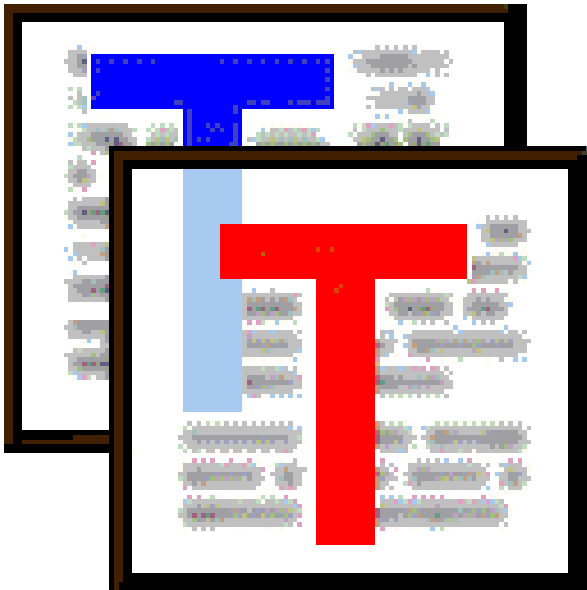


# Delphi2C#

© 2020 Dr. Detlef Meyer-Eltz



# 1 Introduction

## Short description

*Delphi2C#* helps to convert Delphi source code to C#. Both languages are designed by the Danish software engineer Anders Hejlsberg and have a lot of concepts and even classes in common. However C# uses managed code while some basic units of the Delphi RTL heavily makes use of pointers. The core of the Delphi RTL is rebuilt for C# and some classes to simulate pointers are provided too. So the translation of "normal" user code works well. Nevertheless a manual post-processing of the produced code still will be required. It is aim of the program to keep the amount of the post-processing as small as possible.

*Delphi2C#* is built on the experience with the earlier "Delphi2Cpp" and the current "DelphiXE2Cpp11". The main difference between these converters to C++ and *Delphi2C#* concerns the use of pointers. A more detailed comparison of *Delphi2C#* and *Delphi2Cpp/DelphiXE2Cpp11* is here.

## Availability

The actual version of *Delphi2C#* can be obtained from the TextTransformer websites:

<http://www.TextTransformer.com>

<http://www.TextTransformer.de>

# 2 Installation

The installation is done by the installer *Delphi2C#Install.exe*. All files for projects, examples, source code etc. are copied into the chosen installation directory.

The default path is a sub-folder *Delphi2C#* in the user documents folder, like:

C:\Users\User\Documents\Delphi2C#

Regardless of the path, that you chose for the installation, the license file *Delphi2C#Lic.dat* will be written at that default path.

# 3 Registration

If you have bought a license of *Delphi2C#*, **you will get a link to a version of *Delphi2C#*, which you can register.**

The **registration** of *Delphi2C#*, i.e. the removal of trial limitations and the permanent activation of the features, has to be done by the menu: Help->*Registration*. Following dialog is shown:

The screenshot shows a 'Register' dialog box. At the top right, there is a 'Buy Now' button. Below it, a yellow text box contains the message: 'Please insert the registering information as you got it by e-mail'. The main area of the dialog contains several input fields: 'Username', 'Company', 'Program ID' (with a placeholder 'xxxx-xxxx-xxxx-xxxx-xxxx-xxxx-xxxx-xxxx' and a copy icon), 'Status' (set to 'Trial'), and 'Key'. At the bottom, there are 'Cancel' and 'Register' buttons.

If you are on line and click the **Buy Now** button, a webpage is show, where you can transmit a **user name** (at least eight characters), a **company name** and your address details and the details on the method of payment. In addition the **program ID** is required, which is shown in the dialog instead of "xxxx-xxxx-xxxx-xxxx-xxxx-xxxx-xxxx-xxxx". . This program ID is copied into the clipboard if you click the button at the right.

The program ID is specific for your hardware configuration. It's also called the *machines fingerprint*. The registered software can be executed only on the computer on which it originally was installed.


After the check of your credit card has been carried out, an **e-mail** which includes the registration data (user name, company and key) is sent to you automatically. You also will get a link to a version of Delphi2C#, which you can register.

**It is important to know that if you are downloading Delphi2C# to use on a different computer than the one on which you originally downloaded it, you should transfer it immediately onto removable media, and *not* register it on the first computer.**

**User name, Company** and the **key** then have to be copied unchanged from the e-mail into the corresponding entry fields of the dialog box. Then a click on the **button Register** will close the dialog automatically and a message appears, which confirms the success of the registration. A license file *Delphi2C#Lic.dat* is created now in the user documents Delphi2C# folder.

If the program is registered already the **Register button** will not be shown any more.

## 4 How to start

You will get good C# translations of your Delphi code only, if you make the correct settings in dialog for the translation options, which can be shown by the button .

### 1. Paths to your code

Delphi2C# has to know the types and signatures of procedures and functions in your Delphi source code to make a correct translation. That's no problem as far as these information stems from your own source code. You simply have to set the paths to your source code at the in the options dialog.

### 2. Paths to the Delphi RTL/VCL

The paths to the files of the used Delphi RTL/VCL also have to be set into a Delphi2C# project. If you are using Delphi2C# for the first time and you are curious to get some first results, you may select the paths to the original Delphi RTL/VCL as search path for the files not to convert. But unfortunately the original Delphi source code has bugs and in longer term it is recommended, that you prepare a copy of Embarcadero's code.

### 3. Extended System.pas

You should use also an "extended System.pas". This file corrects and completes the original "System.pas".




### 4. Setting the correct definitions


If you have selected the search paths to the Delphi RTL/VCL, your code still will not be translated correctly, if you haven't set the necessary definitions.

As default *MSWINDOWS* is defined. If that would not be the case, even the original Sysutils.pas could be parsed, because e.g. the following code, would not be valid:

```
function AdjustLineBreaks(const S: string; Style: TTextLineBreakStyle =
    {$IFDEF LINUX} tlbsLF {$ENDIF}
    {$IFDEF MSWINDOWS} tlbsCRLF {$ENDIF}): string;
```

### 5. Starting the translation

After you have set your translation options you can save them by the button  and open the first file to to translate with the button . The source file is shown in the left window of the user interface. You can start the translation with the button . As soon as it is finished the C# source code is shown in the windows on the right side of the application. Also the content **on the left side** might have changed:

now the preprocessed Delphi code is shown there. You can save the translated code by the button .

## 5 User interface

The main window of the Delphi2C# application consists in a menu, a tool bar and in three windows for the input and for the output.

--



By this button the texts in all windows is cleared and then you are asked, whether the type information that was learned from the previous translations shall be cleared too.

--



This button does the same as the previous and than inserts the frame for a new unit. So you can quickly write some code snippets into the frame, to translate them.

--

You can load a Delphi source file into the first window by CTRL+O or by the button:



--

Before you start the translation, you can set some options in the according dialog, which is shown by the button



--

Options can be saved and reloaded by the buttons



--

There are two buttons which can have two states each. If the *PP*-button is down, the preprocessor is enabled, if the *PP*-button is up, the preprocessor is disabled. If the *T*-button is down, the translator is enabled, if the *T*-button is up, the translator is disabled.



You can disable the translator either to check the preprocessing of a source file. But the feature to disable the translator mainly has been implemented, to give you the possibility to create a

preprocessed copy of the VCL or your Delphi source files, by means of the file manager. By use of preprocessed files the repeated **translation can be accelerated**. If you chose the search paths to the directories with the preprocessed VCL and you also select your preprocessed Delphi sources, only enabling of the translator suffices for translation and the time for the pre-processing is saved. **If parts of your files aren't preprocessed, you have to enable both, the preprocessor and the translator.** This will still be faster than don't to use preprocessed files, because the preprocessor hardly needs time to preprocess files again, which already were preprocessed. The initial state of these buttons is saved with the options. The *overwritten System.pas* gets always preprocessed, even if the button is disabled.

--

The translation is started with F9 or



--

The dialog for the translation of groups of files is shown by the button:



--

All information that once has been obtained from the interface parts of the processed files is remembered for the translation of further files. Types and variables can be cleared by the button:



--

Finally you can save the generated C# code by CTRL+S or by



At first a file dialog for the header appears and as soon as you have saved the header file the dialog appears again for the C# source file. If the translated file is a library, the file dialog appears for a third time, to save a module definition file.

--



Shows a dialog to find expressions in the text of the actual window.

--



Shows the position, where the parser found an error in the Delphi code.

--

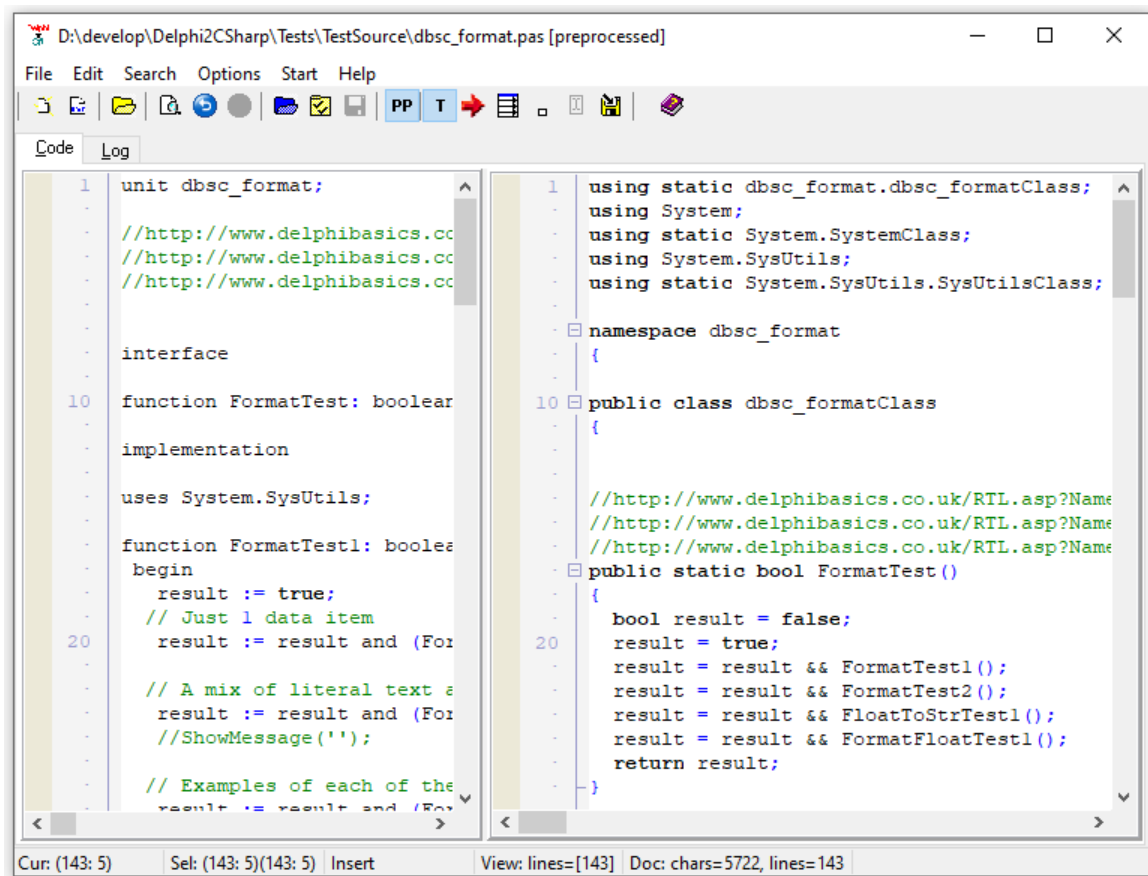
This help is shown with F1 or by the button



## 5.1 Windows

There are two windows in the user interface:

1. the left window shows the Delphi source code or the pre-processed code, after a translation has been executed.
2. the right window shows the generated C# source code

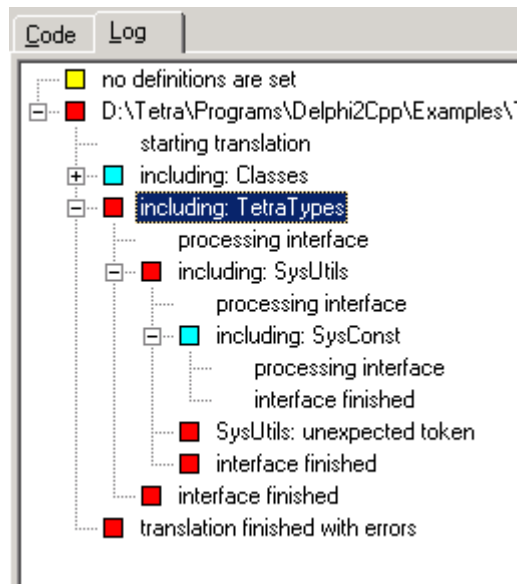


```
1 unit dbsc_format;
.
. //http://www.delphibasics.co
. //http://www.delphibasics.co
. //http://www.delphibasics.co
.
. interface
.
. 10 function FormatTest: boolean;
.
. implementation
.
. uses System.SysUtils;
.
. function FormatTest1: boolean;
. begin
.   result := true;
.   // Just 1 data item
. 20   result := result and (FormatTest1());
.
.   // A mix of literal text and
.   result := result and (FormatTest1());
.   // ShowMessage('');
.
.   // Examples of each of the
.   result := result and (FormatTest1());
.
. 1 using static dbsc_format.dbsc_formatClass;
. using System;
. using static System.SystemClass;
. using System.SysUtils;
. using static System.SysUtils.SysUtilsClass;
.
. namespace dbsc_format
. {
.
. 10 public class dbsc_formatClass
. {
.
.   //http://www.delphibasics.co.uk/RTL.asp?Name=
.   //http://www.delphibasics.co.uk/RTL.asp?Name=
.   //http://www.delphibasics.co.uk/RTL.asp?Name=
. public static bool FormatTest()
. {
.   bool result = false;
.   result = true;
.   result = result && FormatTest1();
.   result = result && FormatTest2();
.   result = result && FloatToStrTest1();
.   result = result && FormatFloatTest1();
.   return result;
. }
```

Cur: (143: 5) Sel: (143: 5)(143: 5) Insert View: lines=[143] Doc: chars=5722, lines=143

## 5.2 Log panel

The Log panel displays logging messages and errors.



The kind of a message is marked by the colored boxes, which are displayed to the left of the node's labels:

- neutral message
- starting the translation without errors
- results of the preprocessor
- including another file
- success
- warning
- error

The picture above is a typical example:

The first line occurs, because no definitions are set in the options.

The red box in front of the filename in the second line means, that there were errors when the file was processed. The cause of the error is marked by the innermost error *SysUtils: unexpected token*. This error is propagated to it's parent nodes.

When *SysUtils.pas* is opened and the translation is started, it stops at:

```
function AdjustLineBreaks(const S: string; Style: TTextLineBreakStyle =
): string;
```

This is a wrong result of the preprocessor. You can reload the original *SysUtils.pas* and find the position of *TTextLineBreakStyle*:

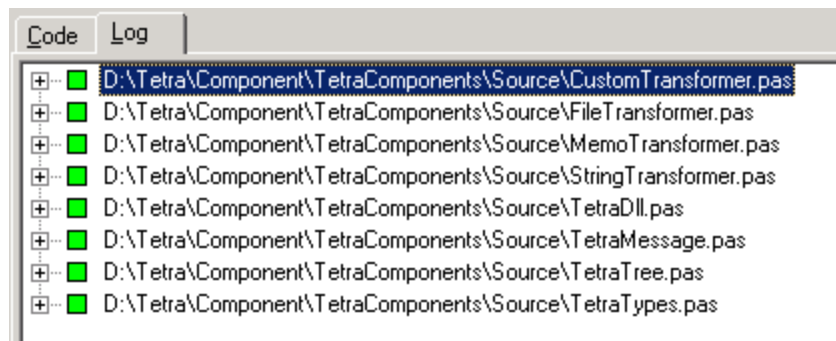
```
function AdjustLineBreaks(const S: string; Style: TTextLineBreakStyle =
  {$IFDEF LINUX} tlbsLF {$ENDIF}
  {$IFDEF MSWINDOWS} tlbsCRLF {$ENDIF}): string;
```

Because neither *LINUX* nor *MSWINDOWS* had been defined, after preprocessing there is no value assigned to *TTextLineBreakStyle*.

--



In the next image you can see an example of the Log panel after use of the file manager, The results of all files are listed in the tree:

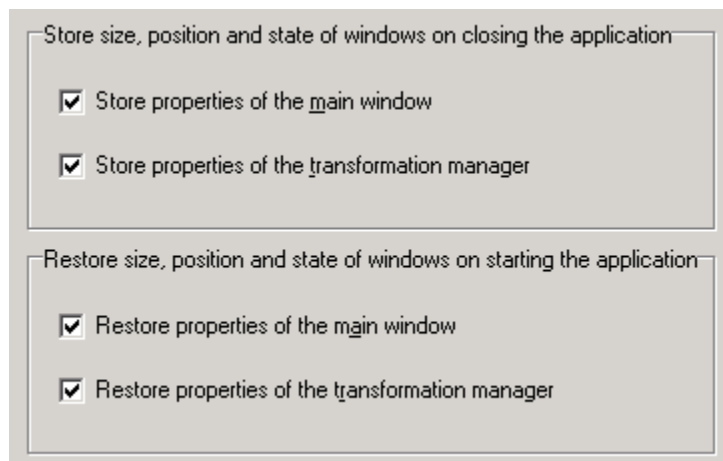


## 5.3 User options

User options can be accessed in the Options menu at the item "Show user options". These options are saved in the Windows registry and thus persist between different sessions with Delphi2C#.

Window positions  
Customization

### 5.3.1 Window positions



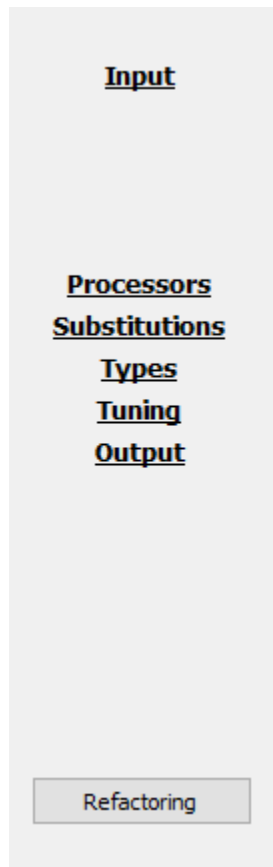
Size positions and state of the main window and the file manager can be stored into the registry and restored from the registry. You can decide to store the values once and then to deactivate a new storage. So the windows will at a new start of Delphi2C# always have the properties that were stored, even if they were change in the previous session.

### 5.3.2 Customization



### 5.4 Translation options

In the options dialog there are six groups of options and a button to open a refactoring dialog:



Input options

Processor options  
Substitution options  
Type options  
Tuning options  
Output options  
Refactoring

You can save and reload the translation options as a project file (\*.prj).

### 5.4.1 Input options

The input options are part of the translation options and specify all contents which either shall be translated or which are required for a translation.

The screenshot shows a dialog box with the following sections:

- Folders**: Two search path fields. The first is labeled "Search paths for files not to convert (RTL/VCL)" and the second is "Search paths for files to convert". Both have a "..." button to the right.
- Unit Scope Names**: A text input field with a "..." button to the right.
- Own or extended "System.pas (internally renamed to d2c\_system)"**: A text input field with a "..." button to the right.
- Conditions**: A field labeled "Definitions" with a "..." button to the right.

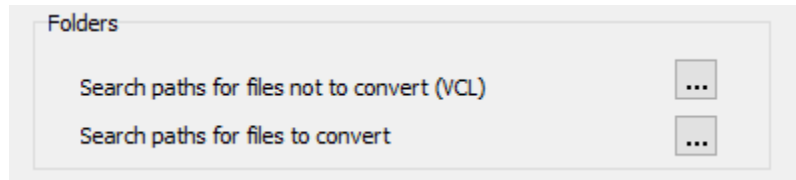
Search paths  
Unit Scope Names  
System.pas

While the items above specify the input files, the definitions determine, which parts of a file are used.

Definitions

### 5.4.1.1 Search paths

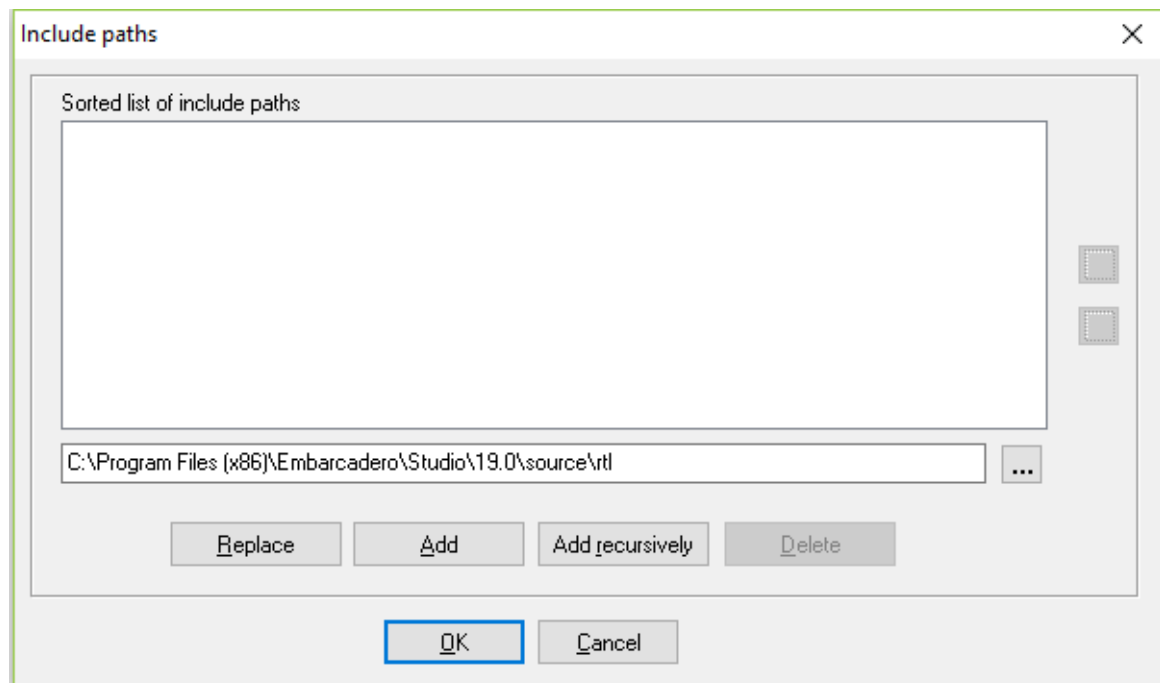
For a correct translation of a Delphi source file the type information of used constants, variables, functions etc. is necessary. If this information is not contained in the actual file, the other used files have to be scanned. As far as these files are in the folder of the source file, they will be found automatically. The folders for other used files have to be specified explicitly - this also applies to files in subdirectories. You can select these folders at the input options of the options dialog.



These directories are separated into

the folders of files for which only the interfaces are needed and  
the folders of files, which really shall be translated.

Both kinds of folders are to be set in a dialog like the one below:



As soon as you have clicked at the "..."-Button and select a folder, you have the option either to add this folder only or this directory recursively together with all of its sub-directories. Once a folder is in the list the "Add"- and the "Add recursive"-button will be disabled for this item. If you want to add sub-directories of an existing item recursively, you first have to delete the item from the list. This behavior

prevents duplicates in the list.

#### 5.4.1.1.1 Paths to the VCL/RTL

If you use C# Builder, there is already a converted version of the VCL. So you don't have to translate the according files. Nevertheless the translator has to know the interface parts of the original Delphi VCL, to make a correct translation of the files, which depend on the VCL. So you have to set the folders of the original or of the preprocessed VCL.

There might be other files, which don't have to be converted, perhaps because you already have translated them. The paths to those files should be set here too.

The paths of the VCL may look like:

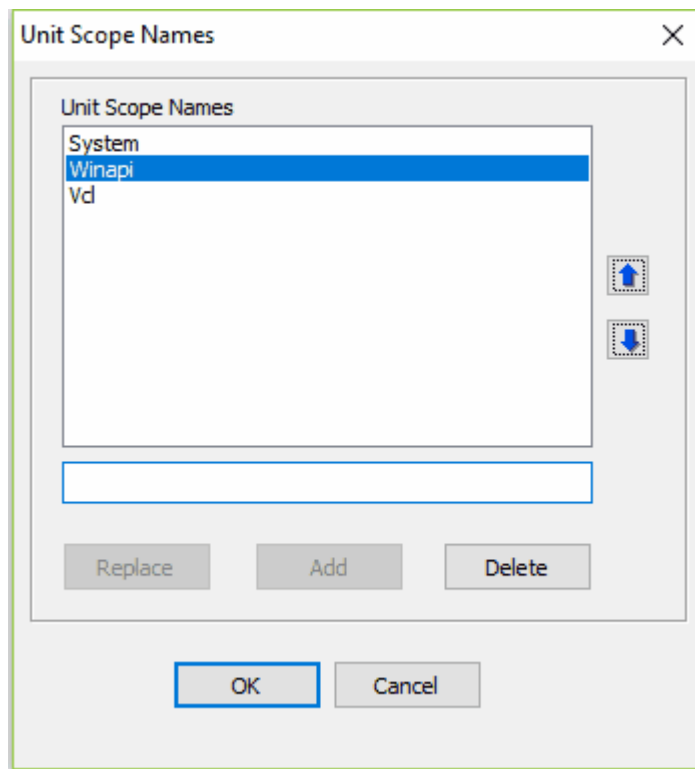
```
C:\Program Files (x86)\CodeGear\RAD Studio\6.0\source\Win32\vcl
C:\Program Files (x86)\CodeGear\RAD Studio\6.0\source\Win32\rtl\common
C:\Program Files (x86)\CodeGear\RAD Studio\6.0\source\Win32\rtl\sys
C:\Program Files (x86)\CodeGear\RAD Studio\6.0\source\Win32\rtl\win
```

#### 5.4.1.1.2 Paths to the source files

The paths to the folders of the files, which shall be translated, can be set by a second dialog, analogously the paths to the VCL. Delphi2C# - as Delphi - doesn't make a recursive lookup in the folders. So sub-folders have to be set explicitly too.

### 5.4.1.2 Unit scope names

A list of unit scope names, which help to find used file, can be entered in the following dialog, which can be opened at the Input-Options.



These identifiers are prefixes in dotted unit names. E.g. *System* is the prefix of the unit *System.Classes* whose file is *System.Classes.pas*. If a unit uses a file it suffices to indicate the name without the prefix, if the prefix is in the list of Unit scope names. At the example above:

```
uses Classes;
```

instead of

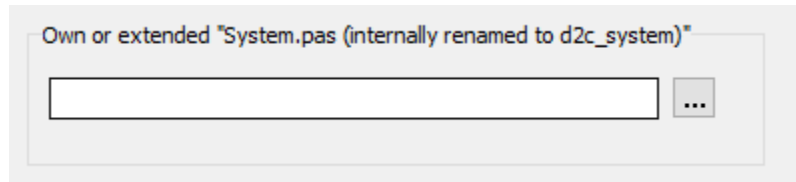
```
uses System.Classes;
```

So, if *System* is in the list of unit scope names, Delphi2C# nevertheless will lookup the file *System.Classes.pas*.

### 5.4.1.3 Extended "System.pas"

"**System.pas**" is a source file of special importance in Delphi projects. Fundamental type definitions, procedures and functions are defined in the *System* unit, which is implicitly included in every unit. For example *TObject* is defined there. There are other intrinsic definitions like the *Read*, *Write* or *Str* function, which are accessible in every unit too. These intrinsic functions are built into the Delphi compiler. *Delphi2C#* must know the signatures of such intrinsic functions and tries to find them in the *System.pas*. So the original incomplete *System.pas* either has to be replaced by an extended copy or the original *System.pas* has to be supplemented by an additional source file.

In the options dialog you can set the name of such an additional *System.pas* extension file.



Such an individual *System.pas* called *d2c\_system.pas* is in the *Source* folder of the *Delphi2C#* installation. No matter which name the file has, it internally is renamed to "d2c\_system". With this name it is shown in the log-tree.

If an individual *System.pas* is used, the specially treated RTL/VCL functions and some compile time functions (*Abs*, *High*, *Low*, *Odd*, *Pred*, *Succ*) might have to be defined in this file for types, that cannot be handled by the built-in translation alternatives.

The overwritten *System.pas* gets always preprocessed, even if the option to pre-process files is disabled for all other files.

Because this file is very basic, **it may not use other files.**

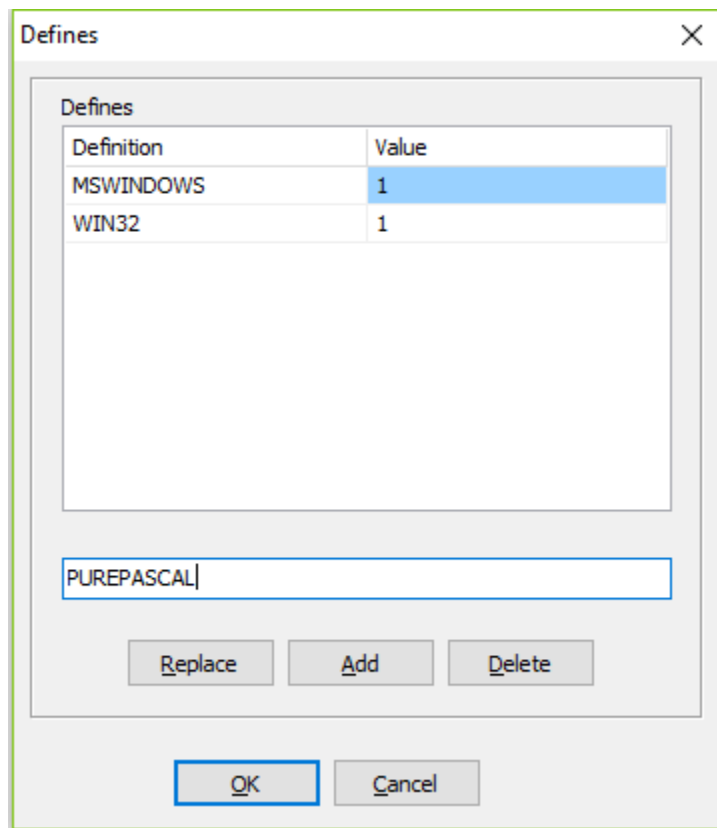
### Lookup algorithm

Delphi2C# looks up system types and functions etc. in following order::

1. *Delphi2C#* will look for declarations at first in your own *System.pas*, if it exists.
2. If the declaration is not found there, *Delphi2C#* will look in the *System.pas* of your Delphi installation, if the path to this file is set in the options..
3. If neither an own *System.pas* exists nor the path to the original *System.pas* is set, *Delphi2C#* simulates the most important parts of this file.

#### 5.4.1.4 Definitions

Delphi code often contains directives for conditional compilation of parts of the source text. Delphi2C# evaluates such directives too. You can set the definitions in the option dialog



There are limitations for the evaluation of such expressions.

If code of the Delphi RTL shall be translated, it is recommended to set *PUREPASCAL* defined, to avoid problems with inline assembler code.

