Delphi2Cpp

1 Introduction

There is no extra help for Delphi2CB, which is an low-priced extract of *Delphi2Cpp* for users of *C++Builder*.

Short description

Delphi2Cpp helps to convert Delphi source code to C++. In contrast to the first version of *Delphi2Cpp*, the current *Delphi2Cpp* 2.xx *) processes not only Delphi 7 code but all Delphi language expansions which were added since then. *Delphi2Cpp 2* also uses the new features of C++11 and later to improve the translation results. Nevertheless a manual post-processing of the produced code still will be required. However, it is aim of the program to keep the amount of the post-processing as small as possible. Some principle flaws are listed here.

A comparison of *Delphi2Cpp 2* and *Delphi2Cpp 1* is here.

Availability

The actual version of Delphi2Cpp 2 can be obtained from the TextTransformer websites:

http://www.TextTransformer.com

http://www.TextTransformer.de

*) For a while this version was called DelphiXE2Cpp11

2 Installation

The installation is done by the installer Delphi2CppInstall.exe. All files for projects, examples, source code etc.are copied into the chosen installation directory.

The default path is a sub-folder *Delphi2Cpp* in the user documents folder, like:

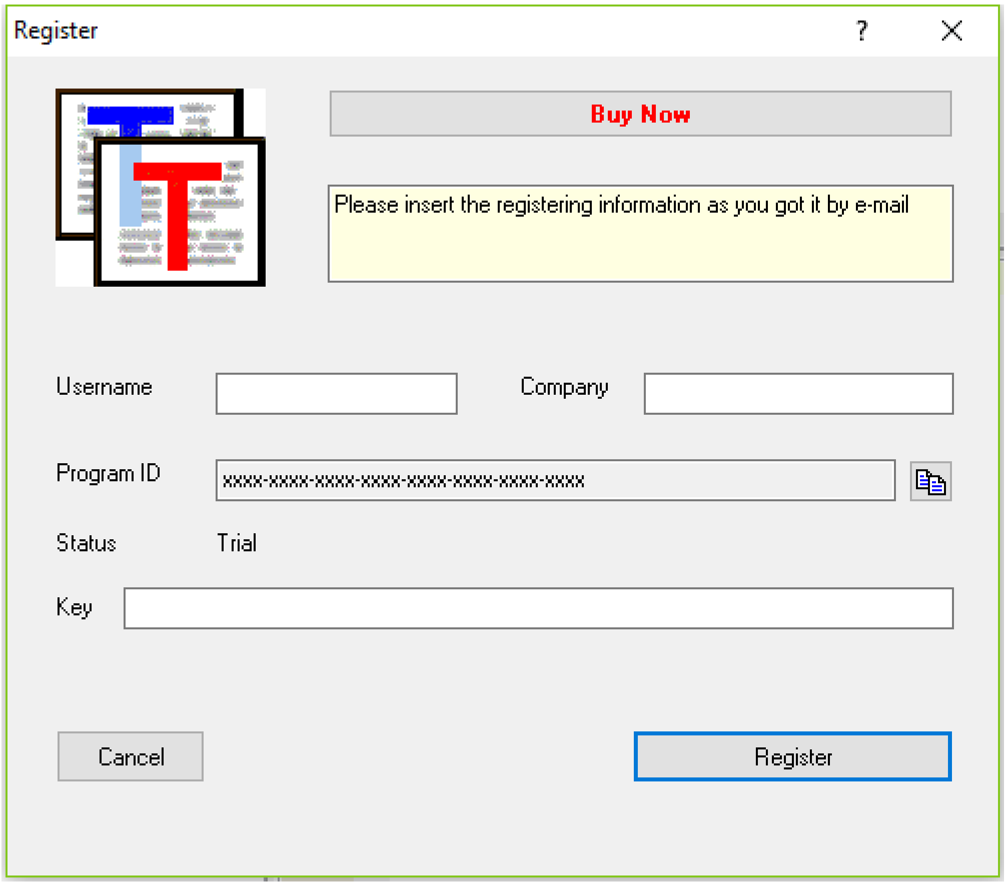
C:\Users\User\Documents\Delphi2Cpp

Regardless of the path, that you chose for the installation, the license file *Delphi2CppLic.dat* will be written at that default path.

3 Registration

If you have bought a license of Delphi2Cpp, **you will get a link to a version of Delphi2Cpp, which you can register.**

The **registration** of Delphi2Cpp, i.e. the permanent activation of the features, has to be done by the menu: Help->*Registration*. Following dialog:



Please insert the registering information as you got it by e-mail

Program ID	xxxx-xxxx-xxxx-xxxx-xxxx-xxxx-xxxx-xxxx
Status	Trial
Key	

Before you can get a key for the registration, you will have to open this dialog in the unregistered program. There you will see the **program ID**,which is shown in the dialog instead of "xxxx-xxxx-xxxx-xxxx-xxxx-xxxx-xxxx-xxxx". The program ID is specific for your hardware configuration. It's also called the *machines fingerprint*. This ID together with the **user name** (at least eight characters) and a **company name** is needed to create the key for the registration. The program ID is copied into the clipboard if you click the button at the right. You either will have to enter these three values into an online form or send them to the manufacturer.

When you have payed for the program, you will get the key via mail. **User name, Company** and the **key** then have to be copied unchanged from the e-mail into the corresponding fields of the dialog.After a click on the ***Register*** button , the program will be closed and restarted automatically.. A license file *Delphi2CppLic.dat* is created now in the user documents DelphiXE2Cpp11 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 . There are two main decisions to make.

1. C++ Builder or other compiler

The translation result depends on the C++ compiler you use. The main difference is between the C++Builder and all other compilers. C++-Builder has it's own C++ version of the Delphi RTL/VCL and Delphi2Cpp tries to optimize the translated code to work together with these libraries. So, depending on the used compiler the desired string type also has to be be chosen. C++ Builder has classes for *AnsiStrings* and *WideStrings*, which are very similar to the original Delphi types. For other compilers it is recommended to use *std::string* and *std::wstring* instead, if you don't want to write your own Delphi like string classes.

2. Choosing the correct source for the RTL/VCL:

Delphi2Cpp 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.
But all Delphi code implicitly also includes the *System* unit and most Delphi code uses at least the *Sysutils* unit too. Already translated C++ code for these both units is part of the Delphi2Cpp installation. In the same folder there are pas-files with the Delphi interface parts of these units. If no other units from the Delphi RTL/VCL are used in your code, you will get the best translation results, if you select the path to these pas-files as search path for the files not to convert.
Mostly your code will depend on more units of the Delphi RTL/VCL. If you are using Delphi2Cpp 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.

If you make use of the original Delphi RTL/VCL, you should use also an "extended System.pas". This file corrects and completes the original "System.pas".

3. Setting the correct definitions

If you have selected the search paths to the Delphi RTL/VCL, your code still might 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 cannot 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;
```

4. Preparation of the RTL/VCL code

It might be necessary to define some substitutions of ampersand-expressions and unfortunately the

RTL/VCL code has flaws, which have to be corrected, if it has to be used.

5. Creating a dummy application or make a complete translation

Normally a complete translation will be made. But if your code is incomplete, you might chose to create a dummy application at first.

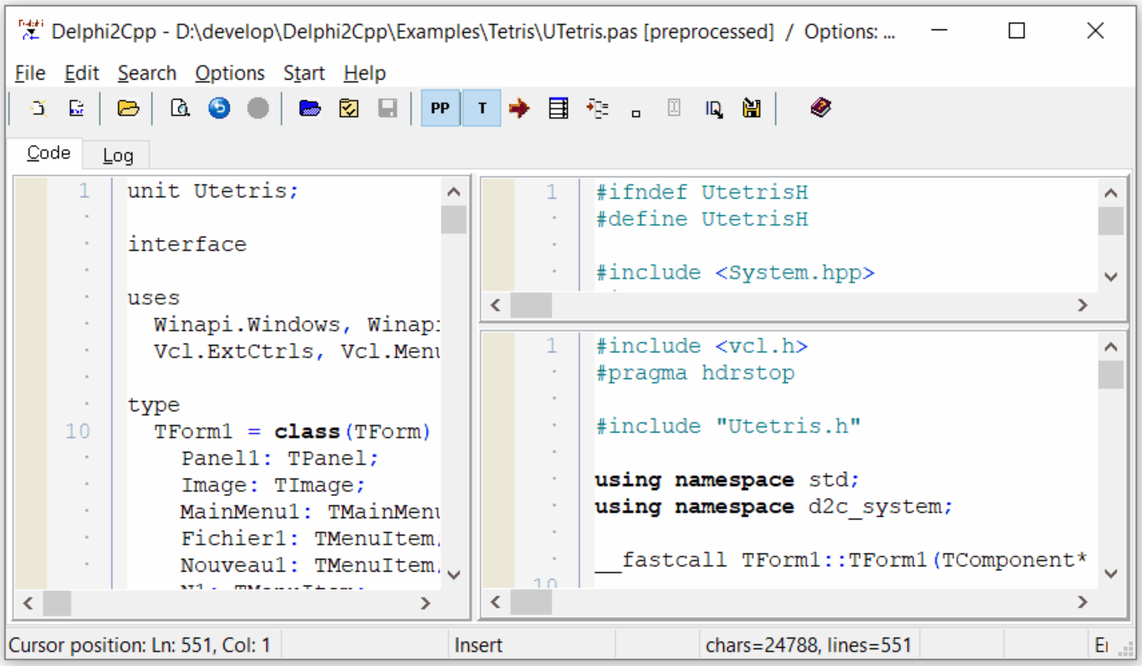
6. Starting the translation

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++ header and the C++ source code are 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

There are three 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 upper window on the right side shows the generated C++ header code
3. the lower window on the right side shows the generated C++ source code



Delphi2Cpp - D:\develop\Delphi2Cpp\Examples\Tetris\UTetris.pas [preprocessed] / Options: ...

```
unit Utetris;

interface

uses
  Winapi.Windows, Winap
  Vcl.ExtCtrls, Vcl.Menu

type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Image: TImage;
    MainMenu1: TMainMenu
    Fichier1: TMenuItem.
    Nouveau1: TMenuItem.
```

```
#ifndef UtetrisH
#define UtetrisH

#include <System.hpp>
```

```
#include <vcl.h>
#pragma hdrstop

#include "Utetris.h"

using namespace std;
using namespace d2c_system;

__fastcall TForm1::TForm1(TComponent*
```

Cursor position: Ln: 551, Col: 1 | Insert | chars=24788, lines=551 | Ei

The most important actions can be started from the toolbar
The log panel shows which files were converted with or without errors.

5.1 Toolbar

The main window of the Delphi2Cpp 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.

--

Switch between unit and form.

--

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:

--

The next button is used to start a recursive translation:

--

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:

--

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

--

When the pre-processor found new identifiers, their notations can be saved via the tool button:

The identifiers are written into a text file, which can be included then into the project options.

--

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.

--

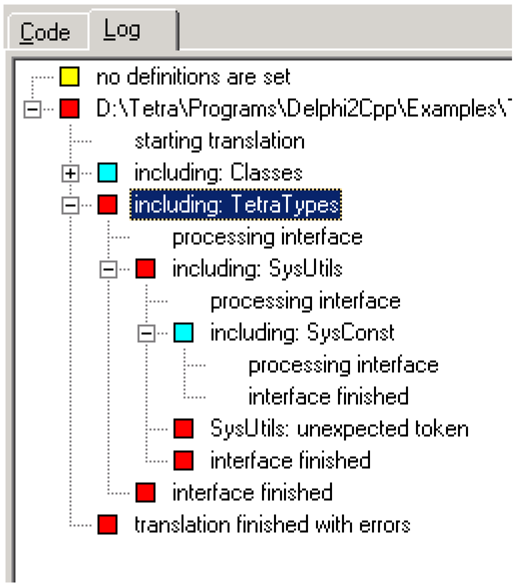
Shows a dialog to find expressions in the text of the actual window.

--

This help is shown with F1 or by the button

5.2 Log panel

The Log panel displays logging messages and errors.



```
🟨 no definitions are set
🟥 D:\Tetra\Programs\Delphi2Cpp\Examples\
     starting translation
   🟦 including: Classes
   🟥 including: TetraTypes
        processing interface
      🟥 including: SysUtils
           processing interface
         🟦 including: SysConst
              processing interface
              interface finished
         🟥 SysUtils: unexpected token
         🟥 interface finished
      🟥 interface finished
   🟥 translation finished with 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
🟪	missing file
🟧	subsequent error (due to previous error or missing file)

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;
```

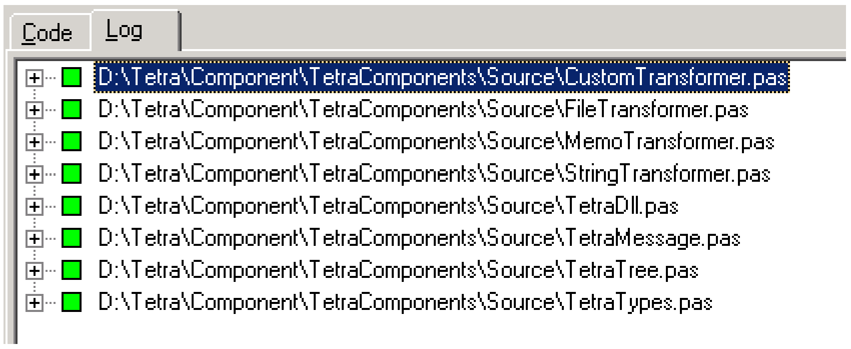
This is a wrong result of the preprocessor. You can reload the original *SysUtils.pas* and find the position of *TTextLineBreakStyle*:

```
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:



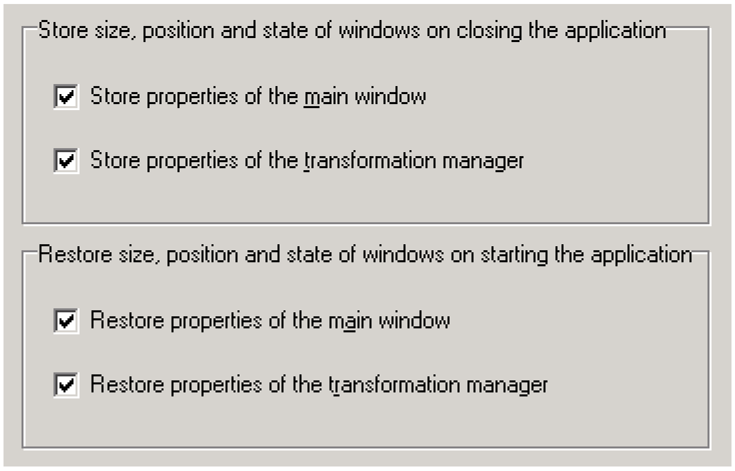
```
D:\Tetra\Component\TetraComponents\Source\CustomTransformer.pas
D:\Tetra\Component\TetraComponents\Source\FileTransformer.pas
D:\Tetra\Component\TetraComponents\Source\MemoTransformer.pas
D:\Tetra\Component\TetraComponents\Source\StringTransformer.pas
D:\Tetra\Component\TetraComponents\Source\TetraDll.pas
D:\Tetra\Component\TetraComponents\Source\TetraMessage.pas
D:\Tetra\Component\TetraComponents\Source\TetraTree.pas
D:\Tetra\Component\TetraComponents\Source\TetraTypes.pas
```

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 Delphi2Cpp.

