RAD Studio's "Tools" menu:



These are the available menu items for Pascal Expert:



You can also use a toolbar with Pascal Expert buttons. You can toggle it on/off in the Options dialog.

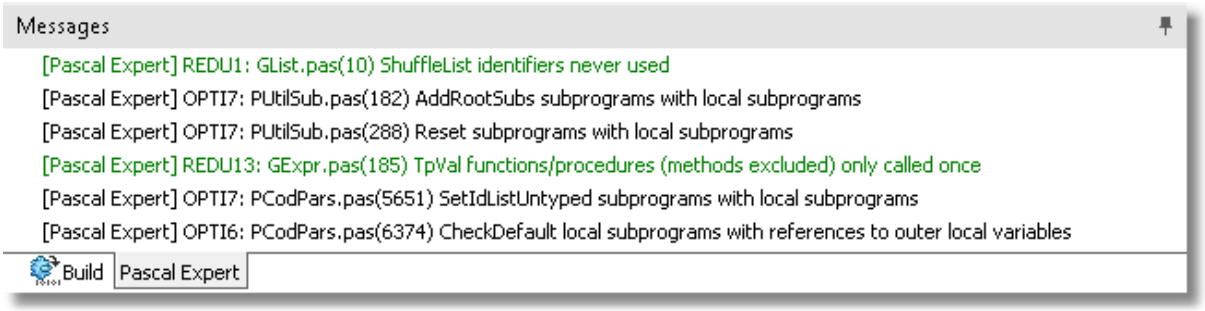
Analyze project
Analyze module
Quick analysis of module
Stop
Options
Help for Pascal Expert
About Pascal Expert

See also:

Introduction
Known limitations

5.1 Analyze project

Selecting this menu will let Pascal Expert analyze the currently active project and the active configuration (like Debug, Release etc) and target (Win32, Win64, OSX etc). Messages will be written to the output window:



If you have not activated your copy of Pascal Expert, you will be prompted to do so. Normally your copy has been activated during installation. But if you skipped it during installation, for example because you need to activate with a manual file, you can here create an activation request file. The request file is sent to us and we will return a response file which you use to activate manually (without needing access to Internet).

When analyzing, the source code will be scanned. Pascal Expert will read the project's settings from its *.dproj file. Any units specified in the *uses* statement of a unit, will be read, if they are found. This is a recursive process, and potentially a lot of source code can be involved. Any form files (DFM) will be parsed and examined as well.

If any include directives (like {$I MYSOURCE.INC}) are found in the source code, these files will also be read and parsed.

Even if Pascal Expert can detect many syntax errors, it is required that the code is possible to compile. Otherwise, the results may be incorrect and possibly misleading. So, as a rule, always make sure that the code compiles, before running Pascal Expert.

Please note 1: You should save all editor and project files before analyzing. The reason is that Pascal Expert reads all files directly from disk, even if they are currently loaded in the Delphi IDE. This means, that if you have created a new project and never saved it, Pascal Expert will fail with an error message, when trying to load the expected project files.

Please note 2: When starting a new analysis, the results of the previous analysis will be cleared. If you want to keep them, for example, press **Ctrl+A** to select all messages and then **Ctrl+C** to copy them to the clipboard.

If you for some reason do not want Pascal Expert to analyze a code block; use the conditional define _PEGANZA_ for this purpose:

```
{$IFNDEF _PEGANZA_}

.. code

{$ENDIF}
```

Since _PEGANZA_ is always defined automatically by Pascal Expert, the code in the example above will not be parsed. In all other cases it will be included (unless you explicitly define _PEGANZA_, which would be pointless).

See also:

Menu items
Analyze module
Quick analysis of module
Stop
Options
Help for Pascal Expert
About Pascal Expert

5.2 Analyze module

This menu item will start an analysis of the currently focused module (source unit) in the editor. It will only report issues for this module, but it will scan all dependant source files.

Some of the issues are either not relevant or subject to false negative or positive results, and will not be examined for this kind of analysis. For best and complete results: Select Analyze project instead.

Issues that are always excluded:

STWA2-Ambiguous unit references
WARN1-Interfaced identifiers that are used, but not outside of unit
WARN2-Interfaced class identifiers that are public/published, but not used outside of unit

Issues that are excluded if the examined identifier is declared in the interface section, or if the examined identifiers has references to unknown identifier:

REDU1-Identifiers never used
REDU12-Functions called only as procedures (result ignored)
REDU13-Functions/procedures (methods excluded) only called once
REDU14-Methods only called once from other method of the same class

OPTI5-Virtual methods (procedures/functions) that are not overridden

Issues that are often excluded if the examined identifiers has references to unknown identifier:

WARN3-Variables that are referenced, but never set
WARN5-Variables that are set, but never referenced
WARN7-Local variables that are referenced before they are set
WARN9-Var parameters that are used, but never set
WARN11-Value parameters that are set
WARN40-Local for-loop variables read after loop
WARN46-Local variables that are set but not later used
WARN51-Out parameter is read before set

REDU2-Local identifiers only used at a lower scope
REDU3-Local identifiers only used at a lower scope, but in more than one subprogram
REDU4-Local identifiers that are set and referenced once
REDU6-Local identifiers that are set more than once without referencing in-between

OPTI8-Parameter is "var", can be changed to "out"

See Analyze project for more information about how analyzing works.