Incomplete definition can lead to hard to find bugs, as for example in *System.Windows.pas*

#### 5.4.1.4.1 Windows.pas

If there is no Definition set either of *CPUX86* or of *Win64* the *Windows.pas* cannot be parsed. That's because of the following code:

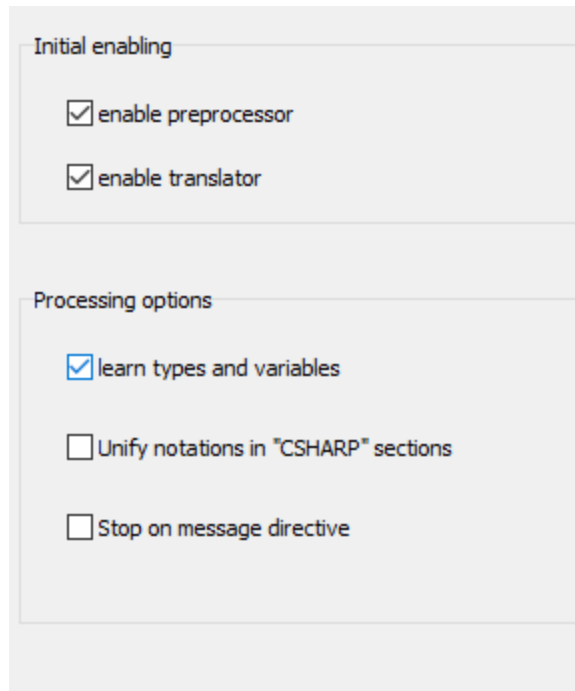
```
function InterlockedBitTestAndComplement(Base: PInteger; Offset: Integer): ByteBool;
{$IFDEF CPIX86}
...
{$ENDIF CPIX86}
{$IFDEF Win64}
...
{$ENDIF CPIX64}
```

There will remain a function declaration only and the parser will regard all following functions as sub-functions to this declaration. So nearly the whole file gets parsed, before the missing function body is discovered. This bug is very hard to find.



## 5.4.2 Processor options

The processor options are part of the translation options and specify the kinds of processing during the translation from Delphi to C#.



When Delphi code is translated, normally the source at first is preprocessed to remove parts of the code, which aren't defined. But it is possible too, to disable either the preprocessor or the translator. That can be done by the according buttons in the tool bar. The initial state of these buttons after the options are loaded can be set here.

The *overwritten System.pas* gets always preprocessed, even if the option to do so is disabled.

Normally the **learning option** is enabled. So the variables and types of every interface are remembered, once the interface was parsed and the interface has not to be processed again. However, there are cases, that the definitions are not constant for all common interfaces. A definition of a current file might enable or disable definitions of a common file. So the result of the conditional compilation will change too and finally different types and variables might be declared of the same unit, which is used in different other units. **When the learning option is disabled**, included units are preprocessed for every new file again and the result will be correct for each file, but the **total processing time increases very much**.

The option Unify notations in "CSharp" sections determines the case sensitivity in "CSharp"-sections.

The option Stop on message directive determine what happens, if a message directive would remain in the pre-processed code.

### 5.4.2.1 Unification of CSHARP-sections

Unify notations in "CSHARP" sections

This option is part of the processor options. It determines how identifiers in "CSHARP"-sections are treated. If the box is checked, the identifiers are unified as all other unifiers in the rest of the code to. If the box is unchecked the identifiers will be written unchanged into the output.

### 5.4.2.2 Stop on message directive

Stop on message directive

This option is part of the processor options. If the is enabled the pre-processor will stop as soon as such a message will remain in the code, that means, that the conditions for this code section are true. It will not stop, if the conditions for the code section with the message aren't true.

Delphi message directive are used in most cases to indicate, that something is wrong in the code. A typical example of such a directive is:

```
{MESSAGE ERROR 'Unknown platform'}
```

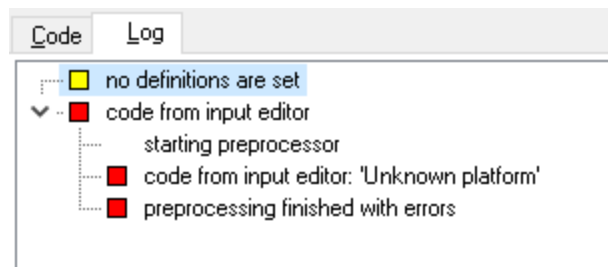
If correct definitions are set, such messages normally will be part of code sections for which the conditions are false. The option to stop on message directives therefore will not apply. But e.g. the recommended *PUREPASCAL* definition is problematic. If it is defined, the definition of *ASSEMBLER* should be avoided. But for example in the following code snippet there is no *PUREPASCAL* alternative. Therefore the function definition would be reduced to a function declaration.

```
function Get8087CW: Word;
  { $IF defined(CPUX86) and defined(ASSEMBLER) }
asm
    PUSH    0
    FNSTCW [ESP].Word
    POP     EAX
end;
  { $ELSEIF defined(CPUX64) and defined(ASSEMBLER) }
asm
    PUSH    0
    FNSTCW [RSP].Word
    POP     RAX
end;
  { $ELSE }
  { MESSAGE ERROR 'Unknown platform' }
  { $ENDIF }
```

->

```
function Get8087CW: Word;
  { MESSAGE ERROR 'Unknown platform' }
```

If another function follows, DelphiXE2Cpp11 will regard it as a sub function of the remained function declaration and the parser will not stop. The parsing error occurs at a much later position then and the real cause of the error is difficult to find. If the option to stop on messages is enabled, the true error position is set. DelphiXE2Cpp11 stops and the message is shown on the log-panel:



On the other side, there are messages which you might want to ignore. In the following case DelphiXE2Cp11 isn't able to calculate the correct result of the condition:

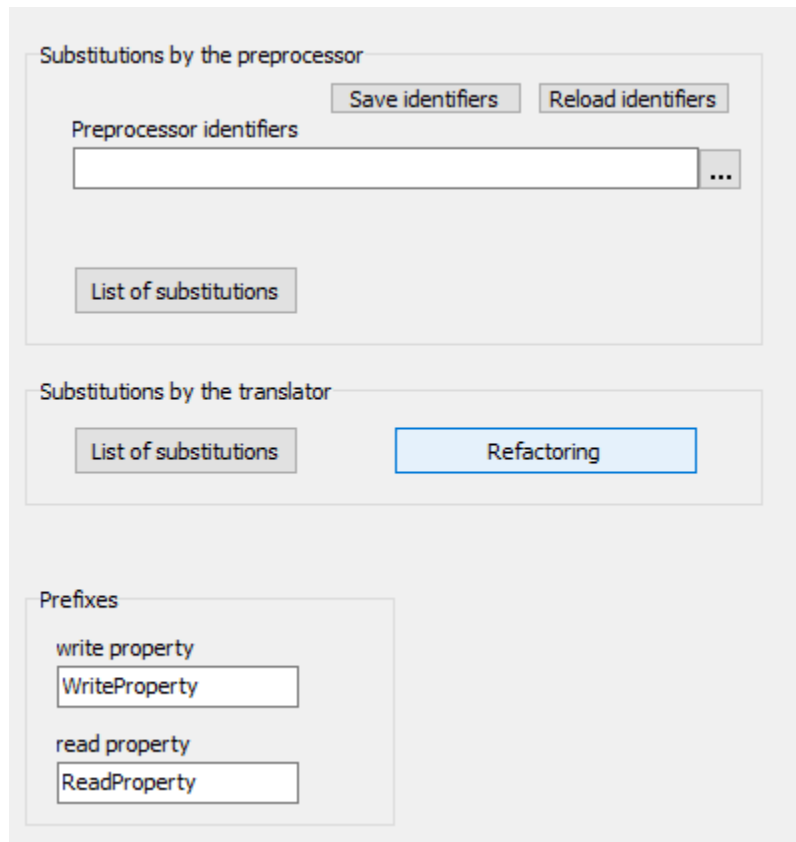
```
{ $IF SizeOf(Extended) <> SizeOf(TExtended80Rec) }  
  { $MESSAGE ERROR 'TExtended80Rec has incorrect size' }  
{ $ENDIF }
```

The consequences of the option to stop on message directives depend on the level of the current file. If this option is enabled and if this message appears in the actual file, the whole translation for this file will be stopped. If the message appears in a dependant file, only the processing of that file will be stopped and the message will be shown without stopping the translation of the actual file.

If the definitions cannot be changed such that the message directives disappear, it's the best to prepare your Delphi source code accordingly.

### 5.4.3 Substitution options

The substitution options are part of the translation options and allow to edit lists of identifiers which are used for different kinds of substitutions during the translation process.



There are two possibilities how the pre-processor can substitute identifiers.

1. The notation of identifiers are unified according to a list of given notations
2. Identifiers can be substituted to different ones,

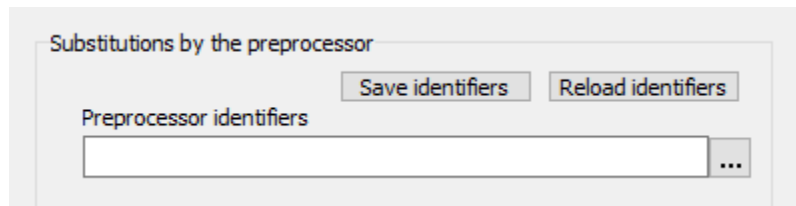
The pre-processor does its work, before the Delphi parser starts. Therefore, you have to take care, that the pre-processor substitutions leave the Delphi code intact. On the contrary

3. the substitutions by the translator are executed after the code already has been parsed.
4. also some kinds of refactoring can be done.
5. You can change the prefixes for special function names here.

#### 5.4.3.1 List of identifiers

After one or several files have been processed the list of identifiers can be saved, which was created by the preprocessor to unify their notations: The list can be loaded again for another session, so that the notations of the identifiers in the generated C# output are the same as in the previous files.

The path to such a list is set on the third register page of the option dialog and is saved together with the other options.



If the path is saved as part of the options, the list is loaded at the same time as the options are loaded.

Whenever additional files are translated and new identifiers were found, you are asked to save them. If you accept, at first a dialog appears by which you can select a file for the list. If the path to the file is different to the path which is set in the options or if no path is set there at all, you are asked whether you want to insert the new path into the options.

You can edit such a list in an external editor or even create such a list by hand. Every line has to consist in just one identifier. E.g.

```
...
SetLength
Setscrollinfo
SetSelection
...
```

If you change "Setscrollinfo" to "SetScrollInfo", all appearances of this identifier will be unified to the second form.

If the same identifier occurs more than one time in the list, the latest occurrence will be taken.

If you edit the list in an external editor, you have to reload the list by the button **Reload identifiers**, otherwise the changes will not have an effect in the current session.

There also are some fixed identifiers, which cannot be modified by the list of identifiers.

#### 5.4.3.1.1 Fixed identifiers

The notation of most identifiers can be determined by the list of identifiers, which is set in the options. However there are some identifiers whose notations are fixed:

```
Char
String
break
continue
```

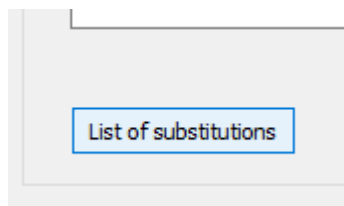
```
implicit
explicit
negative
positive
inc
dec
logicalnot
trunc
round
in
equal
notequal
greaterthan
```

greaterthanorequal  
lessthan  
lessthanorequal  
add  
subtract  
multiply  
divide  
intdivide  
modulus  
logicalor  
bitwiseor  
logicalxor  
bitwisexor  
logicaland  
bitwiseand  
leftshift  
rightshift  
  
MinComp  
MaxComp  
NaN  
Infinity  
NegInfinity  
  
Sum  
SLICE

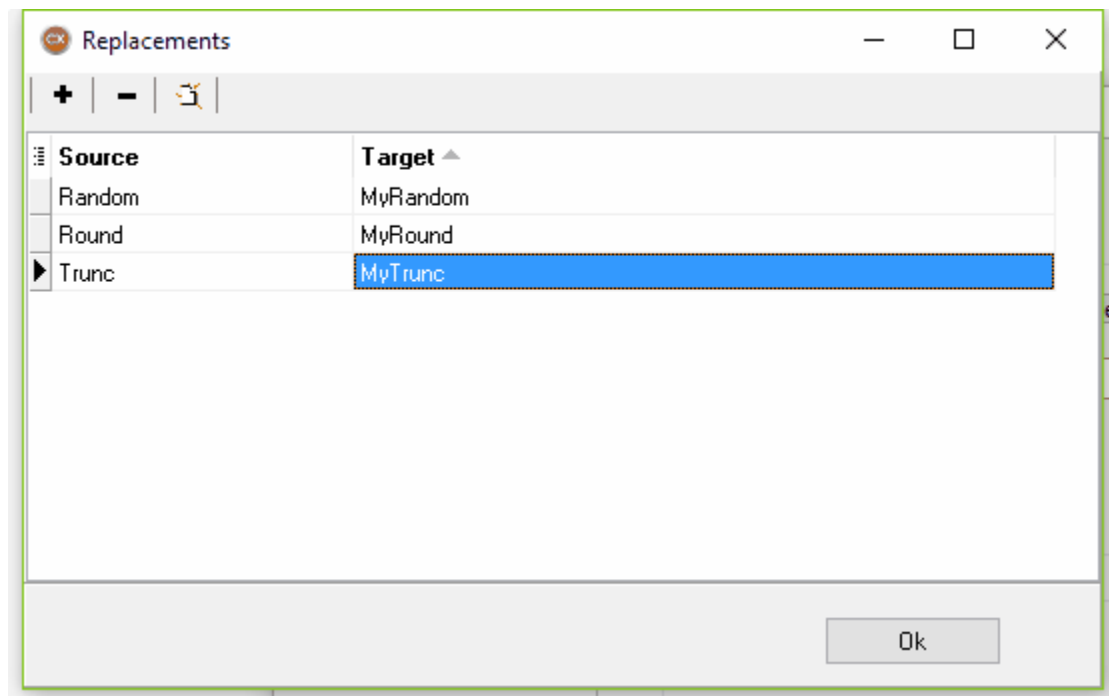
Sorry, the list may not be complete




#### 5.4.3.2 Substitutions in the preprocessor

A substitution table for the preprocessor can be shown, if you click on the button "List of substitutions" in the group-box for preprocessor substitutions.



If you click on the button, the following grid is shown.



-  add a new row
-  remove the actual row
-  clear the whole table

In the first column the identifiers are listed, which shall be replaced by the preprocessor and in the second column identifiers are listed, which are inserted in the code instead of the found identifiers of the first column. The preprocessor recognizes text sections as identifiers, which start with a letter or a underlined and on which an arbitrarily number of letters, numbers or underlines can follow; i.e. as well the real Delphi identifiers as the Delphi keywords.

The substitution of identifiers during the pre-processing of the code can fulfill two purposes:

1. a desired notation of the identifiers can be forced.

The same purpose is accomplished by use of the list of identifiers and this method should be preferred normally. However the items of this list are overwritten by the items of the substitution table. This may be a method to quickly check other notations.

2. completely other names can be assigned to certain identifiers.

So e.g., Delphi function names could be replaced by different names of equivalent C# functions.

For example it is recommended to make such substitutions for ampersand-expressions.

### 5.4.3.3 Substitutions of the translator

Similar to the substitution table for the preprocessor there is a second substitution table for the translator.

There are two differences to the substitutions, which are carried out by the preprocessor:

1. While the preprocessor cannot distinguish identifiers, which are keywords from other identifiers, the translator does. Only the latter are substituted by the translator, i.e. the names for variables, functions etc. Therefore, the translator can substitute such names, which are keywords in C#. Without this substitution, there would be errors in the translated code. E.g.

```
double float; -> double float_value; .
```

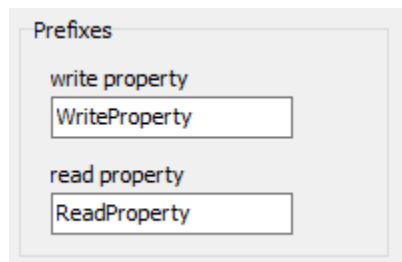
2. The identifier is already recognized by the translator before the substitution takes place. Therefore it can be substituted by something completely different, without affecting the translation process. E.g.

```
StringOfChar -> AnsiString::StringOfChar
```

This translation table also is applied to the **names of helping variables** which are needed for the definition of implicitly defined types, e.g. in set's. So a adjustment of the according names in the C# Builder VCL is possible, which can be different there from version to version. Also the set type "System::Set" can be renamed this way now, e.g. for a comfortable integration of Daniel Flower's TSet.

### 5.4.3.4 Prefixes for properties

In some cases Delphi2C# cannot create direct pendants to Delphi properties in C# and generates public access methods instead. The prefixes for the names of such methods can be set here:

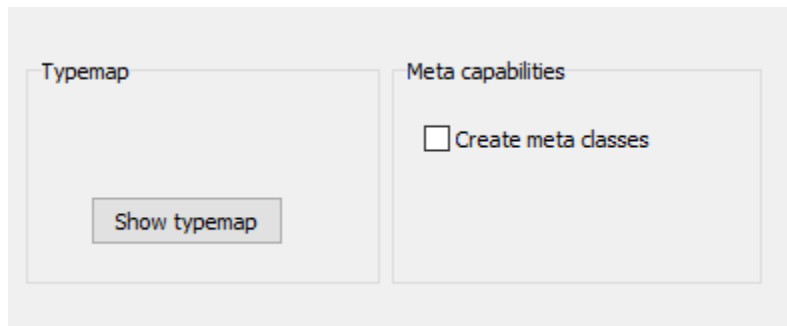


variable indexed properties

### 5.4.4 Type options

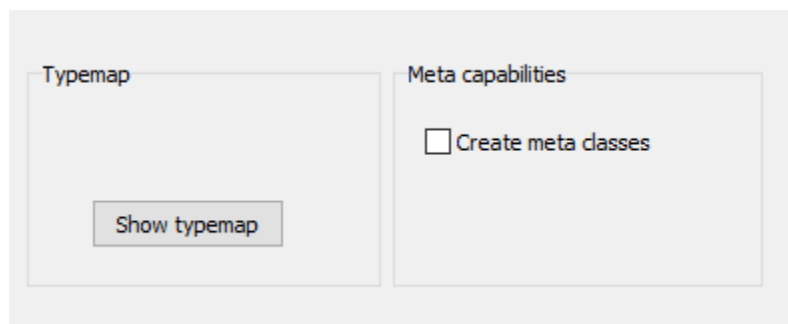
The type options are part of the translation options. Currently there are no options but only a button by which a type-map can be shown.





#### 5.4.4.1 Type-map

At the type options a type map can be shown. If *use user type-map* is checked, the cells of the shown grid can be edited.



In the first column of the type map the names of Delphi built-in types and the second column the according names of the C# types are listed. In the further columns some properties of the C# types are given:

Delphi Typename	C++ Typename	Size	Minimum	Maximum	In System
ansichar	AnsiChar	1	0	255	<input checked="" type="checkbox"/>
ansistring	AnsiString	4	0	-1	<input checked="" type="checkbox"/>

Size: size of the type in bytes  
 Minimum: minimum value of the type  
 Maximum: maximum value of the type  
 In System: true, if the type is defined in d2c\_system or in System.h, else false.

The last column determines, whether the System namespace is prepended to the according type name in a header.

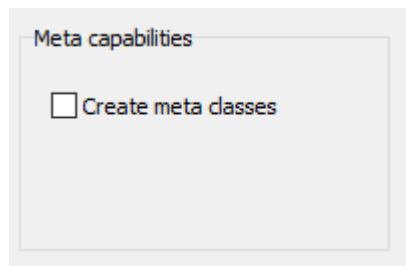
For example *BOOL* is a *Windows* type and therefore has not to be defined in the *System* namespace. E.g.:

```
longbool    BOOL    4        -2147483648    2147483647    false
```

Under Linux however *BOOL* is unknown and could be defined in *d2c\_systypes.h*

```
longbool    BOOL    4        -2147483648    2147483647    true
```

#### 5.4.4.2 Meta capabilities



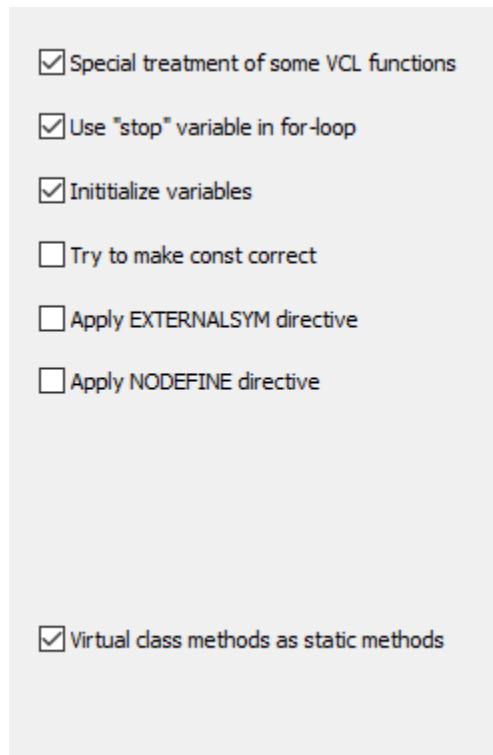
##### Create meta classes

If the option *Create meta classes* is enabled at the type options, Delphi2C# creates for each class on demand an instance of additional meta class (= class reference type). These class reference instances can be used for factory functions, to create different class types in dependence of the class reference parameters. These class reference instances also are needed if overridden virtual class methods have to be used.

To enable this option has drawbacks however. More manual post-processing will be necessary. One reason for that is, that **a creation of class instances from class references is possible only, if the class has a standard constructor.**

#### 5.4.5 Tuning options

The tuning options are part of the translation options and specify special details at the translation from Delphi code to C#.



Special treatment of some VCL functions

*Use "stop" variable in for-loop*

Initialize Variables

Try to make const correct

Apply EXTERNAL directive

Apply NODEFINE directive

.

#### 5.4.5.1 Special treatment of some VCL functions

Some Delphi VCL functions are made to member functions in the C#Builder VCL.. *Delphi2C#* converts the generated C# code accordingly for some of the frequently used function. You can switch off this special treatment and write your own C# functions instead.

#### 5.4.5.2 Use stop-variable in for-loop

The tuning option *Use "stop" variable in for-loop* determines the output for for-loops

#### 5.4.5.3 Initialize Variables

If the tuning option *Initialize variables* is chosen, default values are assigned to all variables.

The initialization of variables in Delphi and C# is the same. Local automatic variables and normal variables of a class aren't initialized, while global and static (class) variables are initialized to zero. Nevertheless Delphi2C# offers the option to initialize all variables explicitly, either to achieve reproducible behavior or just to suppress compiler warnings.

#### 5.4.5.4 Try to make const correct

By the tuning option *Try to make const correct* the generated code can be made more C#-like.

Delphi doesn't know the concept of const-correctness. However it is an important concept in C#. If this option is enabled, *Delphi2C#* makes the getter methods of properties constant as well as the methods which are called inside of these getter methods. In most cases this will work correctly, but, if member variables are changed in such a method, the compiler will produce an error

#### 5.4.5.5 Apply EXTERNALSYM directive

If the tuning option *"Apply EXTERNALSYM directive"* is enabled, type declarations, which are marked with this directive aren't written into the generated code.

Symbols that are defined in the C++ API of the operation system often have to be redefined in Delphi. The other way round, if C++ code is generated from Delphi, such symbols have to be omitted. For this purpose the `$EXTERNALSYM` directive is used. This directive tells the C++Builder that the according symbol already exists in C++. *DelphiXE2Cpp11* don't writes such symbols into the output. If the option *"Apply EXTERNALSYM directive" is enabled,*

See also

#### 5.4.5.6 Apply NODEFINE directive

If the tuning option *"Apply NODEFINE directive"* is enabled, type declarations, which are marked with this directive aren't written into the generated code.

See also

#### 5.4.5.7 Virtual class methods as static methods

Because in C# methods cannot be static and virtual at the same time, Delphi virtual class methods either have to be converted to static non-virtual methods or to virtual non-static methods. This is determined by the tuning option *Virtual class methods as static methods*, which is set to true by default. This is the best option for the frequent case, that there aren't overridden versions to the method at all. In this case a method like:

```
class procedure ClassVirtual; virtual;
```

simply become a non-virtual static function:

```
static virtual void ClassVirtual();
```

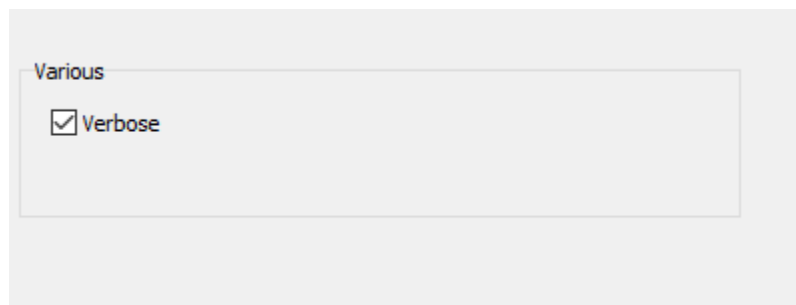
If there are overridden Delphi virtual class methods, the option *Virtual class methods as static methods* has to be disabled. The method then becomes

```
virtual void ClassVirtual(); ///#static
```

*Delphi2C#* then takes care, that the method is called from an *ClassRef*-instance of the according class. This works only, if the creation of meta-classes is enabled.

## 5.4.6 Output options

The output options are part of the translation options and specify the style of the generated output. At the moment there is only the *Verbose* option.



### 5.4.6.1 Verbose

Per default the *Verbose* option is set. That means, that comments are inserted into the output at critical places, where the translation might cause errors. Often such comments simply are quotations of the original Delphi code, which allow a quick comparison.

To distinguish these comments from converted comments, which stem from the Delphi source code, they are marked with a hash character (octothorpe) '#'.

E.g.:

```
WORD Words[4/# range 0..3*/];
```

## 5.4.7 Refactoring

The refactoring dialog is reached from the button on the options dialog. The Dialog shows the list of refactoring items:

Substitution of types

Actions

+ | ✎ | - | ↺

	Original name	New name	Kind of type	Original type	New type	Original	New poi	Unit
	Max	Math.Max	function			0	0	Syst
▶	Min	Math.Min	function			0	0	Syst

Another dialog with the details of a refactoring item is shown, if a new item is added or an existing item is edited:

The screenshot shows the "Refactoring" dialog box in Delphi. It has a title bar with the Delphi logo and the text "Refactoring". The dialog is divided into several sections:

- Original name:** A text box containing "Min".
- New name:** A text box containing "Math.Min".
- Original type is:** A group box containing radio buttons for "array", "builtin type", "class", "dynamic array", "enum type", "function" (selected), "interface", "procedure", "record", "set", and "unspecified".
- Original type:** A dropdown menu.
- New type:** A dropdown menu.
- Original pointer:** A spinner box showing "0".
- New pointer:** A spinner box showing "0".
- Original unit (optional):** A text box containing "System.math".
- Using:** A text box containing "System.math".
- Options:** Two checkboxes: "Rename original declaration" and "Remove original declaration", both of which are unchecked.
- Buttons:** "Ok" and "Cancel" buttons at the bottom.

Variables, functions and constants which shall be changed are looked up according to the criteria, which are given by the control elements the on the left side of the dialog. At least the original name has to be specified, the other criteria are optional. On the right side of the dialogs the resulting properties can be set. Again at least a new name has to be set and the other properties are optional.

#### "Original name" and "New name"

The original name of a variable, function or constant in the Delphi source code will be changed to the

new name in the C# output. The input in the field is treated case insensitive in the same way as the source code by the pre-processor. For example with input in the image above all occurrences of "Min" in the source code will be changed to "Math.Min", regardless whether "Min"; "min", "mIN" or any other case occurs in the code.

If the identifier for the original name isn't contained in the list of notations, its notation will be used for all notations of the identifier in the generated code.

#### "Original type is:"

The general kind of type of the variable, function or constant which shall be changed can be specified, to exclude all other kinds from this refactoring. If, as in the image above, "Min" is specified as a function, variables or constants with the name "Min" will not be changed. If all occurrences of "Min" shall be changed regardless of the kind, it can be set to "unspecified":  
In contrast to the other fields in the dialog, the general kind of type cannot be changed and will remain the same in the output as in the source code.

#### "Original type" and "New type"

If "function" is selected "Original type" and "New type" are specifying the result type of the function. otherwise "Original type" and "New type" specify the type of an built-in type, if this item is selected. Normally the type should be identical, but there might be cases where it is desired to avoid or to force typecasts by means of a change of the result type.

For the new type also a free identifier can be set.

#### "Original pointer" and "New pointer"

Again, normally the pointer of the type should not be changed.

#### Original unit

The input in the field for the original unit is treated case insensitive in the same way as the source code by the pre-processor and "Original name" in thid dialog.

#### Using

In contrast to the "Original unit" field, the input in the "Using" field is case sensitive. Therefore the "System.math" will produce the output lines

```
using System.math;  
using static System.math.mathClass;
```

Rename original declaration  
Remove original declaration

not implemented yet



## 5.5 Translation

The translation of the loaded Delphi source file to C# starts with the button:



Three steps are executed for a translation:

1. the code is preprocessed
2. the included files are scanned for type information and global variables
3. a parse tree for the actual file is created from which the C# code is written into the output windows.

### 5.5.1 Preprocessing

A preprocessor fulfils two tasks:

1. the conditional compilation
2. the unification of the notations of identifiers

#### 5.5.1.1 Conditional compilation

Delphi2C# uses a preprocessor (pretranslator), which prepares the source text so that directives for the conditional compilation are evaluated and removed.

Conditional expressions like

```
{ $IF CompilerVersion >= 17.0 }
```

are evaluated too, but there are some limitations. Only integer values are evaluated and only operators, which also exist in C#. Sizeof-expressions like the following are evaluated too

```
{ $IF SizeOf(Extended) >= 10 }  
  { $DEFINE EXTENDEDHAS10BYTES }  
{ $ENDIF }
```

The size is taken from the type-map.

If there is an expression, which cannot be evaluated, a warning is written into the code:

```
// pre-processor can't evaluate ...
```

The source code has to be corrected by hand then.

Include directives are executed too.

```
{I filename}  
{$INCLUDE filename}
```

The file *filename* is included into the source.

The definitions can be set in the options dialog.

### 5.5.1.2 Unification of notations

While Delphi code is case insensitive, C# code is case sensitive. So different notations of identifiers have to be unified. Delphi2C# uses a simple approach to do that. As soon a a new identifier is recognized it is put into a list and all further notations of this identifier are replaced by the first one (exception: see below). Identifiers used at the refactoring also have an impact on the notations in the output.

After one or several files have been processed the list can be saved.

This unification is done by the preprocessor, which also is responsible for the conditional compilation. For "Cpp"-sections, there is a special option.

Some notations have a special meaning in C# and are fixed. i.e. they are not controlled by the list of identifiers. These identifiers are:

- Char
- String
- break
- continue
- explicit
- implicit

The following identifiers are fixed, because they denote C# UnicodeString methods:

- BytesOf
- ByteType
- c\_str
- cat\_printf
- cat\_sprintf
- cat\_vprintf
- CodePage
- Compare
- CompareIC
- CurrToStr
- CurrToStrF
- data
- Delete
- ElementSize
- EnsureUnicode
- FloatToStrF
- FmtLoadStr
- Format
- FormatFloat
- Insert

IntToHex  
IsDelimiter  
IsEmpty  
IsLeadSurrogate  
IsPathDelimiter  
IsTrailSurrogate  
LastChar  
LastDelimiter  
Length  
LoadStr  
LoadString  
LowerCase  
Pos  
printf  
RefCount  
SetLength  
sprintf  
StringOfChar  
SubString  
swap  
t\_str  
ToDouble  
ToInt  
ToIntDef  
Trim  
TrimLeft  
TrimRight  
Unique  
UpperCase  
vprintf  
w\_str

### 5.5.2 Scanning dependencies

Most Delphi units depend on other units, which are included in the uses clause. Delphi2C# scans the included files in so far, as they are placed either in the same directory as the actual file or in a directory, which is set in the search paths.