Window positions
Customization

5.3.1 Window positions

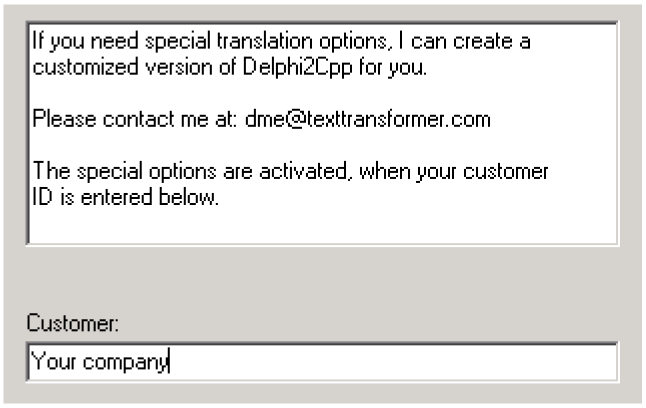


Store size, position and state of windows on closing the application
- [✓] Store properties of the main window
- [✓] Store properties of the transformation manager

Restore size, position and state of windows on starting the application
- [✓] Restore properties of the main window
- [✓] Restore properties of the transformation manager

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 than to deactivate a new storage. So the windows will at a new start of Delphi2Cpp always have the properties that were stored, even if they were change in the previous session.

5.3.2 Customization



If you need special translation options, I can create a customized version of Delphi2Cpp for you.

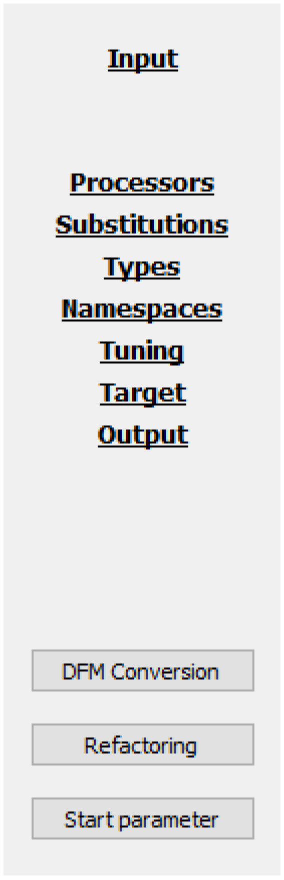
Please contact me at: dme@texttransformer.com

The special options are activated, when your customer ID is entered below.

Customer: Your company

5.4 Translation options

The options dialog can be opened by the blue buttons in the toolbar or via the options menu. there are eight groups of options and three buttons to open additional special option dialogs:



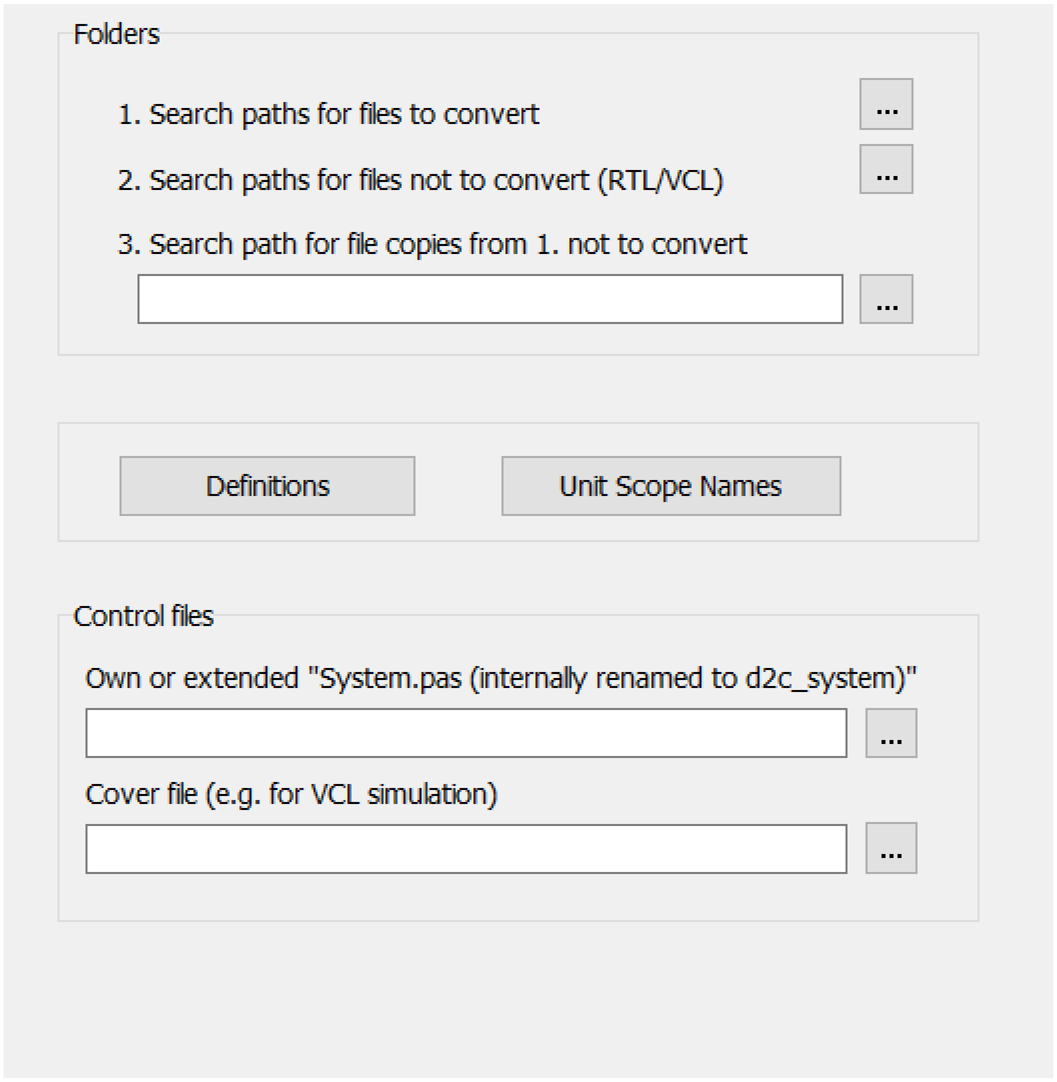
Input options
Processor options
Substitution options
Type options
Namespace options
Tuning options
Target options
Output options

DFM Conversion
Refactoring
Start parameter

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. They specify all contents which either shall be translated or which are required for a translation.


Folders

1. Search paths for files to convert

2. Search paths for files not to convert (RTL/VCL)

3. Search path for file copies from 1. not to convert

Control files

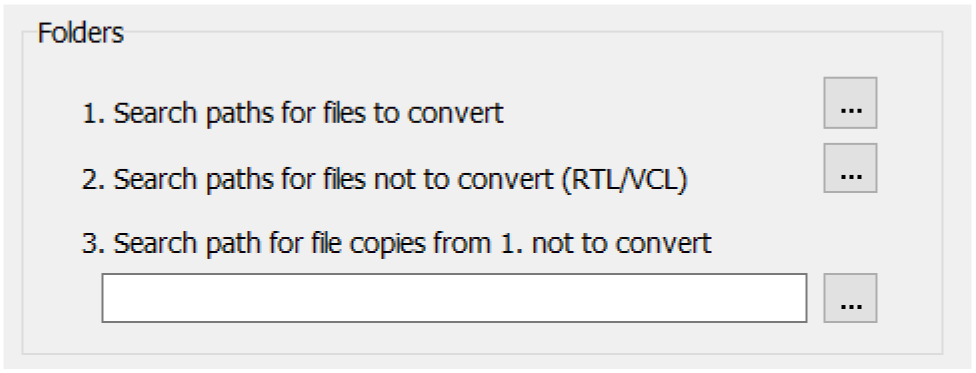
Own or extended "System.pas (internally renamed to d2c_system)"

Cover file (e.g. for VCL simulation)

Search paths
Definitions
Unit Scope Names
Own or extended system.pas
RTL/VCL cover file

5.4.1.1 Search paths

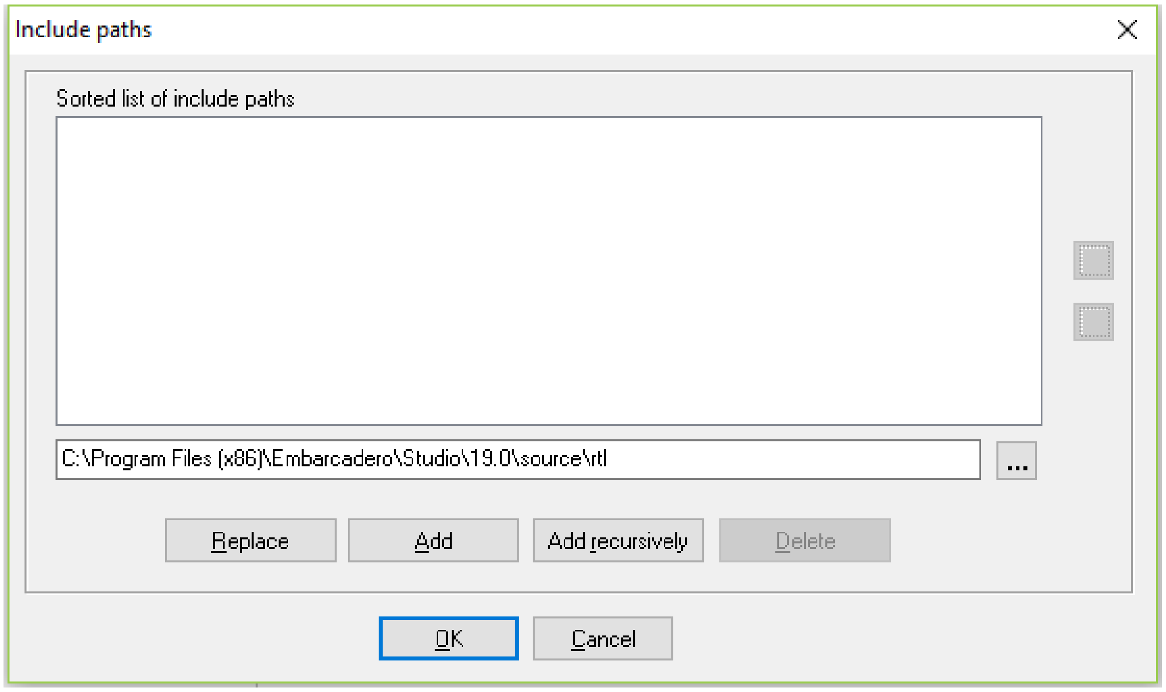
For a correct translation of a Delphi source file the type information of used constants, variables, functions etc.is necessary. If this information is not contained in the actual file, the other used files have to be scanned. As far as these files are in the folder of the source file, they will be found 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.


Folders
1. Search paths for files to convert
...
2. Search paths for files not to convert (RTL/VCL)
...
3. Search path for file copies from 1. not to convert
...

These directories are separated into

1. the folders of files, which really shall be translated.
2. the folders of files for which only the interfaces are needed
3. an optional folder with file copies from 1., which shall not be translated
4. there also are some special case like the system unit

The folders for 1. and 2. are to be set in a dialog like the one below:



C:\Program Files (x86)\Embarcadero\Studio\19.0\source\rtl

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 it's 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 items in the list.

5.4.1.1.1 Paths to the source files

The paths to the folders of the files, which shall be translated or might be translated in the case of a recursive translation, can be set by a second dialog, analogously the paths to the RTL/VCL.

5.4.1.1.2 Paths to the VCL\RTL

If you use C++ Builder, there is already a converted version of the RTL/VCL. So you don't have to translate the according files. Nevertheless the translator has to know the interface parts of the original Delphi RTL/VCL to make a correct translation of the files, which depend on these libraries. So you have to set the folders of the original or - better - of the preprocessed RTL/VCL.These path are set as part of the input options.

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 RTL/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
```

If C++Builder is the target compiler, the files from these folders are included as hpp-Files. E.g.

```
#include <System.Classes.hpp>
#include "SynEdit.hpp"
```

In this case the installed SynEdit components are used. If SynEdit.pas would be translated itself, it's path would have to be set in the paths of the source files and the header would be included as:

```
#include "SynEdit.h"
```

For other compilers as C++Builder only the .h-extension is used.

5.4.1.1.3 Files, not to translate

In rare cases it may be desirable to exclude individual files from the translation that are located in the folders with files to be translated. These files may be copied into a common folder, whose location can be set here. Non-translatable form files would be a possible examples of such files to be excluded. Another example would be files for which there is already a manual translation.

5.4.1.1.4 Special headers

Delphi2Cpp tries to parse ***System.pas*** always in addition to the other included files. *System.pas* contains the declaration of *TObject* and many other frequently used functions, procedures, records and classes.
If *System.pas* cannot be found in the specified serarch paths, a part of the content of this file is simulated.
You also can include your *own extended System.pas*.

The following concerns translation of old Delphi code only,

In old versions of Turbo Pascal / Delphi the units ***WinProcs*** and ***WinTypes*** were used. In Delphi, these two units were merged into the single unit *Windows*. If these files are not found Delphi2Cpp substitutes *WinProcs* and *WinTypes* by *Windows*, so that "# include <Windows.hpp>" will appear in the translated code. In addition, this file is interpreted a little differently in a C-like manner than the other pas files: structures are passed here as parameter to a function by the address of the structure and not as reference as in the other files.

`foo(&StructureType)` instead of `foo(StructureType)`

The unit ***BDE*** is used in database units, but there is no *BDE.pas*. The Delphi compiler doesn't need this file because there is a *BDE.dcu* . The interface is declared in the file *BDE.int* instead.*Delphi2Cpp* also will look for *BDE.int* in the paths to the VCL/RTL The folder for this file has to be set there, e.g. C++Builder6/Doc.

The file ***dsgnintf.pas*** is called *designintf.pas* in the C++Builder VCL.

The namespace ***Windows*** is omitted at the translation since the corresponding functions mostly don't exist there in the C++Builder counterpart. (Also "System." in front of the Move function is left out.)

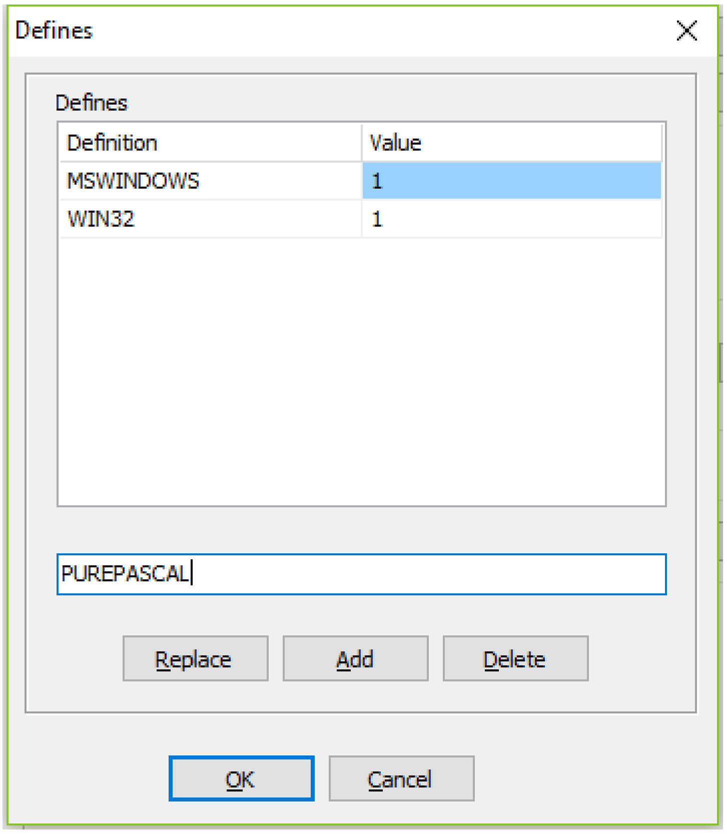
The file ***ShellApi.pas*** is treated in the same C-like manner as *Window.pas*.

Files like **Windows.pas** and ***ShellApi.pas*** are translations of the Windows files **Windows.h** and **ShellApi.h** to Delphi. They should not be translated back to C++; the original files should be used instead.

If you have difficulties with your VCL, please contact the author.

5.4.1.2 Definitions

Delphi code often contains directives for conditional compilation of parts of the source text. Delphi2Cpp evaluates such directives too. You can set the definitions in the option dialog



Defines

Definition	Value
MSWINDOWS	1
WIN32	1

PUREPASCAL

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

5.4.1.2.1 Windows.pas

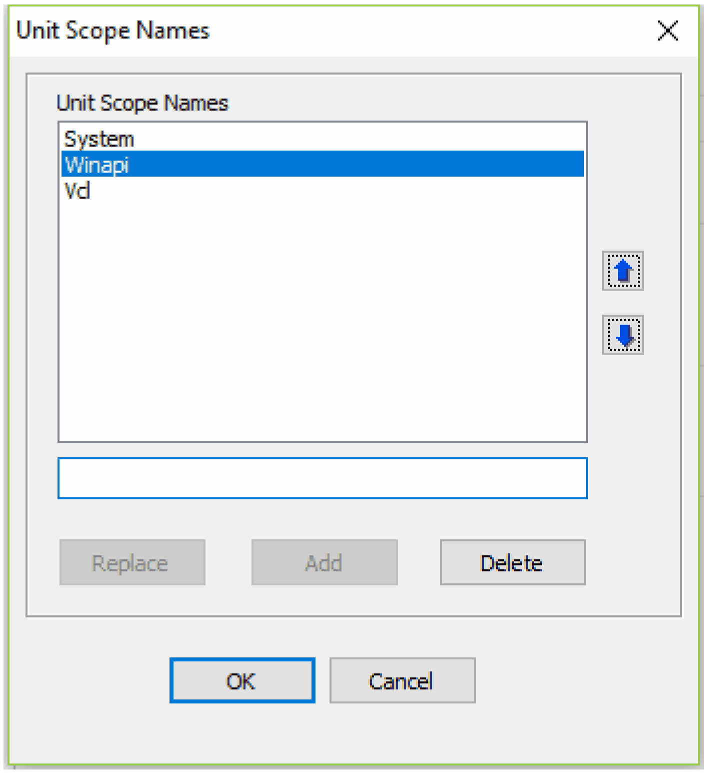
If there is no Definition set either of CPUX86 or of Win64 the Windows.pas cannot be parsed. That's because of the following code:

```
function InterlockedBitTestAndComplement(Base: PInteger; Offset: Integer): ByteBool;
{$IFDEF CPUX86}
...
{$ENDIF CPUX86}
{$IFDEF Win64}
...
{$ENDIF CPUX64}
```

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.

5.4.1.3 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

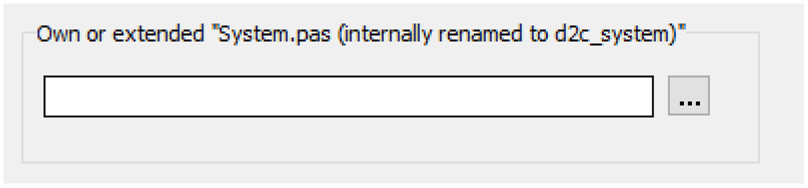
```
uses System.Classes;
```

So, if *System* is in the list of unit scope names, Delphi2Cpp nevertheless will lookup the file *System.Classes.pas*.

5.4.1.4 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 each unit too. These intrinsic function are built into the Delphi compiler. *Delphi2Cpp* 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 a 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.


Own or extended "System.pas (internally renamed to d2c_system)"
...

Such an individual *System.pas* called *d2c_system.pas* is in the *Source* folder of the *Delphi2Cpp* 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. Such a case is the incrementation of values of enumerated types. Of course, these definitions are only needed, if such cases really appear in the source code.

Some examples are explained in the following topics:

procedure *SetString*
Memory management
procedures Inc and Dec

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

Delphi2Cpp looks up system types and functions etc. in following order::

1. *Delphi2Cpp* will look for declarations at first in your own *System.pas*, if it exists.
2. If the declaration is not found there, *Delphi2Cpp* 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, *Delphi2Cpp* simulates the most important parts of this file.

Mostly Delphi2Cpp cannot distinguish different elements with the same name. Delphi2Cpp takes just the first declaration it finds. If there are several functions with the same name the translator tries to match the declaration found first.

5.4.1.4.1 SetString

SetString doesn't exist in the C++Builder VCL. If this function is used in the translated code, an implementation of one's own is required. According to the Delphi help the declaration is:

```
procedure SetString(var s: string; buffer: PChar; len: Integer);
```

Also according to the Delphi help this declaration should be found in the *System.pas*. But only the following exists there:

```
procedure _SetString(s: PShortString; buffer: PChar; len: Byte);
```

Delphi2Cpp uses such declarations - by removing the underscore - if nothing else is found. Indeed, just for the *SetString* function. *Delphi2Cpp* corrects this declaration internally. But with the definition in d2c_system.pas, you don't need to write your own C++ implementation.

In *d2c_system.pas* there are three declarations of *SetString*.

```
procedure SetString(var S: AnsiString; Buffer: PChar; Len: Integer); overload;
procedure SetString(var S: WideString; Buffer: PWideChar; Len: Integer); overload;
procedure SetString(var S: ShortString; Buffer: PChar; Len: Integer); overload;
```

When the Delphi2Cpp translator finds a call of *SetString*, it cannot distinguish between these declarations and will take just the first one it finds. That doesn't matter, because all three declarations have at first a variable string parameter, then a character pointer and then an integer parameter. This vague signature is all, that Delphi2Cpp needs. But later the C++ compiler can chose the right alternative for the according string type.

The implementations of the procedures for *AnsiStrings* and *WideStrings* are quite trivial More interesting is the implementation for *ShortStrings*:

```
procedure SetString(var S: AnsiString; Buffer: PChar; Len: Integer);
begin
  (*_
  S[0] = Len;
  if ( Buffer != NULL )
      memmove( &S[1], Buffer, Len );  _*)
end;
```

The translation with Delphi2Cpp results in:

```
void __fastcall SetString( AnsiString& S, char* Buffer, int Len )
{
  S[0] = Len;
  if ( Buffer != NULL )
      memmove( &S[1], Buffer, Len );
}
```

5.4.1.4.2 Memory management

The function for the memory management *GetMem*, *ReallocMem* and *FreeMem* are defined in *d2c_system.pas*.

```
procedure GetMem(var P: Pointer; Size: Integer);
procedure FreeMem(var P: Pointer; Size: Integer = -1);
procedure ReallocMem(var P: Pointer; Size: Integer);
```

These functions are defined there by use of the C functions *malloc*, *realloc* and *free*.
It is often warned against mixing *malloc* and *new*. (Delphi2Cpp translates the construction of VCL classes with *new*.) But there is no danger, if both are used coherently, i.e. that memory that was allocated with *new* is freed with *delete* and memory that was allocated with *malloc*. is freed with *free.* Memory that was allocated with *malloc* can be *reallocated*, but a reallocation of memory that was allocated with *new* is not possible. That's why it sometimes may be difficult to abstain from using *malloc*.

As already explained for the procedure *SetString*, the translator needs the Delphi declarations to adapt parameters accordingly. For the memory managing procedures there are additional implementations inserted in the C++ code, which are made as templates. E.g.:

```
template <class T>
void GetMem(T*& P, int Size)
{
  P = ( T* ) malloc(Size);
}
```

The advantage is, that there will be no problems with type casts.

BTW: the original *System.pas* contains only the functions:

```
function _FreeMem(P: Pointer): Integer;
function _GetMem(Size: Integer): Pointer;
function _ReallocMem(var P: Pointer; NewSize: Integer): Pointer;
```

5.4.1.4.3 Inc and Dec

As for the procedures for memory management there are template functions for *Inc* and *Dec, e.g.:*

```
template <class T>
T Inc(T& xT)
{
  int t = (int) xT;
  t++;
  xT = (T) t;
  return xT;
}
```

For integer types *Inc and Dec* are converted automatically to the C++ incrementing and decrementing operators. E.g.

```
Inc( i )  -> i++
```

However in cases, where *i* is an enumerated type the operators cannot be used in C++. So the translator lets a call like *Inc(i)* unchanged and the template function are called in C++. By the temporary conversions of the enumerates types to integers the *Inc* and *Dec* functions will work for enumerated types too.

5.4.1.5 RTL/VCL cover file

The Visual Component Library (VCL) is a Delphi library for an easy development of Windows user interfaces (GUI). Thsi library exists in Delphi only. A translation to C++ might be possible, but would be a very big task even with Delphi2C++, because the code of the VCL is a link to the Windows API and Delphi2Cpp would have to know the exact specifications of this API to make a correct translations. Much parts of the Many parts of the RTL are also hardly translatable.

The use of an RTL/VCL cover file allows to simulate code parts that are difficult to translate and then to substitute them in C++. A very simple would be the following:

```
unit VclCover;

interface

//uses ...;

type

TCustomControl = class
public
  Width: Integer;
  Height: Integer;
  Left: Integer;
  Top: Integer;
end;

end.
```

If then the following code would have to be translated:

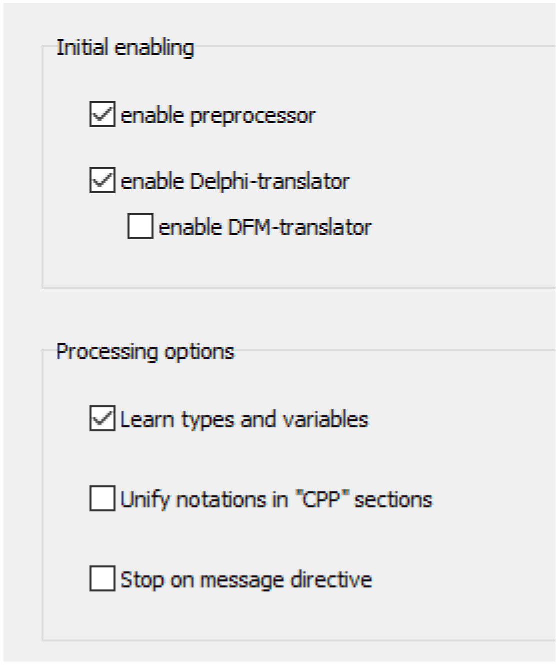
```
var
  Control : TCustomControl;
begin

  Control.With := ...
```

Delphi2C# would know by use of the cover file that on the right side of the assignment an integer is expected. All symbols, that Delphi2C# tries to look up in files that are used for the translation, but are not to be translated themselves, are tried to be looked up in the cover file first. If the symbol is found there a further lookup isn't made.

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++.



Initial enabling

- [✓] enable preprocessor
- [✓] enable Delphi-translator
  - [ ] enable DFM-translator

Processing options

- [✓] Learn types and variables
- [ ] Unify notations in "CPP" sections
- [ ] Stop on message directive

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 Delphi-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. When the Delphi-translator is enabled, also the DFM-translator can be enabled or disabled. (If the Delphi-translator is disabled the check-box for the dfm-translator vanishes, because the dfm files cannot be processed then.)

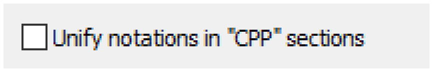
The *overwritten System.pas* gets always preprocessed, even if the option to do so is disabled.

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 "CPP" sections determines the case sensitivity in "CPP"-sections.

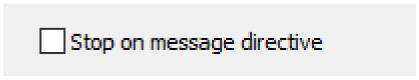
The option Stop on message directive determine what happens, if a message directive would remain in the pre-processed code.

5.4.2.1 Unification of CPP-sections



This option is part of the processor options. It determines how identifiers in "CPP"-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 om 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'}
```

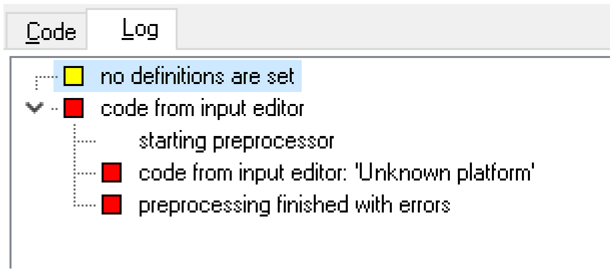
If correct definitions are set, such messages normally will be part of code sections for which the conditions are false.The option to stop on message directives therefore will not apply. But e.g. the recommended *PUREPASCAL* definition is problematic. If it is defined.the definition of *ASSEMBLER* should 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, Delphi2Cpp 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. Delphi2Cpp stops and the message is shown on the log-panel:



🟨 no definitions are set
🟥 code from input editor
starting preprocessor
🟥 code from input editor: 'Unknown platform'
🟥 preprocessing finished with errors

On the other side, there are messages which you might want to ignore. In the following case Delphi2Cpp isn't able to calculate the correct result of the condition:

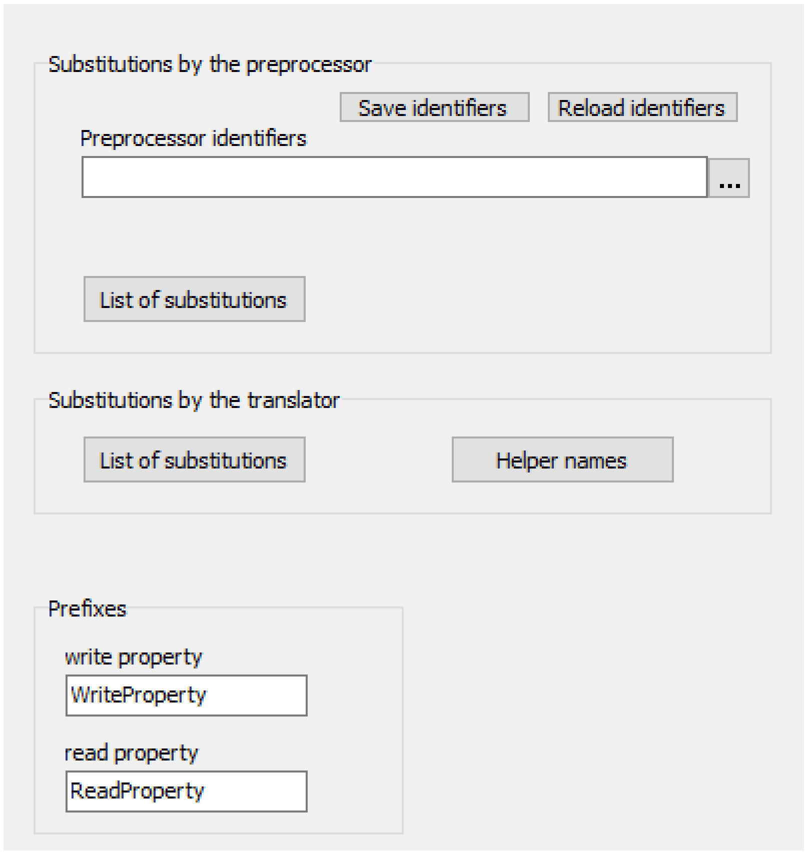
```
{$IF SizeOf(Extended) <> SizeOf(TExtended80Rec)}
  {$MESSAGE ERROR 'TExtended80Rec has incorrect size'}
{$ENDIF }
```

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.



write property: WriteProperty

read property: ReadProperty

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

- The notation of identifiers are unified according to a list of given notations
- 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

- the substitutions by the translator are executed after the code already has been parsed.
- also some kinds of refactoring can be done now.

Substitutions of helper names are useful for C++Builder users to synchronize numbered helper names for enumeration types in the C++ code of C++Builder on one side and of Delphi2Cpp on the other side. For example *System::Sysutils.hpp* contains the following definition:

```
    enum DECLSPEC_DENUM System_Sysutils__85 : unsigned char { rfReplaceAll, rfIgnoreCase };
```

but when *Delphi2Cpp* parses System.*SysUtils.pas, it generates*

```
    enum SysutilsEnum__0 {rfReplaceAll,  rfIgnoreCase };
```

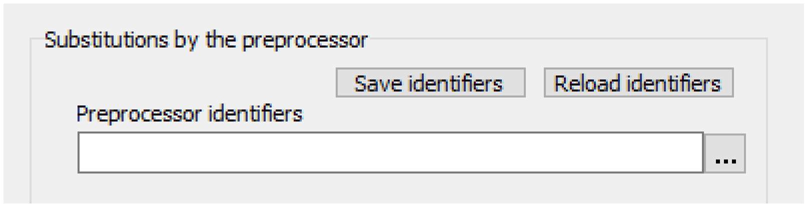
With the option to substitute helper names, *SysutilsEnum__0* automatically can be substituted by *System_Sysutils__85*.
.