See also:

Menu items
Analyze project
Quick analysis of module
Stop
Options
Help for Pascal Expert
About Pascal Expert

5.3 Quick analysis of module

This menu item will start an analysis of the currently focused module (source unit) in the editor. It will only report issues for this module, and it will only scan the module itself.

It will be much quicker than the other types of analysis, but the results will not be so complete.

Some of the issues are either not relevant or subject to false negative or positive results, and will not be examined for this kind of analysis. For best and complete results: Select Analyze project instead.

Issues that are always excluded:

STWA2-Ambiguous unit references

WARN1-Interfaced identifiers that are used, but not outside of unit
WARN2-Interfaced class identifiers that are public/published, but not used outside of unit

Issues that are excluded if the examined identifier is declared in the interface section, or if the examined identifiers has references to unknown identifier:

REDU1-Identifiers never used
REDU12-Functions called only as procedures (result ignored)
REDU13-Functions/procedures (methods excluded) only called once
REDU14-Methods only called once from other method of the same class

OPTI5-Virtual methods (procedures/functions) that are not overridden

Issues that are often excluded if the examined identifiers has references to unknown identifier:

WARN3-Variables that are referenced, but never set
WARN5-Variables that are set, but never referenced
WARN7-Local variables that are referenced before they are set
WARN9-Var parameters that are used, but never set
WARN11-Value parameters that are set
WARN40-Local for-loop variables read after loop
WARN46-Local variables that are set but not later used
WARN51-Out parameter is read before set

REDU2-Local identifiers only used at a lower scope
REDU3-Local identifiers only used at a lower scope, but in more than one subprogram
REDU4-Local identifiers that are set and referenced once
REDU6-Local identifiers that are set more than once without referencing in-between

OPTI8-Parameter is "var", can be changed to "out"

See Analyze project for more information about how analyzing works.

See also:

Menu items
Analyze project
Analyze module
Stop
Options
Help for Pascal Expert
About Pascal Expert

5.4 Stop

Select this menu item to stop the process, for example when you realize that you want to change your selections. Pascal Expert will immediately stop sending messages to the

output window. Messages that already have been displayed will still be available.

See also:

Menu items
Analyze project
Analyze module
Quick analysis of module
Options
Help for Pascal Expert
About Pascal Expert

5.5 Options

This is where you select options for Pascal Expert. The settings are automatically saved to an INI file, so will be available the next time you start the Delphi IDE.

Each Delphi version that runs Pascal Expert will have its own INI-file. For example, Delphi XE8 will save its settings in PEXXE8.INI.

The options dialog is divided in a number of tab pages:

General settings
Report settings
Alerts
Reductions
Optimizations
Conventions

The tab page that was last selected will be selected the next time you enter this dialog (during the same session).

OK
Confirms your selections and saves them.

Cancel
Cancels your selections. Whatever selections you have made in the dialog, those will be lost.

Set Defaults
Loads default values (factory settings).

Load Options
Loads options from the selected file.

Use **Load Options** and **Store Options** to use custom set of settings for different purposes.

E.g. for one particular project MyProj you may want to report all types of problems, and exclude some folders.
Then store the settings for this project in the file MyProj.ini. When analyzing this project, make sure to first load MyProj.ini.

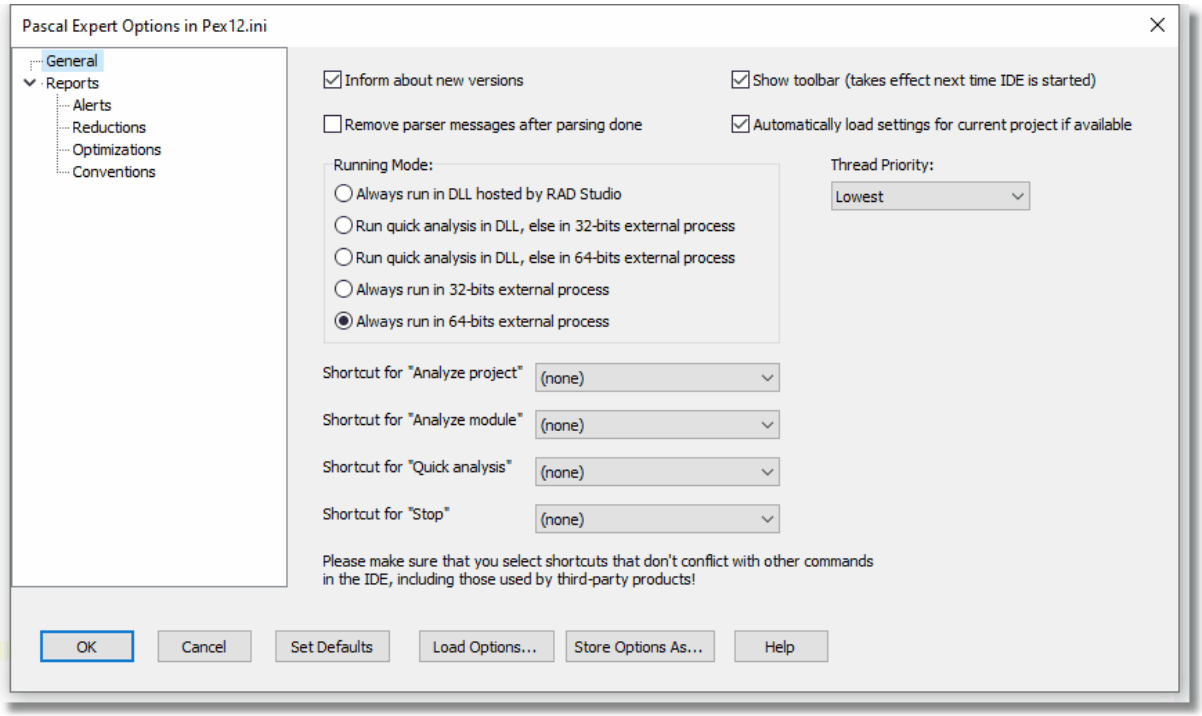
Store Options As
Stores current options to the selected file.