The translation will produce the best results if the **Delphi VCL** is included. In this case, however, **the translations of the first files will slow down significantly**. All information that once has been obtained from the interface parts of the processed files is remembered for the translation of further files.

The information can be cleared by the according command in the start menu or the tool bar button .

.

### 5.5.3 Writing the C++ code

The original Delphi file is split into a C# header and a C# source file. These parts are output into the two windows on the right side of the main window. The header is written into the upper window and the source code is written into the lower window.

## 5.6 File manager

The file manager is a dialog, by which you can translate whole directories or other groups of files. You can reach the file manager either by the menu item *File manager* of the *Start* menu or by the according button in the tool bar:



The button in the tool bar of the manager for executing the translations is deactivated until translation options are set and source files are selected. Before starting the translations, you can check the list of the files which will be produced. There is a page of his own for each of these steps in the file manager:

1. Translation options
2. Source files
3. Preview of the list of target files
4. Results

The settings, inclusive of the select folders and files, can be stored as a management and loaded when required newly.

### 5.6.1 Translation options

If you like to use the file manager for the translation of your source files, you have to decide where the resulting files shall be written. The edit box for the target path which is shown in the picture below, is shown in your application only, if you have selected an option to write the resulting files into a different place as the source files.

Paths of management files are calculated relatively to the current project path:

**C:\Users\HOME\Documents\Delphi2C#\Projects\**

Write target files into the according source folders

Write all target files into a common folder

Calculate target files from the source paths

additionally allow individual target names or folders

#### Select root for the target files

Select target path

1. The most simple case is to write the C# files just in the same place, where the source file is.
2. All files can be written into a common target directory, regardless of the place of the source file
3. The relative paths of files in a common root directory can be reproduced in the target folder.

If case one or two are selected the field for the target folder/root are shown and a dialog for the selection of a the target directory is opened by the  button:.

If target files shall be written outside of the common target path/root, the checkbox to allow individual file names or folders can be enabled. In that case an additional column for individual targets are shown on the source page.

#### Warning






At the top of this options page either a default path or the path to the currently loaded project is shown. If you save or load the source paths, they are calculated relatively to this project path. This allows the exchange the "management"-files between different drives or computers. But you have to pay attention that source folders and project path fit to each other.

## 5.6.2 Selecting source files

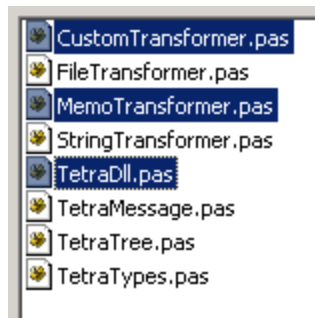
The files which shall be transformed are selected on the second page of the file manager and are shown in a table.

Source files				
No	Path	File name or filter (with wildcards: *, ?)	Recursive	Exclude
1	D:\Tetra\Component\TetraComponents\Source	*.pas	<input type="checkbox"/>	<input type="checkbox"/>

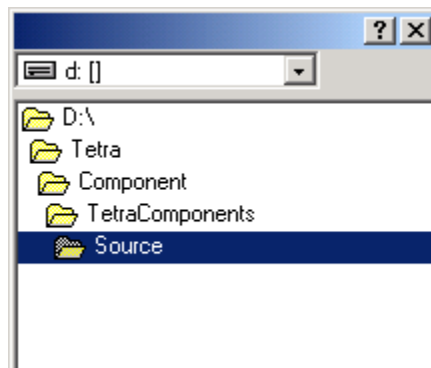
The page has a tool bar of its own with the buttons:

-  Insert an empty row
-  Select a single source file
-  Select a whole source directory
-  Deleting a row
-  Clear the whole table

The choice of a file or a folder is carried out respectively with a corresponding selection box. Several files also can be selected at once in the selection box.



After the confirmation of the choice a new row is inserted in the table below the tool bar for every file or every folder.



There are five columns in the table:

**No**

a simple counter

**Path**

The absolute path of the file or folder.

**Filename or filter**

For files the file name can be seen here (with extension).  
For folders a filter can be specified here. The default filter is "\*.pas".

**Recursive**

The check box in this field can be activated only for folders. If it is activated, then all files in the sub-folders of the shown directory are transformed too.

**Exclude**

Normally the check box of this field remains deactivated. However, it can be that you want to except some files or folders from the translation of a folder. This is possible by producing rows of their own for these exceptions in the table and activating the excluding check box by mouse.

**Target file or folder**


This column is shown only, if on the options page the box "allow individual file names or folders" is checked. Here for each file an arbitrary path or file name as target can be set. If the source is specified by wildcards, an arbitrary target path can be set.

### 5.6.3 Preview of the target files

The list of the files which will be produced are shown on the third tab-page of the file manager.

No	Filename
1	D:\Tetra\Component\TetraComponents\CppSource\CustomTransformer.pas
2	D:\Tetra\Component\TetraComponents\CppSource\FileTransformer.pas
3	D:\Tetra\Component\TetraComponents\CppSource\MemoTransformer.pas
4	D:\Tetra\Component\TetraComponents\CppSource\StringTransformer.pas
5	D:\Tetra\Component\TetraComponents\CppSource\TetraDll.pas
6	D:\Tetra\Component\TetraComponents\CppSource\TetraMessage.pas
7	D:\Tetra\Component\TetraComponents\CppSource\TetraTree.pas
8	D:\Tetra\Component\TetraComponents\CppSource\TetraTypes.pas

### Actualize

You can refresh the list of files by the button  .

## 5.6.4 Starting the translation

The translation of the selected files in the file manager is started by the menu item *Start translation* or by the button in the main tool bar



When the translations are started, the page is changed to the Results-page automatically.

## 5.6.5 Results






The rows of the table on the result page of the file manager contain messages which arise during the translation of files.

Every message is immediately written into a new row of the table after the message was created. So, the growing row number of the table at the same time shows the progress of the translations.

S...	Date	Time	Message
	11.11.2009	01:14:49	Starting N:N Transformation
	11.11.2009	01:14:49	Starting D:\Tetra\Component\TetraComponents\Source\CustomTransformer.pas
	11.11.2009	01:15:34	Starting D:\Tetra\Component\TetraComponents\Source\FileTransformer.pas
	11.11.2009	01:15:42	Starting D:\Tetra\Component\TetraComponents\Source\MemoTransformer.pas
	11.11.2009	01:15:42	Starting D:\Tetra\Component\TetraComponents\Source\StringTransformer.pas
	11.11.2009	01:15:42	Starting D:\Tetra\Component\TetraComponents\Source\TetraDll.pas
	11.11.2009	01:15:46	Starting D:\Tetra\Component\TetraComponents\Source\TetraMessage.pas
	11.11.2009	01:15:47	Starting D:\Tetra\Component\TetraComponents\Source\TetraTree.pas
	11.11.2009	01:15:51	Starting D:\Tetra\Component\TetraComponents\Source\TetraTypes.pas
	11.11.2009	01:15:51	Last transformation finished



In the first row the status of the message is shown as a color.

Color	Status
	new source file
	neutral information
	success message
	warning
	error message

### 5.6.6 Management

The sum of the settings of the file manager is called a management here.

By the menu item: **Save management as**, you can save a management

By the menu item: **Open management**, you then can reload a management.

Managements are save with the extension "ttm". They are written in the same format as TextTransformer managements.

The syntax for a management was designed as scarce and simple as possible, so that it also can be written by hand. A management consists in the extreme case in only one file path.

## 6 Use in command line mode

Delphi2C#.exe can be called from the command line too. You then have to pass some parameters.

### 6.1 Parameter

Delphi2C#.exe can be controlled either by a management, which was produced with the file manager or by parameters for the source and target files.

In the first case a call has the form:

```
Delphi2C# -p PROJECT -m MANAGEMENT
```

and in the second case:

```
Delphi2C# -p PROJECT -s SOURCE [-t TARGET] [-r]
```

Expressions in brackets are optional.

If a path contains spaces, it has to be quoted.

Parameter	Meaning	Examples
-p PROJECT	Delphi2C# project	C#Builder_vcl_ge.prj

-m MANAGEMENT	a project file made with the file-manager	my_management.ttm
-s SOURCE	Source file(s)	C:\dir\*.pas
-t TARGET	Target file or directory	C:\dir2\target
-r RECURSIVE	recursively including the files of the sub-folders	
-pause	after processing waiting for a key	

### -p PROJECT

The parameter -p must be followed by the path of the Delphi2C# project, with the options by which the files of the source directory shall be translated.

### -m MANAGEMENT

The parameter -m is followed by the path to a Delphi2C# management, which specifies the source and target files.

If an -m parameter is provided, -s, -t and -r are ignored.

### -s SOURCE

The parameter -s must be followed by a specification of the files, which shall be translated. In the simplest case this a specification is the path of a single file, like "C:\dir\source.pas". To transform all "pas" files of a directory, you can use a mask like: "C:\dir\\*.pas;\*.dpr". If there is no directory specified in the mask, all according files of the actually directory will be translated. If there is no special extension specified in the mask, all files of the directory will be translated. E.g.: "ab?\*" will chose all files of the directory beginning with "ab" followed by a single character, e.g. "ab1.pas", "ab2.pas" and "ab\_.pas". **Attention:** in this case Delphi2C# will try to translate also files with other extensions than "\*.pas". This will lead to errors for "\*.txt" files or "\*.inc"-files etc.

### -t TARGET

The specification of a target is optional. If there is no, all translated files will be written into the directory of the source files. A target directory has to be specified, if the files shall be preprocessed only.

### -r RECURSIVE

By the optional parameter "-r" you can force a recursive search for source files in all subdirectories.

### -pause

With the optional parameter "-pause" you can keep the console window opened until a key is pressed. So you can read the messages, which were produced. Without this parameter the console window is closed as soon as the translations are finished.

## 7 What is translated

Delphi2C# handles nearly all kinds of the Delphi syntax.

- Tokens
- File layout
- Types
- Variables
- Operators
- Assignments
- Routines
- Special RTL/VCL functions
- Properties
- Statements
- Reading and Writing
- Method pointers

New features since Delphi 7

### 7.1 Tokens

At the token level following points have to be regarded:

- Case sensitivity
- Ampersands
- Simple substitutions
- String constants vs. single characters

#### 7.1.1 Case sensitivity

Expressions which are different only by case are regarded as identical in Delphi. Therefore a preprocessor is executed before the real translation. The preprocessor replaces all later occurrences of expressions which are different from the first occurrence only by the notation found first. The preprocessor provides the conditional compilation of the code at the same time.

Unification of the notations isn't applied to the code areas, where CSHARP is defined.

If an identifier for the original name at a refactoring item isn't contained in the list of notations, it's notation will be used for all notations of the identifier in the generated code.

There are some fixed identifiers, which cannot be modified by the list of identifiers.

#### 7.1.2 Ampersand

By means of an ampersand Delphi keywords can be used as identifiers, e.g. `\Embarcadero\Studio\19.0\source\rtl\win\winrt\WinAPI.ShlObj.pas` line 11032:

```
type
```

```
tagDROPDESCRIPTION = record
  &type: TDropImageType;
```

In such cases it is recommended either to let the pre-processor substitute such expressions or to modify the source code. Otherwise Delphi2C# simply ignores such ampersands, that means "&type" becomes "type". In that case the parser will stop at that position, because of the unexpected *type* keyword. If "&type" is substituted for example by "amps\_type" everything works well. You even can let the translator make a second substitution from "amps\_type" to "type", if you like. This substitution is made after "amps\_type" has been recognized as an identifier.

Another example is in \Embarcadero\Studio\19.0\source\rtl\win\winrt\WinAPI.DataRT.pas line 598:

```
property &Implementation: Xml_Dom_IXmlDomImplementation read get_Implementation;
```

\rtl\win\winrt\WinAPI.Devices.pas line 6078:

```
property &Function: Word read get_Function;
```

\rtl\win\winrt\WinAPI.CommonTypes.pas line 138/439/544/6163...

```
&End
```

### 7.1.3 Simple substitutions

Many key words and operators can be replaced one to one. There is a long list of such substitutions. A few examples are:

begin	{
end	}
record	struct
:=	=
=	==
<>	!=
and	&&
boolean	bool

### 7.1.4 String constants and single characters

The apostrophes of the string constants are replaced by quotation marks. The treatment of the characters is more difficult. Depending on context the apostrophes are left or replaced by quotation marks.

'1' :	->	case '1' :
string_id + '1'	->	string_id + "1"

## 7.2 File layout

*Delphi2C#* creates for each Delphi source file an according C# target file. Records and classes of the Delphi interface part, simply become according records and classes in the generated C# file. But in contrast to Delphi in C# constants, variables and routines cannot be declared outside of a class. Therefore in C# two classes are declared, which contain the global elements as static members. The first of these classes contains the constants, variables and routines of the Delphi interface part and the second class those of the implementation part. If there are records or classes declared in the implementation part of the Delphi code, these element also become member of the second class.

In the following example the placements of typical code elements in Delphi on the one side and in C# on the other side are contrasted:

### Delphi

```
unit test;

interface

type
  TFoo = class
  private
    procedure foo;
  end;

const
  foo = 0;
  procedure foo_proc;

implementation

const
  bar = 0;

type
  TBar = class
  private
    procedure foo;
  end;

procedure TFoo.foo;
begin
end;

procedure foo_proc;
begin
end;

procedure TBar.foo;
begin
```

### C#

```
using static test.testInterface;
using static test.testImplementation;
using System;
using static System.SystemInterface;

namespace test
{
  public class TFoo : TObject
  {
    private void foo()
    {
    }

    public TFoo() {}
  };

  public class testInterface
  {
    public const int foo = 0;
    public static void foo_proc()
    {
    }

  } // class testInterface

  public class testImplementation
  {
    public const int bar = 0;

    public class TBar : TObject
    {
      private void foo()
      {
      }

      public TBar() {}
    };
  };
}
```

```

end;

procedure bar_proc;          public static void bar_proc()
begin                        {
end;                          }

end.                          } // class testImplementation
                               } // namespace test

```

This diagram also shows, that in C# there is no difference between declaration and implementation of a routine; the function body of a routines is written immediately at the place where the Delphi declaration had been.

At the beginning of the C# file there are two uses clauses, which allow the use of the public elements of *testInterface* and of *testImplementation* inside of the whole file *test.cs* without qualification. Next *System.cs* is included, so that the classes, constants and routines which correspond to the according entities of the Delphi system can be used without qualification too. Finally with the clause "using static System.SystemInterface" the constants, variables and routines of the interface part of the System namespace can be accessed, if the current test file would be used by another unit, accordingly uses clauses would be created:

Comments can appear at many places in a file,

## 7.2.1 Uses clauses

Delphi uses clauses become to using directives in C#. For example:

```

unit test;

interface

uses System.SysUtils;
...

```

becomes to:

```

using System.SysUtils;
using static System.SysUtils.SysUtilsInterface;
using static test.testInterface;
using static test.testImplementation;
using System;
using static System.SystemClass;

namespace test
{
...

```

There are two using directives for all used files, because all files that were translated from Delphi with Delphi2C# contain an extra class with the constants and routines of the interface part of the original file, as described for the general file layout.

## 7.2.2 System Namespace

At the translation of Delphi code with *Delphi2C#* the C# System namespace is extended by the entities of the Delphi System. Therefore the *Delphi2C#* installation provides the file *System.cs*. *System.cs* corresponds to *System.pas* and has roughly the following structure:

```
namespace System
{
    // System classes
    ...

    public class TObject
    {
        ...

    public class TDateTime
    {
        ...

    public class SystemInterface
    {
        // System constants

        public const int MaxInt = Int32.MaxValue;
        ...

        // System functions

        public static string Concat(params string[] strings)
        {
            ...

        public static void SetLength(ref string xs, int xi)
        {
            ...

        static void System_initialization()
        {
            ...

        static void System_finalization()
        {
            ...

    } // SystemInterface
} // namespace System
```

*System.cs* starts with the definitions of some fundamental Delphi classes like *TObject*, *TDateTime* etc. and some helper classes like *PChar*, *Pointer* and so on. The second part of *System.cs* consists of the definition of a class called *SystemInterface* which contains public constants and static functions like e.g. *MaxInt*, *Concat* and *SetLength*, which correspond to the constants and functions of the Delphi System with the same name.

All C# files which are generated from Delphi with *Delphi2C#* are using *System.cs* by means of the lines:

```
using System;
using static System.SystemInterface;
```

*System.cs* cannot be generated automatically from *System.pas*, because *System.pas* uses special Delphi internal conventions. For example the *SetLength* function is declared there as:

```
procedure _SetLength(s: _PShortStr; newLength: Byte);
```

Moreover, *System.pas* is incomplete on the one hand and on the other hand it contains stuff, which isn't needed in C#.

There is an extended *System.pas*, which has to be set in the translation options, to let Delphi2C# know the signatures of the system routines.

### 7.2.3 Comments

All comments are output essentially unchanged at the corresponding positions. Line comments remain totally unchanged, while bracketing is translated from

```
{...}
or
(*...*)
to
/*...*/
```

Sometimes there are difficulties to output comments at the desired positions. The following example shows on the right side what Delphi2C# does with the Delphi source on the left side:

<pre>unit comments;  interface  type   TFoo = class   private     function foo : boolean;   end;</pre>	<pre>using static comments.commentsInterface; using static comments.commentsImplementation; using System; using static System.SystemInterface;  namespace comments {   public class TFoo : TObject   {     /*-----     Name   :     Result :     -----*/     private bool foo()     {       bool result = false;       result = true;       return result;     }      public TFoo() {}   };   /*-----</pre>
--	---



```

function foo2: boolean;

implementation
var
  bar : integer = 0; // comment to bar

(*-----
Name   :
Result :
-----*)
function TFoo.foo: boolean;
begin
  result := true;
end;

(*-----
Name   : Foo2
Result :
-----*)
function foo2: boolean;
begin
end;

end.

Name   : Foo2
Result :
-----*/
public static bool foo2()
{
  bool result = false;
  return result;
}

} // class commentsInterface

public class commentsImplementation
{
  public static int bar = 0; // comment to bar
} // class commentsImplementation
} // namespace comments

```

The Delphi member function declaration *foo* and its definition in C# are combined into one text at the position of the original declaration. A comment before the definition has to be shifted to that place too. The same applies to the declaration and definition of the free function *foo2*. The comment to the variable *bar* is output directly behind of the variable declaration. Sometimes Delphi2C# cannot clearly decide, which comment line belongs to a multi line comment in front of a function definition and a comment which explains an entity afterwards. For the second case Delphi2C# only takes single line comments, which follow directly on a statement.

## 7.3 Types

There are built-in types in Delphi and also new types can be defined in sections of a source file which begin with the *type* keyword.

The most simple form of a type definition is just to define another name for an existing type. E.g.:

```

WCHAR = WideChar;

var
  c : WCHAR;

```

In C# there is no true equivalent of such a type definition. Delphi2C# ignores such definitions and if an identifier is found which is the name of a type Delphi2C# takes and outputs that name. From the three lines of code above just remains:

```
char w;
```

Other types that can be defined are:

- Records, Classes and Interfaces
- Arrays
- Enumerated types
- Ranges
- Sets

### 7.3.1 Records, Classes, Interfaces

Delphi and C++ have is the same concept of classes. Delphi records become to structures in C++. Their concepts are similar too. In C# interface types can be defined similar to Delphi.

#### 7.3.1.1 Record

A record mainly consists in public data elements, but also may have methods and sub-records. In Delphi a record also may have a variant part.

In C# a record may not have a constructor without parameters.

Delphi records can contain arrays with a fixed size, e.g.:

```
TFormatSettings = record
...
  ShortMonthNames: array[1..12] of string;
```

A C# structure however only can contain arrays with undefined size.

```
public struct TFormatSettings
...
  public string[] ShortMonthNames;
```

In such and similar case Delphi2C# inserts a method called "CreateRecordMembers" into the structure, which creates the array with defined size:

```
public void CreateFixedArrays()
{
  ShortMonthNames = new string[12/*# range 1..12*/];
  ...
}
```

In addition for all structures a public static function called "CreateRecord" is inserted:

```
public static TFormatSettings CreateRecord()
{
    TFormatSettings tmp = new TFormatSettings();
    tmp.CreateRecordMembers();
    return tmp;
}
```

*CreateRecord* is similar to a Delphi constructor, but it is not called "Create", because the records often already have "Create" functions, as it is the case for TFormatSettings.

At the places where record variable are declared in the Delphi code, Delphi2C# now writes code like:

```
TFormatSettings result = TFormatSettings.CreateRecord();
```

Every record is initialized in that way, regardless whether it contains fixed array or not. For records which don't have fixed sized arrays the *CreateRecord* method simply looks like:

```
public struct TRecord
{
    ...
    public static TRecord CreateRecord(){return new TRecord();}
};
```

However, if a record type is replaced by another one by refactoring, the initialization is done by "new":

```
Guid result = new Guid();
```

#### 7.3.1.1.1 Variant parts in records

C# structures cannot have variant parts as Delphi records can have. However by means of the StructLayout(LayoutKind.Explicit) and FieldOffset attributes, the behavior can be reproduced.

```
TRect = packed record
    case Integer of
        0: (Left, Top, Right, Bottom: Longint);
        1: (TopLeft, BottomRight: TPoint);
    end;

->

[StructLayout(LayoutKind.Explicit)]
public struct TRect
{
    /*# 0*/
    [FieldOffset(0)]
    public int Left;
    [FieldOffset(4)]
    public int Top;
    [FieldOffset(8)]
    public int Right;
    [FieldOffset(12)]
    public int Bottom;
    /*# 1*/
    [FieldOffset(0)]
    public TPoint TopLeft;
    [FieldOffset(4)]
    public TPoint BottomRight;
    public static TRect CreateRecord(){return new TRect();}
```

```
};
```

### 7.3.1.2 Class

A typical class consists may have following additional elements:

- Ancestor
- Constructor
- Destructor
- Class methods
- Abstract methods

#### 7.3.1.2.1 Ancestors

If no ancestor type is specified when declaring a new object class, Delphi automatically uses *TObject* as the ancestor. In C# *TObject* has to be quoted explicitly.

```
TNewClass = class ...
->
class TNewClass : TObject ...
```

#### 7.3.1.2.2 Constructors

In Delphi a declaration of constructors start with the keyword *constructor* followed by an arbitrary name. In C# is the name of the of the class also the name of the constructor.

```
constructor classname.foo; -> classname::classname ( )
```

- Constructor of the base class
- Initialization lists
- Addition of missing constructors
- Virtual constructors
- Problems with constructors

##### 7.3.1.2.2.1 Constructor of the base class

In Delphi and C# the order of construction of the derived and the base classes is differently. In Delphi the derived class is constructed first, while in C# the constructors of the base classes are executed automatically, before the constructor of the derived class is executed. If the base class has no standard constructor (= constructor without parameters) the base class constructor has to be called in the initialization list with the according parameters. The constructors of the ancestor classes are executed in Delphi only, if they are called explicitly from in the written code. In such cases *Delphi2C#* tries to find this call and puts it into the initialization list:

```
type
TFoo = class
  constructor Create(Owner: TComponent);
```

```

end;

implementation

constructor TFoo.Create(Owner: TComponent);
begin
    inherited Create(Owner);
end;

->

public class foo : TObject
{
    public foo(TComponent Owner)
    : base(Owner)
    {
    }
};

```

There is a second reason, why this shift is necessary: in C# the explicit call of an ancestor constructor in the derived constructor has no effect. (A temporary instance of the base class will be created only.)

Base class constructors without parameters are called automatically in C#. *Delphi2C#* preserves the original calls of such constructors as line comments.

```

constructor foo.Create();
begin
    inherited Create;
end;

->

__fastcall foo::foo ( )
{
    // inherited::Create;
}

```

The example above was shortened. In fact *Delphi2C#* for each constructor creates a second function with the function body of the Delphi constructor.

```

type
TFoo = class
    constructor Create(Owner: TComponent);
end;

implementation

constructor TFoo.Create(Owner: TComponent);
begin
    inherited Create(Owner);
end;

->

public class foo : TObject
{
    public foo(TComponent Owner)
    : base(Owner)
    {
        Create(Owner);
    }
    public void Create(TComponent Owner)
    {
        // # base.Create(Owner);
    }
}

```

```

    public foo() {}
};

```

This additional method isn't a good programming style and even can produce errors, but it is needed, when a constructor is called directly to initialize an existing instance again.

```

var
  foo : TFoo;
begin
  foo := TFoo.Create;
  foo := TFoo.Create(...);

```

#### 7.3.1.2.2.2 Initialization lists

In Delphi and C# member variables like other variables too are initialized automatically with according default values. But array with fixed size - as *FCoord* in the following example - have to be created at construction of a class instance. If a field is a record, it will be initialized too by a call of *CreateRecord*.

<u>Delphi source</u>	<u>C# translation</u>
<pre> TBase = class public   constructor Create(arg : Integer);   destructor Destroy; private   FList : TList;   FCoords: array[0..3] of Longint;   FTimeout: Longint; end;  constructor Base.Create(arg : Integer); begin end; </pre>	<pre> public class TBase : TObject {   public TBase(int arg)   {     Create(arg);   }   public void Create(int arg)   {   }   ~TBase()   private TList FList;   private int[] FCoords = new int[4/*# range 0.. 3*/];   private int FTimeout;    public TBase() {} }; </pre>

If the members are initialized explicitly in Delphi, *Delphi2C#* tries to find the according statements and puts them into the initialization list of the class constructor:

<pre> constructor Base.Create(arg : Integer); begin   FList := TList.Create;   FI := arg;   if arg &lt;&gt; \$00 then     FTimeout := arg   else     FTimeout := DefaultTimeout; end; </pre>	<pre> __fastcall Base::Base( int arg ) : FI(arg),   FList(new TList),   FTimeout(0) {   if ( arg != 0x00 )     FTimeout = arg;   else     FTimeout = DefaultTimeout; } </pre>
--	---

The use of initialization lists is more efficient in C# than to initialize the variables in the body of the constructor. But sometimes there is a problem with this method. For example, the initialization of the member *FTimeout* depends of the value of the *arg* parameter. As shown in the next example *Delphi2C#* tries to take care about such cases. But *Delphi2C#* cannot find all such hidden dependencies, as in the following example:

```

constructor Derived.Create;
var
  i : Integer;
begin
  i := SomeFunction;
  inherited Create(i);
end;

__fastcall Derived::Derived( )
: inherited( i ),
  FB(false)
{
  int i = 0;
  i = SomeFunction;
}

```

In such cases constructors have to be corrected manually like:

```

__fastcall Derived::Derived( )
: inheritd( SomeFunction() )
{
}

```

Unfortunately, there is another problem with the order of the initializations. in C# the order in the initializer list is ignored. Member variables are always initialized in the order they appear in the class declaration. In the following example:

```

TInit = class(TObject)
  FName1, FName2, FName4, FName3 : String;
  constructor Create(Name1, Name2, Name3 : String);
end;

implementation

constructor TInit.Create(Name1, Name2, Name3 : String);
begin
  FName1 := Name1;
  FName2 := Name2;
  FName3 := Name3;
  FName4 := FName3;
end;

```

a strict initialization of the member variables in the order in which they are declared would lead to:

```

__fastcall TInit::TInit( String Name1, String Name2, String Name3 )
: FName1(Name1),
  FName2(Name2),
  FName4(FName3),
  FName3(Name3)
{
}

```

Obviously, this is not correct. Therefore Delphi2C# uses the following strategy: as long as the initialization statements in the constructor are in the order of the declarations, they are shifted into the initializer list. For all other member variables follows initialization code in the body of the constructor.

```

__fastcall TInit::TInit( String Name1, String Name2, String Name3 )
: FName1(Name1),
  FName2(Name2),
  FName3(Name3)
{
  FName4 = FName3;
}

```

## 7.3.1.2.2.3 Addition of missing constructors

Unlike in Delphi, constructors of base classes cannot be called directly in C#. Additional constructors have to be defined in the derived class. Delphi2C# inserts missing constructors in C# automatically. So, resuming the previous example, an additional standard constructor is created, which can be used with all classes, which are derived from *TObject*:

Unlike in Delphi, constructors of base classes cannot be called directly in C#. If there are public constructors in the base classes with different signatures as any constructor of the derived class, these constructors are generated for the derived class too. Especially in Delphi all classes are derived from *TObject* and inherit its default constructor. Therefore Delphi2C# generates a default constructor for each derived class, even if such a constructor doesn't exist in the original Delphi code.

Sometimes a lot of additional code has to be produced for C# classes. For example a class, which is derived from *Exception* has more than ten constructors. Inside of each constructor the constructor of the base class has to be called in the initialization list

```
public class MyException : Exception
{
    public EArgumentException(string Msg) : base(Msg) {}
    public EArgumentException(string Msg, object[] Args) : base(Msg, Args) {}
    .
    .
};
```

## 7.3.1.2.2.4 Virtual constructors

In Delphi constructors can be used like virtual functions in C#. This can be demonstrated at the example, which is also used in the section about class method. A class method might be called for a base class and another class derived from it:

```
pBase := TBase.Create;
pDerived1 := TDerived1.Create;

pDerived1->ClassMethod( pDerived1, 1 );
```

Inside of the class method a new object of the class is created:

```
class function TBase.ClassMethod(xi: Integer): Integer;
begin
    with Create do <-- new object from virtual constructor
    begin
        Init; <-- virtual method
        Done;
        Free;
    end;
    result := xi;
end;
```

The *Init* method might be virtual. In this case the *Init* method of *TDerived1* will be called. That means, an instance of *TDerived1* has been created, because *ClassMethod* was called for a *TDerived1* object. If *ClassMethod* were called for a *TBase* object, a *TBase* object would have been created and *TBase.Init* would have been called.



## 7.3.1.2.2.5 Problems with constructors

Summarizing, there remain two problems for which the translated constructors have to be checked:

1. the order of construction of the derived and the base classes is differently in Delphi and C#
2. member variables should be initialized in at the beginning of the constructor code in the initialization list. But sometimes the value can depend on other calculations and *Delphi2C#* cannot recognize this.

There is still another problem with special constructors. In Delphi there can be several constructors with the same signature

```
TCoordinate = class(TObject)
public
    constructor CreateRectangular(AX, AY: Double);
    constructor CreatePolar(Radius, Angle: Double);
private
    x,y : Double;
end;

constructor TCoordinate.CreateRectangular(AX, AY: Double);
begin
    x := AX;
    y := AY;
end

constructor TCoordinate.CreatePolar(Radius, Angle: Double);
begin
    x := Radius * cos(Angle);
    y := Radius * sin(Angle);
end
```

After translation the two constructors become ambiguous:

```
public class TCoordinate : TObject
{
    public TCoordinate(double AX, double AY)
    {
        CreateRectangular(AX, AY);
    }
    public void CreateRectangular(double AX, double AY)
    {
        x = AX;
        y = AY;
    }
    public TCoordinate(double Radius, double Angle)
    {
        CreatePolar(Radius, Angle);
    }
    public void CreatePolar(double Radius, double Angle)
    {
        x = Radius * cos(Angle);
        y = Radius * sin(Angle);
    }
    private double x;
    private double y;

    public TCoordinate() {}
};
```

In such cases the conflict has to be avoided manually.