If you create C++ code for another compiler than C++Builder all properties are replaced by pairs of functions. You can change the prefixes for the 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: On the page for the substitution options 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.

Your 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 identifiers 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.

Some directives may have an impact on the requires notation.

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

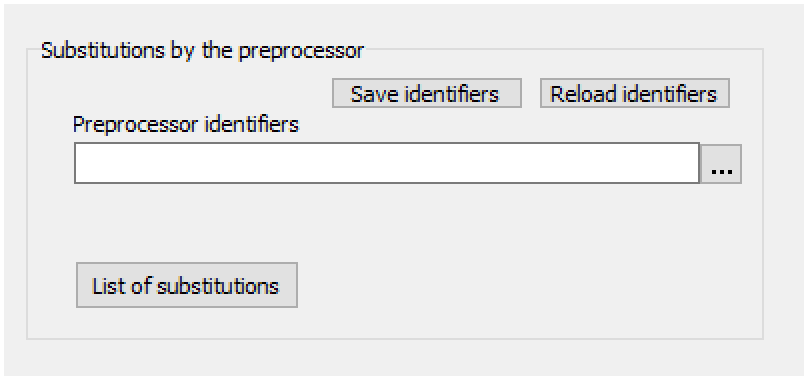
implicit
explicit
negative
positive
inc
dec
logicalnot
trunc
round
in
equal
notequal
greaterthan
greaterthanorequal
lessthan
lessthanorequal
add
subtract
multiply
divide
intdivide
modulus
logicalor
bitwiseor
logicalxor
bitwisexor
logicaland
bitwiseand
leftshift
rightshift

MinComp
MaxComp
NaN
Infinity
NegInfinity

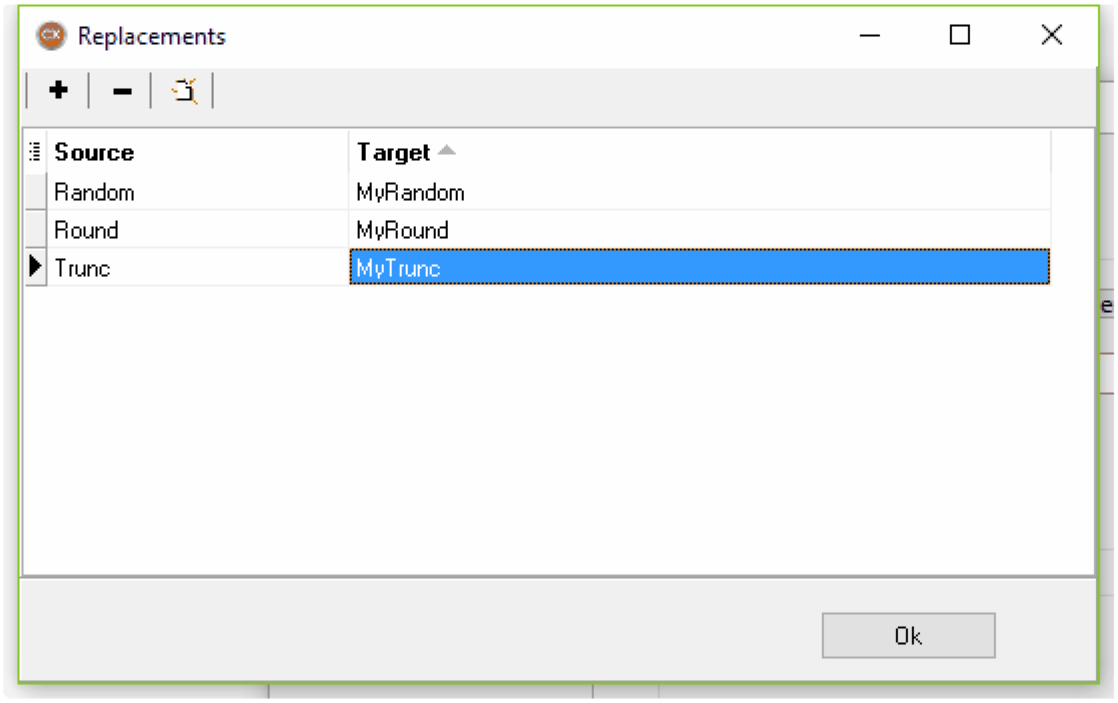
Sum
SLICE
Winapi // minwindef.h: #define WINAPI   __stdcall

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.



Source	Target
Random	MyRandom
Round	MyRound
Trunc	MyTrunc

✚	**add a new row**
━	**remover 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.

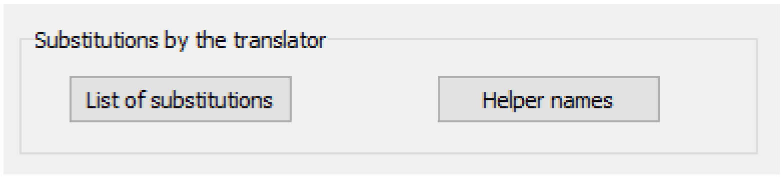
2. completely other names can be assigned to certain identifiers.

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.

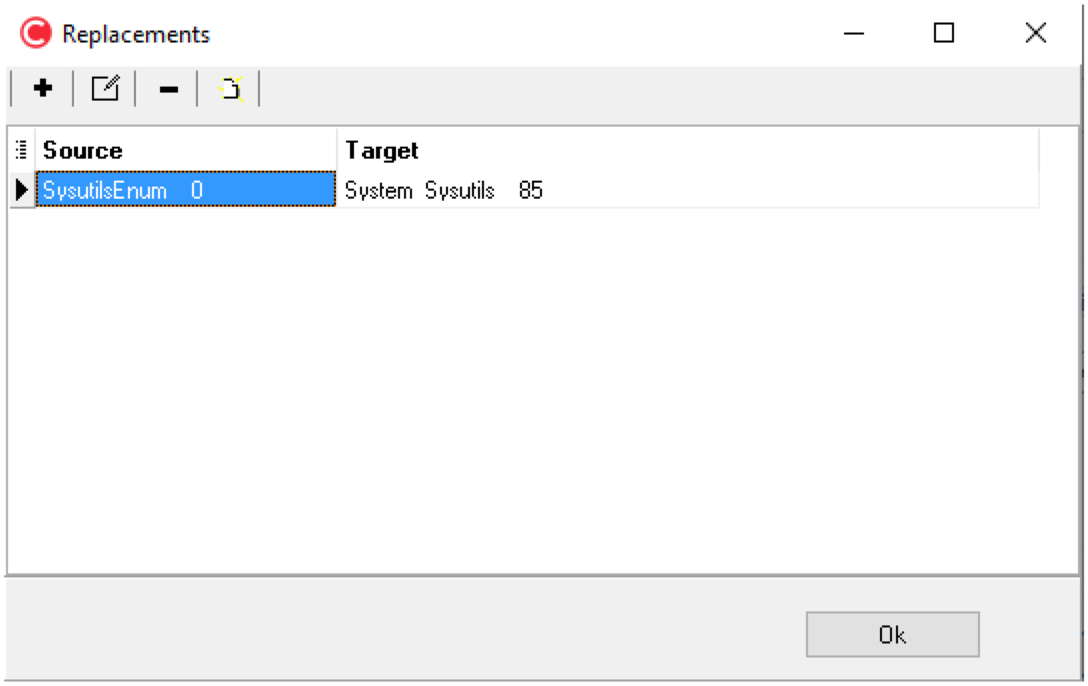
   ```
   StringOfChar -> AnsiString::StringOfChar
   ```

Helper names

There is an additional map for the substitution of helper names which are created for the definition of implicitly defined types, For example in System.SyUutils.hpp the following enum type is defined:

```
enum DECLSPEC_DENUM System_Sysutils__85 : unsigned char { rfReplaceAll, rfIgnoreCase };
```

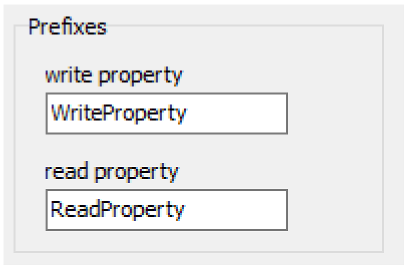
When Delphi2Cpp parses System.SysUtils.pas, initially it cannot know this name and gives another name to this type: *SysutilsEnum__0*.
The mapping between these two names can be defined in the helper substitution dialog as shown below:



Source	Target
SysutilsEnum__0	System_Sysutils__85

5.4.3.4 Prefixes for properties

If you create C++ code for another compiler than C++Builder all properties are replaced by pairs of functions. You can change the prefixes for the function names at the substitution options.

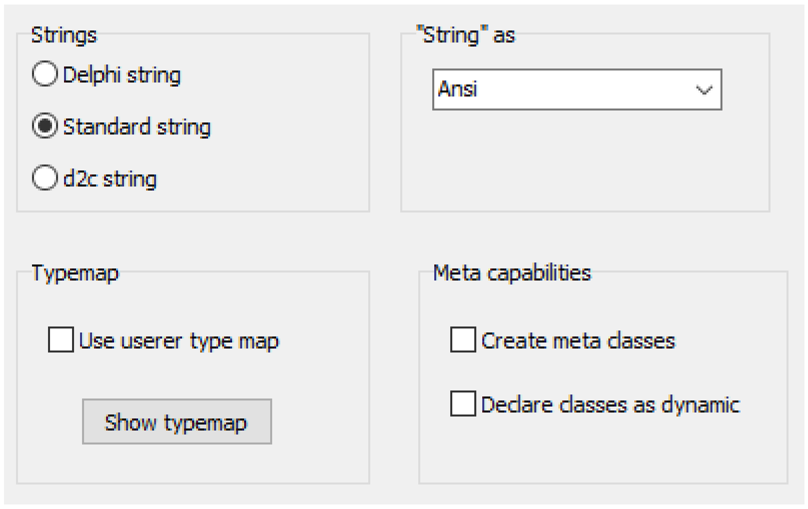


write property: WriteProperty
read property: ReadProperty

if the default prefixes *ReadProperty* and *WriteProperty* are left, then it is very unlikely that there will be conflicts with existing names in the code.
Here the consequences of changing these prefixes are described

5.4.4 Type options

The type options are part of the translation options and specify how Delphi types are converted.



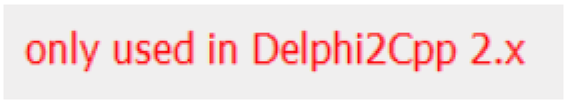
You have to chose how the string types *AnsiString*, *WideString* and *String* are translated.

the insertion of macros to access runtime class information

d2c_config.h !!!

```
const int StringBaseIndex = 0;
```

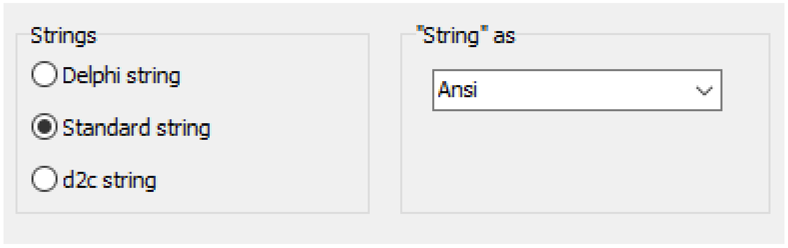
Delphi2CB doesn't support any type options. Therefor only the following hint is shown on this page:



only used in Delphi2Cpp 2.x

5.4.4.1 String types

At the type options you can chose how the string types *AnsiString*, *WideString* and *String* are translated.



Strings: ( ) Delphi string [✓ Standard string] ( ) d2c string

"String" as: Ansi

If ***Delphi string*** is selected, the translated code will use classes for *AnsiString*, *WideString* and *UnicodeString*. In C++Builder these classes are provided. If you chose this option for other compilers, you have to create these classes yourself. They have to be 1 based and have to obey the specifications from Embarcadero:

http://docwiki.embarcadero.com/RADStudio/Tokyo/en/String_Types_(Delphi)

If ***Standard string*** is selected, the following typedef's are needed:

```
typedef std::string  AnsiString
typedef std::wstring WideString
typedef std::wstring UnicodeString
```

Delphi functions for strings will be converted to functions for AnsiString/UnicodeString or std::string/std::wstring. Examples:

```
var
  s1, s2 : String;
begin
  Length(s1);
  SetLength(s1, 10);
  s1 := '12345678';
  s2 := copy(s1, 3, 4);
  Delete(s1, 3, 2);
  Pos(s1, s2);
```

->

```
  Delphi string                        Standard string

  s1.Length();                         s1.size();
  s1.SetLength(10);                    s1.resize(10);
  s1 = L"12345678";                    s1 = L"12345678";
  s2 = s1.SubString(3, 4);             s2 = s1.substr(3-1, 4);
  s1.Delete(3,  2);                    s1.erase(3-1,  2);
  s2.Pos(s1);                          s2.find(s1);
```

d2c string is an experimental own AnsiString/UnicodeString based on std::string/std::wstring

According to the chosen ***"String" as*** option *String* will be treated either as *AnsiString* or as *UnicodeString*.

```
var
S: String:
begin
S := 'hallo';
```

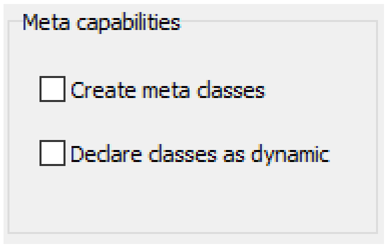
is translated for an ***Ansi*** association to:

```
String S;
S = "hallo";
```

and for the ***Unicode*** association to

```
String S;
S = L"hallo";
```

5.4.4.2 Meta capabilities



Create meta classes

If the option *Create meta classes* is enabled at the type options, Delphi2Cpp creates for each class an 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.

Declare classes as dynamic

This feature from the first version of Delphi2Cpp hasn't been re-implemented in Delphi2Cpp II yet. If you need this feature please contact me.

Alternatively you can use MFC-like macros. In the Microsoft Foundation Classes (MFC) the macros *DECLARE_DYNAMIC* and *IMPLEMENT_DYNAMIC* give access to runtime class information, similar to the runtime information that is provided in the VCL by the accordingly overwritten functions of TObject.

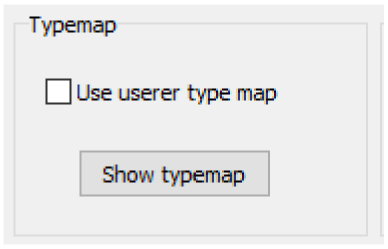
The macros can be renamed by means of the substitution table of the translator. An obvious alternative would be to use the macros "DECLARE_DYNCREATE" and "IMPLEMENT_DYNCREATE" also defined for the MFC in the file "afx.h".