See also:

Menu items
Analyze project
Analyze module
Quick analysis of module
Stop
Help for Pascal Expert
About Pascal Expert

5.5.1 General settings

On this tab page, you will find general settings for Pascal Expert.



Inform about new versions
Default=Yes

If you want to be notified when there is an upgrade for your Pascal Expert installation,

select this option. Whenever you select "Analyze", a message box will be displayed, if there is a newer version of Pascal Expert. For this to work, needless to say, you will need an active Internet connection.

Show toolbar (takes effect next time IDE is started)
Default=Yes

Mark this checkbox if you want a toolbar to be used. Normally, you can use the normal customize feature in RAD Studio under View|Toolbars to turn on/off the toolbar. But sometimes it is needed to use this checkbox to effectively remove the toolbar.

Remove parser messages after parsing
Default=Yes

If this option is selected, Pascal Expert will clear the output window from the status messages generated while parsing, before outputting report results. This will reduce the number of lines in the output window.

The status messages generated while parsing, which are removed, normally look like:

"Parsing unit interface in ..."
"Parsing unit implementation in ..."

Automatically load settings for current project if available
Default=No

This is an advanced feature. It only makes sense to turn on if you have created additional settings files (with the "Store Options As"-button), and saved those to the default folder for settings (C:\Documents and Settings\<acc>\My Documents\Peganza\Pascal Expert).

Activating this option will then let Pascal Expert automatically load settings when analyzing a project (not analyzing a module or a quick analysis).
But it requires that the INI-file with settings has the same name as the analyzed project.

Example:

You have created a settings file MyProj.ini and saved it in the default folder for settings. When you make MyProj.dproj the selected project and starts an analysis, Pascal Expert will load settings from MyProj.ini.

Running Mode
Default=Always run in external process

Introduced in version 8.1, this option allows you to select in what way the analysis will be performed. As default, analysis is done by an external process. There are two different EXE files that will run the analysis: PexRunner32.exe or PexRunner64.exe. When using a 64-bit system, it is recommended that you run the analysis with PexRunner64.exe.

Thread Priority
Default=Normal

This option sets the priority for the thread that is created when "Analyze" is selected. The higher priority you give this thread, the faster the analysis will run. A drawback will be that other operations during the analysis will be slower.

This setting has most importance when the analysis is done in a DLL (see "Running Mode" above).

Shortcut for "Analyze project"
Default=(none)

Select the shortcut you want to use for the menu item "Analyze project".

Shortcut for "Analyze module"
Default=(none)

Select the shortcut you want to use for the menu item "Analyze module".

Shortcut for "Quick analysis"
Default=(none)

Select the shortcut you want to use for the menu item "Quick analysis".