## 7.3.1.2.3 Destructors

In Delphi a declaration of destructors start with the keyword *destructor* followed by an arbitrary name. In C# the name of the of the class is also the name of the destructor preceded by the the character '~'.

```
destructor classname.foo;    ->    classname::~~classname ( )
```

Delphi2C# tempts to find calls of destructors of the base class and to comment them out in C#. Thereby is assumed that the destructor of the base class is virtual. This has to be checked by the user.

```
destructor foo.Destroy();    ->    foo::~~foo ( )
begin
  FreeAndNil(m_Messages);
  inherited Destroy;
end;
                                {
                                FreeAndNil ( m_Messages );
                                // todo check:  inherited::Destroy;
```

## 7.3.1.2.4 class methods

Delphi class methods are similar to C# static methods, but there are two differences:

1. Delphi class methods can be virtual, C# static methods cannot. Therefore *Delphi2C#* has to use a tricky construction to reproduce this ability of Delphi.
2. In the defining declaration of a class method, the identifier *Self* represents the class where the method is called. In C++ however inside of a static function there is no counterpart to Delphi's *Self* (*this* isn't defined her).

Therefore the two cases have to be distinguished at the translation of Delphi class methods to C++ and Delphi's *Self*-instance requires additional treatment:

```
non virtual class methods
virtual class methods
Self instance
```

## 7.3.1.2.4.1 non virtual class methods

Delphi non virtual class methods are converted to C# static methods. They can be called through a class reference or an object reference:

```
type
  TBase = class(TObject)
  public
    class function ClassMethod(xi: Integer): Integer;
  end;
...
var
  pBase: TBase;
  i : Integer;
begin
  i := TBase.ClassMethod(0); // calling through a class reference
  // ...
```

```
i := pBase.ClassMethod(0); // calling through an object reference
```

This is translated in the following way:

```
public class TBase : TObject
{
    public static int ClassMethod(int xi)
    {
        ...
    };
    ...

    TBase* pBase = NULL;
    int i = 0;
    TBase.ClassMethod(0); // calling through a class reference
    // ...
    TBase.ClassMethod(0); // calling through an object reference
}
```

The class method cannot be called through an object reference.

#### 7.3.1.2.4.2 virtual class methods

Because there are no virtual static methods in C# Delphi2C# has an option, which allows to convert virtual class methods either to static non-virtual methods or to virtual non-static methods.

The first case results into the same code as for non-virtual class methods. If the virtual class methods aren't overridden, this is obviously the best option. But if the methods are overwritten, the virtual class methods have to be converted to virtual C# methods. Then these methods cannot be called through a class type expression in C# any more, If they are called that way in the Delphi code, an adequate instance of the class has to be provided in C#. If the option to create meta classes is enabled Delphi2C# provides these instances automatically:

```
TBase = class(TObject)
public
    class function ClassVirtual(xi: Integer): Integer; virtual;

var
    base : TBase;
begin
    base.ClassVirtual(0);
    TBase.ClassVirtual(0);
    TDerived.ClassVirtual(0);
end;

->

public class TBase : TObject
{
    public /*#static*/ virtual int ClassVirtual(int xi)
    {
        ...
    };
    ...

    base.ClassVirtual(0);
    ClassRef<TBase>.getClassInstance().ClassVirtual(xi);
    ClassRef<TDerived>.getClassInstance().ClassVirtual(xi);
}
```

By calling *ClassVirtual* through the *TBase* pointer *base*, the correct version of *ClassVirtual* will be called as for for non-static methods too. The correct version of *ClassVirtual* will be called in C# too, if the class name is used in Delphi, because Delphi2C# replaces the class names by the according class reference instances.

#### 7.3.1.2.4.3 Self instance

Like the "this" pointer in C++ is an implicit parameter to all member functions, in Delphi the "Self" instance is an implicit parameter to class functions. Other class methods can be called there through this instance and they can be called by hidden use of "Self". "Self" must not appear in the code. For example:

```
class function TBase.ClassMethod(xi: Integer): Integer;
begin
with Create do <-- new object from a virtual constructor of Self
begin
  Init;
  Done;
  Free;
end;
result := xi;
end;
```

Delphi2C# can convert this code adequately only, if the option to create meta classes is enabled. The code then becomes to:

```
public static int ClassMethod(int xi)
{
  int result = 0;
  /*# with Create do */
  {
    TBase with0 = (TBase) SCreate();
    with0.Init();
    with0.Done();
    with0.Free();
  }
  result = xi;
  return result;
}
```

"SCreate" is a static method, which returns a new instance of *TBase*.

#### 7.3.1.2.5 abstract methods

Like Delphi also C# knows abstract methods. The most natural way of translation is for example:

```
function Get(Index: Integer): Integer; virtual; abstract;
->
abstract public int Get(int Index)/*# virtual */;
```

## 7.3.1.2.6 Visibility of class members

In Delphi a private or protected member is visible anywhere in the module where its class is declared.  
In C# a private or protected member is visible only in the class.

## 7.3.1.2.7 Creation of instances of classes

VCL classes have to be created with `new` in C#.

```
TList.Create(NIL)    ->    new TList(NULL)
```

## 7.3.1.3 Interfaces

In **Delphi** interface types can be defined like in the following lines of code:

```
IConverter = interface
  ['{GUID}']
  function convert(Source : String): String;
end;

TConverter = class(TInterfacedObject, IConverter)
public
  //...
  function convert(Source : String): String;
end;
```

**C#** also knows this keyword, but the GUID has to be written differently:

```
[Guid("GUID"), ComVisible(true)]
public interface IConverter : IInterface
{
  string convert(string Source);
};

public class TConverter : TInterfacedObject, IConverter
{
  //...
  public string convert(string Source)
  {
    ...
  };
};
```

All Interfaces inherit from at least from *IInterface*.

## 7.3.2 Arrays

Delphi distinguishes between Static arrays with a fixed size and Dynamic arrays with a variable size. Both can be passed to routines as parameters. There is a third kind of array: Open arrays, which can be passed to routines. Open arrays are arrays of unspecified size with elements, that all have the same type.

Some of the following examples stem from: <http://www.delphibasics.co.uk/RTL.asp?Name=array>

### 7.3.2.1 Static arrays

Static arrays can have one or more dimensions. The declarations in C# can be derived from the Delphi declarations in a straightforward manner.:

```
var
  // Define static arrays
  wordArray : Array[Word] Of Integer;    // Static, size=High(Word)
  multiArray : Array[byte, 1..5] Of char; // Static array, 2 dimensions
  rangeArray : Array[5..20] Of string;   // Static array, size = 16

->
  // Define static arrays
  int[] wordArray = new int[65536/*# word*/]; // Static, size=High(Word)
  char[,] multiArray = new char[256/*# byte*/, 5/*# range 1.. 5*/]; // Static array, 2 dimensions
  string[] rangeArray = new string[16/*# range 5.. 20*/]; // Static array, size = 16
```

While in Delphi the lower bound and the upper bound have to be defined, in C# arrays are always zero based, Array indices are corrected by *Delphi2C#*.

### 7.3.2.2 Dynamic arrays

Dynamic arrays become C# arrays:

```
var
  // Define dynamic arrays
  byteArray : Array Of byte; // Single dimension array
  multiArray : Array Of Array Of string; // Multi-dimension array
```

```
->
// Define dynamic arrays
byte[] byteArray = new byte[]{}; // Single dimension array
string[][] multiArray = new string[][]{}; // Multi-dimension array
```

Dynamic arrays are accepted as parameters only, if the type of the array is defined explicitly and if the function expects this type.

```
System.Generics.Collections
FSysLangs: TArray<TLangRec>;
private List<TLangRec> FSysLangs = new List<TLangRec>();

    TLangRec tmp0 = FSysLangs[FSysLangs.Count - 1 /*# High(FSysLangs) */];
    tmp0.FName = GetLocaleDataW(AID, (uint) LOCALE_SLANGUAGE);
    FSysLangs[FSysLangs.Count - 1 /*# High(FSysLangs) */] = tmp0;
```

### 7.3.2.3 Array indices

While in Delphi the lower bound and the upper bound of a static array have to be defined, in C# arrays are always zero based, i.e. the undermost index is 0 and the topmost index is the size of the array minus 1.

If the lower bound of an array isn't null, Delphi2C# corrects an index by which the array is accessed automatically by subtraction of the lower bound.

Example:

```
var
arr : array [1..3] of integer;
i : integer;
begin
    for i := low(arr) to high(arr) do
        arr[i] := 0;
    end;
```

is translated to:

```
int arr [ 3 ];
int i;
for ( i = 1; i <= 3; i++)
    arr[i - 1] = 0;
```

### 7.3.2.4 Initializing arrays

The initialization of arrays in Delphi and C# looks very similar. For example the initialization of an array of *TStyleRecords*:

```
type
TStyleRecord = record
    Name : string;
    Color : TColor;
    Style : TFontStyles;
end;
TStylesArray = Array[0 .. 2] Of TStyleRecord;
```

```

const

DefaultStyles : TStylesArray = (
  (Name : 'tnone';   Color : clBlack;  Style : []),
  (Name : 'tstring'; Color : clMaroon; Style : []),
  (Name : 'tcomment'; Color : clNavy;  Style : [fsItalic])
);

->

public struct TStyleRecord
{
  public string Name;
  public int /*-0x7FFFFFFF - 1..0x7FFFFFFF*/ Color;
  public TSet Style;
  public static TStyleRecord CreateRecord(){return new TStyleRecord();}
};

public static readonly TStyleRecord[] DefaultStyles = {new TStyleRecord{Name="tnone", Color=clBlack, S
new TStyleRecord{Name="tstring", Color=clMaroon, Style=new TSet()},
new TStyleRecord{Name="tcomment", Color=clNavy, Style=new TSet() << (int) TArrays_FontStyle.fsaItalic}

```

### 7.3.2.5 Array parameters

Static and dynamic arrays can be passed in Delphi to the same function, if it expects an open array parameter. In the C# translation static and dynamic arrays are incompatible types. Static arrays are passed to functions as open array. Dynamic array can be passed to a function only, if the type of the dynamic array is defined explicitly and the function expects this type. Array of const parameters allow to pass an array on the fly.

#### 7.3.2.5.1 Open array parameters

The concept of open arrays allow arrays of different sizes to be passed to the same procedure or function.

```

function Sum(Arr: Array of Integer): Integer;
var
  i: Integer;
begin
  Result := 0;
  for i := Low(Arr) to High(Arr) do
    Result := Result + Arr[i];
end;

```

In the C# translation the array member functions *GetUpperBound* and *GetLowerBound* are used instead of the Delphi functions *High* and *Low*.

```

public static int Sum(int[] Arr)
{
  int result = 0;
  int i = 0;
  int stop = 0;
  result = 0;
  for(stop = Arr.GetUpperBound(0), i = Arr.GetLowerBound(0); i <= stop; i++)
  {
    result = result + Arr[i];
  }
}

```



```
    return result;
}
```

If a temporary *set* of values can be passed as open array parameter too:

```
procedure Log(strings : array of string)
begin end;

procedure foo;
begin
Log(['one', 'two', 'three']);
end;

->

public static void Log(string[] strings)
{
}

public static void foo()
{
    Log(new string[]{"one", "two", "three"});
}
```

#### 7.3.2.5.2 Static array parameter

A static array is passed to functions as an open array parameter. The additional second parameter for the upper bound of the array is inserted into the declaration of the function automatically and is passed to the function automatically too. The upper bound of the array is calculated by means of a macro:

```
#define MAXIDX(x) (sizeof(x)/sizeof(x[0]))-1

procedure foo(Arr: Array of Char)
begin end;

procedure bar();
var
    chararray : Array [1..10] of Char;
begin
    foo(chararray);
end;
```

is translated to:

```
public static void foo(char[] Arr)
{
}

public static void bar()
{
    char[] charArray = new char[10/*# range 1.. 10*/];
    foo(charArray);
}
```

#### 7.3.2.5.3 Dynamic array parameter

A Delphi function accepts a dynamic array as parameter, if it is defined explicitly:

```
type
strarray = Array of String;
procedure Check(aSources : strarray)
begin end
```

```
->
public static void Check(string[] aSources)
{
}
```

#### 7.3.2.5.4 array of const

"Array of const" parameters look similar to open array parameters.

```
procedure foo(Args : array of const);
```

However, while all elements of an open array have the same type, elements of different types can be passed as an *array of const*. Indeed the *array of const* is an open array of *TVarRec* elements and *TVarRec* is a variant type which can contain the single values of different types. For C# *TVarRec* hasn't to be re-coded. Because in C# all value types can be converted to *object*, *object* can be used instead of *TVarRec*.

```
procedure foo(Args : array of const)
begin end;

->

public static void foo(object[] Args)
{
}
```

A following complex example is from:

[\*\*\*\*]

<https://stackoverflow.com/questions/19532228/how-do-i-build-an-array-of-const>

```
function VarRecToStr( AVarRec : TVarRec ) : string;
const
  bool : array[boolean] of string = ('False', 'True');
begin
  case AVarRec.VType of
    vtInteger:   result := IntToStr(AVarRec.VInteger);
    vtBoolean:  result := bool[AVarRec.VBoolean];
    vtChar:     result := AVarRec.VChar;
    vtExtended: result := FloatToStr(AVarRec.VExtended^);
    vtString:   result := AVarRec.VString^;
    // vtPointer
    vtPChar:    result := AVarRec.VPChar;
    vtObject:   result := AVarRec.VObject.ClassName;
    vtClass:    result := AVarRec.VClass.ClassName;
    vtWideChar: result := AVarRec.VWideChar;
    //vtPWideChar: Result := AVarRec.PWideChar^;
    vtAnsiString: result := string(AVarRec.VAnsiString);
    vtCurrency:  result := CurrToStr(AVarRec.VCurrency^);
    vtVariant:  result := string(AVarRec.VVariant^);
    //vtInterface: (VInterface: Pointer);
    vtWideString: result := string(AVarRec.VWideString);
    //vtInt64:    Result := AVarRec.PInt64^;
    vtUnicodeString: result := string(AVarRec.VUnicodeString);
    //Reserved1: NativeInt;
    //VType:     Byte;
  else
    result := '';
  end;
end;

function VarArrayToStr( AVarArray : array of const ) : string;
var
  i : Integer;
```

```
begin
  result := '';
  for i := 0 to High(AVarArray) do
    result := result + VarRecToStr( AVarArray[i] );
  end;

procedure foo;
var
  s : string;
begin
  s := VarArrayToStr([1, true, 'a', 1.2, 'hello']);
end;
```

The conversion to an object is implicit and it's called boxing. The reverse *unboxing* operation, the conversion of an object to the contained value type, is explicit, that means for each value type the an according cast has to be done.

```
public static string VarRecToStr(object AVarRec)
{
  string result = string.Empty;
  string[] bools = new string[2/*# boolean*/]{"False", "True"};
  switch(Type.GetTypeCode(AVarRec.GetType()))
  {
    case vtInteger:
      result = ((int)AVarRec).ToString();
      break;
    case vtBoolean:
      result = bools[Convert.ToInt32(((bool)AVarRec))];
      break;
    case vtChar:
      result = ((char)AVarRec).ToString();
      break;
    case vtExtended:
      result = FloatToStr(((double)AVarRec));
      break;
    case vtString:
      result = ((string)AVarRec);
      break;
    // vtPointer
    case vtPChar:
      result = ((string)AVarRec);
      break;
    case vtObject:
      result = AVarRec.ClassName;
      break;
    case vtClass:
      result = AVarRec.ClassName;
      break;
    case vtWideChar:
      result = ((char)AVarRec).ToString();
      break;
    //vtPWideChar: Result := AVarRec.PWideChar^;
    case vtAnsiString:
      result = ((string) ((string)AVarRec));
      break;
    case vtCurrency:
      result = CurrToStr(((decimal)AVarRec));
      break;
    case vtVariant:
      result = ((string) AVarRec);
      break;
    //vtInterface: (VInterface: Pointer);
    case vtWideString:
      result = ((string) ((string)AVarRec));
      break;
    //vtInt64: Result := AVarRec.PInt64^;
    case vtUnicodeString:
      result = ((string) ((string)AVarRec));
      break;
    //Reserved1: NativeInt;
    //VType: Byte;
```

```

        default:
            result = "";
            break;
    }
    return result;
}

public static string VarArrayToStr(object[] AVarArray)
{
    string result = string.Empty;
    int i = 0;
    int stop = 0;
    result = "";
    for(stop = AVarArray.Length - 1 /*# High(AVarArray) */, i = 0; i <= stop; i++)
    {
        result = result + VarRecToStr(AVarArray[i]);
    }
    return result;
}

public static void foo()
{
    string s = string.Empty;
    s = VarArrayToStr(new object[]{1, true, 'a', 1.2D, "hello"});
}

```

There are ambiguous cases however. For example in C# the different string types are not distinguished. Delphi2C# cannot decide, which string case is the best. Here manual post-Processing is necessary.

#### 7.3.2.5.5 Array of const vs. set

DelphiXE2Cpp11 decides by the expected parameter type how the set argument is translated:

```

type
    TCharSet = set of Char;

procedure foo(arr : array of const);
procedure bar(set : TCharSet);

foo(['hello', 'world']);
bar(['hello', 'world']);

->

foo(new object[]{'h', 'w'});
bar(new TSet() << 'h' << 'w');

```

### 7.3.3 Enumerated types

The explicit definition of enumeration types is easy to translate.

```

Day = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
->
public enum Day {Mon,
                Tue,
                Wed,
                Thu,

```

```
    Fri,  
    Sat,  
    Sun };
```

However, an implicit definition is also possible in object Pascal within a variable declaration. It is decomposed for C# into an explicit type definition and the real declaration of the variable. The name of the type is derived from the name of the unit (...) by appending two underscores and a counter.

```
procedure foo;  
var  
  Day : (Mon, Tue, Wed, Thu, Fri, Sat, Sun);  
begin end;  
  
->  
  
public enum test__0 {Mon,  
    Tue,  
    Wed,  
    Thu,  
    Fri,  
    Sat,  
    Sun };  
  
public class testClass  
{  
  
  public static void foo()  
  {  
    test__0 Day = test__0.Mon;  
  }  
}
```

If the size of an array is specified by an enumerated type, the size is evaluated from the smallest and greatest value of the type.

```
type  
  TEnum = (cm1, cm2, cm3, cm4, cm5, cm6);  
  
var  
  foo : Array[TEnum] Of String;  
  
->  
  
public enum TEnum {cm1,  
    cm2,  
    cm3,  
    cm4,  
    cm5,  
    cm6 };  
public static string[] foo = new string[6/*# TEnum*/];
```

### 7.3.4 Ranges

Numeric ranges don't exist in C#. If a variable has a range type Delphi2C# deduces an underlying type of the range - mostly integer - and writes the original limits into the translated code as a comment:

```
type  
  TYearType = 1..12;  
  
var  
  year : TYearType;  
  
->  
  
private static int /*1..12*/ year = 0;
```

### 7.3.5 Sets

A Delphi set is simulated in the C# VCL by the class *TSet* in *DelphiSets.cs*.

```
public class TSet : IEnumerable
```

Every *TSet* consists of 256 bits, which can be set or unset.

```
var
  CharSet: set Of 'a'..'z';
  NumberSet: set Of 1..10;

->

private static TSet CharSet = new TSet();
private static TSet NumberSet = new TSet();
```

A set is created on the fly, if there is no explicit type-declaration of a set, as e.g. in:

```
MySet := ['a', 'b', 'c'];

->

MySet = new TSet() << 'a' << 'b' << 'c';
```

An example of a set constant is:

```
type
  TDay = (dMon, dTue, dWed, dThu, dFri, dSat, dSun);

const
  cDays: set Of TDay = [dMon .. dSun];

->

public enum TDay {dMon,
                 dTue,
                 dWed,
                 dThu,
                 dFri,
                 dSat,
                 dSun };

public static TSet cDays = TSet() <<
  (int) dMon << (int) dTue << (int) dWed << (int) dThu << (int) dFri << (int) dSat <<
  (int) dSun;
```

### 7.3.6 Order of lookup

The order by which symbols are looked up is different in Delphi and C#. Delphi tries to find a symbol in the last used unit at first and if isn't there Delphi will continue with the previous used unit. If both used units contain the same symbol, but defined differently, this doesn't matter, because Delphi will take just the definition, that it finds first. In the following example *MyType* will be an integer:

```
uses lookupinunits2, // LType = TestRecord;
    lookupinunits1; // LType = Integer;

Type
  MyType = LType;

implementation

procedure foo;
var
  i : integer;
  m : MyType;
  t : TestRecord;
begin
  m := i;
  // m := t; E2010 incompatible types: 'Integer' and 'TestRecord'
end;
```

In the translated C# code there are no type definitions, but *m* correctly becomes an integer, not a *TestRecord*.

```
public static void foo()
{
  int i = 0;
  int m = 0;
  TestRecord T = null;
  m = i;
  // m := t; E2010 incompatible types: 'Integer' and 'TestRecord'
}
```

Remark:

In C++ both definitions of *LType* would conflict. If *LType* would denote pointer types, then there would be an ambiguity.

## 7.4 Variables

In Delphi declarations of variables are done in a section of code which begins with the *var* keyword. A single declaration then consists in a name followed by a double point and the type:

```
var
  str : AnsiString;
```

In C# the type is followed by the name.

```
AnsiString str;
```

But beneath these "normal" variables, special kinds of variables also can be declared in sections

starting with:

```
threadvar
resourcestring
```

### 7.4.1 threadvars

In Delphi the keyword *threadvar* is used to declare variables using the thread-local storage.

```
threadvar
x: Integer;
```

In C# the ThreadStatic attribute can be used:

```
[ThreadStatic]
int x;
```

### 7.4.2 Resource strings

Delphi compiler has built-in support for resource strings. In Delphi there is a complex management of modules *Delphi2C#* makes resource strings to normal string constants

```
resourcestring
SIndexError = 'Index out of bounds: %d';
```

gets translated to:

```
public static string SIndexError = "Index out of bounds: %d";
```

In the code for *Delphi2C#* there is an additional makeshift. *TResStringRec* is rewritten as:

```
public struct TResStringRec
{
//public Pointer<HMODULE> Module;
//public Pointer<uint> Module;
//public uint Identifier;

public static implicit operator TResStringRec(string s)
{
return new TResStringRec() { FMessage = s };
}

public string FMessage {get; set;}
};
```

The implicit operator lets compile calls like:

```
ConvertError(new Pointer<TResStringRec>(SFormatTooLong));
```

In a constructor of an Exception the message is fetched by:

```
public Exception(Pointer<TResStringRec> ResStringRec)
: base(ResStringRec.Deref().FMessage)
{
}
```



## 7.5 Operators

Some of the names of Delphi operators are the same in C# as for example '>' and '>=', others are named differently as for example the assignment operator ':=' is '=' in C# and the equality operator '=' is '==' in C#. At the translation from Delphi to C# for most operators it suffices just to substitute the name of the operator. But there are two difficulties:

In C# two manners of use of the Delphi operators "and" and "or" have to be distinguished. The operator precedence in Delphi and C# is different. The in-operator has to be substituted in a special ways.

Also operator overloading has a different syntax.

### 7.5.1 boolean vs. bitwise operators

In C# two manners of use of the Delphi operators "and" and "or" have to be distinguished.

If these operators are between expressions which result in boolean values, then the complete expression results in a boolean value in accordance with the boolean logic. The boolean "and" operator in C# is "&&" and the boolean "or" operator in C# is "||".

If the "and" operator or the "or" operator is, however, enclosed by expressions which don't yield boolean values, then the results are connected bitwise. In this case the corresponding C# operators are "|" and "&".

### 7.5.2 operator precedence

In complex expressions, rules of precedence determine the order in which operations are performed. Delphi has four levels:

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

The first level is the highest precedence and the fourth level is the lowest. The equivalent operators are spread in C# like in C++ on 11 levels.

level	operators
1.	(address) & ! ~ // dereference *, unary + -
2.	* / %
3.	+ -
4.	<< >>
5.	< > <= >=
6.	== !=
7.	&
8.	^
9.	

```
10.    &&  
11.    ||
```

To reproduce the order in which expressions are performed in Delphi appropriate parenthesis must be inserted in C#.

For example, while in Delphi the *And* and *Or* operators have a higher priority than the equality operators, in C# equality operators are evaluated first. So at the translation of the following condition:

```
if attr And flag = flag then
```

according parenthesis are set in the C# output:

```
if( ( attr & flag ) == flag )
```

### 7.5.3 in-operator

The in-operator of Delphi is replaced by the "Contains" function of the Set class in C#. There is a special translation of the in-Operator in a for-in loop.

## 7.6 Assignments

A simple assignment statement in Delphi looks like:

```
A := B;
```

This becomes in C# to

```
A = B;
```

However, some simple assignments in Delphi are producing warnings or even bugs in C#. Therefore

explicit casts

are necessary in C#.

### 7.6.1 Explicit casts

Generally, if a variable of one type is assigned to another variable with another type this is possible without problems, if no information is lost. For example, if a shortint variable is assigned to an integer variable, there is no problem, because the size of shortint is one byte and the size of an integer variable is at least two bytes. If the assignment goes the other way round however in C# an explicit cast is necessary:

```
si : shortint;  
i  : integer;  
begin  
  i := si;  
  si := i;
```

becomes to:

```
sbyte si = 0;
int i = 0;
i = si;
si = (sbyte) i;
```

Delphi2C# always inserts the according casts, also when such casts are necessary to pass parameters to functions.

## 7.7 Routines

There are two kinds of routines in Delphi: procedures and functions. Both kinds may be declared first and defined later.

If a routine has no parameters in contrast to Delphi the calls of the routine in C# have to end with parenthesis.

```
foo;    ->  foo();
```

There are different kinds of parameters, which have to be translated accordingly. Sometimes parameters cannot be passed directly as in Delphi, but a temporary variable has to be created at fist, which then is passed.

Delphi nested routines also can be reproduced in C#.

### 7.7.1 Procedures and functions

Procedures are translated to void-functions

```
procedure foo; -> void foo();
```

The translation of functions is more complicated, because there aren't return-statements in Object-Pascal. Instead, the return value is assigned to a variable *Result*, which is implicitly declared in each function. In C# this variable must be declared explicitly and returned at the end of the function. Also to the Exit-function has to be replaced by a return-statement in C#.

```
function foo(i : Integer) : bar;    ->    bar __fastcall foo ( int i )
begin
  Result := 0;
  if i < 0 then
    EXIT
  else
    Result := 1;
end;
{
  bar result;
  result = 0;
  if ( i < 0 )
    return result;
  else
    result = 1;
  return result;
}
```

In addition, the function name itself acts as a special variable that holds the function's return value, as does the predefined variable *Result*. So the same translation as above results from:

```
function foo(i : Integer) : bar;
begin
  foo := 0;
  if i < 0 then
    EXIT
  else
    foo := 1;
end;
```

## 7.7.2 Declaration and definition

Routines may be declared first in the interface part of a unit and defined later in the implementation part. If a routine is the member of a class, the declaration always has to be separated from the definition.

```
interface
type
  TFoo = class
  private
    procedure foo;
  end;

  procedure foo;

implementation

procedure TFoo.foo;
begin
  ...
end;

procedure foo;
begin
  ...
end;
```

In C# this difference doesn't exist, there is only one place for a routine.

```
public class TFoo : TObject
{
  private void foo()
  {
    ...
  }

  public TFoo() {}
};

public class ...Class
{
  public static void foo()
  {
    ...
  }
} // ...Class
```

Member routines are written inside of the class definition and free routines become public static members of the extra class which is created for each unit.

## 7.7.3 Parameter types

Parameters either are passed to routines by value or by reference. Strings are passed as references, but behave as if they were passed by value (because of its copy-on-write technique). Further there are

constant parameters and untyped parameters - and array parameters. The different cases of single parameters and how they are translated are listed below. The Delphi *var* keyword becomes to *ref* in C# and *const* doesn't exist in C#.

```

type
MyRecord = record
end;

PInteger = ^Integer;

procedure Foo(param : Integer);
procedure Foo(const param : Integer);
procedure Foo(var param : Integer);
procedure Foo(out param : Integer);

procedure Foo(param : String);
procedure Foo(const param : String);
procedure Foo(var param : String);
procedure Foo(out param : String);

procedure Foo(param : Pointer);
procedure Foo(const param : Pointer);
procedure Foo(var param : Pointer);
procedure Foo(out param : Pointer);

procedure Foo(param : PChar);
procedure Foo(const param : PChar);
procedure Foo(var param : PChar);
procedure Foo(out param : PChar);

procedure Foo(param : PInteger);
procedure Foo(const param : PInteger);
procedure Foo(var param : PInteger);
procedure Foo(out param : PInteger);

procedure Foo(param : MyRecord);
procedure Foo(const param : MyRecord);
procedure Foo(var param : MyRecord);
procedure Foo(out param : MyRecord);

// untyped parameters
procedure Foo(const param);
procedure Foo(var param);
procedure Foo(out param);

```

->

```

public static void Foo(int param);
public static void Foo(int param);
public static void Foo(ref int param);
public static void Foo(ref int param);

public static void Foo(Pointer param);
public static void Foo(Pointer param);
public static void Foo(ref Pointer param);
public static void Foo(ref Pointer param);

public static void Foo(string param);
public static void Foo(string param);
public static void Foo(ref string param);
public static void Foo(ref string param);

public static void Foo(PChar param);
public static void Foo(PChar param);
public static void Foo(ref PChar param);
public static void Foo(ref PChar param);

public static void Foo(Pointer<int> param);
public static void Foo(Pointer<int> param);

```

```

public static void Foo(ref Pointer<int> param);
public static void Foo(ref Pointer<int> param);

public static void Foo(MyRecord param);
public static void Foo(MyRecord param);
public static void Foo(ref MyRecord param);
public static void Foo(ref MyRecord param);

// untyped parameters
public static void Foo(Pointer param);
public static void Foo(ref Pointer param);
public static void Foo(ref Pointer param);

```

*Pointer*, *PChar* and *Pointer<T>* are classes to simulate pointers.

### 7.7.4 Temporary variables

The function *StrUpper* expects an *PWideChar* parameter. In Delphi an array of char can be passed too. In C# a temporary *PChar* variable is created from the array. After the call the result is converted back to the character array.

```

var
  S: array[0..20] of char = 'A fUnNy StRiNg';
begin
  upper := StrUpper(S);

```

->

```

char[] S = "A fUnNy StRiNg".ToCharArray();
PChar tmp0 = new PChar(S);
upper = StrUpper(new PChar(tmp0));
S = tmp0.ToCharArray();

```

### 7.7.5 Calls of inherited procedures and functions

For each class, which inherits from another a typedef is inserted into the C# code, like

```

class foo: public bar {
  typedef bar inherited;

```

So, if in Object Pascal "inherited" is followed by a method identifier, it can be translated easily to C#.

```

inherited.foo -> inherited::foo()

```

When "inherited" has no identifier after it, it refers to the inherited method with the same name as the enclosing method. In this case, inherited can appear with or without parameters; if no parameters are specified, it passes to the inherited method the same parameters with which the enclosing method was called. For example,

```

procedure foo.bar(b : BOOLEAN);
begin
  inherited;
end;

```

```
->
void __fastcall foo::bar ( bool b )
{
    inherited::bar( b );
}
```

## 7.7.6 Nested routines

In Delphi functions can be nested. Fortunately C# local functions are quite similar.

```
type
TNested = class
public
    iClassVar : Integer;
    function Test(iOuterParam, iTwiceParam : Integer; s : String): Integer;
end;

implementation

function TNested.Test(iOuterParam, iTwiceParam : Integer; s : String): Integer;
const
    cSeparate = ':';
var
    iFunctionVar : Integer;

    procedure NestedTest(iInnerParam, iTwiceParam : Integer);
    begin
        result := iClassVar + iOuterParam + iFunctionVar + iInnerParam + iTwiceParam;
    end;

begin
    iClassVar := 1;
    iFunctionVar := 2;
    NestedTest(3, 4);
    result := result + iTwiceParam;
end;
```

->

```
public class TNested : TObject
{
    public int iClassVar;
    public int test(int iOuterParam, int iTwiceParam, string s)
    {
        int result = 0;
        const char cSeparate = ':';
        int iFunctionVar = 0;
        void NestedTest(int iInnerParam, int iTwiceParam_1)
        {
            result = iClassVar + iOuterParam + iFunctionVar + iInnerParam + iTwiceParam_1;
        };
        iClassVar = 1;
        iFunctionVar = 2;
        NestedTest(3, 4);
        result = result + iTwiceParam;
        return result;
    }

    public TNested() {}
};
```

There are some restrictions in C# however. In the example the outer and the inner routine both have a parameter called "iTwiceParam". In C# this isn't allowed. the parameter for the inner routine is renamed to "iTwiceParam\_1" therefore. The "\_1" stands for level one.