The following table compares the class names and functions of the MFC and Delphi:

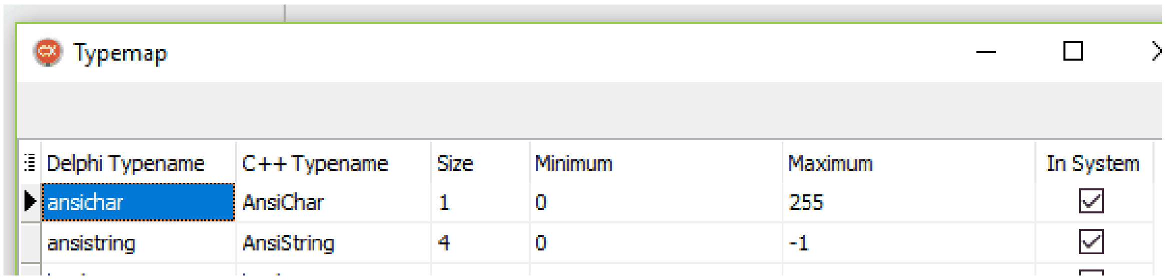
class CObject	class TObject
struct CRuntimeClass	class TMetaClass
CObject::GetRuntimeClass	TObject::ClassType
CRuntimeClass::IsDerivedFrom	TMetaClass::InheritsFrom
CObject::IsKindOf	TObject::InheritsFrom
CRuntimeClass::CreateObject	TMetaClass::Create
CObject::CreateObject	TObject::Create

5.4.4.3 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	✓
ansistring	AnsiString	4	0	-1	✓

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
```

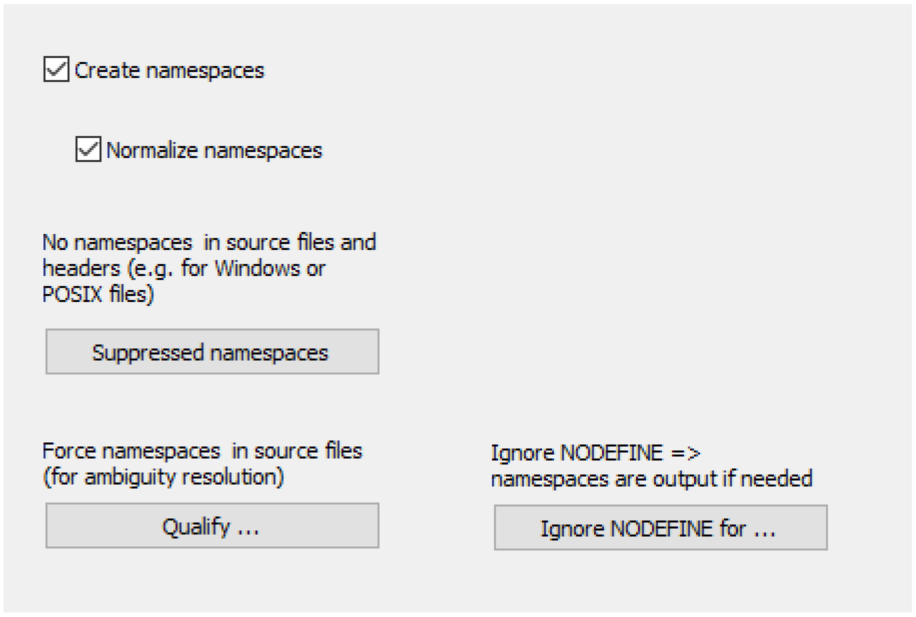
size_t

In addition to the built-in types there is a *size_t* item, though no corresponding type exists in Delphi. The reason is, that sometimes Integer types are converted to *size_t* types in C++ and the properties of *size_t* determine whether some casts are written into the resulting code, which avoid warnings from the C++ compiler.

For C++Builder sometimes simple type identifiers are needed, because no space is allowed inside of a type identifier.

5.4.5 Namespace options

The namespace options are part of the translation options and specify which namespaces have to be created in the resulting code.


[✓] Create namespaces

[✓] Normalize namespaces

No namespaces in source files and headers (e.g. for Windows or POSIX files)

Force namespaces in source files (for ambiguity resolution)

Ignore NODEFINE => namespaces are output if needed

Namespaces are created if the option *Create namespaces* is activated.

The namespaces are normalized if the option Normalize namespaces is activated.

If the button *Suppressed namespaces* is clicked a dialog is shown, where you can enter file names for which the creation of namespaces shall be suppressed.

If the button Qualify ... is clicked a dialog is shown, where you can enter type identifiers, which always shall be qualified.

If the button Ignore NODEFINE for ... is clicked a dialog is shown, where you can enter type identifiers, for which the NODEFINE directive shall be ignored.

For C++Builder no namespaces are created for files where a form is defined and also no namespaces are written for types in published sections, because the form parser of the C++Builder cannot process them.

5.4.5.1 Normalize namespaces

If the option Create namespaces is activated, the option *Normalize namespaces* determines their notation. If the option is activated, the namespaces are written like the C++Builder does, that means the first letter of the identifier is capitalized and the rest of the letters are written in lower case. For example for the file:

```
System.SysUtils
```

the following namespace will be created:

```
namespace System
{

namespace Sysutils
```

```
    {
```

If the normalize option is not activated the notation depends on list of identifiers. If the identifiers are defined there they are output accordingly. Otherwise the notations of the identifiers are determined by their first occurrence in the code or by the file name.

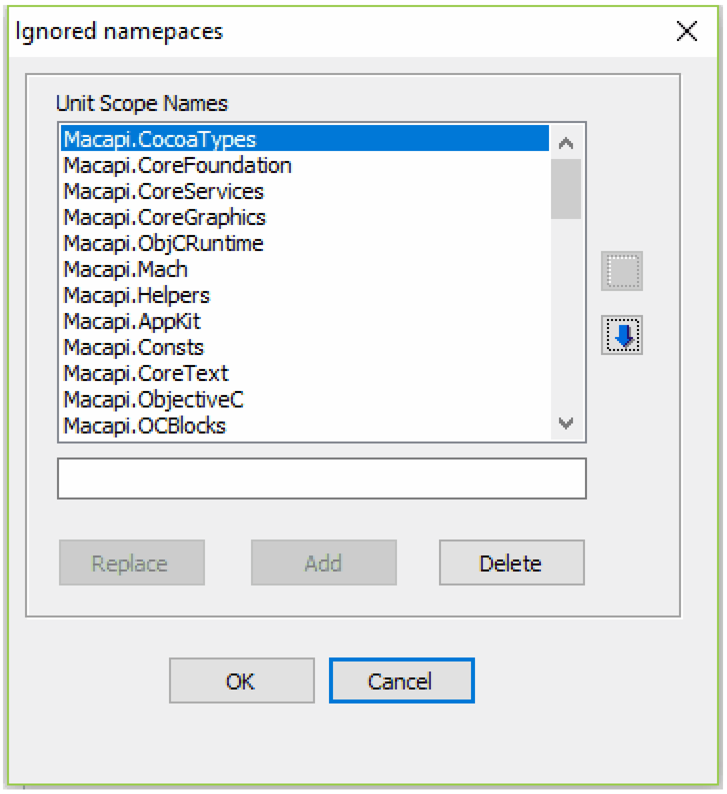
In contrast to the namespace, the file name will not be changed. E.g.

```
    #include "System.SysUtils.h"
```

5.4.5.2 Suppressed namespaces

One of the namespace options is, to suppress special namespaces.

If the button *No namespaces* is clicked a dialog is shown, where you can enter namespaces, that shall be suppressed.



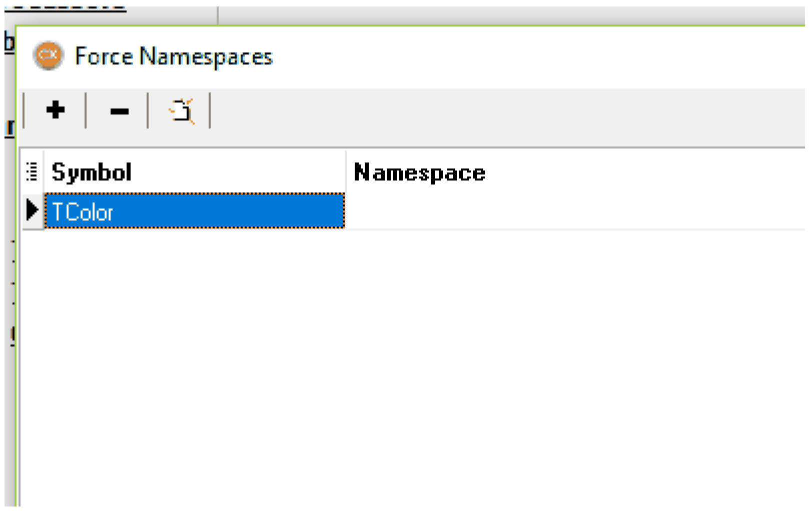
Macapi.CocoaTypes
Macapi.CoreFoundation
Macapi.CoreServices
Macapi.CoreGraphics
Macapi.ObjCRuntime
Macapi.Mach
Macapi.Helpers
Macapi.AppKit
Macapi.Consts
Macapi.CoreText
Macapi.ObjectiveC
Macapi.OCBlocks

It is recommended to suppress the namespace for API files. The BOOL type is a striking example of why this should be done

In the picture the namespaces for files of the OSX-API are suppressed. For Windows it is recommended to suppress the namespaces of the files in rtl/Win, i.e. the namespaces, which start with "Winapi".

5.4.5.3 Forced namespaces

One of the namespace options is, to force the output of namespaces for listed types. This applies also for the source files, where normally the output of the namespaces because of using-clauses is not needed.

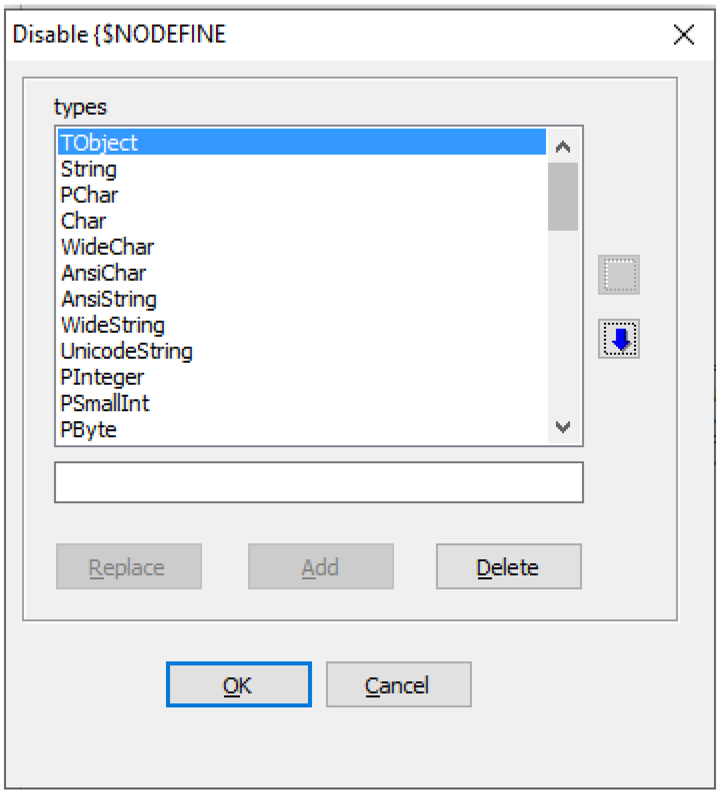


Symbol	Namespace
TColor	

If the cell for the namespace is empty, *Delphi2Cpp* will lookup the namespace, if it has a value, this is set as namespace.

5.4.5.4 Ignore NODEFINE

One of the namespace options is to ignore the NODEFINE directive for listed types.



Type identifiers listed in this box are treated as if there werde no NODEFINE directive. If the tuning option to apply NODEFINE is not disabled all types specified with this directive "disappear" in the file were they are defined. Nevertheless they my be defined in another file (this is often the case for C++Builder). When thes tyes then have to be disambiguited or simple if they are used in headers. they will not be qualified with the namespace of their original unit any more.
E.g. instead of:

```
void foo(const System::String& s);
```

the following line is output:

```
void foo(const String& s);
```

If the NODEFINE directive is ignored, the first declaration, will be written again.

There is a special problem with old version of the RTL. In System.pas for RAD Studio 10.2 Tokyo there are many types defined with wrong NODEFINE specifications. E.g.

```
{NODEFINE    string       'UnicodeString' } {$OBJTYPENAME string   'NUnicodeString'} { defined in ustri
```

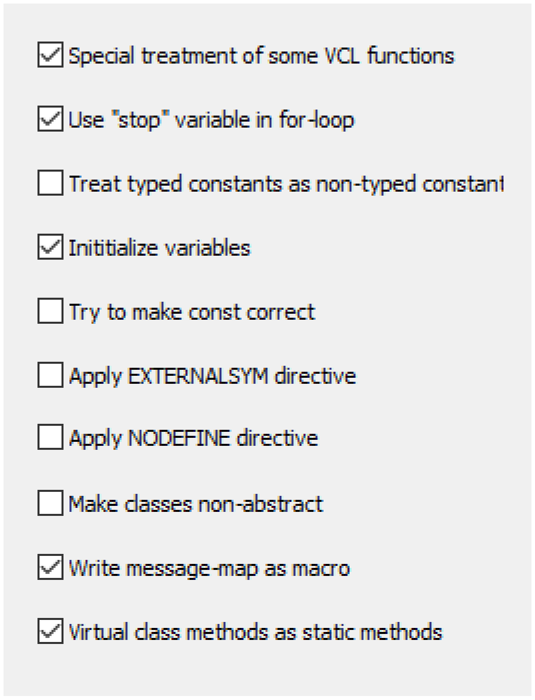
instead of the correct version in RAD Studio 11.1 Alexandia:

```
{$NODEFINE    string       'UnicodeString' } {$OBJTYPENAME string   'NUnicodeString'} { defined in ustr
```

Delphi2Cpp always ignored the wrong specifications in the Tokyo version. Therefore always the namespaces were written.

5.4.6 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
- [ ] Treat typed constants as non-typed constant
- [✓] Inititialize variables
- [ ] Try to make const correct
- [ ] Apply EXTERNALSYM directive
- [ ] Apply NODEFINE directive
- [ ] Make classes non-abstract
- [✓] Write message-map as macro
- [✓] Virtual class methods as static methods

Special treatment of some VCL functions

Use "stop" variable in for-loop

Treat typed constants as non-typed constants

Initialize Variables

Try to make const correct

Apply EXTERNAL directive

Apply NODEFINE directive

Make classes non-abstract

Write message-map as macro
.
Virtual class methods as static methods

.

5.4.6.1 Special treatment of some VCL functions

Some Delphi VCL functions are made to member functions in the C++Builder VCL.. *Delphi2Cpp* 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.

5.4.6.2 Use stop-variable in for-loop

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

5.4.6.3 Treat typed constants as non-typed constants

The tuning option Treat typed constants as non-typed constants concerns typed constants like

```
const
  tc : integer = 7;
```

In Delphi 7 such typed constants were writable like variables. *Delphi2Cpp* imitates this behavior when the option to treat typed constants as non-typed constants is deactivated. The constant then becomes an extern variable in C++. The definition is written into the header:

```
extern int tc;
```

and the implementation is written into the source file:

```
int tc = 7;
```

If option to treat typed constants as non-typed constants is activated, the constants of sinmple types are treated as a non-typed constant. (An example of a non-typed constant is: "const c = 7;".) There is only one line as output:

```
const int tc = 7;
```

In the more current versions of Delphi typed constants are writable only if the {$J+} directive is set.

5.4.6.4 Initialize Variables

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

The initialization of variables is 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 Delphi2Cpp offers the option to initialize all variables explicitly, either to achieve reproducible behave or just to suppress compiler warnings.

5.4.6.5 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, *Delphi2Cpp* 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

5.4.6.6 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++. *Delphi2Cpp* don't writes such symbols into the output. If the option " *Apply EXTERNALSYM directive" is enabled*,

See also

5.4.6.7 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.

You also can ignore the NODEFINE directive for selected types only.

See also

5.4.6.8 Make classes non-abstract

The tuning option *Make classes non-abstract* is used to create a kind of mock function bodies in abstract classes.

Of course, this option should be used temporarily only.

5.4.6.9 Write message-map as macro

The tuning option *Write message-map as macro* allows to pretty-print message maps by means of macros. If you want to step through the code with a debugger macros should be avoided.

5.4.6.10 Create additional 'this' parameter for class methods

The tuning option *Create additional 'this' parameter for class methods* is set to false by default. If it is true in the generated code the parameters of class methods are preceded by an extra parameter, which represents the Delphi *Self* type, as explained here. If Self isn't used by your code and if the code also doesn't use virtual class methods, this options may be unchecked.

5.4.6.11 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();
```

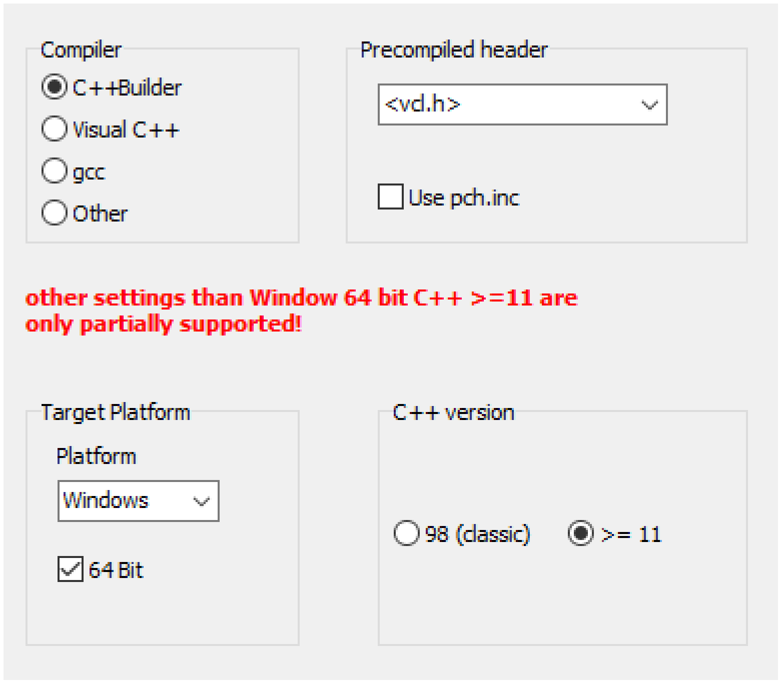
If there are overridden Delphi virtual class methods, the option *Virtual class methods as static methods* has to be disabled. The method then becomes

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

Delphi2Cpp 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.