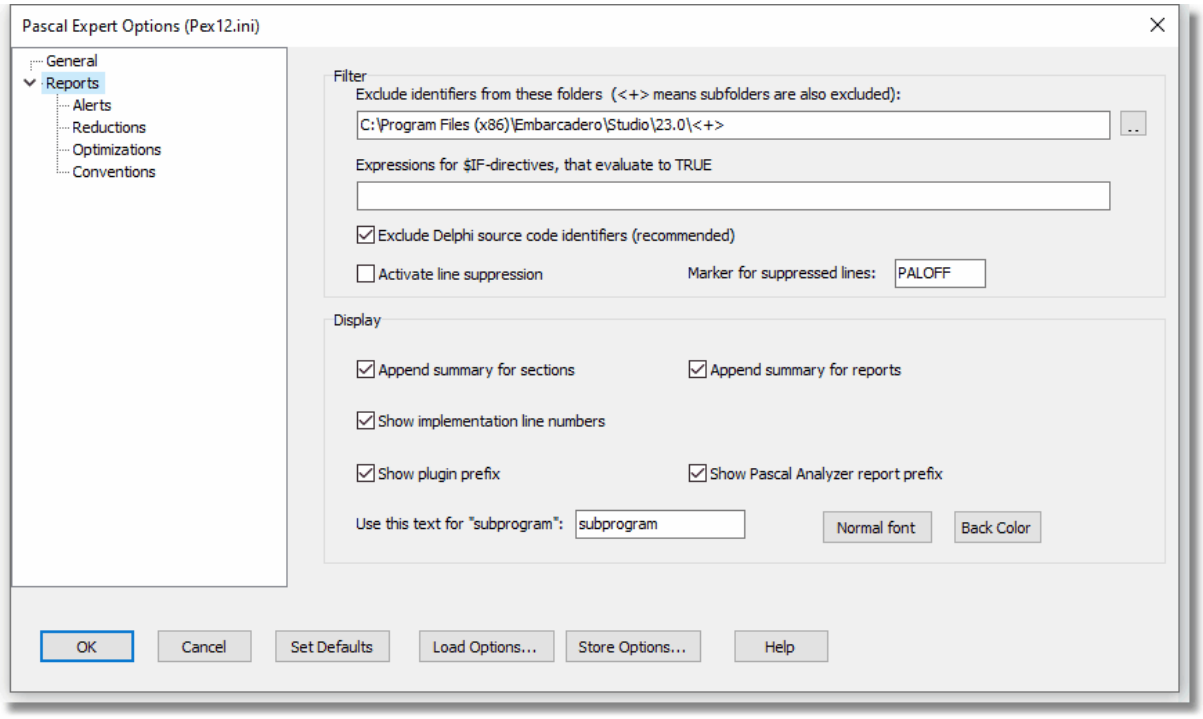
Shortcut for "Stop"
Default=(none)

Select the shortcut you want to use for the menu item "Stop".

See also:

Options menu

5.5.2 Report settings



On this tab page there are options for how reports are generated:

Exclude identifiers from these folders (<+> means subfolders are also excluded)

Identifiers declared in source code from these folders will not appear in the output.

Enter excluded folders separated with a semicolon e. g.:

c:\source\myunits;c:\source\generic

Alternatively, press the ellipsis button to select the folders.

It is possible to select that an exclude folder should also apply to its subfolders. When subfolders are excluded, the folder name is suffixed with "<+>".

Environmental variables set in the Delphi IDE, may also be used.

Expressions for $IF-directives, that evaluate to TRUE
An $IF-directive in code is followed by an expression, that evaluates to TRUE or FALSE. If you use $IF-directives, you must supply all expressions that evaluate to TRUE, because Pascal Expert cannot always determine the value of an expression. Enter the expressions separated with semicolons, like:

RTLVersion > 14;Declared(Windows)

Please observe that you do not need to include Defined-directives like "Defined(MSWINDOWS)", because Pascal Expert manages to evaluate those directives.

When Pascal Expert's parser finds a $IF-directive in code, it will try to evaluate it. If it is an expression that you have supplied, it will be evaluated to TRUE, otherwise it will be evaluated as FALSE.

Exclude Delphi source code identifiers (recommended)
Default=Yes

If checked, identifiers in Delphi source code will not be reported. For this to work, the source code should be installed in the same folder branch as Delphi itself (normally under "C:\Program Files (x86)", which is the default location). If not, you should add this folder to the list of excluded folders (see "Exclude identifiers from these folders" above).

It is recommended to keep this option turned on, to avoid a lot of warnings for Delphi source code, which you probably are not interested in fixing anyway.

Activate line suppression
Default = False
Check this option if you want to suppress lines that are marked with the suppressed lines maker (see the following text block).

N.B. Only activate this option if you really have lines marked, because the process will be slower.

Suppressed lines-Marker for suppressed lines
Default = PALOFF

By adding a comment:

```
//PALOFF
```

.. as a comment to a source code line, means that Pascal Expert will not report any issues encountered on that line. If you place the comment on a line where an identifier is declared, that identifier will not be reported. This is the most effective way to get rid of all issues for an identifier.

Note also that you can use curly brackets, like "{PALOFF}". Like for "//" blanks are allowed, for example "{ PALOFF }".

Example

```
type
  TGlobalDLLData = record
    Path : string[255];
    LineNr : Integer; //PALOFF (1-based line number in source module)
  end;
```

In the code example above, LineNr will not be reported. Observe that the marker must be first in the comment string on the line, and that it is allowed to add comment text to

the right of the marker.

In some report sections you will find that even if you place the comment on the line which you believes make Pascal Expert report the identifier, it will not have any effect. Then try placing the comment on the line where the identifier is declared.