## 7.8 Special RTL/VCL-functions

Some functions of the *Delphi RTL/VCL* either don't exist in the *C#Builder* counterpart or have become to member functions of the *String* classes. The conversion of calls of the latter kind of functions into calls of the according member functions is done automatically by *Delphi2C#*. For Delphi I/O routines there is a ready translated C# file. In addition the calls of some compile time functions and some other special functions is done automatically. See the following examples:

```

var
  i, j : Integer;
  p1 : Pointer;
  s1, s2 : String;
  iset : set Of int;
  obj : TObject;
  e :TEnum;

                                                                    / std::string
begin
Assigned( obj );           -> ( obj != NULL );
Copy(s1, i, j);           -> s1.SubString( i, j ); / s1.substr( i - 1, j );
Dec(i);                   -> i--;
Dec(i, j);                 -> i -= j;
Dec(e1);                   -> e1--;
Delete(s1, i, j);         -> s1.Delete( i, j ); / s1.erase( i - 1, j );
Dispose(p1);              -> delete p1;
Exclude(iset, i);         -> iset >> i;
FreeAndNil(p1);          -> delete p1; p1 = NULL;
High(TEnum);             -> /*# High(TEnum) */ 2;
High(strarray);         -> strarray.High;
High(type);              -> High<type>(); // defined in d2c_system.pas
Inc(i);                   -> i++;
Inc(i, j);                -> i += j;
Inc(e1);                  -> e1++;
Include(iset, i);        -> iset << i;
Insert(s1, s2, i);       -> s2.Insert( s1, i ); / s2.insert( i - 1, s1 );
Length(s1);              -> s1.Length( ); / s1.length( );
Length(strarray);       -> strarray.Length;
Low(TEnum);              -> /*# Low(TEnum) */ 0;
Low(strarray);          -> strarray.Low;
Low(type);               -> Low<type>(); // defined in d2c_system.pas
New(obj);                -> obj = new obj;
PAnsiChar(s1);          -> s1.c_str();
Pos(s1, s2);             -> s2.Pos( s1 ); / no longer from 1.4.9 on: s2.find( s1 ); (at least
SetLength(s1, i);        -> s1.SetLength( i ); / s1.resize( i );
Str(d:8:2, S);           -> Str( d, 8, 2, S );

RegisterComponents(s1, [a,b,c]); ->

TComponentClass classes[ 4 ] = { __classid( a ), __classid( b ), __classid( c ) };
RegisterComponents( s1 , classes, 3 );

```

You can switch off the special treatment of this functions..

### 7.8.1 I/O routines

Delphi has text and file I/O library routines, which are quite different from C# I/O routines. So they cannot be substituted automatically by according routines of the C# standard library. A direct counterpart of the Delphi in C# was made instead by translation and adaptation of the according parts of the *free pascal FCL*. It is contained in the files *d2c\_sysfile.h* and *d2c\_sysfile.cpp* in the source folder of the Delphi2C# installation. The *GNU Lesser General Public License* which apply to the FCL also applies to these files. The translation was made for *Windows* with the *0x86* processor. The best



matching declarations are contained in *d2c\_system.pas*.

With *d2c\_file.h* and *d2c\_sysfile.cpp* the behavior of the Delphi I/O routines is reproduced in C# quite exactly. For example:

```
var
  t : TextFile;

begin
  AssignFile(t, 'Test.txt');
  Rewrite(t);
```

becomes:

```
TTextRec t;
AssignFile( t, "Test.txt" );
Rewrite( t );
```

There are differences however in the cases, that *Read(Ln)/Write(Ln)* routines are called with several arguments and that formatting parameters are appended in the *Write(Ln)* routines.

The *BlockRead* and *BlockWrite* routines **only work with plain old data types** (POD types), which don't contain pointers to data. In C#, types may not be POD types any longer, which in Delphi are such types. E.g. structures containing Strings will not be POD types in C# any longer.

## 7.8.2 Read(Ln)/Write(Ln) routines

The *Read(Ln)/Write(Ln)* routines can be called in *Delphi* with an arbitrary number of arguments. *Delphi2C#* divides them into a series of function calls:

```
WriteLn('Hello ', name, '!');
```

becomes:

```
WriteLn( "Hello " ); WriteLn( name ); WriteLn( '!' );
```

## 7.8.3 Formatting parameters

The *Write(Ln)* and the *Str* routines can be called with Width and Decimals formatting parameters in Delphi, by use of a special syntactical extension:

```
Write(t, d:8:2);
Str(d:8:2, S);
```

In the translated code, the Width and Decimals become normal comma separated parameters.

```
Write( t, d, 8, 2 );
Str( d, 8, 2, S );
```

This is possible also for the *Write(Ln)* procedure, which accepts further output parameters too, because such calls are divided into a series calls by *Delphi2C#*.

## 7.9 Properties

Delphi allows to access class fields or arrays via properties. Each class may have one default array-property which can be accessed in a simplified notation.

### 7.9.1 Field properties

There are properties in C# similar to properties in Delphi. The Delphi read and write access via properties become to get and set property accessors in C#. The following example is taken from the Embarcadero documentation:

```

type
  THeading = 0..359;
  TCompass = class(TControl)
  private
    FHeading: THeading;
    procedure SetHeading(Value: THeading);
  published
    Property Heading: THeading read FHeading write SetHeading;
    // ...
  end;
->

public class TCompass : TControl
{
  private int /*0..359*/ FHeading;
  /**# private void SetHeading(int /*0..359*/ Value);
  /*property Heading : THeading read FHeading write SetHeading;*/
  public int /* 0.. 359*/ Heading
  {
    get
    {
      return FHeading;
    }
    set
    {
      SetHeading(value);
    }
  }
  // ...
};

```

C# also has e pendant to Delphi's default

.

### 7.9.2 Indexed properties

Values which are specified by an index can be set or get by an indexed property. The index either can be a constant as in the example below or a variable as in the example following afterwards:

```

TRectangle = class
private
  fCoords: array[0..3] of LongInt;
  function GetCoord(Index: Integer): LongInt;
  procedure SetCoord(Index: Integer; Value: LongInt);
public
  Property Left : LongInt Index 0 read GetCoord write SetCoord;
  Property Top : LongInt Index 1 read GetCoord write SetCoord;

  Property Right : LongInt Index 2 read GetCoord write SetCoord;
  Property Bottom : LongInt Index 3 read GetCoord write SetCoord;
end;

```

```
->

public class TRectangle : TObject
{
    private int[] fCoords = new int[4/*# range 0.. 3*/];
    /*# private int GetCoord(int Index);
    /*# private void SetCoord(int Index, int Value);
    /*property Left : int read GetCoord write SetCoord;*/
    public int Left
    {
        get
        {
            return GetCoord(0);
        }
        set
        {
            SetCoord(0, value);
        }
    }
    /*property Top : int read GetCoord write SetCoord;*/
    public int Top
    {
        get
        {
            return GetCoord(1);
        }
        set
        {
            SetCoord(1, value);
        }
    }
    /*property Right : int read GetCoord write SetCoord;*/
    public int Right
    {
        get
        {
            return GetCoord(2);
        }
        set
        {
            SetCoord(2, value);
        }
    }
    /*property Bottom : int read GetCoord write SetCoord;*/
    public int Bottom
    {
        get
        {
            return GetCoord(3);
        }
        set
        {
            SetCoord(3, value);
        }
    }
    public TRectangle() {}
};
```

The get and set accessors are looking similar as those of simple values, but there is an additional constant parameter in the called getter and setter methods.

If the index value isn't constant one might think, that the C# indexer notation could be a good candidate as counterpart in C#. However there can be only one indexer in a C# class, but there can be several indexed properties in Delphi. Therefore *Delphi2C#* reserves the indexer syntax for the Delphi default array-property. For other indexed properties with variable index *Delphi2C#* creates two public methods which redirect the parameters to the private getter and setter methods. The name of these additional methods are constructed from the name of the C# property with a prefix, which can be set in the project options. The default suffices are "Readproperty" and "Writeproperty":

```

TRectangle = class
private
    fCoords: array[0..3] of LongInt;
    function GetCoord(Index: Integer): LongInt;
    procedure SetCoord(Index: Integer; Value: LongInt);
public
    Property Coords[Index: Integer] : LongInt read GetCoord write SetCoord;
end;

->

public class TRectangle : TObject
{
    private int[] fCoords = new int[4/*# range 0.. 3*/];
    /*# private int GetCoord(int Index);
    /*# private void SetCoord(int Index, int Value);
    /*property Coords [Index: integer]: int read GetCoord write SetCoord;*/
    public int ReadPropertyCoords(int Index) { return GetCoord(Index);}
    public void WritePropertyCoords(int Index, int Value){SetCoord(Index, Value);}

    public TRectangle() {}
};

```

### 7.9.3 Default array-property

If a class has a default property, you can access that property in Delphi with the abbreviation object [index], which is equivalent to object.property[index]. C# has an analogue indexer notation which Delphi2C# uses to translate default array-properties

```

type
    // Class with Indexed properties
    TRectangle = class
    private
        fCoords: array[0..3] of Longint;
        function GetCoord(Index: Integer): Longint;
        procedure SetCoord(Index: Integer; Value: Longint);
    public
        property Coords[Index: Integer] : Longint
            read GetCoord write SetCoord; Default;
    end;

->

private class TRectangle : TObject
{
    private int[] fCoords = new int[4/*# range 0.. 3*/];

    /*# private int GetCoord(int Index);

    /*# private void SetCoord(int Index, int Value);
    /*property Coords [Index: integer]: int read GetCoord write SetCoord default ;*/
    public int ReadPropertyCoords(int Index) { return GetCoord(Index);}
    public void WritePropertyCoords(int Index, int Value){SetCoord(Index, Value);}
    public int this[int Index]
    {
        get
        {
            return GetCoord(Index);
        }
        set
        {
            SetCoord(Index, value);
        }
    }

    public TRectangle() {}
};

```

In addition Delphi2C# creates the same *ReadProperty* and *WriteProperty* methods as for non default indexed properties. The existence of these additional methods makes it easier translate calls to indexed properties in a general way.

## 7.10 Statements

The translation of most statements is straightforward. There are some specials with:

- for loop's
- finally-statements
- with-statements
- Initialization/Finalization

### 7.10.1 for loop's

In Delphi there are for-loops where a variable is incremented or decremented to or down to a special value and there are for-in loops. For the first kind of loops the for-loop parameters are evaluated only once, before the loop runs. This complicates a correct translation to C++ a little bit. The number of loops in the following example is determined by the variable *n*:

```
procedure test;
var
  I, n : Integer
begin
  n := 10;
  for I:=1 to n do
  begin
    DoSomething;
    n := 11;
  end;
end;
```

A straightforward translation of this code would be;

```
int I = 0, n = 0;
n = 10;
for ( I = 1; I <= n; I++)
{
  DoSomething();
  n = 11;
}
```

However, in C# an additional loop would be executed, because *n* is changed in the loop and the number of loops is recalculated with this new value. Therefore a correct translation has to remember the original loop count like in the following code:

```
int I = 0, n = 0;
n = 10;
for ( int stop = n, I = 1; I <= stop; I++)
{
  DoSomething();
  n = 11;
}
```

Delphi2C# can produce both code variants, depending on the option to *Use "stop" variable in for-loop* or not..

### 7.10.1.1 for-in loop

for-in loops are a special kind of Delphi for-loops which have the syntax:

```
var
  a : typename;
begin
  for a in B do
    DoSomething(a);
```

where 'a' may be a character in a string 'B' or 'a' may be an element of an array 'B' or 'a' may be a member of a set 'B'. All these cases are translated to a C# for\_each statements:

```
typename a;
foreach (typename element_0 : B)
{
  a = element_0;
  DoSomething(a);
}
```

### 7.10.2 with-statements

In C# there are no with-statements. Therefore Delphi2C# inserts a temporary helping variable of the with-type. This type is easily obtained by use of the C#11 *auto* keyword:

```
type TDate = record          ->      struct TDate {
  Day: Integer;              int Day;
  Month: Integer;            int Month;
  Year: Integer;             int Year;
end;                          };

procedure test(OrderDate: TDate);    void Test(TDate OrderDate)
begin                                  {
  with OrderDate do                 /*# with OrderDate do */
  begin                               {
    if Month = 12 then               {
      begin                           auto& with0 = OrderDate;
        Month := 1;                   if(with0.Month == 12)
        Year := Year + 1;              {
      end                               {
        else                             with0.Month = 1;
        Month := Month + 1;           with0.Year = with0.Year + 1;
      end;                             }
    }                                   else
  }                                     with0.Month = with0.Month + 1;
}                                       }
}
```

### 7.10.3 finally

The finally keyword after a try block opens a block of code, which is executed regardless of what happened in the try block. Here some cleanup can be done and acquired resources can be freed. Fortunately C# has this keyword too.

## 7.10.4 Initialization/Finalization

There isn't any direct counterpart for the sections "initialization" and "finalization" of a Unit in C#. These sections are therefore translated as two functions which contain the respective instructions. In addition, a global variable of a class is defined. In the constructor of this class the initialization routine is called and in destructor the routine for the finalization is called.

```
initialization
pTest := CTest.Create;

finalization
pTest.Free();
```

->

```
void Tests_initialization()
{
    pTest = new CTest;
}

void Tests_finalization()
{
    delete pTest;
}

class Tests_unit
{
public:
    Tests_unit(){ Tests_initialization(); }
    ~Tests_unit(){ Tests_finalization(); }
};
Tests_unit _Tests_unit;
```

## 7.11 class-reference type

In Delphi methods of a class can be called without creating an instance of the class at first. That's similar to C++ static methods. But in C++ it is not possible to assign classes as values to variables and then to create instances of the class by calling a virtual constructor function from such a class reference. This is possible in Delphi however, as shown in the following example code:

```
type
    TBase = class
    end;

    TBaseClass = class of TBase;

    TDerived = class(TBase)
    end;

    TDerivedClass = class of TDerived;
```

```
function make(Base: TBaseClass): TBase;
begin
    result := Base.Create; // will create TBase or TDerived in dependence of the passed parameter
end;
```

The variables *TBaseClass* and *TDerivedClass* are called "class references" of *TBase* or *TDerived* respectively.

There is no direct counterpart to class references in C#, but if the option to create meta-classes is enabled, Delphi2C# creates a framework to simulate class references. Parallel to the existing classes, a second hierarchy of class references "ClassRef<T>" is created then and additional methods are written into the original classes, which allow to simulate the most important basic class functions as for example the *Create*-method. For exceptions there is another but similar hierarchy of *ExceptionRef<T>* classes.

### 7.11.1 ClassRef

In *System.cs* of the accompanying code to Delphi2C# there is a class *TMetaClass* defined. *TMetaClass* is the class reference type for *TObject* and it is the base class of all class reference types of all other classes. These class references are defined as instances of a class *ClassRef*, which is a generic class:

```
public sealed class ClassRef<Class> : TMetaClass where Class : TObject, new()
```

where the template parameter denotes the original class. That way for a hierarchy of classes, which are derived one from another, there is a parallel hierarchy of class references. The class references are implemented as singletons and only created, if needed. The exact definition of the *ClassRef* class is tricky and works only, because Delphi2C# also inserts some additional helper code into every class declaration. The following code demonstrates how a small class factory using class references is converted from Delphi to C++:

```
type
    TBase = class
    public
        function GetName: String; virtual;
    end;

    TBaseClass = class of TBase;

    TDerived = class(TBase)
    public
        function GetName: String; override;
    end;

    TDerivedClass = class of TDerived;

implementation

function make(Base: TBaseClass): TBase;
begin
    result := Base.Create;
end;

function testTactory: boolean;
```



```
var
  s : String;
  p : TBase;
begin
  p := make(TDerived1);
  result := p.GetName = 'TDerived';
end;
```

->

```
public class TBase : TObject
{
    //...

    public TBase() {}
    public override string ClassName() {return "TBase";}
    public override TMetaClass ClassType(){return class_id<TBase>();}
    public override TMetaClass ClassParent(){return class_id<TObject>();}
    public override TObject Create(){return new TBase();}
    public static new TBase SCreate() {return new TBase();}
};

// ClassRef<TBase> TBaseClass;

public class TDerived : TBase
{
    //...

    public TDerived() {}
    public override string ClassName() {return "TDerived";}
    public override TMetaClass ClassType(){return class_id<TDerived>();}
    public override TMetaClass ClassParent(){return class_id<TBase>();}
    public override TObject Create(){return new TDerived();}
    public static new TDerived SCreate() {return new TDerived();}
};

// ClassRef<TDerived> TDerivedClass;

public class TestInterface
{

} // class TestInterface

public class TestImplementation
{

public static TBase Make(TMetaClass Base)
{
    TBase result = null;
    result = (TBase) Base.Create();
    return result;
}
```

```

}

public static bool testTactory()
{
    bool result = false;
    string S = string.Empty;
    TBase P = null;
    P = Make(TDerived1->ClassType());
    result = P.GetName == "TDerived";
    return result;
}

```

The central point in this code is the call of the *class\_id*-function:

```
P = make(class_id<TDerived>());
```

The *class\_id*-function delivers class references. In the example the *class\_id*-function delivers the class reference to the class TDerived.

If TDerived wouldn't have a standard constructor, instead of the line

```
static TDerived* Create() {return new TDerived();}
```

the line

```
static TDerived* Create() {ThrowNoDefaultConstructorError(ClassName()); return nullptr}
```

would have been written. If TDerived were an abstract class, the line would have been:

```
static TDerived* Create() {ThrowAbstractError(ClassName()); return nullptr}
```

Other uses of Delphi class references are reproduced in C++ too. For example:

```

ClassRef := Sender.ClassType;

while ClassRef <> NIL do
begin
    s := ClassRef.ClassName();
    ClassRef := ClassRef.ClassParent;
end;

```

is converted to:

```

TClass ClassRef = Sender->ClassType();

while(ClassRef != nullptr)
{
    s = ClassRef->ClassName();
    ClassRef = ClassRef->ClassParent();
}

```

However only a minimal frame for class reference manipulations is created and there have to be standard constructors for all classes with used class references.

## 7.11.2 ExceptionRef

There is a special problem with exceptions: while *Delphi* exceptions as all other classes are derived from *TObject*, this isn't possible in C#. In C# only classes derived from *System.Exception* can be thrown and caught. Therefore *ClassRef*'s, which are made to create *TObject* types cannot be used for exceptions. *Delphi2C#* uses an analogously *ExceptionRef<T>* instead. The definition of this class is inserted on top of the manually prepared *System.External.ExcUtils.cs*. Like *ClassRef<T>* *ExceptionRef<T>* is derived from *TMetaClass*, but instead of the function

```
public override TObject Create()
```

*ExceptionRef<T>* has the function:

```
public override System.Exception Create(string s)
```

This function uses the *CreateInstance* function from *System.Reflection* to create exceptions. The base function:

```
public virtual Exception Create(string s)
```

has been added to *TMetaClass* for this reason.

By use of the *ExceptionRef<T>* hierarchy for example the following line of code in *System.Sysutils*:

```
E := ExceptTypes[ExceptMap[ErrorCode].EClass].Create(ExceptMap[ErrorCode].EIdent);
```

is translated to C# quite natural to:

```
E = (System.SysUtils.Exception) ExceptTypes[(int) ExceptMap[ErrorCode] - 3].EClass].Create(ExceptMap[ErrorCode] - 3].EIdent);
```

## 7.12 Reading and Writing

*Delphi* has *Stream* classes to read and write files similar to those in C#. But there are also an classic, non-object oriented Pascal routines for this purpose. For this classic approach there are three file types, which have no counterpart in C#

1. File; declares an untyped file to read or write binary data
2. Text or TextFile; declares a text file to read or write ASCII data
3. File of [type]; declares a typed file to read and write sequences of that type (records).

*Delphi2C#* tries to convert this classic approach to the C# object-oriented approach. For all three file types in C# a *Stream* is created at first. In a second step in dependence of the different *Delphi* file types different kinds of *Streams* are created from the first basic stream. For example a *StreamWriter* will be created, if ASCII data shall be written to a Text file:

```
var
  myFile : Text;

begin
  AssignFile(myFile, 'Test.txt');
  Rewrite(myFile);

->

  Stream myFile_stream;
```

```

myFile_stream = new FileStream("Test.txt", FileMode.Create);
using (StreamWriter myFile = new StreamWriter(myFile_stream)) {
    ...
}

```

In Delphi the access mode of these file types are specified by the FileMode variable. In C# there also exists a FileMode, but this variable specifies how to open a file, eg. create a new file or append to an existing file. Therefore the Delphi FileMode variable cannot be defined in the C# code. In translated code the Delphi FileMode is passed as explicit parameter to the file opening commands, "FileMode.Create" in the example above. "FileMode.CreateOrOpen" is the FileMode for readers. The kind of opening a file in Delphi is determined by the command which is used to open the file. The ReWrite command and the Reset command are truncating an existing file or creating a new file, whereas the Append command opens a file to add output to the existing content of a file.

The default FileMode in Delphi is fmOpenReadWrite (=2). but in C# a stream cannot have read and write access at the same time. Delphi2C# therefore looks at the subsequent use of the stream. If writing operations are following a Writer is created else a Reader is created. If the Delphi code really reads and writes to the file without resetting it, the translation will fail.

If a Write function is called with a file as first parameter, the output will be written into that file. Otherwise the Output is written to the console:

```

WriteLn(myFile, 'Hello');
WriteLn('Hello');

```

->

```

myFile.WriteLine("Hello");
Console.WriteLine("Hello");

```

The same for Read-functions:

```

Read(myFile, Letter);
Read(Letter);

```

->

```

Letter = (char) myFile.Read();
Letter = (char) Console.Read();

```

For Files and Files of a type Delphi2C# creates BinaryReader and BinaryWriter. Delphi2C# only converts uses of File of [builtin type] An automatic treatment of files of records might partly be possible, but hasn't be done so far.

```

var
  myWord : WORD;
  myFile : File of WORD;

begin
  AssignFile(myFile, 'Test.cus');
  Rewrite(myFile);

  While not Eof(myFile) do
  begin
    Read(myFile, myWord);
  end;

  CloseFile(myFile);

```

->

```
ushort myWord = 0;
Stream myFile_stream;

myFile_stream = new FileStream("Test.cus", FileMode.OpenOrCreate);
using (BinaryReader myFile = new BinaryReader(myFile_stream, System.Text.Encoding.ASCII)) {

while(!(myFile.PeekChar() < 0))
{
    myWord = (ushort) myFile.ReadInt16();
}
myFile.Close();
} // using
```

Delphi2C# also converts Rename commands.  
The width and precision arguments in write operations aren't converted correctly yet.

## 7.13 Method pointers

*Delphi's* event handling is implemented by means of method pointers. Such method pointers are declared by addition of the words "of object" to a procedural type name. E.g.

```
TNotifyEvent = procedure(Sender: TObject) of object;
```

Delphi2C# converts them into delegates:

```
public delegate void TNotifyEvent(TObject Sender);
```

**At the current state Delphi2C# doesn't deal with event handling.**

## 8 New features since Delphi 7

The Delphi language has been extended since Delphi 7 by following items:

- Unicode
- Unit scope names (Dotted filenames)
- Operator overloading
- Class helpers
- Class-like records
- Nested classes
- Anonymous methods
- Generics

## 8.1 Unicode

Delphi2C# is able to process Delphi files which uses non ANSI characters for identifiers or in comments. For example:

```

unit Unicode;

interface

(* Delphi2C# 的 Unicode *)

type

  TRecord = record
    S1: string;
    S2 : string;
  end;

implementation

  // 的 的 的 的 (xìngzhì)
  procedure T (A: TRecord);
  begin
    WriteLn(A.S1);
    WriteLn(A.S2);
  end;

end.

```

become to:

```

using static Unicode.UnicodeClass;
using System;
using static System.SystemClass;

namespace Unicode
{
  public struct TRecord
  {
    public string S1;
    public string S2;
    public static TRecord CreateRecord(){return new TRecord();}
  };
  public class UnicodeClass
  {
    /* Delphi2C# 的 Unicode */

    // 的 的 的 的 (xìngzhì)

    public static void T (TRecord A)
    {
      WriteLn(A.S1);
      WriteLn(A.S2);
    }

  } // UnicodeClass

} // namespace Unicode

```

## 8.2 Unit scope names

Delphi2C# is able to process names with unit scopes. For example:

```
System.SysUtils
```

does express, that the unit *SysUtils* is part of the unit scope *System*. Delphi2C# as well can open files with such dotted names as well as it can process such names correctly.

## 8.3 Operator Overloading

[http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Operator\\_Overloading\\_\(Delphi\)](http://docwiki.embarcadero.com/RADStudio/Tokyo/en/Operator_Overloading_(Delphi))

The following table maps the signatures of Delphi operators to the signatures of the according operators in C#:

Delphi Declaration Signature	Symbol Mapping	C# Declaration Signature
Implicit(a : type) : resultType;	implicit	public static implicit operator resultType (type a)
Explicit(a: type) : resultType;	explicit	public static explicit operator resultType (type a)
Negative(a: type) : resultType;	-	public static type operator - (type a)
Positive(a: type): resultType;	+	public static type operator + (type a)
Inc(a: type) : resultType;	Inc	public static type operator ++ (type a)
Dec(a: type): resultType;	Dec	public static type operator -- (type a)
LogicalNot(a: type): resultType;	not	public static resultType operator ! (type a)
Trunc(a: type): resultType;	Trunc	public static resultType Trunc(type Value)
Round(a: type): resultType;	Round	public static resultType Round(type Value)
In(a: type; b: type) : Boolean;	in	public static bool IsContained(type a, type b)
Equal(a: type; b: type) : Boolean;	=	public static bool operator == (type a, type b)
NotEqual(a: type; b: type): Boolean;	<>	public static bool operator != (type a, type b)
GreaterThan(a: type; b: type) Boolean;	>	public static bool operator > (type a, type b)
GreaterThanOrEqual(a: type; b: type): Boolean;	>=	public static bool operator >= (type a, type b)
LessThan(a: type; b: type): Boolean;	<	public static bool operator < (type a, type b)

LessThanOrEqual(a: type; b: type): Boolean;	<=	public static bool operator <= (type a, type b)
Add(a: type; b: type): resultType;	+	public static resultType operator + (type a, type b)
Subtract(a: type; b: type): resultType;	-	public static resultType operator - (type a, type b)
Multiply(a: type; b: type): resultType;	*	public static resultType operator * (type a, type b)
Divide(a: type; b: type): resultType;	/	public static resultType operator / (type a, type b)
IntDivide(a: type; b: type): resultType;	div	public static resultType operator / (type a, type b)
Modulus(a: type; b: type): resultType;	mod	public static resultType operator % (type a, type b)
LeftShift(a: type; b: type): resultType;	shl	public static resultType operator << (type a, type b)
RightShift(a: type; b: type): resultType;	shr	public static resultType operator >> (type a, type b)
LogicalAnd(a: type; b: type): resultType;	and	public static resultType operator && (type a, type b)
LogicalOr(a: type; b: type): resultType;	or	public static resultType operator    (type a, type b)
LogicalXor(a: type; b: type): resultType;	xor	// doesn't exist
BitwiseAnd(a: type; b: type): resultType;	and	public static resultType operator & (type a, type b)
BitwiseOr(a: type; b: type): resultType;	or	public static resultType operator   (type a, type b)
BitwiseXor(a: type; b: type): resultType;	xor	// doesn't exist

Examples for:

- binary operators
- unary operators.
- conversion operators
- Finally there are more operators in Delphi like *Trunc* or *In* which aren't operators in C#.

### 8.3.1 binary operators

The translation of overloaded binary operators is straightforward. This is shown in the following example:

```
class operator TMyClass.Add(a, b: TMyClass): TMyClass;
var
  returnrec : TMyrClass;
begin
  returnrec.payload := a.payload + b.payload;
  Result:= returnrec;
end;
```

becomes in C# to:

```
public static TOperatorClass operator + (TOperatorClass a, TOperatorClass b)
{
  TOperatorClass result = TOperatorClass.CreateRecord();
  TOperatorClass returnrec = TOperatorClass.CreateRecord();
  returnrec.payload = a.payload + b.payload;
  result = returnrec;
  return result;
}
```



```
}

```

Problematic are the operator *IntDivide* and *LogicalXor* because they don't have counterparts in C#. Delphi2C# converts *IntDivide* to a normal *Divide* operator `/`. As long as there isn't an additional *Divide* operator this will work. There is no automatic for *LogicalXor* yet.

### 8.3.2 unary operators

Example of a *negative* operator:

```
class operator TMyClass.Negative(a: TMyClass): TMyClass;
var
  b : TMyClass;
begin
  b:= -a.payload;
  Result:= b;
end;
```

Delphi2C# converts this to:

```
public static TMyClass operator - (TMyClass a)
{
  TOperatorClass result = TMyClass.CreateRecord();
  TOperatorClass b = TMyClass.CreateRecord();
  b = -a.payload; // Use the implicit conv here?
  result = b;
  return result;
}
```

### 8.3.3 conversion operators

A class may be converted into another type:

```
class operator TMyClass.Implicit(a: TMyClass): Integer;
var
  myint : integer;
begin
```

```

    myint:= a.payload;
    Result:= myint;
end;

```

This becomes in C# to:

```

public static implicit operator int (TMyClass a)
{
    int result = 0;
    int myint = 0;
    myint = a.payload;
    result = myint;
    return result;
}

```

If the other way round another type is converted to the class::

```

class operator TMyClass.Implicit(a: Integer): TMyClass;
var
    returnrec : TMyClass;
begin
    returnrec.payload:= a;
    Result:= returnrec;
end;

```

this becomes to:

```

public static implicit operator TMyClass (int a)
{
    TMyClass result = TMyClass.CreateRecord();
    TMyClass returnrec = TMyClass.CreateRecord();
    returnrec.payload = a;
    result = returnrec;
    return result;
}

```

For explicit operators:

```

class operator TMyClass.explicit(a: TMyClass): double;
var
    b : double;
begin
    b:= a.payload;
    Result:= b;
end;

```

->

```

public static explicit operator double (TMyClass a)
{
    double result = 0.0D;
    double b = 0.0D;
    b = (double) a.payload;
    result = b;
    return result;
}

```

```

class operator TOperatorClass.explicit(a: TOperatorClass): boolean;

```

```
begin
  Result:= True;
end;
```

->

```
public static explicit operator bool (TOperatorClass a)
{
  bool result = false;
  result = true;
  return result;
}
```

### 8.3.4 more operators

In Delphi there the operators *Round*, *Trunc* and *In*, which have no counterparts in C#. These operators are defines as static member functions in C#.

```
public static long Round(TOperatorClass Value)
{
  long result = 0;
  result = (long) Math.Round(((double) Value), 0); // cast to double prevents from cycle
  return result;
}
```

At positions, where these operators are used, Delphi2C# creates explicit calls to the member function. For example:

```
var
  x: TMyClass;
  d : Double;
begin
  d := Round(x);
```

becomes to:

```
TMyClass X = {0};
double d = 0.0;

d = TMyClass.Round(X);
```

## 8.4 Class helpers

A quite similar feature to Delphi's helper records and helper classes are the extension methods of C#. Therefore these helper are converted to classes containing the according extension methods. Taking the example from:

<http://delphi.about.com/od/oopindelphi/a/understanding-delphi-class-and-record-helpers.htm>

```
TStringsHelper = class Helper for TBase
private
    function GetTheObject(const AString: String): TObject;
    procedure SetTheObject(const AString: String; const Value: TObject);
public
    property ObjectFor[const AString : String]: TObject Read GetTheObject Write SetTheObject;
end;
```

becomes with Delphi2C# to

```
public static class TStringsHelper
{
    public static TObject GetTheObject(this TStrings helped, string aString)
    {
        ...
    }
    public static void SetTheObject(this TStrings helped, string aString, TObject Value)
    {
        ...
    }
}
/*property ObjectFor [aString: string]: TObject read GetTheObject write SetTheObject;*/
public static TObject ReadPropertyObjectFor(this helped, string aString) { return GetTheObject(helped, aString); }
public static void WritePropertyObjectFor(this helped, string aString, TObject Value){SetTheObject(helped, aString, Value); }
};
```

Regardless whether a helper class is defined in the interface part of a Delphi source file or in the implementation part, the generated C# class is put in front of the C# file outside of the class that is constructed for global parts of the unit, because extension methods must be defined in a top level static class.