5.4.7 Target options

The target options are part of the translation options and specify the operation system where the resulting C++ code shall be executed as well as the compiler which shall be used..



Compiler
[✓ C++Builder] Visual C++ gcc Other

Precompiled header
<vcl.h>
☐ Use pch.inc

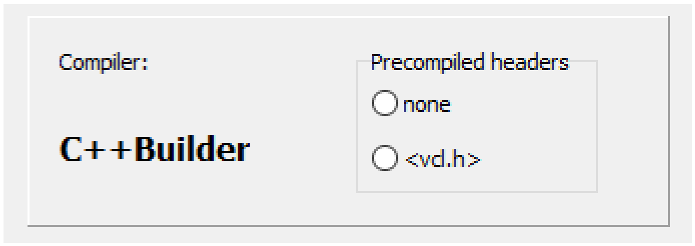
other settings than Window 64 bit C++ >=11 are only partially supported!

Target Platform
Platform: Windows
☑ 64 Bit

C++ version
98 (classic) [✓ >= 11]

Compiler

Delphi2CB only supports the C++Builder. Therefore the top of the page looks like:



Precompiled header

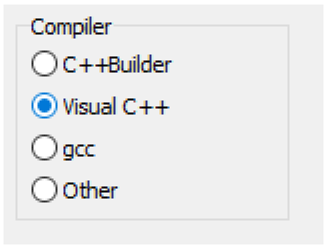
Target platform

C++ version

There is a rough distinction between old C++98 compilers and new compilers for C++11 or later (C+ +14, C++ 17 ...). For C++Builder this distinction corresponds to the distinction between the classic compiler and clang. A finer distinction is made in the auxiliary code.
The selection of the version determines how arrays-of-const are constructed, whether structures are zero-initialized etc.
The support for C++98 isn't as complete as that for C++11. In cases, where there is no solutionn for C ++98 the solution for C++11 is output. For example this is the case for nested functions.

5.4.7.1 Compiler

At the target options you can chose the kind of c++-compiler, for which the output shall be produced.



Compiler: C++Builder [✓ Visual C++] gcc Other

C++Builder

C++Builder is made on top of a Delphi-Compiler and has some C++ extensions to cope with language features of Delphi, which cannot be reproduced adequately with the standard C++.

Visual C++/gcc/Other

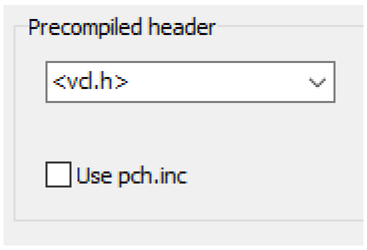
At the moment there is nearly no difference in the options to produce code for *Visual C++*, *gcc* or any

other compiler. Only *threadvars* are treated differently for *gcc*. In future there might be more compiler specific conversions.
If the generated C++ code shall be used with other compilers than the C++Builder, properties are eliminated and the *__fastcall* directives are left out. You can change the prefixes of the names for the functions which are created instead of the properties.

Delphi2CB only supports C++Builder

5.4.7.2 Precompiled header

Some compilers allow header files to be precompiled into a precompiled header, which then hasn't to be recompiled in future compilations. The point up to which the code is precompiled is marked by a specific file or a pragma. At the target options you can chose a marker, which Delphi2Cpp then will insert into the generated code.



There are three options:

1. <vcl.h>

normally used with C++Builder. Delphi2Cpp also appends the line:

```
#pragma hdrstop
```

if this option is chosen.

2. "stdafx.h"

normally used with Visual C++.

3. No marker for a precompiled header at all

for other compilers like gcc.

If the options "Use pch.inc" is activated, no include directives are written into the C++ output, with exception of the header of the actual source file. The user can include the pch.inc file into the file for the precompiled headers or into the *stdafx.h* instead.

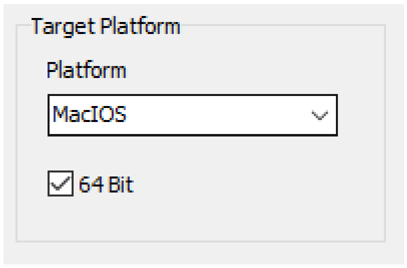
5.4.7.2.1 pch.inc

If the file manager was used, a list of all header files, which were included in the processed files is written into the root folder of the last target files. The file with this list is called "pch.inc" and can be used for inclusion into the "stdafx.h" of Visual C++ or an according file for C++Builder.

There is an option which prevents that include directives are written to into the files, if the "pch.inc" shall be used instead.

5.4.7.3 Target platform

At the target options you can chose the target platform.



Platform: MacIOS

64 Bit [✓]

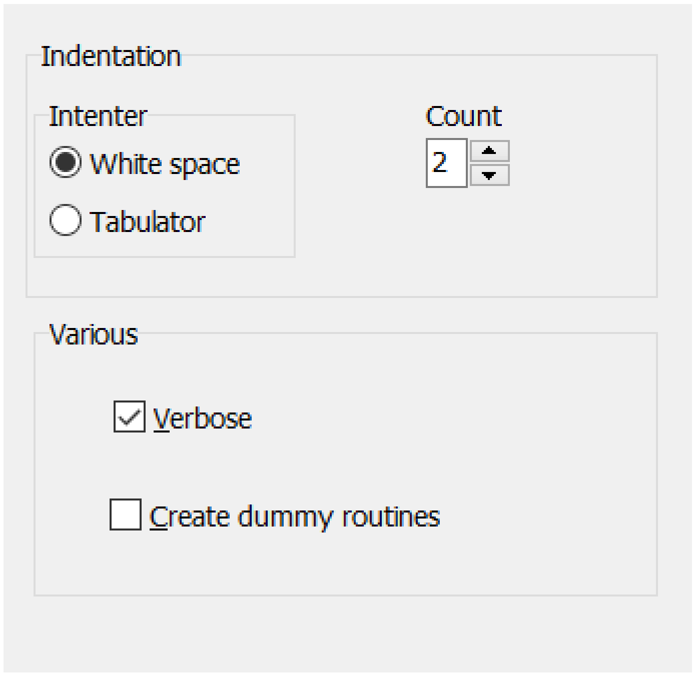
The alternative platforms are: Window, Linux or MacIOS. The selected platform makes no big difference in the generated C++ code, because *Delphi2Cpp* aims to generate portable C++ code. But nevertheless some types are functions might be named differently for different platform. The source code for other compilers contains different conditions for the three platforms.

More important is the *64 Bit* option. Depending on the chosen option some values in the type-map are different.

Delphi2CB only supports Windows as target platform

5.4.8 Output options

The output options are part of the translation options and specify the style of the generated output.


Indentation

Intenter

(●) White space

( ) Tabulator

Count: 2

Various

[✓] Verbose

[ ] Create dummy routines

Indentation

Indentation can be done either by white space characters or by tabulators. The *Count* field controls how much characters are used, when the indentation is increases or decreased.

Verbose option
Create dummy code

5.4.8.1 Verbose

Per default the *Verbose* option is set in the output options. 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*/];
```

5.4.8.2 Create dummy code

A frame of an application only will be created, if the checkbox to create dummy routines is activated in the output options dialog. To create a frame only might be useful if the application uses code that can not or shall not be translated, e.g. if only the working code of a GUI application shall be converted.

Such a frame application can be relatively easily made to compile and link. Once there is a running frame application, it can then be expanded piece by piece into a working application.

In such a frame:

1. function bodies aren't written, default values are returned. Example:

```
String Translate(String AText)
{
  return L"";
}
```

2. unknown types are output as *TObject*-type. Example:

If the application uses third party code, which is not used for the translation, the types defined in that code, e.g. a type named "TThirdPartyType" cannot be found. It will be replaced by "TObject" then:

```
virtual void foo(/*#TThirdPartyType*/ System::TObject* AThirdParty);
```

If the files to the VCL are in the folder for files, which shall be used for the translation, but shall not be translated themselves, the VCL types will be found and written into the output. e.g.

```
virtual void SetPicture(Vcl::Graphics::TBitmap* const APicture) = 0;
```

As there is no C++ counterpart of the VCL the VCL types aren't defined there. In this case the code can be made compile, by creation of a file, which defines these types. It could look like:

```
#ifndef MissingTypesH
#define MissingTypesH

#include "System.h"
#include "d2c_system.h"

//namespace System {
// namespace Classes
// {
//       typedef System::TObject TStream;
// }
//}

namespace Vcl {

   namespace Stdctrls {
         typedef System::TObject TCheckBox;
         typedef System::TObject TComboBox;
   } // Stdctrls

   namespace Graphics {
         typedef int TColor;
```

```
          typedef System::TObject TBitmap;
      } // Graphics

   } // Vcl


   #endif // MissingTypesH
```

3. using directives of unknown units or of units not to translate are commented out. Example:

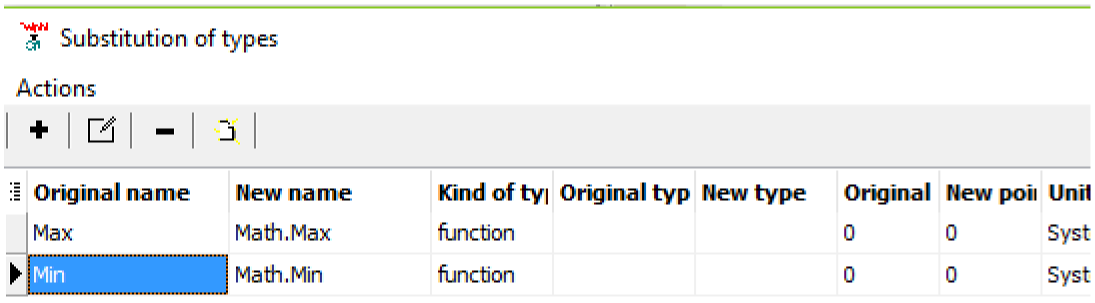
```
//# #include "Vcl.Controls.h"
//# #include "System.Classes.h"
//# #include "Winapi.Windows.h"
//# #include "Vcl.Graphics.h"
//# #include "Winapi.Messages.h"
//# #include "System.Contnrs.h"
```

In addition to the two cases above there is a third case which consists in the code of the RTL. Most parts of the RTL are converted to C++. If this code shall not be used for the dummy application, the missing types can be defined as the types of the VCL. If the code shall be used the according commenting out must be undone for the lines in question

```
//# #include "Vcl.Controls.h"
#include "System.Classes.h"
//# #include "Winapi.Windows.h"
//# #include "Vcl.Graphics.h"
//# #include "Winapi.Messages.h"
//# #include "System.Contnrs.h"
```

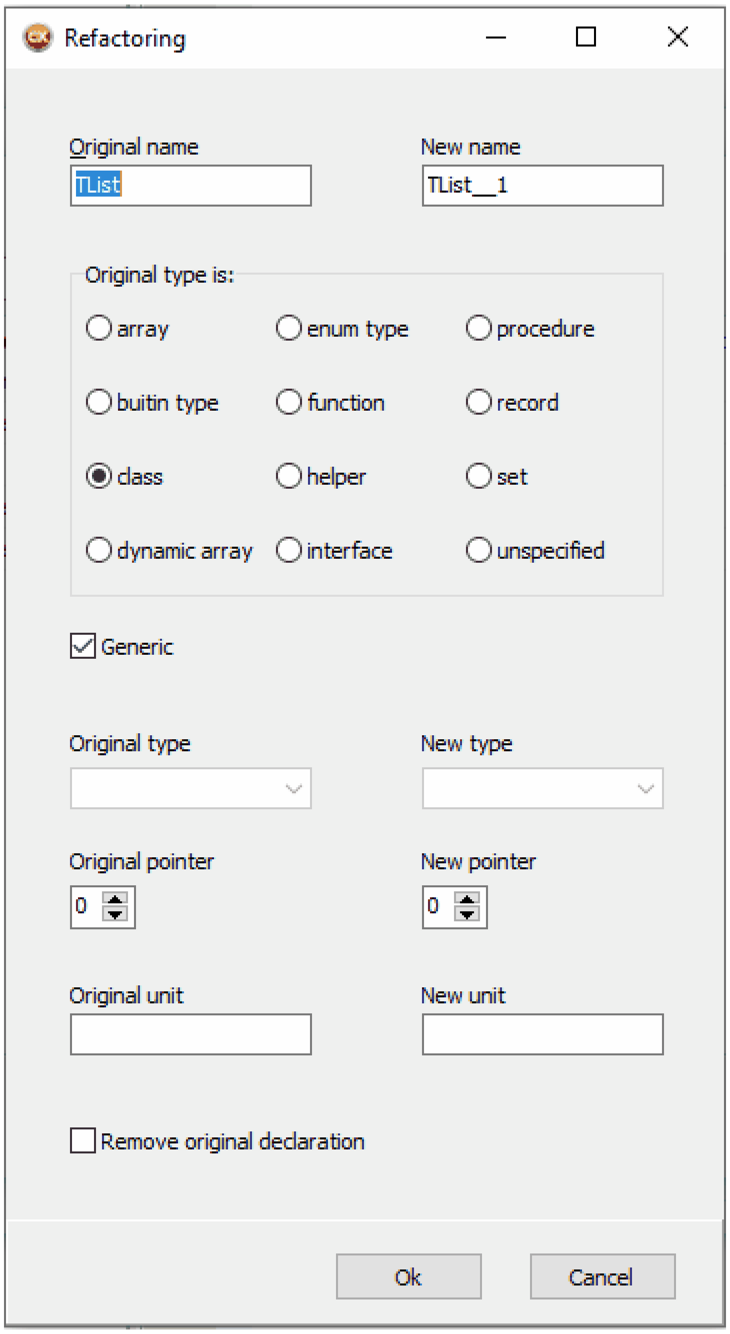
5.4.9 Refactoring

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



Original name	New name	Kind of ty	Original typ	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:



Original name: TList
New name: TList__1

Original type is:
( ) array ( ) enum type ( ) procedure
( ) buitin type ( ) function ( ) record
(●) class ( ) helper ( ) set
( ) dynamic array ( ) interface ( ) unspecified

[✓] Generic

Original type:
New type:

Original pointer: 0
New pointer: 0

Original unit:
New unit:

[ ] Remove original declaration

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. If the identifier for the original name isn't contained in the list of notations, it's 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. For example there is no according type to "ULONG" or "unsigned long" in Delphi, but it may be needed in C++. Using this identifer you can refactor:

```
function _AddRef: Integer; stdcall;
```

to

```
ULONG __stdcall AddRef()
```

"Original pointer" and "New pointer"

If the original type is a pointer the counter for the original pointer should be set accordingly. So the original pointer type will be recognized. But if the new type is defied as a pointer the value for the new type should remain Null. It has to be set to 1 only, if the new type is not a pointer, but the result shall be a pointer to the new type. For example:

```
PVSFixedFileInfo                  tagVS_FIXEDFILEINFO
1                                 1
```

=>

```
var
FI: PVSFixedFileInfo;

->

tagVS_FIXEDFILEINFO* FI = nullptr;
```

Generics

By use of the "Generics" option Delphi2Cpp distinguishes between generic and non generic types. In the following example only the generic type is substituted:

```
uses System.Classes, System.Generics.Collections;

var
  L1 : TList;
  L2 : TList<Integer>;
```

->

```
  TList* L1 = nullptr;
  TList__1<int>* L2 = nullptr;
```

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. The ".pas" extension hasn't to be appended. If there is a ".pas" extension, it will be removed.

New unit

Here the name of a header file can be set, where the new type is defined. If for example *MyList.h* is set, the following additional include directive is written into the output:

```
#include "MyList.h"
```

In contrast to the *Original unit* field, an extension has to be set here. If needed, the new type is specified then with the "Mylist" scope.

For C++Builder a ".hpp" extension is regarded as belonging to a C++ header, which automatically has been produced by the C++Builder compiler from an included Delphi unit. Therefore the name of the file is not used for scope specification.

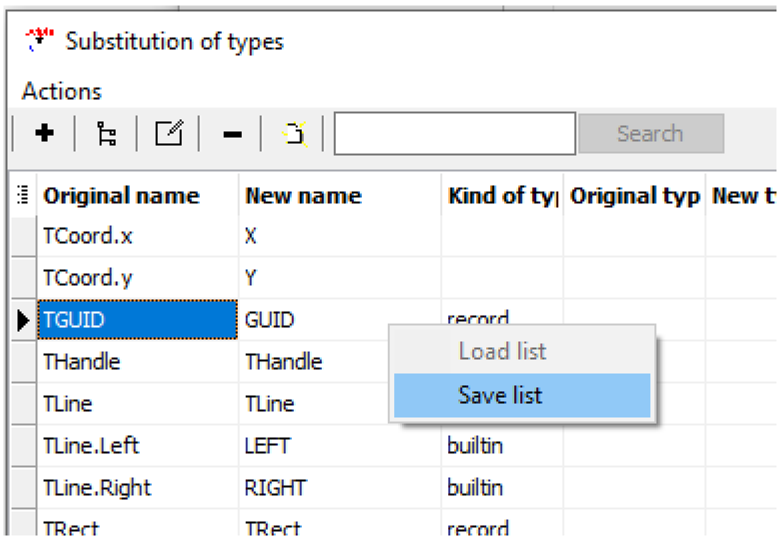
Remove original declaration

If this option is set the original declaration of the refactored type is omitted at the translation of the file where it was declared.

The table items can be loaded and saved via the popup menu.

5.4.9.1 Load/Save refactoring

The refactoring elements are saved in the project files along with the other options. However, they can also be saved separately in a text file so that exchange between different projects is easier. These files are loaded and saved via a pop-up menu in the refactoring dialog.

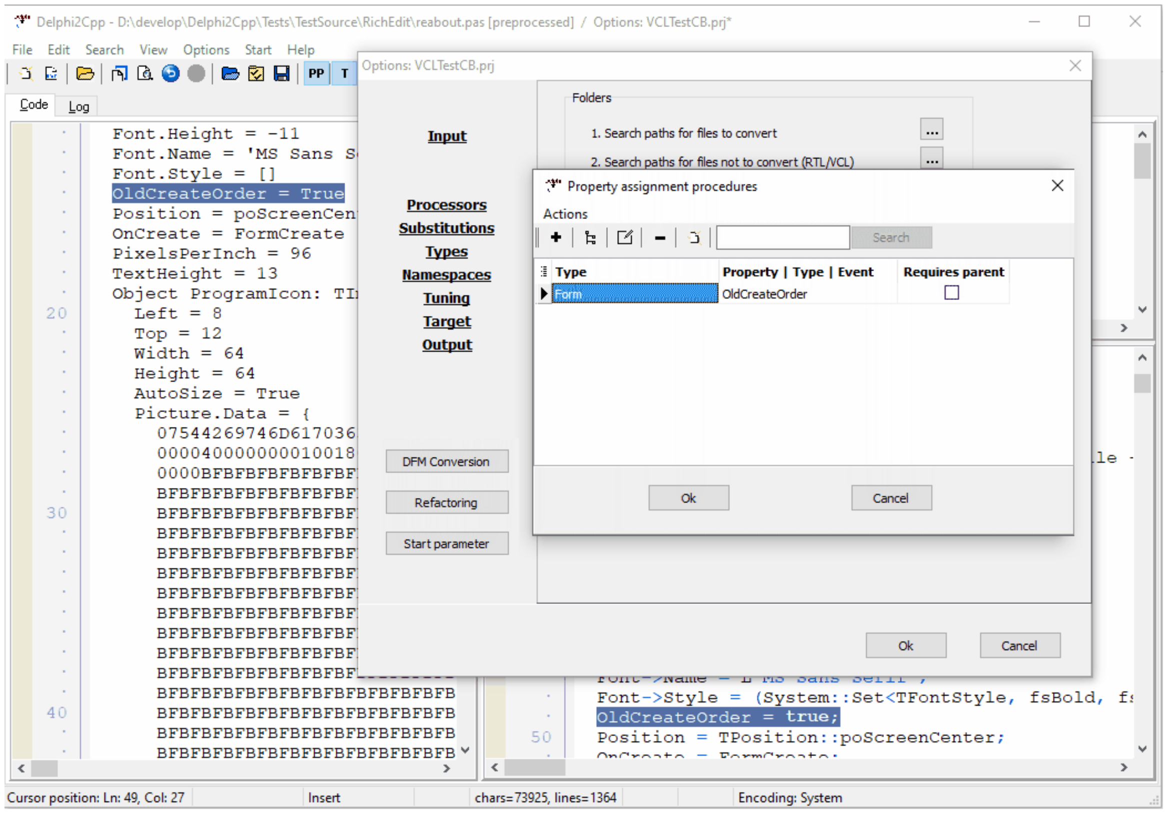


Original name	New name	Kind of ty	Original typ	New t
TCoord.x	X			
TCoord.y	Y			
TGUID	GUID	record		
THandle	THandle			
TLine	TLine			
TLine.Left	LEFT	builtin		
TLine.Right	RIGHT	builtin		
TRect	TRect	record		

The menu item for loading the file is only enabled if the list of functions is empty.

5.4.10 DFM Conversion

When the DFM translation is enabled in the processor options, per default all lines of the *DFM* code are converted to C++ assignment statements. However, when the Delphi compiler reads the *DFM* code, more can actually happen than simple assignments. To reproduce these additional effects, Delphi2Cpp can be configured to issue special DFM conversion routines when properties of certain types are to be assigned values. The dialog in which these types and properties are listed appears when you click the "*DFM Conversion*" button at the bottom left of the options dialog.



Delphi2Cpp - D:\develop\Delphi2Cpp\Tests\TestSource\RichEdit\readout.pas [preprocessed] / Options: VCLTestCB.prj*

```
Font.Height = -11
Font.Name = 'MS Sans S
Font.Style = []
OldCreateOrder = True
Position = poScreenCen
OnCreate = FormCreate
PixelsPerInch = 96
TextHeight = 13
Object ProgramIcon: TI
  Left = 8
  Top = 12
  Width = 64
  Height = 64
  AutoSize = True
  Picture.Data = {
    07544269746D617036
    000040000000010018
    0000BFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBFBFBFBFBFBF
    BFBFBFBFBFBFBFBFBFBFBFBFBFBF
```

Options: VCLTestCB.prj

Folders
1. Search paths for files to convert
2. Search paths for files not to convert (RTL/VCL)

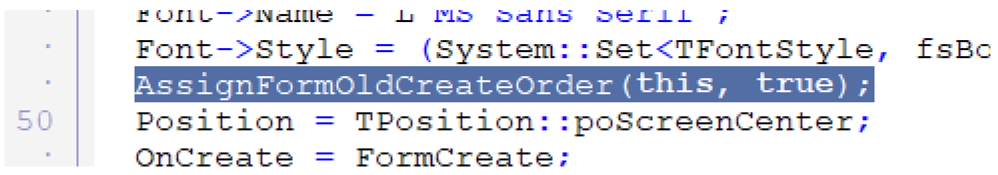
Property assignment procedures

Type	Property | Type | Event	Requires parent
Form	OldCreateOrder	☐

```
Font->Style = (System::Set<TFontStyle, fsBold, f
OldCreateOrder = true;
Position = TPosition::poScreenCenter;
```

Cursor position: Ln: 49, Col: 27 | Insert | chars=73925, lines=1364 | Encoding: System

In the example, a conversion is created for the *OldCreateOrder* property of a form. If the code is retranslated after the new routine has been inserted, the output window:shows the following line:



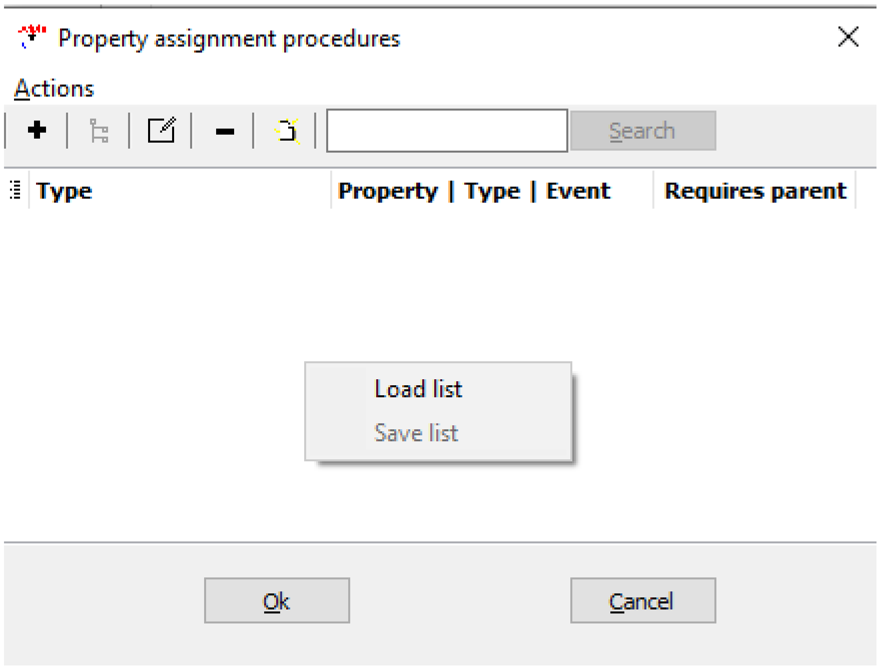
```
Font->Style = (System::Set<TFontStyle, fsBc
AssignFormOldCreateOrder(this, true);
Position = TPosition::poScreenCenter;
OnCreate = FormCreate;
```

Remark: Normally the name of a specific type starting with 'T' should be entered in the Type column. However, the example shows the special case in which a routine is defined for all classes derived from TForm.

The table items can be loaded and saved via the popup menu.

5.4.10.1 Load/Save DFM routines

The name parts from which the routines for converting DFM files are formed are saved in the project files along with the other options. However, they can also be saved separately in a text file so that exchange between different projects is easier. These files are loaded and saved via a pop-up menu in the dialog for setting up the DFM conversion routines.

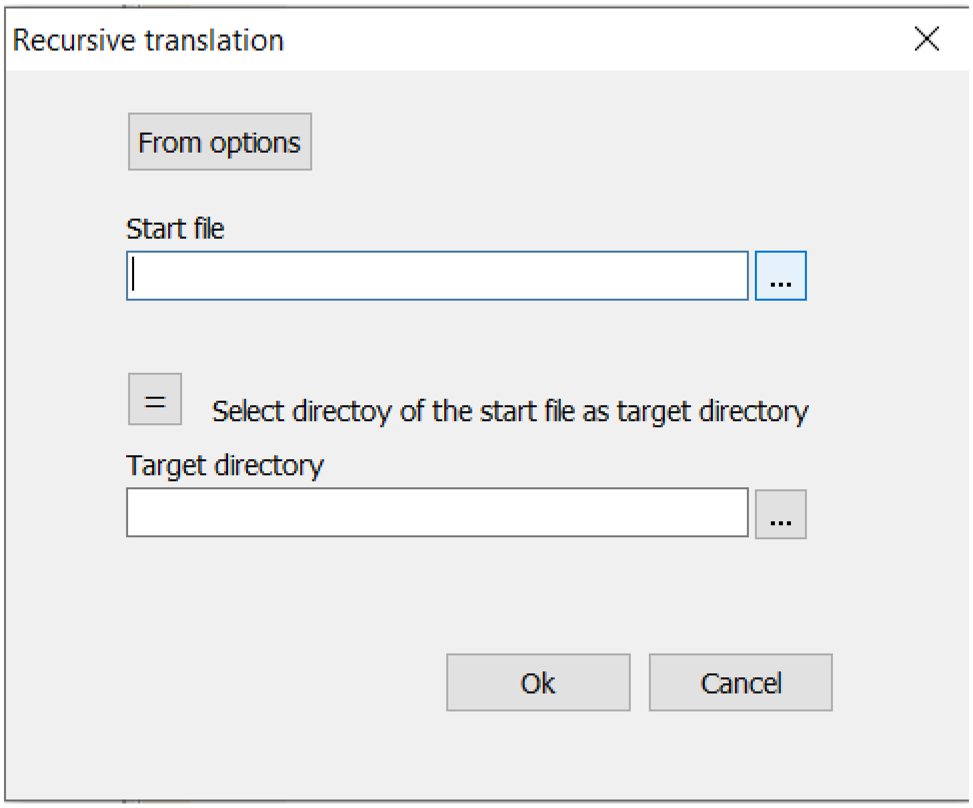


The menu item for loading the file is only enabled if the list of functions is empty.

The *DfmRoutines.txt* file in the project folder contains the list of routines used in applications from which the DFM feature of Delphi2Cpp was developed. The associated C++ code is in the file d2c_dfm.h/cpp.

5.4.11 Start parameter

The start parameter dialog is reached either from the button on the options dialog or from the menu or tool bar button . The parameters are entered here that are required, starting from a start file, to automatically compile not only this unit itself, but also all units on which it depends.



From options

The parameters to be set in the following two fields are set in the options when this dialog is invoked from the options dialog. From there they can be picked up again using the "From options" button.

Start file

Into this edit field the complete path of the start file has to be selected.

Target directory

The target directory is the directory where the translated start file will be written. The files on which this output unit depends are written into directories whose relative position to the target directory corresponds to the relative position of the source directory. The files on which the start file depends are searched in the directories specified for the translation.

OK button

When you press the OK button, different things can happen depending on where this dialog was accessed from.

- if the dialog had been called from the options dialog, the parameters are set into the options
- if the dialog had been called from the menu or from the tool bar.the recursive translation is started immediately inside of the IDE

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

Delphi2Cpp uses a preprocessor (pre-translator), which prepares the source text so that directives for the conditional compilation are evaluated and removed.
However, unlike in Delphi conditional compilation, the Delphi2Cpp pre-processor only can evaluate conditional symbols, but usually cannot evaluate constants in the Delphi language and such constants never can be set by conditional compilation.

For example, conditional expressions like

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

are evaluated, but integer values are evaluated and only operators, which also exist in C++. Sizeof-expressions like tho following are evaluated too

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

as long, as the size can be taken from the type-map. In System pas the is the following code:

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

"TExtended80Rec" is not defined in the type-map and therefore Delphi2Cpp cannot evaluate the expression.

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.

The Delphi2Cpp pre-processor cannot set new Delphi constants as in the following code from the Jedi components in JvConsts.pas

```
const
...
{$IFDEF DELPHI26}
SDelphiKey = 'Software\Embarcadero\BDS\20.0';
{$ENDIF DELPHI26}

{$IF not declared(SDelphiKey)}
  {$MESSAGE FATAL 'Declaration for SDelphiKey is missing'}
{$IFEND}
```

Even if DELPHI26 is set, SDelphiKey would not be declared

Include directives are executed correctly.