It is possible to select what string should be used as the marker. Default value for the suppression marker is "PALOFF", but you can change this to something else. Blank spaces between "//" and the suppression marker are allowed. You can also have more text to the right of the marker, like:

```
//PALOFF because false warning otherwise
```

To select only some report sections that will be excluded use this syntax:

Examples:

//PALOFF WARN8 *(report section WARN8 will not be reported)*

//PALOFF WARN2;OPTI8;OPTI2 *(report sections WARN2, OPTI8, OPTI2 will not be reported)*

//PALOFF OPTI *(all report sections for the Optimization Report will be excluded)*

//PALOFF STWA2;WARN;OPTI4 *(STWA2 and OPTI4 will be excluded, plus all sections in the Warnings Report)*

Append summary for sections
Default = No

If selected, also total issues for report sections will be displayed.

Append summary for reports
Default = No

If selected, also total issues for reports will be displayed.

Show implementation line numbers
Default = Yes

This option determines if the line number where a subprogram is declared is displayed in the reports. If set to Yes, instead implementation line number is displayed. It has meaning if you double-click on the line to jump to the source code. Either it will then take you to the declaration or the implementation line. Often you would probably prefer to reach the implementation. If so, then set this option to Yes.

Show Pascal Analyzer report prefix
Default = Yes

This option lets each issue include a prefix, like "WARN12", to show which report and section the issue is related to.

Show plugin prefix
Default = Yes

This option prefixes each issue with "[Pascal Expert]".

Use this text for "subprogram"
Default = subprogram

If you want to use another string than "subprogram" in output messages, you can enter the string here, for example "function". Enter it as the singular term with small letters.

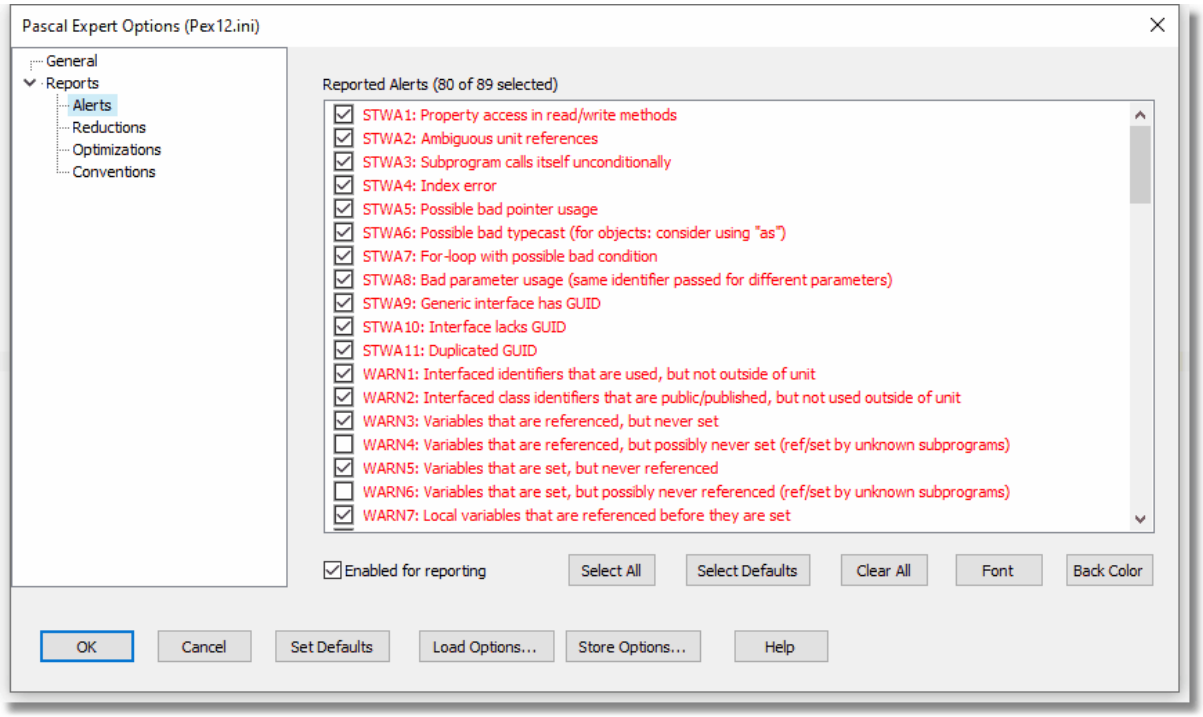
Normal font
Select font settings, including foreground color for status messages in the output window.

Back Color
Select the background color for status messages in the output window.

See also:

Options menu
General settings
Alerts
Reductions
Optimizations
Conventions

5.5.2.1 Alerts



Select which alerts that you want to display. See Alerts.

Enable for reporting
Default=Yes

This checkbox is a convenient way to temporarily turn off items for reporting, without clearing the selections.

Select All
Marks all report sections as selected.

Select Defaults
All sections that should by default be selected, are checked, otherwise unchecked. Only selections on the currently active tab page will be affected. If you want to revert **ALL** settings to the default factory settings, press the "Set Defaults" button in the bottom of the dialog.

Clear All
Uncheck all report sections.

Font
Select font settings, including foreground color for the sections in this report category. This determines how the messages will appear in the output window. The settings are (except for size), reflected in the list box.