If fields of the helped type shall be changed by an extension method, the "this" parameter has to be passed by reference.

### Remark

Till the Delphi compiler 10 Seattle it was allowed to access private members of the helped class via its class helper regardless in which unit the helped class was declared. With the just described C# pendant this is not possible. However, this possibility broke OOP encapsulation rules and was regarded as a bug, which was fixed with Delphi compiler 10.1 Berlin. You can read more about this bug fix here:

<http://blog.marcocantu.com/blog/2016-june-closing-class-helpers-loophole.html>

### 8.4.1 this ref

For record helpers the the "this" parameter is passed by reference. This allows to change the helped type itself as in the following example:

```
TRec = record
  X, Y: Double;
end;

TRecHelper = record helper for TRec
public
  procedure Add(Y: Double);
end;

procedure TRecHelper.Add(Y: Double);
begin
  X := X + Y;
end;
```

->

```
public static void Add(this ref TRec helped, double Y)
{
  helped.X = helped.X + Y;
}
```

A call of the Add-Function with a value of 10 will increment the value of the X-field of the record.

```
var R: TRec;
begin
  R.X := 10;
  R.Add(R, 10); // => R.X = 20
```

->

```
TRec R = TRec.CreateRecord();
R.X = 10;
R.Add(R, 10);
```

A this-parameter of an extension method may not be passed by reference however, if the helped type is a class. See:

<https://github.com/dotnet/csharplang/blob/master/proposals/csharp-7.2/readonly-ref.md#refin-extension-methods>

In this case Delphi2C# writes the following warning:

```
/*#helped will not be changed!*/
```

## 8.5 Class-like records

Since Delphi 7 the abilities of records have been expanded to more class-like structures with properties, methods and nested types. Here an example from

[http://docwiki.embarcadero.com/RADStudio/Rio/en/Structured\\_Types\\_\(Delphi\)](http://docwiki.embarcadero.com/RADStudio/Rio/en/Structured_Types_(Delphi))

## #Records\_.28advanced.29

```

type
  TMyRecord = record
    type
      TInnerColorType = Integer;
    var
      Red: Integer;
    class var
      Blue: Integer;
    procedure printRed();
    constructor Create(val: Integer);
    property RedProperty: TInnerColorType read Red write Red;
    class property BlueProp: TInnerColorType read Blue write Blue;
  end;

implementation

constructor TMyRecord.Create(val: Integer);
begin
  Red := val;
end;

procedure TMyRecord.printRed;
begin
  Writeln('Red: ', Red);
end;

```

Delphi2C# converts these new features to:

```

public struct TMyRecord
{
  public int Red;
  public static int Blue;
  public void printRed()
  {
    { Write("Red: "); WriteLn(Red); };
  }
  public TMyRecord(int val)
  {
    Red = val;
  }
  public void Create(int val)
  {
    Red = val;
  }
  /*property RedProperty : TInnerColorType read Red write Red;*/
  public int RedProperty
  {
    get
    {
      return Red;
    }
    set
    {
      Red = value;
    }
  }
  /*property BlueProp : TInnerColorType read Blue write Blue;*/
  public static int BlueProp
  {
    get
    {
      return Blue;
    }
    set
    {
      Blue = value;
    }
  }
  public static TMyRecord CreateRecord(){return new TMyRecord();}
};

```

## 8.6 Nested classes

The possibility to work with nested classes is new since Delphi 7. Here an example from:

[http://docwiki.embarcadero.com/RADStudio/Rio/en/Nested\\_Type\\_Declarations](http://docwiki.embarcadero.com/RADStudio/Rio/en/Nested_Type_Declarations)

```
type
  TOuterClass = class
    strict private
      myField: Integer;

    public
      type
        TInnerClass = class
          public
            myInnerField: Integer;
            procedure innerProc;
          end;

      procedure outerProc;
    end;

implementation

procedure TOuterClass.TInnerClass.innerProc;
begin
  // ...
end;

procedure foo;
var
  x: TOuterClass;
  y: TOuterClass.TInnerClass;
begin
  x := TOuterClass.Create;
  x.outerProc;
  // ...
  y := TOuterClass.TInnerClass.Create;
  y.innerProc;
end;
```

Delphi2C# converts this to:

```
public class TOuterClass : TObject
{
  private int myField;

  public class TInnerClass : TObject
  {
    public int myInnerField;
    public void innerProc()
    {
      // ...
    }

    public TInnerClass() {}
  };
  // # public void outerProc();

  public TOuterClass() {}
};

public class testClass
```

```

{
public static void foo()
{
    TOuterClass x = null;
    TOuterClass.TInnerClass y = null;
    x = new TOuterClass();
    x.outerProc();
    //...
    y = new TOuterClass.TInnerClass();
    y.innerProc();
}
} // testClass

```

## 8.7 Anonymous Methods

The corresponding C# feature to Delphi's anonymous methods are delegates. The translation is quite straight forward:

The following examples are taken from

[http://docs.embarcadero.com/products/rad\\_studio/delphiAndcpp2009/HelpUpdate2/EN/html/devcommon/anonymousmethods\\_xml.html](http://docs.embarcadero.com/products/rad_studio/delphiAndcpp2009/HelpUpdate2/EN/html/devcommon/anonymousmethods_xml.html)

- Assignment to a method reference
- Assignment to a method
- Using anonymous methods
- Variable binding
- Use as events

### 8.7.1 Assignment to a method reference

An anonymous method type can be declared as a reference to a method. It becomes in C# to a delegate:

```

type
    TFuncOfInt = reference to function(x: Integer): Integer;

var
    adder: TFuncOfInt;
begin
    adder := function(X: Integer) : Integer
    begin
        Result := X + Y;
    end;
    WriteLn(adder(22)); // -> 42
->

public delegate int TFuncOfInt(int x);

TFuncOfInt adder = null;
adder = delegate(int x)
{
    int result = 0;
    result = x + Y;
}

```



```
    return result;
};
WriteLn(adder(22)); // -> 42
```

Here the example from Embarcadero is simplified to remove a problem, which is discussed in the context of variable binding.

## 8.7.2 Assignment to a method

As well as anonymous methods can be assigned to a method reference (see above), a normal method can be assigned to it. In C# this is done by means of `std::bind`. The expression of this assignment looks quite complicated however, because `std::placeholders` are needed to represent unbound variables.

```
type
  TMethRef = Reference to procedure(X: Integer);

TAn3Class = class(TObject)
  procedure method(X: Integer);
end;

procedure Test;
var
  m: TMethRef;
  i: TAn3Class;
begin
  // ...
  m := i.method;
end;
```

->

```
typedef std::function<void (int)> TMethRef;

public class TAn3Class : TObject
{
  public bool Method(int x)
  {
    ...
  }

  public TAn3Class() {}
};

void Test()
{
  TMethRef m = null;
  TAn3Class i = null;
  // ... todo: i = new TAn3Class();
  m = i.Method;
}
```

Here remains a problem. the assignment of `"i.Method"` only works, if `i` isn't null.

### 8.7.3 Using anonymous methods

Anonymous methods in Delphi as well as lambda expressions in C# can be returned by functions and passed to functions as parameters. The following example demonstrates the use as a parameter:

```
type
  TFuncOfIntToString = Reference to function(X: Integer): String;

procedure AnalyzeFunction(Proc: TFuncOfIntToString);
begin
  Proc(3);
end;
->

public delegate string TFuncOfIntToString(int x);

public static string AnalyzeFunction(TFuncOfIntToString proc)
{
  string result = string.Empty;
  result = proc(3);
  return result;
}
```

The use as return value is demonstrated in the next example.

### 8.7.4 Variable binding

There is a subtle difference between anonymous methods and lambda expressions: while anonymous methods extend the lifetime of captured references, this is not the case for lambda expressions. In the following Delphi code snippet the anonymous method, which is assigned to the variable *adder*, binds the value 20 to the parameter variable *y*. The lifetime of *y* is extended in Delphi, until *adder* is destroyed.

```
type
  TFuncOfInt = reference to function(x: Integer): Integer;

function MakeAdder(y: Integer): TFuncOfInt;
begin
  Result := function(x: Integer) : Integer
  begin
    Result := x + y;
  end;
end;

procedure TestAnonymous1;
```

```
var
  adder: TFuncOfInt;
begin
  adder := MakeAdder(20);
  Writeln(adder(22));
end;
```

->

```
public delegate int TFuncOfInt(int x);

public static void TestAnonymous1()
{
  TFuncOfInt adder = null;
  adder = MakeAdder(20);
  Writeln(adder(22));
}

public static TFuncOfInt MakeAdder(int Y)
{
  TFuncOfInt result = null;
  result = delegate(int x)
  {
    int result_1 = 0;
    result_1 = x + Y;
    return result_1;
  };
  return result;
}
```

## 8.7.5 Use as events

Method reference types can be used as a kind of event in Delphi and become delegates by translation to C#.

```
type
  TAnProc = Reference to procedure;

  TAn4Component = class(TComponent)
  private
    FMyEvent: TAnProc;
  public
    property MyEvent: TAnProc Read FMyEvent Write FMyEvent;
  end;

procedure TestAnonymous4;
var
  C : TAn4Component;
begin
  C := TAn4Component.Create;
```

```
C.MyEvent := procedure
begin
;
end;
end;
```

->

`using static` anonymous4.anonymous4Class;

```
public class TAn4Component : TObject
{
private TAnProc FMyEvent;
public bool FResult;
// MyEvent property serves as an event:
/*property MyEvent : TAnProc read FMyEvent write FMyEvent;*/
public TAnProc MyEvent
{
get
{
return FMyEvent;
}
set
{
FMyEvent = value;
}
}

public TAn4Component() {}
};

public class anonymous4Class
{
public delegate void TAnProc();
public static void TestAnonymous4()
{
TAn4Component c = null;
c = new TAn4Component();
c.MyEvent = delegate()
{
...
};
};
```

## 8.8 Generics

The following discussion of the translation of Delphi generics to C# templates goes along the Embarcadero documentation

[http://docwiki.embarcadero.com/RADStudio/Tokyo/de/Generics\\_-\\_Index](http://docwiki.embarcadero.com/RADStudio/Tokyo/de/Generics_-_Index)

Declaration  
Nested types

Base types  
Procedural types  
Parameterized methods

Delphi2C# cannot distinguish a generic type and a normal type with the same name in the same unit. There are such cases in System-pas. E.g.

```
IEnumerator = interface(IInterface)
IEnumerator<T> = interface(IEumerator)
```

### 8.8.1 Declaration

```
type
  TPair<TKey,TValue> = class
  private
    FKey: TKey;
    FValue: TValue;
  public
    function GetKey: TKey;
    procedure SetKey(key: TKey);
    function GetValue: TValue;
    procedure SetValue(Value: TValue);
    property key: TKey Read GetKey Write SetKey;
    property Value: TValue Read GetValue Write SetValue;
  end;

implementation

function TPair<TKey,TValue>.GetValue: TValue;
begin
  Result := FValue;
end;
```

->

```
public class TPair<TKey, TValue> : TObject
{
  public TKey FKey;
  public TValue FValue;
  public TValue GetValue()
  {
    TValue result;
    result = FValue;
    return result;
  }
  public TPair() {}
};
```

## 8.8.2 Nested types

A nested type within a generic is itself a generic.

```

type
  TFoo<T> = class
    type
      TBar = class
        X: Integer;
        // ...
      end;
    end;

  // ...
  TBaz = class
    type
      TQux<T> = class
        X: Integer;
        // ...
      end;
    // ...
  end;

var
  n: TFoo<Double>.TBar;

```

->

```

private class TFoo<T> : TObject
{
    public class TBar : TObject
    {
        public int X;
        // ...

        public TBar() {}
    };

    public TFoo() {}
};

// ...

private class TBaz : TObject
{
    public class TQux<T> : TObject
    {
        public int X;
        // ...

        public TQux() {}
    };
    // ...

    public TBaz() {}
};
private static TFoo<double>.TBar N = null;

```

A generic can also be declared within a regular class as a nested type:

```

type
  TOuter = class
    type
      TData<T> = class
        Ffoo1: TFoo<Integer>;           // declared with closed constructed type
        Ffoo2: TFoo<T>;                 // declared with open constructed type

```

```

    FFooBar1: TFoo<Integer>.TBar; // declared with closed constructed type
    FFooBar2: TFoo<T>.TBar;      // declared with open constructed type
    FBazQux1: TBaz.TQux<Integer>; // declared with closed constructed type
    FBazQux2: TBaz.TQux<T>;     // declared with open constructed type
    //...
end;
var
    FIntegerData: TData<Integer>;
    FStringData: TData<String>;
end;

```

-&gt;

```

public class TOuter : TObject
{
    public class TData<T> : TObject
    {
        public TFoo<int> FFool;           // declared with closed constructed type
        public TFoo<T> FFoo2;           // declared with open constructed type
        public TFoo<int>.TBar FFooBar1; // declared with closed constructed type
        public TFoo<T>.TBar FFooBar2;   // declared with open constructed type
        public TBaz.TQux<int> FBazQux1; // declared with closed constructed type
        public TBaz.TQux<T> FBazQux2;   // declared with open constructed type
        //...

        public TData() {}
    };
    public TData<int> FIntegerData;
    public TData<string> FStringData;

    public TOuter() {}
};

```

### 8.8.3 Base types

The base type of a parameterized class or interface type might be an actual type or a constructed type

```

type
    TFoo1<T> = class(TBar)           // Actual type
    end;

    TFoo2<T> = class(TBar2<T>)      // Open constructed type
    end;

    TFoo3<T> = class(TBar3<Integer>) // Closed constructed type
    end;

```

-&gt;

```

// Actual type
public class TFoo1<T> : TBar
{
    public TFoo1() {}
};
// Open constructed type
public class TFoo2<T> : TBar2<T>
{
    public TFoo2() {}
};
// Closed constructed type
public class TFoo3<T> : TBar2<int>
{
    public TFoo3() {}
};

```

Class, interface, record, and array types can be declared with type parameters.

```

type
    TRecord<T> = record
        FData: T;
    end;

type
    IAncestor<T> = interface
        function GetRecord: TRecord<T>;
    end;

    IFoo<T> = interface(IAncestor<T>)
        procedure AMethod(Param: T);
    end;

type
    TFoo<T> = class(TObject, IFoo<T>)
        FField: TRecord<T>;
        procedure AMethod(Param: T);
        function GetRecord: TRecord<T>;
    end;

```

->

```

public struct TRecord<T>
{
    public T FData;
    public static TRecord<T> CreateRecord(){return new TRecord<T>();}
};

public interface IAncestor<T> : IInterface
{
    TRecord<T> GetRecord();
};

public interface IFoo<T> : IAncestor<T>
{
    void AMethod(T Param);
};

public class TFoo<T> : TObject, IFoo<T>
{
    public TRecord<T> FField = TRecord<T>.CreateRecord();
    public void AMethod(T Param){...}
    public TRecord<T> GetRecord(){...}

    public TFoo() {}
};

```



## 8.8.4 Procedural types

The procedure type and the method pointer can be declared with type parameters. Parameter types and result types can also use type parameters.

```
type
  TMyProc<T> = procedure(Param: T);
  TMyProc2<Y> = procedure(Param1, Param2: Y) of object;
type
  TFoo = class
    procedure Test;
    procedure MyProc(X, Y: Integer);
  end;

procedure sample(Param: Integer);
begin
  ...
end;

procedure TFoo.MyProc(X, Y: Integer);
begin
  ...
end;

procedure TFoo.Test;
var
  X: TMyProc<Integer>;
  Y: TMyProc2<Integer>;
begin
  X := sample;
  X(10);
  Y := MyProc;
  Y(20, 30);
end;

procedure Test;
var
  F: TFoo;
begin
  F := TFoo.Create;
  F.Test;
  F.Free;
end;
```

->

```
public delegate void TMyProc<T>(T Param);
public delegate void TMyProc2<Y>(Y Param1, Y Param2);

private class TFoo : TObject
{
  public void Test()
  {
    TMyProc<int> X = null;
    TMyProc2<int> Y = null;
    X = Sample;
    X(10);
    Y = MyProc;
    Y(20, 30);
  }
}
```

```

    }

    public void MyProc(int X, int Y)
    {
        ...
    }

    public TFoo() {}
};

public static void Sample(int Param)
{
    ...
}

public static void Test()
{
    TFoo F = null;
    F = new TFoo();
    F.Test();
    F = null;
}

```

### 8.8.5 Parameterized methods

Methods can be declared with type parameters. Parameter types and result types can use type parameters.

```

type
  TFoo = class
    procedure Test;
    procedure CompareAndPrintResult<T>(X, Y: T);
  end;

procedure TFoo.CompareAndPrintResult<T>(X, Y: T);
begin
end;

procedure TFoo.Test;
begin
  CompareAndPrintResult<String>('Hello', 'World');
  CompareAndPrintResult('Hello', 'Hello');
  CompareAndPrintResult<Integer>(20, 20);
  CompareAndPrintResult(10, 20);
end;

procedure Test;
var
  F: TFoo;
begin
  F := TFoo.Create;
  F.Test;
  ReadLn;
  F.Free;
end;

```

->

```

private class TFoo : TObject
{
  public void Test()
  {
    CompareAndPrintResult<string>("Hello", "World");
    CompareAndPrintResult("Hello", "Hello");
    CompareAndPrintResult<int>(20, 20);
  }
}

```

```
        CompareAndPrintResult(10, 20);
    }

    public void CompareAndPrintResult<T> (T X, T Y)
    {
    }

    public TFoo() {}
};

public static void Test()
{
    TFoo F = null;
    F = new TFoo();
    F.Test();
    ReadLn();
    F = null;
}
```

## 9 What is partially translated

Some features of Delphi can be translated partly only.

Variants in records

Visibility of class members

Virtual class methods

Abstract classes cannot be created, they have to be made non-abstract before

Not all PInvoke calls are working without post-processing of the used signatures.

### 9.1 Pointers

Pointers may be used in C# in an unsafe context only. For the translation of Delphi code using pointers in normal context Delphi2C# provides three pointer classes, which simulate the behavior of pointers. The first is class for void pointers `Pointer`, the second a generic class ***Pointer<T>***, where T is the type at the memory address that the Delphi-pointer denotes. For the special case of character types, the third pointer class is used. As name for this third class ***PChar*** is reused. The C# `PChar` class simulates the behavior of Delphi's `PChar` and of the other Delphi character pointers.

The `Pointer` class is more as a placeholder than a real substitution for the original void-pointer and has to be corrected by hand. The typed pointer classes simply reflect the memory layout of pointers. A pointer to a type T is the address of a variable of this type in the memory. There may be more such variables at adjacent memory positions. In that case the pointer just points to the first element of an array of variables of the type T. The typed pointer classes are working on copies of such arrays into data containers. The location to which the pointer points is reproduced as an integer offset to the first element. The operations that can be done with a pointer are reproduced by member functions of the pointer classes. For example the incrementation of a pointer becomes the incrementation of the index member and dereferencing the pointer becomes a call of the `Deref` member function, which returns the element at the current index. The Delphi2C# translator is responsible for the correct substitution of pointer operations to the according member functions.

All pointer classes have a common interface: ***IPointer***. `Pointer<T>` and `PChar` differ on the used data containers.

IPointer  
 Pointer  
 Pointer<T>  
 PChar

How the translation works can be seen in the following example:

```

var
  pc : PChar;
  s  : string;
begin
  s := 'hello';
  pc := PChar(s);
  Inc(pc);
  pc^ := 'a';

```

->

```

PChar pc = null;
string s = string.Empty;
s = "hello";
pc = new PChar(s);
++pc;
pc.Assign('a');
pc.Synchronize(ref s);

```

The code is translated line by line, but at the end there is an additional line in C#. By the Synchronize method the changed pointer content is copied back to the original string. The result is the same as in Delphi, where the string data are manipulated directly in the memory.

### Limitations of the pointer classes

- While Delphi2C# automatically adds a call of the Synchronize method at the end of the code block, where a character pointer is defined from a string, such synchronization isn't guaranteed for other pointers and all cases. For example the first pointer might be assigned to a second one. Then changes of the content of one of them will not affect the content of the other.
- Sometimes Delphi uses a general-purpose pointer type **Pointer**. For a *Pointer* no type is specified to which it points. Such a Pointer might be simulated too, but in Delphi it is used interoperably with other typed pointers. This feature cannot be reproduced in C#.

## 9.1.1 IPointer

IPointer is the common interface of the two C# classes *Pointer<T>* and *PChar*, which are used for the simulation of Delphi pointers.

```
public interface IPointer<T>
{
    int Length
    {
        get;
    }

    int Position
    {
        get;
    }

    bool IsNull();
    void SetNull();
    T Deref();
    void Assign(T c);
    void Inc(int i);
    void Dec(int i);
};
```

### 9.1.2 Pointer

The *Pointer* class is a quite bad placeholder for the original void-pointer. Code where these Pointers seldom works without post-processing. Sometimes a simple object is a better candidate than *Pointer*, but often *Pointer* can be substituted by typed *Pointe<T>*. However Delphi2C# cannot determine automatically which type is the most adequate type for T.

### 9.1.3 Pointer<T>

The generic class *Pointer<T>* is used to simulate Delphi pointers to a variable of type T or to an array of such variables. However, for the case that this type is a character type, there is a special *PChar* class. Both classes have the common *IPointer* interface.

*Pointer<T>* has a *T[]* member variable and a second variable for the current index:

```
public class Pointer<T>: IPointer<T> where T : new()
{
    protected T[] m_Array = null;
    protected int FPosition = -1;
}
```

## 9.1.4 PChar

The *PChar* class is used to simulate Delphi's *PChar* type or other character pointer types.

*PChar* can store a character array either as a string or as a *StringBuilder* and has a second member variable for the current index.

```
public class PChar : IPointer<char>
{
    private StringBuilder m_StringBuilder = null;
    private string m_String = null;
    private int FPosition = -1;
```

Whether the string or the *StringBuilder* is used depends on the construction and on the use of *PChar*. When *PChar* is constructed from a string, *m\_String* becomes a reference of the original string. That means, there are minimal costs to represent the character array of a Delphi string.

```
public PChar(string S, int Index = 0)
{
    m_String = S;
    FPosition = Index;
}
```

As long as there is read only access to the pointer nothing but the index variable is changed. But as soon as there is a writing access to *PChar*, *m\_String* will be copied into *m\_StringBuilder* and all further operations will be done on it. For example the the following code is executed if a character is assigned to the dereferenced pointer:

```
public void Assign(char c)
{
    if(m_StringBuilder == null && m_String != null)
    {
        m_StringBuilder = new StringBuilder();
        m_StringBuilder.Append(m_String);
    }

    m_StringBuilder[FPosition] = c;
}
```

But the real purpose of the modifications is the modification of the original string. Therefore the changes at *PChar* finally have to be copied back. For this there is the *Synchronize* method.

```
public void Synchronize(ref string s)
```

Delphi2C# automatically adds a call of this method at the end of the code block, where the pointer is defined.

The *StringBuilder* member is used also in cases that *PChar* is not constructed from a string:

```
public PChar(int Size)
public PChar(char c)
public PChar(char[] arr, int Index = 0)
```

## 9.2 inline assembler

Inline assembler code isn't converted. It is put into comments instead, so that the translated code will not stop to compile because of invalid assembler parts. In the **professional version** of *Delphi2Cpp*, there is a minimalistic option to convert Delphi comments and Delphi expressions and to substitute identifiers. The option wasn't taken over here to *Delphi2C#*, because it is of little use and because in the actual Delphi *RTL* the definition of *PUREPASCAL* can be set, to avoid the use of assembler code at all,

## 9.3 const-correctness

Compared with the concept the *const*-correctness in C# the use of *const* in Delphi is very limited. In the Delphi *const*-section true constants are declared whose values cannot change and the keyword *const* also can be used to declare constant parameters. No values can be assigned to constant parameters and they cannot be passed to routines, where *var* parameters are expected. But unlike C#, Delphi does not permit methods to be marked as *const*. The VCL pendant of the C#Builder is not designed for C# *const*-correctness.

If the translated Delphi code simply should compile, it would be the best to ignore the *const*-qualifier totally. But it is the aim of *Delphi2C#*, that the created C# code should be C#-like code and the translation also is orientated at the way the C#Builder produces C#-header files from Delphi sources. C#Builder leaves the *const* qualifiers for parameters. For example:

```
TMyClass = class
private
    FObject : TObject;
public
    constructor Create(const Obj: TObject);
```

The declaration of a constructor is translated by C#Builder and accordingly by *Delphi2C#* to

```
__fastcall TMyClass( const TObject* Obj );
```

But this leads to a problem in the body of the constructor, where the parameter is assigned to a member of the class:

```
__fastcall TMyClass::TMyClass( const TObject* Obj )
: FObject(Obj)
{
}
```

Compiling this code produces the error: E2034 conversion of 'const TObject \*' to 'TObject \*' not

possible. So a cast is necessary, which strips the `const` qualifier away:

```
__fastcall TMyClass::TMyClass( const TObject* Obj )
: FObject( (TObject*)Obj)
{
}
```

or more precisely:

```
__fastcall TMyClass::TMyClass( const TObject* Obj )
: FObject( const_cast<TObject*>(Obj) )
{
}
```

This example suggests to leave out the `const`-qualifier at the translation anyway as mentioned above. You can correct the code in this way, but there are other cases where the `const`-qualifier should be preserved.

For other compilers than C#Builder the methods, which are created for the read-specifiers of properties are made `const`-methods.

## 10 Unit Tests

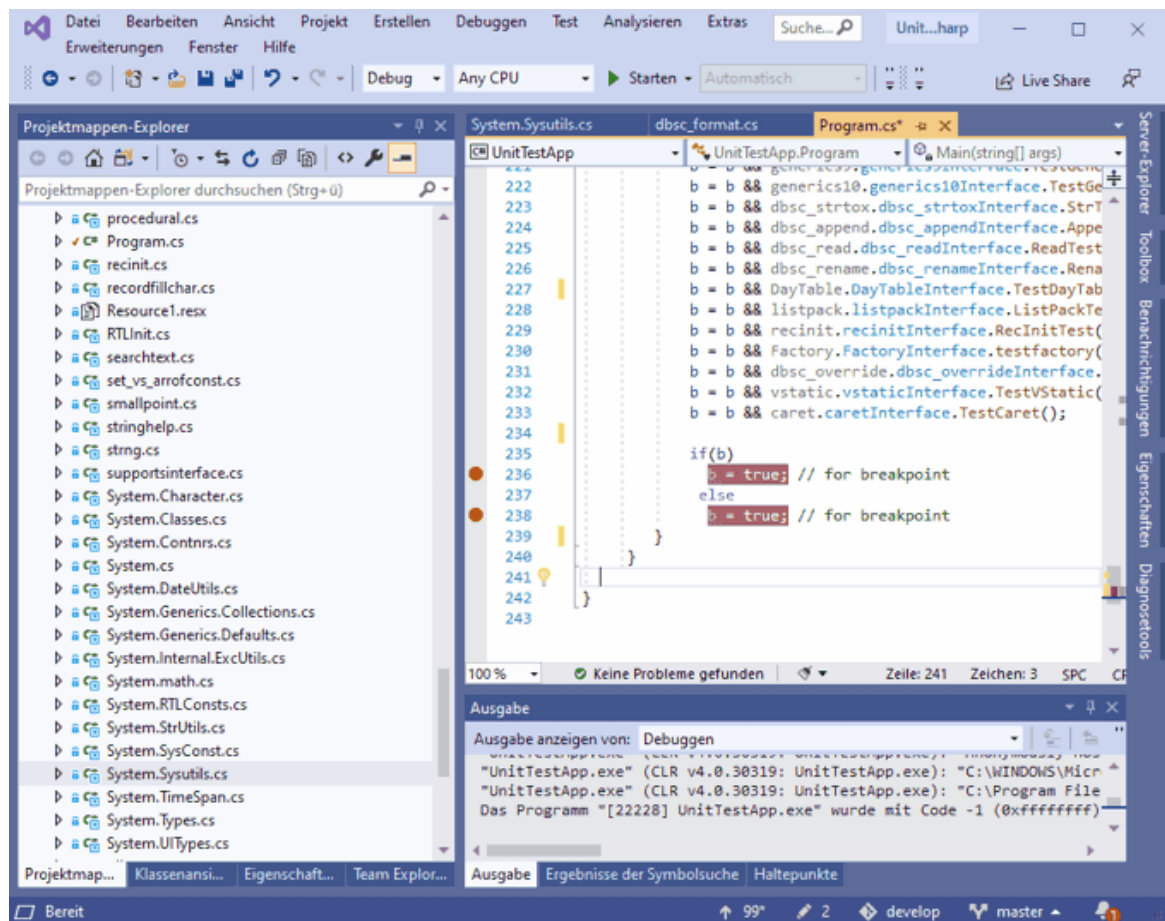
The quality of the translation results of Delphi code to C# with Delphi2C# is guaranteed by a collection of test files. The test cases mostly are modified examples from Embarcadero and from Delphi Basics:

<http://www.delphibasics.co.uk>

The output operations in the examples were replaced by boolean expressions which can be checked at the execution of the tests. The modified files then were inserted into a DUnit application. (DUnit is a testing framework which is integrated into the RAD Studio.)

After verification that the tests are working correctly in Delphi, the code is translated with Delphi2C# to C#. The translated test files then are inserted into a C# test application. There the tests are repeated then in C#.





The examples below are only a small selection of the whole test suite, which comprises more than a hundred of such test files.

Format  
TDictionary  
TStringList

## 10.1 Format

The formatting routines account for a considerable part of the SysUtils unit. Some of them are nested and consist in about 1000 lines of code. Nevertheless the translated code works nearly perfect. Examples to the formatting routines from

<http://www.delphibasics.co.uk/RTL.asp?Name=format>

were modified slightly to be able to use them for test purposes. The code translated with Delphi2C# compiles and works without additional manual processing without faults.

```
using static dbsc_format.dbsc_formatInterface;
using static dbsc_format.dbsc_formatImplementation;
using System;
using static System.SystemInterface;
using System.SysUtils;
using static System.SysUtils.SysUtilsInterface;
```

```

namespace dbsc_format
{

public class dbsc_formatInterface
{
//http://www.delphibasics.co.uk/RTL.asp?Name=Format
//http://www.delphibasics.co.uk/RTL.asp?Name=FloatToStr
//http://www.delphibasics.co.uk/RTL.asp?Name=FormatFloat
public static bool FormatTest()
{
    bool result = false;
    result = true;
    result = result && FormatTest1();
    result = result && FormatTest2();
    result = result && FloatToStrTest1();
    result = result && FormatFloatTest1();
    return result;
}

} // class dbsc_formatInterface

public class dbsc_formatImplementation
{

public static bool FormatTest1()
{
    bool result = false;
    result = true;
    // Just 1 data item
    result = result && (Format("%s", new object[]{"Hello"}) == "Hello");

    // A mix of literal text and a data item
    result = result && (Format("String = %s", new object[]{"Hello"}) == "String = Hello");
    //ShowMessage('');

    // Examples of each of the data types
    result = result && (Format("Decimal          = %d", new object[]{-123}) == "Decimal          = -123")
    result = result && (Format("Exponent         = %e", new object[]{12345.678D}) == "Exponent         = ")
    result = result && (Format("Fixed           = %f", new object[]{12345.678D}) == "Fixed           = ")
    result = result && (Format("General         = %g", new object[]{12345.678D}) == "General         = ")
    result = result && (Format("Number         = %n", new object[]{12345.678D}) == "Number         = ")
    result = result && (Format("Money          = %m", new object[]{12345.678D}) == "Money          = ")
    // makes no sense under C#
    // result := result and (Format('Pointer      = %p', [addr(text)]) = 'Pointer      = 0069FC9
    result = result && (Format("String         = %s", new object[]{"Hello"}) == "String         = Hel
    result = result && (Format("Unsigned decimal = %u", new object[]{123}) == "Unsigned decimal = 123");
    result = result && (Format("Hexadecimal    = %x", new object[]{140}) == "Hexadecimal    = 8C");
    return result;
}

public static bool FormatTest2()
{
    bool result = false;
    result = true;
    // The width value dictates the output size
    // with blank padding to the left
    // Note the <> characters are added to show formatting
    result = result && (Format("Padded decimal   = <%7d>", new object[]{1234}) == "Padded decimal   = <

    // With the '-' operator, the data is left justified
    result = result && (Format("Justified decimal = <%-7d>", new object[]{1234}) == "Justified decimal =

    // The precision value forces 0 padding to the desired size
    result = result && (Format("0 padded decimal = <%.6d>", new object[]{1234}) == "0 padded decimal =

    // A combination of width and precision
    // Note that width value precedes the precision value
    result = result && (Format("Width + precision = <%8.6d>", new object[]{1234}) == "Width + precision =

    // The index value allows the next value in the data array
    // to be changed
    result = result && (Format("Reposition after 3 strings = %s %s %s %1:s %s", new object[]{"Zero", "One

```

```

// One or more of the values may be provided by the
// data array itself. Note that testing has shown that an *
// for the width parameter can yield EConvertError.
result = result && (Format("In line          = <%10.4d>", new object[]{1234}) == "In line
result = result && (Format("Part data driven = <%.4d>", new object[]{10, 1234}) == "Part data drive
result = result && (Format("Data driven     = <%.*d>", new object[]{10, 4, 1234}) == "Data driven
return result;
}

public static bool FloatToStrTest1()
{
    bool result = false;
    double amount1 = 0.0D;
    double amount2 = 0.0D;
    double amount3 = 0.0D;
    result = true;
    amount1 = 1234567890.123456789D; // High precision number
    amount2 = 1234567890123456.123D; // High mantissa digits
    amount3 = 1E100D; // High value number
    result = result && (FloatToStr(amount1) == "1234567890,12346");
    result = result && (FloatToStr(amount2) == "1,23456789012346E15");
    result = result && (FloatToStr(amount3) == "1E100");
    return result;
}

public static bool FormatFloatTest1()
{
    bool result = false;
    double flt = 0.0D;
    result = true;
    // Set up our floating point number
    flt = 1234.567D;

    // Display a sample value using all of the format options

    // Round out the decimal value
    result = result && (FormatFloat("#####", flt) == "1235");
    result = result && (FormatFloat("00000", flt) == "01235");
    result = result && (FormatFloat("0", flt) == "1235");
    result = result && (FormatFloat("#,##0", flt) == "1.235");
    result = result && (FormatFloat(",0", flt) == "1.235");

    // Include the decimal value
    result = result && (FormatFloat("0.####", flt) == "1234,567");
    result = result && (FormatFloat("0.0000", flt) == "1234,5670");

    // Scientific format
    result = result && (FormatFloat("0.0000000E+00", flt) == "1,2345670E+03");
    result = result && (FormatFloat("0.0000000E-00", flt) == "1,2345670E03");
    result = result && (FormatFloat("#.#####E-##", flt) == "1,234567E3");

    // Include freeform text
    result = result && (FormatFloat("\\"Value = \"0.0", flt) == "Value = 1234,6");

    // Different formatting for negative numbers
    result = result && (FormatFloat("0.0", -1234.567D) == "-1234,6");
    result = result && (FormatFloat("0.0 \\\"CR\\\";0.0 \\\"DB\\\"", -1234.567D) == "1234,6 DB");
    result = result && (FormatFloat("0.0 \\\"CR\\\";0.0 \\\"DB\\\"", 1234.567D) == "1234,6 CR");

    // Different format for zero value
    result = result && (FormatFloat("0.0", 0.0D) == "0,0");
    result = result && (FormatFloat("0.0;-0.0;\\\"Nothing\\\"", 0.0D) == "Nothing");
    return result;
}

} // class dbsc_formatImplementation

} // namespace dbsc_format

```

## 10.2 TDictionary

Delphi's class *TDictionary* is defined in the unit `System.Generics.Collections`. It is relatively complex and it uses much parts of the RTL. The correctness of the translation of code in which this class is used is guaranteed by a unit test which is derived from an Embarcadero example.

[>Generics.Collections.TDictionary](http://docwiki.embarcadero.com/CodeExamples/Rio/en/Generics_Collections_TDictionary_(Delphi))

As for all test cases, the output operations have been replaced by boolean expressions which are checked at the execution of the test.

The translation with Delphi2C# doesn't require any further manual post-processing and is shown below.

```
using static docu_tdictionary.docu_tdictionaryInterface;
using static docu_tdictionary.docu_tdictionaryImplementation;
using System;
using static System.SystemInterface;
using System.Types;
using static System.Types.TypesInterface;
using System.SysUtils;
using static System.SysUtils.SysUtilsInterface;
using System.math;
using static System.math.mathInterface;
using System.Generics.Collections;
using static System.Generics.Collections.CollectionsInterface;

namespace docu_tdictionary
{

public class docu_tdictionaryInterface
{

//http://docwiki.embarcadero.com/CodeExamples/Rio/en/Generics_Collections_TDictionary_(Delphi)
public static bool TestDictionary()
{
    bool result = false;
    result = TestDictionary1();
    return result;
}

} // class docu_tdictionaryInterface

public class docu_tdictionaryImplementation
{

public class TCity : TObject
{
    public string country = string.Empty;
    public double Latitude;
    public double Longitude;

    public TCity() {}
    public override string ClassName() {return "TCity";}
    public override TMetaClass ClassType(){return class_id<TCity>();}
    public override TMetaClass ClassParent(){return class_id<TObject>();}
    public override TObject Create(){return new TCity();}
    public static new TCity SCreate() {return new TCity();}
};

public const double Epsilon = 0.0000001D;

public static bool TestDictionary1()
{
    bool result = false;
    TDictionary<string, TCity> Dictionary = null;
    TCity city = null;
    TCity Value = null;
```

```
string Key = string.Empty;
bool bTest = false;
string S = string.Empty;
result = true;
/* Create the dictionary. */
Dictionary = new TDictionary<string, TCity>();
city = new TCity();
/* Add some key-value pairs to the dictionary. */
city.country = "Romania";
city.Latitude = 47.16D;
city.Longitude = 27.58D;
Dictionary.Add("Iasi", city);
city = new TCity();
city.country = "United Kingdom";
city.Latitude = 51.5D;
city.Longitude = -0.17D;
Dictionary.Add("London", city);
city = new TCity();
city.country = "Argentina";
/* Notice the wrong coordinates */
city.Latitude = 0;
city.Longitude = 0;
Dictionary.Add("Buenos Aires", city);

/* Display the current number of key-value entries. */
result = result && (Dictionary.Count == 3);

// Try looking up "Iasi".
if(Dictionary.TryGetValue("Iasi", ref city) == true)
{
    result = result && (city.country == "Romania");
}
else
result = false;

/* Remove the "Iasi" key from dictionary. */
Dictionary.Remove("Iasi");

/* Make sure the dictionary's capacity is set to the number of entries. */
Dictionary.TrimExcess();

/* Test if "Iasi" is a key in the dictionary. */
if(Dictionary.ContainsKey("Iasi"))
    result = false;

/* Test how (United Kingdom, 51.5, -0.17) is a value in the dictionary but
ContainsValue returns False if passed a different instance of TCity with the
same data, as different instances have different references. */
if(Dictionary.ContainsKey("London"))
{
    Dictionary.TryGetValue("London", ref city);
    if((city.country == "United Kingdom") && (CompareValue(city.Latitude, 51.5D, Epsilon) == EqualsValu
        result = result && (city.country == "United Kingdom");
    else
        result = false;
    city = new TCity();
    city.country = "United Kingdom";
    city.Latitude = 51.5D;
    city.Longitude = -0.17D;
    if(Dictionary.ContainsValue(city))
        result = false;
    city = null;
}
else
result = false;

/* Update the coordinates to the correct ones. */
city = new TCity();
city.country = "Argentina";
city.Latitude = -34.6D;
city.Longitude = -58.45D;
Dictionary.AddOrSetValue("Buenos Aires", city);

/* Generate the exception "Duplicates not allowed". */
try
```

```

    {
        bTest = false;
        Dictionary.Add("Buenos Aires", city);
    }
    catch(System.SysUtils.Exception)
    {
        bTest = true;
    }
    result = result && (bTest == true);
    bTest = false;
    /* Display all countries. */
    foreach(TCity element_0 in Dictionary.Values)
    {
        Value = element_0;
        if(Value.country == "Argentina")
            bTest = true;
    }
    result = result && (bTest == true);
    bTest = false;
    /* Iterate through all keys in the dictionary and display their coordinates. */
    foreach(string element_0 in Dictionary.Keys)
    {
        Key = element_0;
        S = FloatToStrF(Dictionary[Key].Longitude, TFloatFormat.ffFixed, 4, 2);
        if(S == "-58,45")
            bTest = true;
    }
    result = result && (bTest == true);

    /* Clear all entries in the dictionary. */
    Dictionary.Clear();

    /* There should be no entries at this point. */
    result = result && (Dictionary.Count == 0);

    /* Free the memory allocated for the dictionary. */
    Dictionary = null;
    city = null;
    return result;
}

} // class docu_tdictionaryImplementation

} // namespace docu_tdictionary

```

### 10.3 TStringList

A frequently used Delphi class is *TStringList*. The translation of the defining code in `System.Classes` needs little manual post-processing. However there are some streaming operations namely in the base class *TPersistent*, which aren't implemented. But the example from

<http://www.delphibasics.co.uk/RTL.asp?Name=tstringlist>

compiles and works without manual post-processing. (Again, the original code has been slightly modified for the testing purpose.)

```

using System.Classes;
using static System.Classes.ClassesInterface;
using static dbcs_tstringlist.dbcs_tstringlistInterface;

```

```
using static dbsc_tstringlist.dbsc_tstringlistImplementation;
using System;
using static System.SystemInterface;

namespace dbsc_tstringlist
{

public class dbsc_tstringlistInterface
{

//http://www.delphibasics.co.uk/RTL.asp?Name=TStringList
public static bool TStringListTest()
{
    bool result = false;
    result = true;
    result = result && TStringListTest1();
    result = result && TStringListTest2();
    result = result && TStringListTest3();
    return result;
}

} // class dbsc_tstringlistInterface

public class dbsc_tstringlistImplementation
{

public static bool TStringListTest1()
{
    bool result = false;
    TStringList animals = null;           // Define our string list variable
    int i = 0;
    result = true;
    // Define a string list object, and point our variable at it
    animals = new TStringList();

    // Now add some names to our list
    animals.Add("Cat");
    animals.Add("Mouse");
    animals.Add("Giraffe");

    // Now display these animals
    // for i := 0 to animals.Count-1 do
    // ShowMessage(animals[i]); // animals[i] equates to animals.Strings[i]
    result = result && (animals[0] == "Cat");
    result = result && (animals[1] == "Mouse");
    result = result && (animals[2] == "Giraffe");

    // Free up the list object
    animals = null;
    return result;
}

public static bool TStringListTest2()
{
    bool result = false;
    TStringList Names = null;           // Define our string list variable
    string ageStr = string.Empty;
    int i = 0;
    result = true;
    // Define a string list object, and point our variable at it
    Names = new TStringList();

    // Now add some names to our list
    Names.CommaText = "Neil=45, Brian=63, Jim=22";

    // And now find Brian's age
    ageStr = Names.ReadPropertyValues("Brian");

    // Display this value
    // ShowMessage('Brians age = '+ageStr);
    result = result && (ageStr == "63");
}

}
```

```

// Now display all name and age pair values
for(i = 0; i <= Names.Count - 1; i++)
{
    //ShowMessage(names.Names[i]+' is '+names.ValueFromIndex[i]);
    if(i == 0)
        result = result && (new PChar(Names.ReadPropertyNames(i)).ToString() == "Neil") && (new PChar(Nam
    if(i == 1)
        result = result && (new PChar(Names.ReadPropertyNames(i)).ToString() == "Brian") && (new PChar(Na
    if(i == 2)
        result = result && (new PChar(Names.ReadPropertyNames(i)).ToString() == "Jim") && (new PChar(Name
}

// Free up the list object
Names = null;
return result;
}

public static bool TStringListTest3()
{
    bool result = false;
    TStringList cars = null;           // Define our string list variable
    int i = 0;
    result = true;
    // Define a string list object, and point our variable at it
    cars = new TStringList();

    // Now add some cars to our list - using the DelimitedText property
    // with overridden control variables
    cars.Delimiter = ' ';           // Each list item will be blank separated
    cars.QuoteChar = '|';          // And each item will be quoted with '|'s
    cars.DelimitedText = "|Honda Jazz| |Ford Mondeo| |Jaguar \"E-type\"|";

    // Now display these cars
    // for i := 0 to cars.Count-1 do
    //     ShowMessage(cars[i]);      // cars[i] equates to cars.Strings[i]
    result = result && (cars[0] == "Honda Jazz");
    result = result && (cars[1] == "Ford Mondeo");
    result = result && (cars[2] == "Jaguar \"E-type\"");

    // Free up the list object
    cars = null;
    return result;
}

} // class dbsc_tstringlistImplementation
} // namespace dbsc_tstringlist

```

## 11 What is not translated

There are some principle problems - listed below - at the conversion of Delphi code to C++ which cannot be resolved by an automatic translator. But even things which Delphi2C#, normally can handle may fail in complex nested cases. Sometimes Delphi2C# generates explicit "todo"-comments where something has to be completed manually.

Some Delphi constructs, which aren't, automatically translated yet are:

- Parts of the RTL operate directly on the virtual method table of objects. These parts aren't reproduced. The most important consequence of this lack is, that streaming of forms and other



types isn't possible in Delphi manner.

- Inline assembler code in Delphi and C# almost are identically. Delphi2C# doesn't translate these parts but only copies them
- virtual class methods.
- *Delphi2C#* always assumes unique names. But e.g. there might be symbols from the operation system, which differ in notation.
- Some problems with constructors remain. E.g. *Delphi2C#* cannot distinguish constructors with equal signatures.
- Class helpers cannot change the fields of the helped class itself, though this is possible for records
- Manual post-processing to achieve const-correctness is necessary.
- The consequences of the ZEROBASEDSTRING directive are not corrected automatically
- While typed pointers can be simulated, the compilation of code using untyped pointers often fails.
- The keyword absolute cannot be converted adequately
- Little effort has been done to test the COM technologies of the Delphi ActiveX framework.
- At the current state Delphi2C# doesn't deal with event handling

Special problems:

lifetime extension of bound variables

Most of the basic input output types and functions are converted, but not all. E.g. the writing and reading "file of record" isn't possible yet. Also the width and precision arguments in write operations aren't converted correctly yet.

## 12 PInvoke

"PInvoke" or "P/Invoke" are the abbreviated notations for "Platform invoke" or "Platform invocation services". Generally that's the technology to access unmanaged code from managed code. With regard to Delphi2C# only the use of the Windows API matters. Especially the big *Winapi.Windows.pas* nearly completely consists in calls of functions of the Windows API, but there are other units of the VCL/RTL too, where such calls take place.

It's often not easy to find the correct signatures for such calls. This is indicated by the amount of tutorials and forum questions to this subject in the internet and there even is a complete website, where a lot of users have assembled and discussed the PInvoke signatures, that they found out.

<http://www.pinvoke.net>

In Delphi the calls of external API's is managed in a systematic way. It should be possible therefore to convert the Delphi signatures to C# PInvoke signatures. At the development of Delphi2C# much time was spent to this subject and there are many API calls that are converted correctly. But unfortunately there also remain a calls, where the signatures have to be post-processed manually. In such cases *pinvoke.net* is a valuable source. Often there also may be a direct pendant in C# for the needed API function.

For call of a Windows API function attributes have to be specified, at least the DllImport attribute. For

parameters and return values further attributes can be added, which specify the manner, how the according types are passed. This treatment is called "marshalling". The most simple case is, that the data types have a common representation in both managed and unmanaged memory. An additional attribute is necessary, if arrays of a fixed size are passed.

The conversion becomes more difficult when pointers have to be passed, because C# hasn't pointers at all. In such cases Delphi2C# creates adapter-functions, which are called with the pointer simulating types. Adapter functions are created too, if a buffer of characters or a string has to be retrieved from the unmanaged code.

The most complicated case is, that an API function uses a function pointer as parameter.

## 12.1 DllImport

For call of a Windows API function attributes have to be specified. The minimal attribute to call a function from from an external dll is "DllImport" with the name of the dll as parameter:

```
[DllImport(kernel32)]
```

In this example the routine is looked up in the *kernel32.dll*.

## 12.2 SetLastError

The default attributes that Delphi2C# creates for Windows API calls are:

```
[DllImport(kernel32, SetLastError=true)]
```

"SetLastError=true" asserts, that additional error information (an unsigned 32 bit error code), which can be retrieved by a call of "GetLastError", is not overwritten with errors, which might be caused of the CLR, after the call returns from unmanaged to managed code.

Though it makes sense to set this attribute only for calls, which can change the last error, Delphi2C# cannot distinguish these cases and always sets this attribute. It doesn't harm in those case, where it wouldn't be useful.

## 12.3 Common datatypes

The most simple case of calling Windows API functions from C# is, that all parameter and return types are isomorphic, i.e. they have a common representation in both managed and unmanaged memory. A simple example is the function *GetCurrentThreadId*. The Delphi code imports this function with the following lines of code:

```
function GetCurrentThreadId: DWORD; stdcall;
{$EXTERNALSYM GetCurrentThreadId}

implementation

function GetCurrentThreadId; external kernel32 name 'GetCurrentThreadId';
```

In C# this simply becomes to:

```
[DllImport(kernel32, SetLastError=true)]
public static extern uint /*stdcall*/ GetCurrentThread();
```

This function can be called in the code e.g. by:

```
uint CurThread = GetCurrentThreadId();
```

The conversion of the Delphi code also is straightforward in the following example of *GetLocalTime*, where a *TSystemTime* var parameter is passed.

```
type
  _SYSTEMTIME = record
    wYear: Word;
    wMonth: Word;
    wDayOfWeek: Word;
    wDay: Word;
    wHour: Word;
    wMinute: Word;
    wSecond: Word;
    wMilliseconds: Word;
  end;
  {$EXTERNALSYM _SYSTEMTIME}
  TSystemTime = _SYSTEMTIME;
  SYSTEMTIME = _SYSTEMTIME;
  {$EXTERNALSYM SYSTEMTIME}

procedure GetLocalTime(var lpSystemTime: TSystemTime); stdcall;
{$EXTERNALSYM GetLocalTime}

implementation

procedure GetLocalTime; external kernel32 name 'GetLocalTime';
```

The *\_SYSTEMTIME* structure, that Delphi2C# generates for C# only consists in fields with data types, which need no special marshalling.

```
public struct _SYSTEMTIME
{
    public ushort wYear;
    public ushort wMonth;
    public ushort wDayOfWeek;
    public ushort wDay;
    public ushort wHour;
    public ushort wMinute;
    public ushort wSecond;
    public ushort wMilliseconds;
    public static _SYSTEMTIME CreateRecord(){return new _SYSTEMTIME();}
};
```

Therefore the *GetLocalTime* function also is converted quite easily:

```
[DllImport(kernel32, SetLastError=true)]
public static extern void /*stdcall*/ GetLocalTime(
    ref _SYSTEMTIME lpSystemTime);
```

*GetLocalTime* is called inside of the *SysUtils.Date*-function:

```
var
    SystemTime: TSystemTime;
begin
    GetLocalTime(SystemTime);
```

Delphi2C# automatically generates the following call:

```
_SYSTEMTIME SYSTEMTIME = _SYSTEMTIME.CreateRecord();
GetLocalTime(ref SYSTEMTIME);
```

## 12.4 Array size attribute

In the next example a record is marshalled, which has array fields with a fixed array size.

```
type
  PCPInfo = ^TCPInfo;
  {$EXTERNALSYM _cpinfo}
  _cpinfo = record
    MaxCharSize: UINT; { max length (bytes) of a char }
    DefaultChar: array[0..MAX_DEFAULTCHAR - 1] of Byte; { default character }
    LeadByte: array[0..MAX_LEADBYTES - 1] of Byte; { lead byte ranges }
  end;
  TCPInfo = _cpinfo;
  {$EXTERNALSYM CPINFO}
  CPINFO = _cpinfo;

  {$EXTERNALSYM GetCPInfo}
  function GetCPInfo(CodePage: UINT; var lpCPInfo: TCPInfo): BOOL; stdcall;

implementation

function GetCPInfo; external kernel32 name 'GetCPInfo';
```

In this case attributes have to be set to the according fields:

```
public struct _cpinfo
{
  public uint MaxCharSize; /* max length (bytes) of a char */
  [MarshalAs(UnmanagedType.ByValArray, SizeConst=2)] public byte[] DefaultChar; /* default character */
  [MarshalAs(UnmanagedType.ByValArray, SizeConst=12)] public byte[] LeadByte; /* lead byte ranges */
  public static _cpinfo CreateRecord(){return new _cpinfo();}
};

[DllImport(kernel32, SetLastError=true)]
public static extern int /*stdcall*/ GetCPInfo(
    uint CodePage,
    ref _cpinfo lpCPInfo);
```

The function is called in Delphi with:

```
var
  AnsiCPInfo: TCPInfo;

GetCPInfo(CP_ACP, AnsiCPInfo);
```

and in C# with:

```
_cpinfo AnsiCPInfo = _cpinfo.CreateRecord();
GetCPInfo((uint) CP_ACP, ref AnsiCPInfo);
```

A special case are character arrays like in the `_OSVERSIONINFOW` record:

```
szCSDVersion: array[0..127] of WideChar; { Maintenance UnicodeString for PSS usage }
```

This is converted to:

```
[MarshalAs(UnmanagedType.ByValTStr, SizeConst=128)] public string szCSDVersion; /* Maintenance AnsiSt
```

## 12.5 Adapter functions

Many API functions have parameters with pointer type. C# hasn't pointers at all, but it has the *IntPtr* type, which is used in such cases. In such cases Delphi2C# creates adapter-functions in addition to the original Delphi function declarations. Simulated pointers are passed to these adapter-functions and inside of the functions they are converted to *IntPtr*'s, which then can be passed again to the real API function. If the pointer is used to retrieve a value from the unmanaged code, the *IntPtr* is converted back to the simulated pointer after the API call. This is a kind of "double-marshalling".

If, for example, the values for an Integer shall be retrieved from the unmanaged code, this can be done by a var parameter of a *PInteger* type:

```
procedure APIfoo(var pi : PInteger);
{$EXTERNALSYM APIfoo}

implementation

procedure APIfoo(var pi : PInteger); external bar32 name 'APIfoo';
```

Delphi2C# creates an adapter function with an "ref Pointer<int>" parameter. The simulated pointer types have a member function "ToIntPtr", which returns the needed *IntPtr*-value. However, this member function isn't called directly, but inside of a global function "ToIntPtr", which asserts that the simulated pointer isn't null.

```
[DllImport(bar32, SetLastError=true)]
public static extern void APIfoo(
    /*ref*/ IntPtr pi);

public static void APIfoo(ref Pointer<int> pi)
{
    APIfoo(ToIntPtr(pi));
}
```

```

    FromIntPtr(pi);
}

```

"ToIntPtr" allocates some memory. This is freed in the subsequent call of "FromIntPtr(lpFileTime)", which also writes the retrieved integer value back to the simulated pointer variable.

Adapter functions are created too, if a buffer of characters or a string has to be retrieved from the unmanaged code.

## 12.6 Retrieve a string

There are many API functions to retrieve strings. In these cases a buffer is passed which then is filled with characters. A C# string cannot be used as such a buffer, because its internal buffer cannot be "pre-allocated". But `StringBuilder` can be used instead.

```

function GetTempPath(nBufferLength: DWORD; lpBuffer: LPWSTR): DWORD; stdcall;
{$EXTERNALSYM GetTempPath}

implementation

function GetTempPath; external kernelbase name 'GetTempPathW';

```

becomes to:

```

[DllImport(kernelbase, SetLastError=true)]
public static extern uint /*stdcall*/ GetTempPath(
    uint nBufferLength,
    StringBuilder lpBuffer);

public static uint /*stdcall*/ GetTempPath(uint nBufferLength, ref char[]
lpBuffer)
{
    StringBuilder tmp3 = new StringBuilder(lpBuffer.ToString(),
lpBuffer.Length);
    uint tmp2 = GetTempPath(nBufferLength, tmp3);
    lpBuffer = tmp3.ToString().ToCharArray();
    return tmp2;
}

```

Delphi2C# creates an adapter function with a "ref char[]" parameter. This buffer must already have enough space for the retrieved characters. Correctly the first parameter *nBufferLength* should be used to set the size for the buffer, but Delphi2C# cannot know this from the pure syntax of the function. But if the original Delphi code works, then the generated C# code will work too.

If an API function has LPWSTR parameters Delphi2C# also creates a second adapter function with a string reference parameter instead of the character array reference:

```

public static uint /*stdcall*/ GetTempPath(uint nBufferLength, ref string lpBuffer)

```

```

{
// StringBuilder tmp1 = new StringBuilder(lpBuffer, lpBuffer.Length);
StringBuilder tmp1 = new StringBuilder((int)nBufferLength);
uint tmp0 = GetTempPath(nBufferLength, tmp1);
lpBuffer = tmp1.ToString();
return tmp0;
}

```

Here the same problem with the buffer size exist. The string has to have the required size before *GetTempPath* is called and it will have this size, if the converted Delphi code had been correct.

```

char[] Path = new char[256];
uint ui = GetTempPath(256, ref Path);

string sPath = string.Empty;
SetLength(ref sPath, 256);
ui = GetTempPath(256, ref sPath);

```

## 12.7 API callback

API function may use callback parameters. In this case not only the parameters of the API function have to be adapted, the parameters of the callback function have to be adapted too. The "EnumSystemLocales" function is such an example:

```

type
TFNLocaleEnumProc = Pointer;

{$EXTERNALSYM EnumSystemLocales}
function EnumSystemLocales(lpLocaleEnumProc: TFNLocaleEnumProc; dwFlags: DWORD): bool; stdcall;

implementation

function EnumSystemLocales; external kernel32 name 'EnumSystemLocales';

function EnumLocalesCallback(LocaleID: PChar): Integer; stdcall;
begin
// ...
end;

procedure CallEnum;
begin
EnumSystemLocales(@EnumLocalesCallback, LCID_SUPPORTED);
end;

```

The callback function "EnumLocalesCallback" has a *PChar* parameter, which becomes the simulated *PChar*-class in C#. Of course the unmanaged code couldn't do anything with it. Delphi2C# therefore creates a sibling function to "EnumLocalesCallback" with the same name, but with parameters, which are conform with unmanaged code and which can be used to call the original "EnumLocalesCallback" function. By means of the function "Marshal.GetFunctionPointerForDelegate<TDelegate>", the "EnumLocalesCallback" sibling can be converted than to a function pointer, that can be called in the unmanaged code.

This is what Delphi2C# makes from the code above:

```

public delegate int callback__0([MarshalAs(UnmanagedType.LPStr)] string LocaleID); // LPWSTR manually

```

```

public static int EnumLocalesCallback([MarshalAs(UnmanagedType.LPStr)] string LocaleID) // LPWStr manual
{
    return EnumLocalesCallback(new PChar(LocaleID));
}

[DllImport(kernel32, SetLastError=true)]
public static extern bool /*stdcall*/ EnumSystemLocales(
    IntPtr lpLocaleEnumProc,
    DWORD dwFlags);

public static int /*stdcall*/ EnumLocalesCallback(PChar LocaleID)
{
    int result = 0;
    // ...
    return result;
}

public static void CallEnum()
{
    EnumSystemLocales(Marshal.GetFunctionPointerForDelegate<callback__0>(EnumLocalesCallback), LCID_SUPPO
}

```

As template parameter "TDelegate" for "GetFunctionPointerForDelegate<TDelegate>" the delegate "callback\_\_0" is used. "callback\_\_0" just has the signature of the "EnumLocalesCallback" sibling.

When "EnumSystemLocales" is called in the managed code, the unmanaged code will call the "EnumLocalesCallback" sibling, which then will call the original "EnumLocalesCallback".

## 13 Pretranslated C# code

Delphi2C# ships with some pre-translated parts of the Delphi RTL/VCL. You also can improve and accelerate your translations, if you prepare parts of your own Delphi code.

### 13.1 Delphi RTL/VCL

User Delphi code is based on the Delphi RTL and the VCL. One could think this isn't a problem, since this code can be translated by *Delphi2C#* as well as the own code. Unfortunately, it is not quite so simple. Particularly the file *System.pas* makes problems. *System.pas* is interlocked with the Delphi compiler narrowly. Some fundamental functions are built into the Delphi compiler and some parts are encoded in a special manner, which are interpreted correctly from the Delphi compiler only. For example the symbol "\_AnsiStr" is used instead of "AnsiString" and the same applies to quite a number of other basic types. *System.pas* further depends partly on assembler code. RTL/VCL sources also convert API functions and types of the operating system such that they conform to Delphi. In C++ this conversion isn't necessary. Also some parts of *System.pas* aren't needed in C++.

Therefore some parts of the Delphi RTL are pre-translated and prepared to use with the code translated by *Delphi2C#*. Because Embarcadero has the copyright of the Delphi RTL/VCL the translated parts cannot be shipped with the *Delphi2C#* installer. However as customer of *Delphi2C#* you certainly will have a license of Delphi too and as owner you also have the right to use the



translated code. So you can get the C# version of the Delphi code, if you send a prove of the Delphi ownership to me or simply send the Delphi RTL code to me.