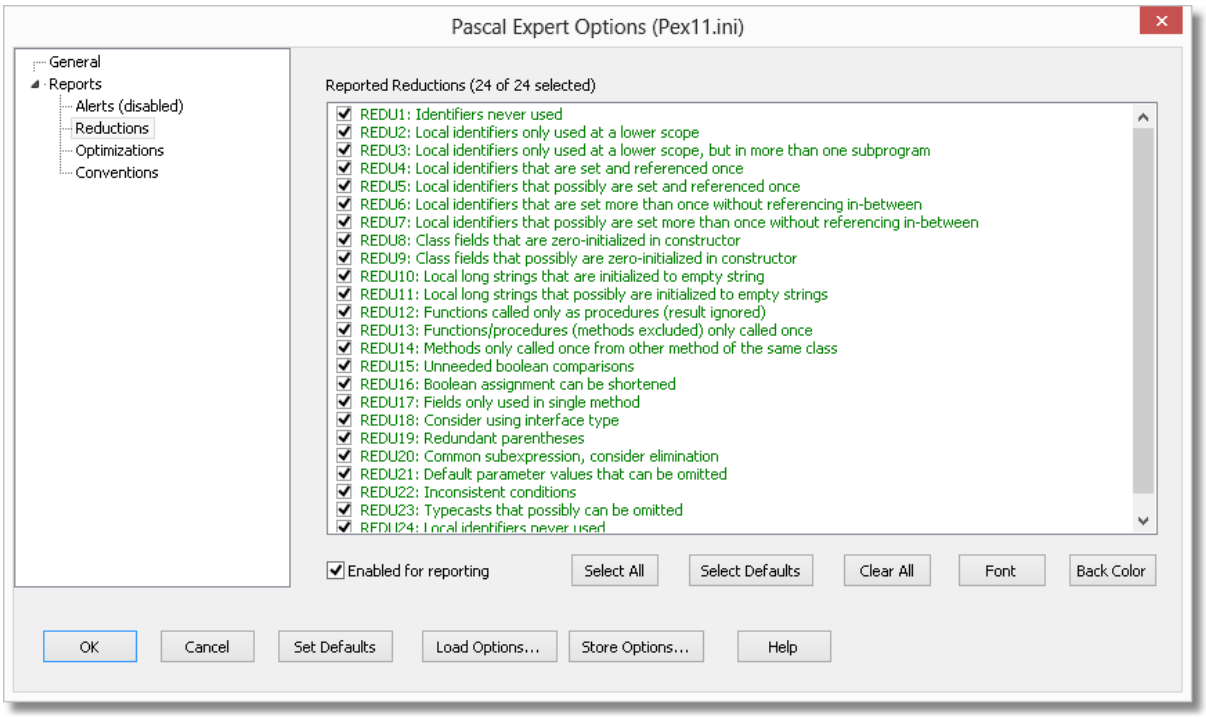
Back Color
Select the background color for the sections in this report category. The selected back color will be used for messages in the output window.

See also:

Reports
Reductions
Optimizations
Conventions

5.5.2.2 Reductions

Select which reductions that you want to display. See Reductions.



Enable for reporting
Default=Yes

This checkbox is a convenient way to temporarily turn off items for reporting, without clearing the selections.

Select All
Marks all report sections as selected.

Select Defaults
All sections that should by default be selected, are checked, otherwise unchecked. Only selections on the currently active tab page will be affected. If you want to revert **ALL** settings to the default factory settings, press the "Set Defaults" button in the bottom of the dialog.

Clear All
Uncheck all report sections.

Font
Select font settings, including foreground color for the sections in this report category. This determines how the messages will appear in the output window. The settings are (except for size), reflected in the list box.