Some helping code is already delivered with the Delphi2C# installer:

```
DelphiSets.cs.  
System.cs
```

### 13.1.1 DelphiSets.cs

DelphiSets.cs is part of the C# version of the Delphi RTL and contains the class TSet to simulate Delphi sets. The public members of this class are listed below:

```
public class TSet : IEnumerable  
  
    public TSet()  
    public TSet(TSet OtherSet)  
    public bool this[int index]  
    public TSet Clear()  
    public void Include(int Index)  
    public static TSet operator <<(TSet ImpliedObject, int Index)  
    public void Exclude(int Index)  
    public static TSet operator >>(TSet ImpliedObject, int Index)  
    public bool Empty()  
    public bool Contains(int Index)  
    public static TSet operator +(TSet ImpliedObject, TSet AddSet)  
    public static TSet operator -(TSet ImpliedObject, TSet SubSet)  
    public static TSet operator *(TSet ImpliedObject, TSet MulSet)  
    public static bool operator ==(TSet ImpliedObject, TSet CmpSet)  
    public static bool operator !=(TSet ImpliedObject, TSet CmpSet)  
    public static bool operator <=(TSet ImpliedObject, TSet CmpSet)  
    public static bool operator >=(TSet ImpliedObject, TSet CmpSet)  
    public override int GetHashCode()  
    public override bool Equals(object other)  
    public IEnumerator GetEnumerator()
```

### 13.1.2 System.cs

The System.cs file that is delivered with the Delphi2C# installer is only a part of the complete System.cs that is contained the C# version of the Delphi RTL that you will get as owner of Delphi2C# and of Delphi. This part of the System.cs only contains the classes to simulate pointers. The public members of these classes are listed below:

```
public interface IPointer<T> : IDisposable  
  
    int Length  
    int Capacity  
    int Position  
    bool IsNull();  
    void SetNull();  
    T Deref();
```

```
void Assign(T c, int Index = 0);  
void Inc(int i);  
void Dec(int i);
```

```
public class PChar : IPointer<char>  
  
    public PChar()  
    public PChar(int Size, bool Alloc)  
    public PChar(bool Value)  
    public PChar(char c)  
    public PChar(double Value)  
    public PChar(short Value)  
    public PChar(int Value)  
    public PChar(long Value)  
    public PChar(float Value)  
    public PChar(ushort Value)  
    public PChar(uint Value)  
    public PChar(ulong Value)  
    public PChar(byte[] bytes)  
    public PChar(string S, int Index = 0)  
    public PChar(char[] arr, int Index = 0)  
    public PChar(PChar Other, int Delta = 0)  
    public PChar(Pointer ptr)  
    public char this[int index]  
    public int Length  
    public int Capacity  
    public int Position  
    public bool IsNull()  
    public void SetNull()  
    public char Deref()  
    public IntPtr ToIntPtr()  
    public void FromIntPtr()  
    public void Assign(char c, int Index = 0)  
    public void Assign(string s)  
    public void Inc(int i)  
    public void Dec(int i)  
    public void DerefAdd(int Value)  
    public void DerefSubtract(int Value)  
    public static PChar operator ++(PChar a)  
    public static PChar operator --(PChar a)  
    public override string ToString()  
    public char[] ToCharArray()  
    public void Synchronize(ref string s)  
    public void Synchronize(ref char[] ca)  
    public string Substring(int from, int count)  
    public void Insert(string s)  
    public void Insert(string s, int startIndex)  
    public static bool operator ==(PChar p1, PChar p2)  
    public static bool operator !=(PChar p1, PChar p2)  
    public static PChar operator +(PChar p, int Value)  
    public static PChar operator -(PChar p, int Value)  
    public override bool Equals(object obj)  
    public override int GetHashCode()  
    public static implicit operator Pointer (PChar ptr)  
    public static implicit operator PChar (Pointer ptr)  
    public void Dispose()
```

```
public class Pointer : IPointer<object>

    public Pointer()
    public Pointer(int size, bool Alloc)
    public Pointer(bool Value)
    public Pointer(char Value)
    public Pointer(double Value)
    public Pointer(short Value)
    public Pointer(int Value)
    public Pointer(long Value)
    public Pointer(float Value)
    public Pointer(ushort Value)
    public Pointer(uint Value)
    public Pointer(ulong Value)
    public Pointer(byte[] bytes)
    public Pointer(IntPtr ptr, int size = 0, bool disposed = true)
    public Pointer(Pointer Other) //, int Delta = 0)
    public bool IsNull()
    public void SetNull()
    public int Length
    public int Capacity
    public int Position
    public void Assign(object t, int Index = 0)
    public void Assign(IntPtr t, int size = 0)
    public object Deref()
    public IntPtr GetIntPtr()
    public byte[] ToBytes()
    public void FromBytes(byte[] bytes)
    public void Inc(int i)
    public void Dec(int i)
    public override bool Equals(object obj)
    public override int GetHashCode()
    public static Pointer operator ++(Pointer a)
    public static Pointer operator --(Pointer a)
    public static bool operator ==(Pointer p1, Pointer p2)
    public static bool operator !=(Pointer p1, Pointer p2)
    public static Pointer operator +(Pointer p, int Value)
    public static Pointer operator -(Pointer p, int Value)
    public static implicit operator IntPtr (Pointer ptr)
    public static implicit operator Pointer (IntPtr ptr)
    public static implicit operator uint (Pointer ptr)
    public static implicit operator Pointer (uint ptr)
    public void Dispose()

public class Pointer<T>: IPointer<T> where T : new()
    public Pointer()
    public Pointer(int Size, bool Alloc)
    public Pointer(byte[] bytes)
    public Pointer(T t)
    public Pointer(IntPtr iptr)
    public Pointer(List<T> l, int Index = 0)
    public Pointer(T[] l, int Index = 0)
    public Pointer(Pointer<T> Other, int Delta = 0)
    public bool IsNull()
    public void SetNull()
    public T this[int index]
```

```

public int Length
public int Capacity
public int Position
public object Object
public void Assign(T t, int Index = 0)
public T Deref()
public IntPtr ToIntPtr()
public byte[] ToBytes()
public void FromBytes(byte[] bytes)
public T FromIntPtr()
public void Inc(int i)
public void Dec(int i)
public override bool Equals(object obj)
public override int GetHashCode()
public static Pointer<T> operator ++(Pointer<T> a)
public static Pointer<T> operator --(Pointer<T> a)
public static bool operator ==(Pointer<T> p1, Pointer<T> p2)
public static bool operator !=(Pointer<T> p1, Pointer<T> p2)
public static Pointer<T> operator +(Pointer<T> p, int Value)
public static Pointer<T> operator -(Pointer<T> p, int Value)
public void Dispose()

```

## 13.2 Preparing Delphi code

Normally a preparation of the Delphi code should not be necessary. But there are three reasons to do so:

- sometimes the RTL/VCL code isn't clean
- some substitutions for ampersand-expressions have to be defined
- parallel updates of Delphi and C# code can be simplified

### 13.2.1 Bugs in the Delphi RTL/VCL

In some cases Delphi2C# cannot process a unit though the Delphi compiler can. That's because the automatically generated parser of Delphi2C# is more strict than the Delphi parser. The Delphi parser might be handwritten and tolerates bugs like the following in the System.pas of RAD Studio 10.2 Tokyo inside of the function "FSetExceptFlag":


```
{ELSEIF defined(CPUX64) and defined(Linux)) }
```

It is obvious, that there is a closing parenthesis too much and the code should be corrected to:

```
{ELSEIF defined(CPUX64) and defined(Linux) }
```

The next bug in the same file is:

```
{IF not (defined(PC_MAPPED_EXCEPTIONS) or defined(SJLJ_BASED_EXCEPTIONS)) or defined(ZCX_BASED_EXCEPTI
```

Such bugs unfortunately exist in all versions of the RTL/VCL at different positions. They can be found inside of the Delphi2C# IDE quite easily, because the position where the preprocessor or the parser stops is shown in the input editor. If you have moved the cursor, the position is shown again by use of the  button.

Here is a list of some flaws in the RTL/VCL of RAD Studio 10.2 Tokyo.

System.ObjAuto.pas line 23:

```
{$IF SizeOf(Extended) >= 10} // 10,12,16
  {$DEFINE EXTENDEDHAS10BYTES}
{$ENDIF}

{$IF SizeOf(Extended) = 10}
  {$DEFINE EXTENDEDIS10BYTES}
{$ENDIF}
```

should be:

```
{$IF SizeOf(Extended) >= 10} // 10,12,16
  {$DEFINE EXTENDEDHAS10BYTES}
{$ENDIF}

{$IF SizeOf(Extended) = 10}
  {$DEFINE EXTENDEDIS10BYTES}
{$ENDIF}
```

Internal.Unwinder.pas:

```
{$IFDEF MACOS}
const
  _U = '_';
  {$EXTERNALSYM _U}
{$ELSE !MACOS}
  _U = '';
  {$EXTERNALSYM _U}
{$ENDIF}
```

could be:

```
{$IFDEF MACOS}
const
  _U = '_';
  {$EXTERNALSYM _U}
{$ELSE !MACOS}
const
  _U = '';
  {$EXTERNALSYM _U}
{$ENDIF}
```

System.pas line 6643:

```
  {$ELSEIF defined(CPUX64) and defined(Linux)} }
->  {$ELSEIF defined(CPUX64) and defined(Linux)} }
```

line 24087:

```
  {$IF not (defined(PC_MAPPED_EXCEPTIONS) or defined(SJLJ_BASED_EXCEPTIONS)) or defined(ZCX_BASED_EXCEPTI
```

```
->
{$IF not (defined(PC_MAPPED_EXCEPTIONS) or defined(SJLJ_BASED_EXCEPTIONS)) or defined(ZCX_BASED_EXCEPTI
```

Vcl.Imaging.GifImg.pas line 2421:

```
SetColors(GetPaletteEntries(Palette, 0, 256, nil^));
->
SetColors(GetPaletteEntries(Palette, 0, 256, nil));
```

WinAPI.DXFile.pas line 37:

```
(*$HPPEMIT '#include "dxfile.h"'{*}
*$HPPEMIT '#include "rmxfguid.h"'{*}
*$HPPEMIT '#include "rmxftmpl.h"'{*}

->

(*$HPPEMIT '#include "dxfile.h"'{*}
*$HPPEMIT '#include "rmxfguid.h"'{*}
*$HPPEMIT '#include "rmxftmpl.h"'{*}
```

ToolsApi/ToolsApi.pas line 123/250/252

```
(*$HPPEMIT 'DEFINE_GUID(IID_IOTASStreamModifyTime,0x49F2F63F,0x60CB,0x4FD4,0xB1,0x2F,0x81,0x67,0xFC,0x79
...
*$HPPEMIT 'DEFINE_GUID(IID_IOTAToolsFilterNotifier,0xCEF1F13A,0xE877,0x4F20,0x88,0xF2,0xF7,0xE2,0xBA,0
*$HPPEMIT 'DEFINE_GUID(IID_IOTAToolsFilter,0x8864B891,0x9B6D,0x4002,0xBB,0x2E,0x1D,0x6E,0x59,0xBF,0xA4
.
*$HPPEMIT 'DEFINE_GUID(IID_IOTATypeLibrary, 0x7A2F5910,0x58D2,0x448E,0xB4,0x57,0x2D,0xC0,0x1E,0x85,0x3

->
(*$HPPEMIT 'DEFINE_GUID(IID_IOTASStreamModifyTime,0x49F2F63F,0x60CB,0x4FD4,0xB1,0x2F,0x81,0x67,0xFC,0x79
...
*$HPPEMIT 'DEFINE_GUID(IID_IOTAToolsFilterNotifier,0xCEF1F13A,0xE877,0x4F20,0x88,0xF2,0xF7,0xE2,0xBA,0
*$HPPEMIT 'DEFINE_GUID(IID_IOTAToolsFilter,0x8864B891,0x9B6D,0x4002,0xBB,0x2E,0x1D,0x6E,0x59,0xBF,0xA4
.
*$HPPEMIT 'DEFINE_GUID(IID_IOTATypeLibrary, 0x7A2F5910,0x58D2,0x448E,0xB4,0x57,0x2D,0xC0,0x1E,0x85,0x3
```

## 13.2.2 Frequent re-translation

Users who like to continue to develop their Delphi code and in parallel also need the C# code updated certainly don't want to post-process the generated code again and again. Therefore Delphi2C# offers the possibility to prepare the Delphi source code such, that Delphi2C# will reproduce the corrected code fragments. These fragments either can be inserted as special [comments](#) (`*#_..._#*`) or can be hidden by conditional compilation with use of the [predefined identifier CSHARP](#). In fact the second method is based on the first, because the Delphi2C# pre-processor converts the CSHARP

part into the special comments and the Delphi2C# translator than simply removes the special brackets (*\*#\_ ... \_#\**).

### 13.2.2.1 Comments (*\*#\_ ... \_#\**)

Delphi2C# interprets the extended Delphi brackets (*\*#\_ ... \_#\**) in a special way. A text in such brackets is taken unchanged into the C# code.

For example an additional header is included into the C# code by the following line:

```
(*_using System.SysUtils;_#*)  
->  
using System.SysUtils;
```

### 13.2.2.2 Predefined identifier CSHARP

In addition to the definitions which the user can set in the translation options the identifier *CSHARP* always is defined in *Delphi2C#*. The preprocessor treats this identifier in a special manner. The preprocessor not simply writes the according code into the pre-processed code, but it puts it into the special brackets (*\*#\_ ... \_#\**). In a second step the translator then removes the brackets.

For example:

```
{$ifdef CSHARP}  
  out << s << endl;  
{$else}  
  WriteLn(s);  
{$endif}
```

The pre-processed code then is:

```
(*_ out << s << endl; _#*)
```

and because of the special treatment of the brackets (*\*#\_...\_#\**), the final C# output is:

```
out << s << endl;
```

*Delphi2C#* ignores the part of code in the *{\$else}*-section completely, but it is visible to the Delphi compiler. So, this special manner of the conditional compilation makes it possible that both the original Delphi code and the generated C# code remain compiling.

The identifiers in these section either can be normalized or can be left untouched. This is controlled by the CSHARP unification option.

### 13.2.3 Delphi directives to support C++Builder

There are four directives defined in Delphi to support the generation of C++ header files for C++Builder. For Delphi2C# these directives are of little interest, because the interaction with the Windows API is controlled by the `PIInvoke` statements. However Delphi2C# uses the `EXTERNALSYM` directives to recognize, for which functions `PIInvoke` adapters have to be constructed.

```
$HPPEMIT  
$EXTERNALSYM  
$NODEFINE  
$NOINCLUDE
```

#### 13.2.3.1 \$HPPEMIT

The `HPPEMIT` directive adds a specified symbol to the C++ header file.

`HPPEMIT` directives are output into the "user supplied" section at the top of the header file in the order in which they appear in the Pascal file.

The `HPPEMIT` directive accepts an optional `END` directive that instructs the compiler to emit the string at the bottom of the header file. Otherwise, the string is emitted at the top of the file.

**Syntax:**

```
{$HPPEMIT string}
```

**Example:**

```
{$HPPEMIT 'Symbol goes to top of file' }.  
{$HPPEMIT END 'Symbol goes to bottom of file'}
```

#### 13.2.3.2 \$EXTERNALSYM

The `EXTERNALSYM` directive prevents the specified Pascal symbol from appearing in C++ header files. This directive is used for types, which already are defined in the API of the operation system. For Delphi these types have to be redefined, for C++ not.

Delphi2C# doesn't output code parts, which are marked with the `EXTERNALSYM` directive if the according option is enabled.

**Syntax:**

```
{$EXTERNALSYM identifier}
```

**Example:**

```
type  
  size_t : LongWord;  
  {$EXTERNALSYM size_t}
```



### 13.2.3.3 \$NODEFINE

The *NODEFINE* directive prevents the specified symbol from being included in the C++ header file of C++Builder, while allowing some information to be output to the OBJ file. For example the types TSize, TPoint and, TRect in System.Types.pas are marked with *NODEFINE*. In C++Builder these types are defined in System.Types.h.

Delphi2C# doesn't output code parts, which are marked with the *NODEFINE* directive if the according option is enabled. In this case it is your responsibility to define the necessary code yourself, One possibility is to do it with *HPPEMIT*:

**Syntax:**

```
{ $NODEFINE identifier }
```

**Example:**

```
type
    Temperature = type double;
    { $NODEFINE Temperature }
    { $HPPEMIT 'typedef double Temperature' }
```

### 13.2.3.4 \$NOINCLUDE

The *NOINCLUDE* directive prevents the specified file from being included in header files generated for C#.

**Syntax:**

```
{ $NOINCLUDE filename }
```

**Example:**

```
{ $NOINCLUDE Unit1 } // removes #include Unit1.
```

## 14 Formatting

The generated C# code should be readable, but little effort was made to make it beautiful. There are free pretty-printers available, which have a lot of options to format the code just as you like it. I recommend:

<http://universalindent.sourceforge.net/>

With UniversalIndentGUI "you change the value of a parameter and directly see how your reformatted code will look like. Save your beauty looking code or create an anywhere usable batch/shell script to reformat whole directories or just one file even out of the editor of your choice that supports external tool calls."

## 15 DelphiC# versus Delphi2Cpp

*Delphi2C#* is based on the experience with the earlier "Delphi2Cpp" and the current "DelphiXE2Cpp11". In contrast to the earlier "Delphi2Cpp", which processes Delphi 7 code only, "DelphiXE2Cpp11" like *Delphi2C#* processes all Delphi language expansions which were added since then.

## 16 TextTransformer

*Delphi2C#* and the previous *Delphi2Cpp* were made from a TextTransformer project, which is based on the Delphi parser and the Delphi pretty-printer, which can be obtained freely from

[http://www.texttransformer.org/Delphi\\_en.html](http://www.texttransformer.org/Delphi_en.html)

[http://www.texttransformer.org/DelphiPrettyPrint\\_en.html](http://www.texttransformer.org/DelphiPrettyPrint_en.html)

## 17 Service

There is also a service to make translations of Delphi source code for you. So you don't have to buy the program:

[http://www.texttransformer.com/D2C\\_TranslationService\\_en.html](http://www.texttransformer.com/D2C_TranslationService_en.html)

or in German at:

[http://www.texttransformer.de/D2C\\_TranslationService\\_ge.html](http://www.texttransformer.de/D2C_TranslationService_ge.html)

I also like make extensions of Delphi2C# or other translators adapted individually for you. The translation results can be increased drastically by such customizations. Please contact me by the contact form at:

[http://www.texttransformer.com/Contact\\_en.html](http://www.texttransformer.com/Contact_en.html)

or in German at:

[http://www.texttransformer.de/Contact\\_ge.html](http://www.texttransformer.de/Contact_ge.html)

# Index

- - -

- 95  
-- 80

- # -

#helped will not be changed 101

- & -

& 43

- ( -

(\*#\_ ... \_#\*) 143  
(\*\_ ... \_) 143  
(\*\_...\_) 14

- \* -

\* 95

- / -

/ 95  
/\*# 29  
//# 29

- [ -

[&] 106  
[=] 106

- \_ -

\_\_closure 93  
\_\_interface 61

- { -

{\$J+} directive 26

- + -

+ 95  
++ 80

- < -

< 95  
<< 80  
<= 95

- > -

>> 80

- A -

Abs 14  
abstract methods 60  
ActiveX 128  
Add 95  
Add include path 12  
Add recursively 12  
AddError 40  
AddMessage 40  
AddWarning 40  
ambiguity 71  
Ampersand 43  
Ancestor 52  
and 73, 95  
anonymous methods 104  
API callback 135  
Array 62  
Array of const 66, 68  
array parameter 78  
Array size 68, 69  
Array size attribute 132  
arrays 63  
Assembler 18, 128  
Assigned 80  
AssignFile 91

Assignments 74  
auto 86

## - B -

Backup 40  
Base class 52, 58  
Beautifier 145  
binary operator 96  
bitwise operator 73  
BitwiseAnd 95  
BitwiseOr 95  
BitwiseXor 95  
BlockRead 80  
BlockWrite 80  
boolean operator 73  
boxing 66  
break 34  
BytesOf 34  
ByteType 34

## - C -

c\_str 34  
C++ Builder 4  
C++ header 35  
C++ source file 35  
C++11 63, 86  
C++Builder 144  
capture 106  
Case sensibility 43  
Case sensitivity 34  
Case-sensitivity 22  
cat\_printf 34  
cat\_sprintf 34  
cat\_vprintf 34  
Char 34  
Character buffer 134  
Class 50  
Class creation 61  
class helper 100  
class method 58  
class reference 88  
class\_id 88  
Class-like records 101  
ClassRef 28, 88  
class-reference 87

Clear types and variables 5, 35  
Clear windows 5  
CloseFile 91  
CodePage 34  
COM technologies 128  
command line mode 41  
Command line parameter 41  
Comments 48  
Compare 34  
CompareIC 34  
Compile time functions 14  
Compiler 4  
Conditional compilation 15, 33  
Conflicting names 128  
Connect 93  
const 26, 76, 119  
const correctness 26  
const parameter 119  
const parameters 76  
const\_cast 119  
constant 26, 119  
const-correctness 28, 119  
Constructor 52, 56  
Constructors of the base class 128  
Contact 146  
continue 34  
conversion operator 97  
Copy 80  
Cpp 14  
Cpp definition 43, 143  
CPUX86 16  
CreateRecord 50  
CreateRecordMembers 50  
CSHARP 18  
CurrToStr 34  
CurrToStrF 34  
Customization 146

## - D -

d2c\_LoadResourceString 72  
d2c\_sysfile.cpp 80  
d2c\_sysfile.h 80  
d2c\_sysobj 56  
d2c\_system 14  
Daniel Flower 70  
data 34  
Dec 80, 95

Decimals 81  
Default array-property 84  
Default.prj 2  
Definition 4, 15  
delegate 104, 107  
Delete 34, 80  
Delphi ActiveX framework 128  
Delphi I/O routines 14  
Delphi RTL/VCL 4, 136  
Delphi2C# 2, 146  
Delphi2Cpp 2, 146  
DelphiSets.cs 136, 137  
DelphiSets.h 70  
DelphiXE2Cpp11 146  
DelphiXE2Cpp11Lic.dat 2  
Demjen 93  
Dependencies 35  
Destructor 58  
Directive 15, 144  
Directives 33  
Dispose 80  
div 95  
Divide 95  
DllImport 130  
dotted file names 13  
Dynamic array 62  
DynamicArray 62

## - E -

E2034 119  
ElementSize 34  
EnsureUnicode 34  
Enumerated types 68  
Equal 95  
equality operators 73  
event 93, 107  
Event handling 93  
exception 91  
ExceptionRef 91  
Exclude 80  
Excluding individual files 38, 39  
Explicit 97  
Explicit casts 74  
Extended "System.pas" 14  
extended System.pas 4  
EXTERNALSYM 28, 144

## - F -

FCL 80  
File 91  
File layout 45  
File manager 36  
File of 91  
FileMode 91  
finalization 87  
Finalization part 128  
finally 86  
Fingerprint 2  
Fixed identifiers 21  
FloatToStrF 34  
FmtLoadStr 34, 72  
for loop 85  
foreach 86  
for-in loop 86  
for-loop 28  
Format 34  
FormatFloat 34  
Formatting 145  
Formatting parameters 81  
FreeMem 14  
FreePascal FCL 80  
friend 61  
function 78  
Function name 24, 75  
Functions 75

## - G -

Generics 108  
GetFunctionPointerForDelegate 135  
GetLowerBound 64  
GetMem 14  
GetUpperBound 64  
GNU Lesser General Public License 80  
GreaterThan 95  
GreaterThanOrEqual 95  
GUID 61

## - H -

hash character 29  
Hejlsberg 2

High 14, 62, 64, 80  
 HPPEMIT 144

## - I -

I/O routines 80  
 Identifier notation 34  
 Implementation-class 45  
 Implicit 97  
 In 95  
 Inc 80, 95  
 Include 80  
 Include directive 33  
 Include paths 12  
 Included files 35  
 Indexed property 82  
 Indexer notation 82, 84  
 Inheritance 52  
 inherited 78  
 initialization 26, 87  
 Initialization lists 54  
 Initialization part 128  
 Initialize Variables 28  
 initializer\_list 63  
 Initializing arrays 63  
 inline assembler 119  
 Inline assembler code 128  
 in-operator 74  
 Input options 11  
 Insert 34, 80  
 Installation 2  
 installation folder 2  
 IntDivide 95, 96  
 Interface 50, 61  
 Interface-class 45  
 IntToHex 34  
 IPointer 116  
 IPointer<T> class 137  
 IsContained 99  
 IsDelimiter 34  
 IsEmpty 34  
 IsLeadSurrogate 34  
 Isomorphic types 130  
 IsPathDelimiter 34  
 IsTrailSurrogate 34

## - K -

Keyword 24

## - L -

lambda expressions 104  
 Last error position 5  
 LastChar 34  
 LastDelimiter 34  
 Learning types and variables 5  
 LeftShift 95  
 Length 34, 62, 80  
 LessThan 95  
 LessThanOrEqual 95  
 Library 5  
 License 2  
 lifetime extension 106  
 LoadResourceString 72  
 LoadStr 34, 72  
 LoadString 34  
 Local function 79  
 Log panel 7  
 Log-file 36  
 LogicalAnd 95  
 LogicalNot 95  
 LogicalOr 95  
 LogicalXor 95, 96  
 Lookup algorithm 14  
 lookup order 71  
 Low 14, 62, 64, 80  
 LowerCase 34  
 LPWSTR parameter 134

## - M -

-m 41  
 Management 36, 41  
 Mangement 41  
 MAXIDX 65  
 MAXIDX(x) 62  
 Memory management 14  
 Message directive 18  
 Method pointers 93  
 method reference 104, 107  
 Missing constructor 56

mod 95  
module definition file 5  
Modulus 95  
MSWINDOWS 4  
Multiply 95

## - N -

N:1 36  
N:N 36  
Names of helping variables 24  
namespace 14, 71  
Negative 95  
Nested classes 103  
Nested functions 128  
Nested routines 79  
New 80  
New features 93  
NODEFINE 28, 145  
NOINCLUDE 145  
not 95  
Notation 22  
NotEqual 95

## - O -

object 66  
octothorpe 29  
Odd 14  
Open array 64, 78  
Operator 73  
operator overloading 95  
operator precedence 73  
Operators 73  
Options 10  
or 73, 95  
Order of lookup 71  
Other compiler 4, 24, 62, 82  
out parameters 76  
Output options 29  
overloading binary operators 96  
overloading conversion operators 97  
overloading unary operators 97  
Overwriting "System.pas" 14

## - P -

-p 41  
P/Invoke 129  
PAnsiChar 80  
Parameter types 76  
-pause 41  
PChar 118  
PChar class 137  
PInvoke 129, 144  
placeholder 105  
plain old data types 80  
Platform invocation services 129  
Platform invoke 129  
POD types 80  
Pointer class 137  
Pointer<T> 117  
Pointer<T> class 137  
Pointers 115  
Pos 34, 80  
Positive 95  
PP-button 5  
precedence of operators 73  
Pred 14  
Prefix 24  
Preprocessed code 4  
Preprocessor 5, 33, 34  
pre-processor can't evaluate 33  
Pretranslated C# code 136  
Pretranslator 33  
Pretty-printer 145  
Preview of the target files 39  
printf 34  
procedure 78  
Procedures 75  
Processor options 17  
professional Version 5, 7, 41  
program ID 2  
Project file 10  
property 24, 82  
PUREPASCAL 15, 18, 119

## - R -

-r 41  
Range 69

Read 82, 91  
 Read procedure 81  
 Reading and Writing 91  
 ReadLn 91  
 ReadLn procedure 81  
 ReadProperty 82, 84  
 ReallocMem 14, 76  
 ReallocMemory 76  
 Record 50, 51  
 record helper 100  
 ref 76  
 Refactoring 29  
 RefCount 34  
 reference to a method 104  
 Refresh 39  
 RegisterComponents 80  
 Registration 2  
 Rename 91  
 Reset 91  
 resource string 72  
 Result 75  
 Results 40  
 return-statement 75  
 Rewrite 91  
 RightShift 95  
 Round 95, 99  
 Routines 75  
 runtime\_error 60

## - S -

-s 41  
 scope 71  
 Search path to the source files 13  
 Search path to the VCL/RTL 13  
 Selecting source files 38  
 Service 146  
 Set 68, 70  
 Set class 70, 74  
 SetLastError 130  
 SetLength 34, 80  
 shl 95  
 shr 95  
 Simple substitutions 44  
 Single characters 44  
 Size of an array 62  
 size\_t 25  
 sprintf 34

Start a translation 40  
 Statements 85  
 Static array 62  
 Static array parameter 65  
 static class method 28  
 std::bind 105  
 std::bind1st 93  
 std::function 93  
 std::mem\_fun 93  
 std::runtime\_error 60  
 std::vector 62  
 stdexcept 60  
 Stop on message directive 18  
 stop variable 28  
 Str procedure 81  
 Starting the translation 5  
 String constant 44  
 string parameter 78  
 String type 4  
 String; 34  
 StringOfChar 34  
 StructLayout 51  
 Substitution in the translator 24  
 Substitution of the preprocessor 22  
 Substitution options 19  
 Substitution table 22  
 SubString 34, 80  
 Subtract 95  
 Succ 14  
 swap 34  
 symbol lookup 71  
 Synchronize 118  
 Synchronizing Delphi and C# code 136  
 System namespace 47  
 System unit 4  
 System.cs 136, 137  
 System.pas 4, 14, 136  
 System::Set 24, 70  
 SystemClass 47  
 Sysutils unit 4

## - T -

-t 41  
 t\_str 34  
 Tamas Demjen 93  
 Target file or folder 38  
 T-button 5



TClass 87, 88  
 TD2CObject 56  
 temporary file 40  
 Temporary variables 78  
 Text 91  
 TextFile 91  
 TextTransformer 146  
 TFoo<int>\* FFoo1;  
     TBaz::TQux<int>\* FBazQux1; 110  
     TBaz::TQux<T>\* FBazQux2; 110  
     TFoo<int>::TBar\* FFooBar1; 110  
     TFoo<T>\* FFoo2; 110  
     TFoo<T>::TBar\* FFooBar2; 110  
 this ref 101  
 ThreadStatic 72  
 threadvar 72  
 ThrowAbstractError 88  
 ThrowNoDefaultConstructorError 88  
 TMetaClass 87, 88  
 TObject 14, 52, 56  
 ToDouble 34  
 ToInt 34  
 ToIntDef 34  
 Tokens 43  
 Tool bar 5  
 Translation 33  
 Translation options 4, 10, 36  
 Translation service 146  
 Translator 5  
 Treat typed constants as non-typed constants 26  
 Trim 34  
 TrimLeft 34  
 TrimRight 34  
 Trunc 95, 99  
 TSet 24, 70  
 ttm 41  
 Tuning options 26  
 TVarRec 66  
 Type options 24  
 typed constant 26  
 Type-map 25  
 Types 49

## - U -

unary operator 97  
 unboxing 66  
 Unicode source file 94

Unification 18  
 Unification of notations 34  
 Unique 34  
 Unit frame 5  
 Unit scope name 95  
 Unit scope names 13  
 UniversalIndentGUI 145  
 Unknown architecture 18  
 Unknown platform 18  
 untyped parameters 76  
 UpperCase 34  
 Use "stop" variable in for-loop 26  
 User options 9  
 Uses clauses 46

## - V -

var 76  
 variable binding 106  
 variable parameters 76  
 Variables 71  
 Variant 51  
 Variant types 128  
 VCL 35  
 VCL-functions 80  
 vector 62  
 Verbose option 29  
 virtual class method 28  
 Virtual class methods as static 28  
 virtual constructor 88  
 Virtual constructors 56  
 Visibility 61  
 vprintf 34

## - W -

w\_str 34  
 Width 81  
 Win64 16  
 Winapi.Windows 129  
 Window position 9  
 Window size 9  
 Windows 80  
 Windows API 144  
 Windows interfaces 144  
 Windows.pas 16  
 with-statement 86

Write 82, 91  
Write procedure 81  
WriteLn 91  
WriteLn procedure 81  
WriteProperty 82, 84

**- X -**

xor 95

**- Z -**

ZEROBASEDSTRING 128