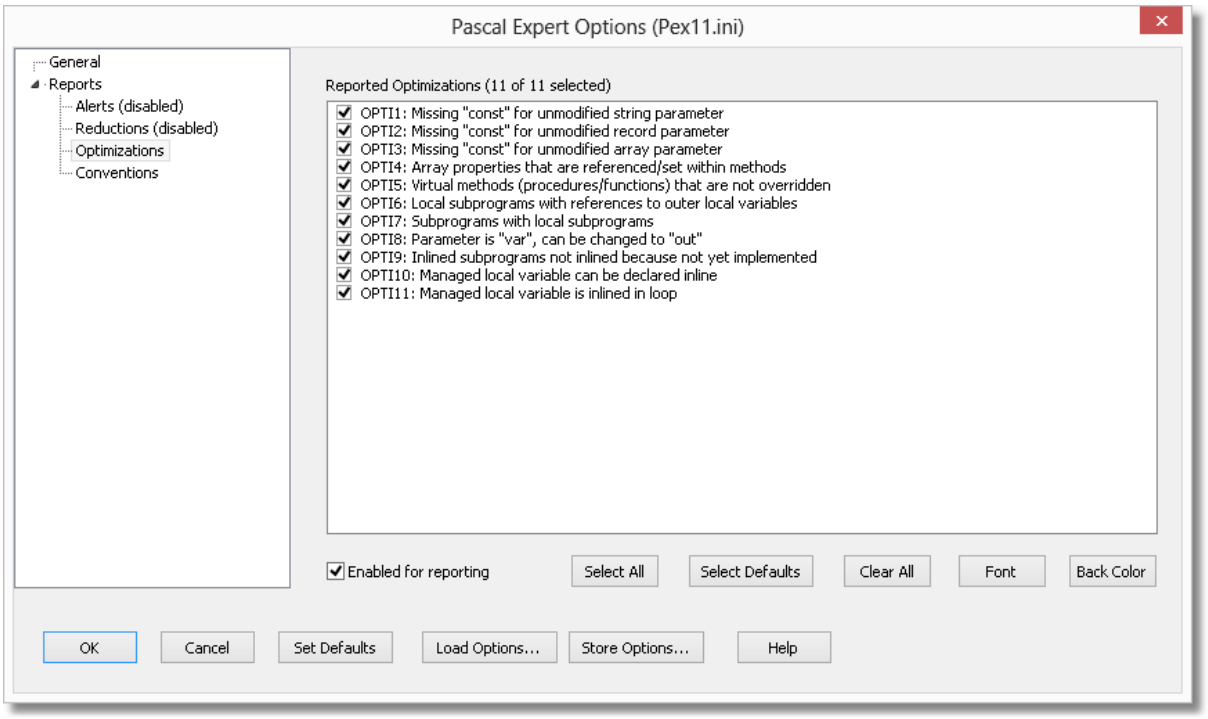
Back Color
Select the background color for the sections in this report category. The selected back color will be used for messages in the output window.

See also:

Reports
Alerts
Optimizations
Conventions

5.5.2.3 Optimizations

Select which optimizations that you want to display. See Optimizations.



Enable for reporting
Default=Yes

This checkbox is a convenient way to temporarily turn off items for reporting, without clearing the selections.

Select All
Marks all report sections as selected.

Select Defaults
All sections that should by default be selected, are checked, otherwise unchecked. Only selections on the currently active tab page will be affected. If you want to revert **ALL** settings to the default factory settings, press the "Set Defaults" button in the bottom of the dialog.

Clear All
Uncheck all report sections.

Font
Select font settings, including foreground color for the sections in this report category. This determines how the messages will appear in the output window. The settings are (except for size), reflected in the list box.

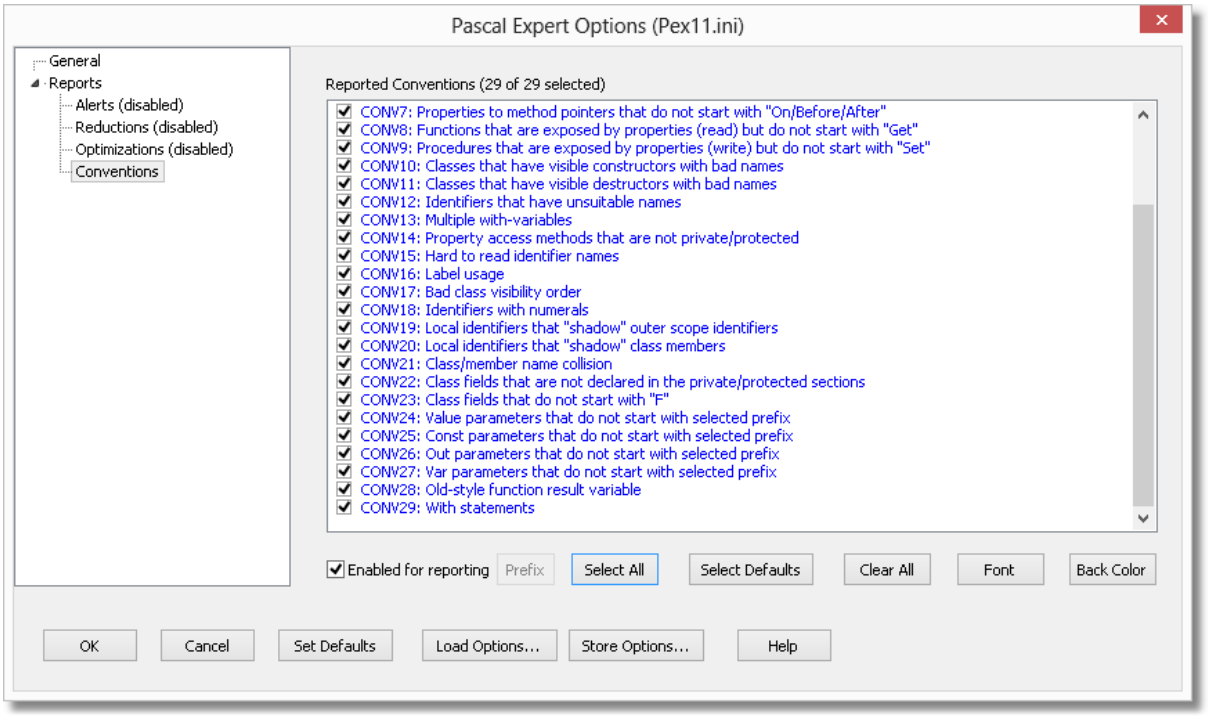
Back Color
Select the background color for the sections in this report category. The selected back

color will be used for messages in the output window.

See also:

Reports
Alerts
Reductions
Conventions

5.5.2.4 Conventions



Select which conventions that you want to display. See Conventions.

Enable for reporting
Default=Yes

This checkbox is a convenient way to temporarily turn off items for reporting, without clearing the selections.

Prefix
When any of the CONV24, CONV25, CONV26 or CONV27 report sections are selected in the list, press this button to set the prefix string.

Select All
Marks all report sections as selected.

Select Defaults
All sections that should by default be selected, are checked, otherwise unchecked. Only selections on the currently active tab page will be affected. If you want to revert **ALL** settings to the default factory settings, press the "Set Defaults" button in the bottom of the dialog.

Clear All
Uncheck all report sections.

Font
Select font settings, including foreground color for the sections in this report category. This determines how the messages will appear in the output window. The settings are (except for size), reflected in the list box.

Back Color
Select the background color for the sections in this report category. The selected back color will be used for messages in the output window.

See also:

Reports
Alerts
Reductions
Optimizations

5.6 Help for Pascal Expert

Display help for Pascal Expert from the PEX.CHM file, located in the program directory for Pascal Expert. You can also press F1 when an output message is selected, and the relevant topic will be displayed.

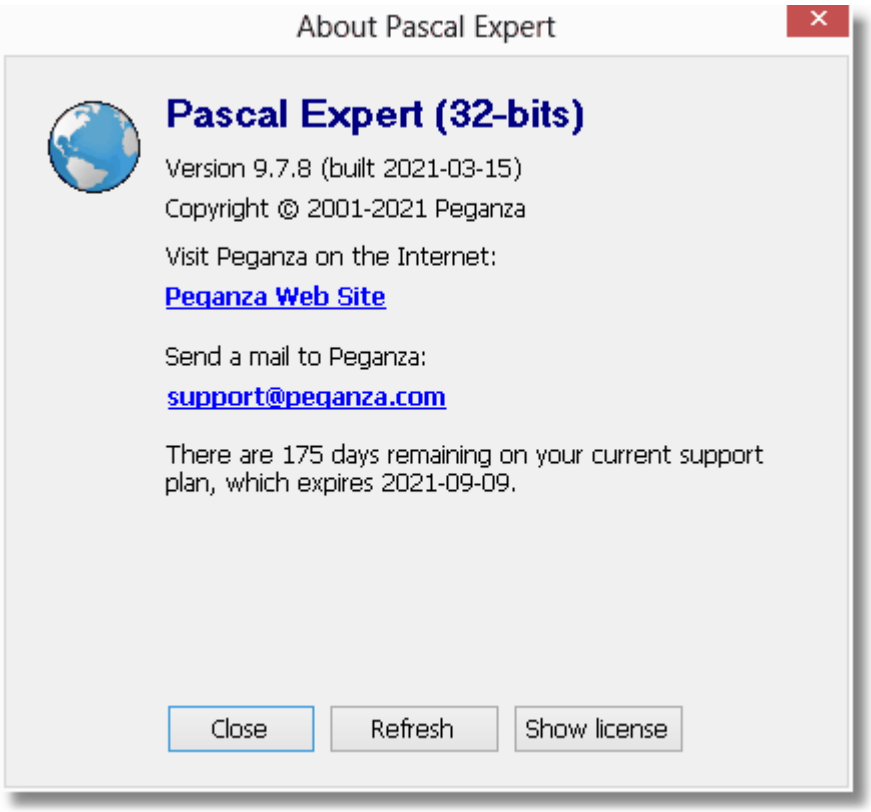
In the program folder for Pascal Expert, you can also find the help texts in a PDF file, if you prefer to read in that format. On our web site there is also an online version of the help system.

See also:

Menu items
Analyze project
Analyze module
Quick analysis of module
Stop
Options
About Pascal Expert

5.7 About Pascal Expert

This menu item displays the about box dialog for Pascal Expert. For example, you can see which version of Pascal Expert that is installed. You can also see when your current support plan expires, or view your license number.



When initially buying Pascal Expert, you get a full year (plus some extra bonus days) of support, including access to all minor and major upgrades. After this period, you can buy support plans for one or more additional years, at our web site. You must afterwards click **Refresh license info** to display the current support plan expiration date. What happens when you click this button, is that Pascal Expert tries to contact our activation servers and refresh your local license information that is displayed.

When selecting "Analyze" in the menu, you will also be prompted to activate your license if needed. This is the case if you have not entered your product key while installing Pascal Expert.

See also:

Menu items
Analyze project
Analyze module
Quick analysis of module
Stop

Options
Help for Pascal Expert

Index

- R -

- S -

- T -

- U -

- V -