```
{$I filename}
{$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. Delphi2Cpp 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. Delphi2Cpp 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:

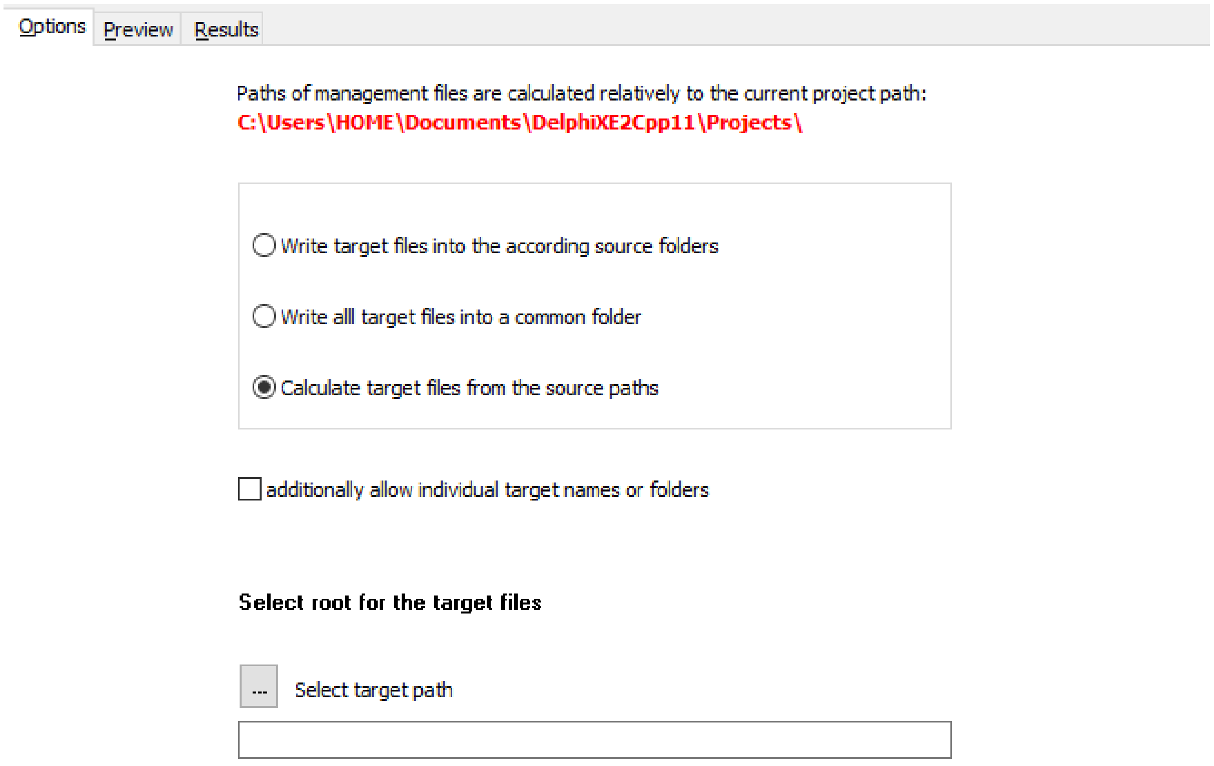
The button in the tool bar of the manager for executing the translations is deactivated until translation options are set and source files are selected. Before starting the translations, you can check the list of the files which will be produced. There is a page of his own for each of these steps in 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\DelphiXE2Cpp11\Projects\

○ Write target files into the according source folders
○ Write alll 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

1. The most simple case is to write the C++ files just to the same place, where the source files are.
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 two or three are selected the field for the target folder/root is shown and a dialog for the selection of a the target directory can be opened by the ... button:.

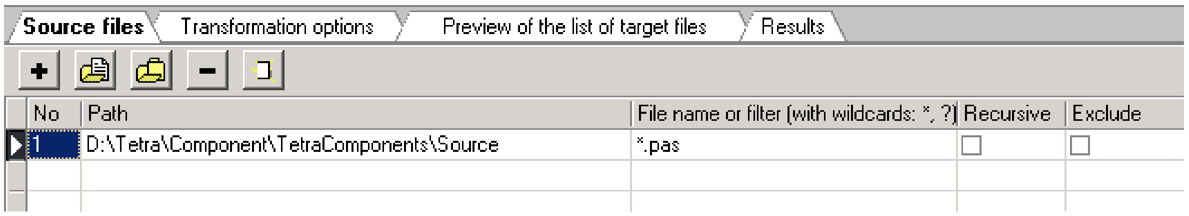
If target files shall be written outside of the common target path/root, the checkbox to allow individual file names or folders can be enabled. In that case an additional column for individual targets are shown on the source page.

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.

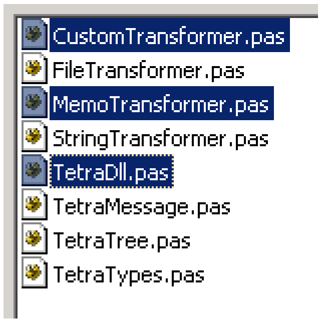


No	Path	File name or filter (with wildcards: *, ?)	Recursive	Exclude
1	D:\Tetra\Component\TetraComponents\Source	*.pas	☐	☐

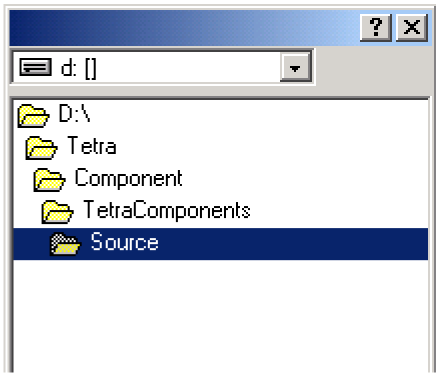
The page has a tool bar of its own with the buttons:

✚	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

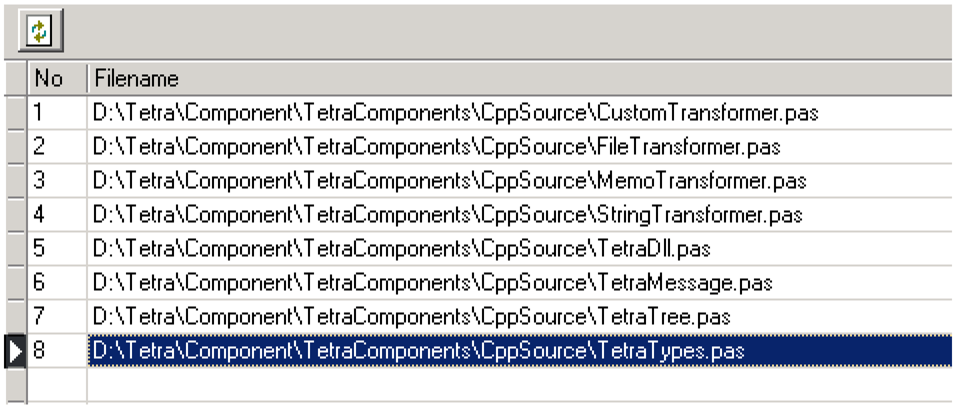
Normally the check box of this field remains deactivated. However, it can be that you want to except some files or folders from the translation of a folder. This is possible by producing rows of their own for these exceptions in the table and activating the excluding check box by mouse.

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 .

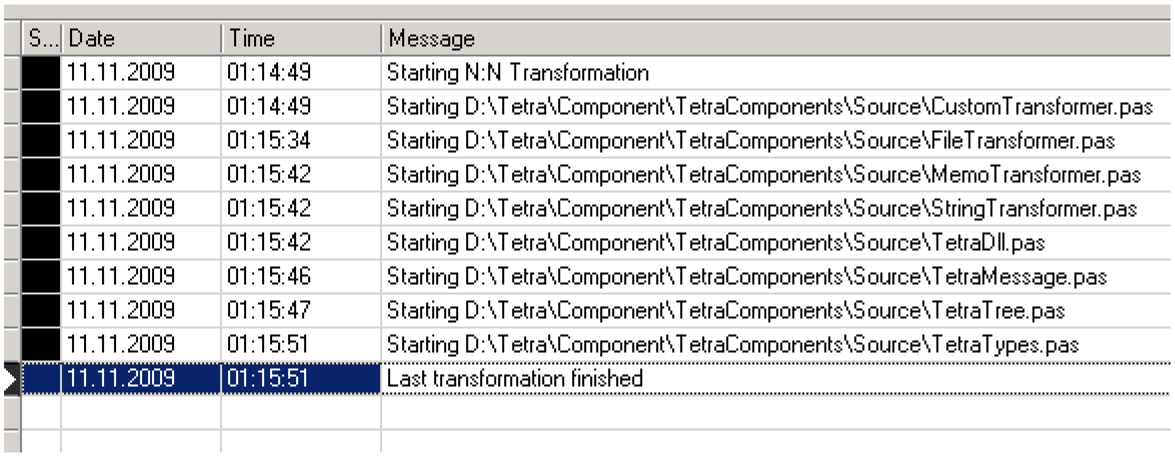
5.6.4 Starting the translation

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

Delphi2Cpp.exe can be called from the command line too. You then have to pass some parameters.

6.1 Parameter

Delphi2Cpp.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:

```
Delphi2Cpp -p PROJECT -m MANAGEMENT
```

and in the second case:

```
Delphi2Cpp -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	Delphi2Cpp 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 Delphi2Cpp 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 Delphi2Cpp management, which specifies the source and target files.
If an -m paramerter 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 Delphi2Cpp 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 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
Indexes
Types
Variables
Operators
Assignments
Routines
Special RTL/VCL functions
Properties
Statements
Manipulations with class-reference types
Reading and Writing
Message handlers
Absolute address
Method pointers
Libraries

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
Simple type identifiers (C++Builder)

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 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 CPP 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.

Some directives may have an impact on the requires notation.

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;
```

or \Embarcadero\Studio\19.0\source\rtl\win\winrt\WinAPI.CommCtrl.pas line 1429:

```
&type: UINT;
```

The Delphi2Cpp II pre-processor and parser can reckognize such tokens as identifiers, but the translation will result in type names that are forbidden in C++:

```
UINT &type;
```

There are four possibilities to handle such cases:

- let the pre-processor substitute such expressions
- let the translator substitute such expressions
- modify the source code
- make manual corrections afterwards

There is a case where the same identifiers are used with and without the ampersand.

```
TState = (Start, &Property, ObjectStart, &Object, ArrayStart, &Array, ConstructorStart, &Constructor, C
```

and e.g.

```
if FCurrentState = TState.Property then
```

then the identifier notations should be defined in the file for the identifiers in a unique way, e.g.:

```
&Array
&Constructor
&Object
&Property
Array
Constructor
Object
Property
```

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
```

Inside of the class *TParallel* in System.Threading there are a lot of overloaded "&For" functions.

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"
```

7.1.5 Simple type identifiers

How built-in type identifiers are substituted at the translation can be seen and set at the type options. However, for C++Builder there are additional restrictions.
While e.g. the type *Cardinal* usually is translated as *unsigned int*, the space inside of the name isn't permitted in the following context:

```
property testprop: cardinal read GetProp;
```

Delphi2Cpp II therefore produces a type definition for a simple identifier:

```
typedef unsigned int unsignedint;
__property unsignedint testprop = { read = GetProp };
```

7.2 File layout

The interface part and the implementation part of a unit are in Object-Pascal put in one file. In C++ they become a header file and a source file.
A file with the minimal frame of a Delphi file which might be called *test.pas* looks like::

```
unit test;

interface

implementation

end.
```

It becomes to *test.h :*

```
C++Builder                        Other Compilers

#ifndef testH                     #ifndef testH
#define testH                     #define testH

#include <System.hpp>             #include "System.h"

#include "d2c_system.h"


namespace test                    namespace test
{                                 {


}  // namespace test              }  // namespace test

#endif // testH                   #endif // testH
```

The header file is enclosed into a sentinel. Then *for C++Builder System.hpp and* d2c_system.h are included or for other compilers System.h..That way the classes, constants and routines which correspond to the according entities of the Delphi system can be used.
test.cpp starts with the selected marker for precompiled headers

```
C++Builder                        Other Compilers

#include <vcl.h>                  #include "stdafx.h"   // for Visual C++
#pragma hdrstop

#include "test.h"                 #include "test.h"

using namespace std;              using namespace std;
using namespace d2c_system;       using namespace System;
using namespace System;

namespace test                    namespace test
{                                 {



}  // namespace test              }  // namespace test
```

Thre creation of the *test* namespace is optional.

If there are uses clauses in the delphi source files the according include directives follow in the C++ files.

Comments can appear at many places in a file,

Variables declared in interface parts are declared extern variables in C++ headers with the implementation in the source file.

7.2.1 System Namespace

Type definitions, routines and constants of the Delphi system are reproduced for C++ in some files with the prefix "d2c_" and the code in these files is put into the namespace "d2c_system" for C++Builder and into the namespace "System" for other compilers.
These files are part of the files, which are installed with *Delphi2Cpp II*. There also is an extended System.pas, which has to be set in the translation options, to let Delphi2Cpp II know the signatures of the system routines.

For C++Builder the d2c-files are included directly, for other compilers there is an extra file "System.h" in which the d2c-files are included:

```
#include "d2c_system.h"
#include "d2c_systypes.h"
#include "d2c_sysconst.h"
#include "d2c_syscurr.h"
#include "d2c_sysdate.h"
#include "d2c_sysobj.h"
#include "d2c_openarray.h"
#include "d2c_sysexcept.h"
#include "d2c_sysmath.h"
#include "d2c_sysstring.h"
#include "d2c_sysfile.h"
#include "d2c_sysmem.h"
```

For C++ Builder in every produced C++ file there is a the statement

```
using namespace d2c_system;
```

For other compilers the "System.h" ends with

```
using namespace System;
```

7.2.2 Uses clauses

References to other units become to include directives in C++ in which the files of the VCL get the extension "hpp" and the extension is "h" for the other header files.

```
uses                      ->   #include "classes.hpp"
  Classes, TetraTypes;         #include "TetraTypes.h"
```

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

```
/*...*/
```

7.2.4 Namespaces

There is an option, to create a namespace for each unit. In C++ header files the scope expressions are put in front of types and constants from other units and in the C++ implementation files according uses clauses are inserted.

Example:

```
unit Namespace1;
interface
type

PInteger = ^integer;
...


unit Namespace2;
interface
uses Namespace1;
type

PInt = PInteger;

implementation
const

_pint1 : PInteger = Nil;
_pint2 : PInt = Nil;

end.
```

Namespace2 is translated to the header:

```
#ifndef Namespace2H
#define Namespace2H

#include "Namespace1.h"

namespace Namespace2
{

typedef Namespace1::PInteger PInt;

}  // namespace Namespace2

#endif //  Namespace2H
```

and the implementation:

```
#include <vcl.h>
#pragma hdrstop

#include "Namespace2.h"

using namespace Namespace1;

namespace Namespace2
{
```

```
PInteger _pint1 = NULL;
PInt _pint2 = NULL;

}  // namespace Namespace2
```

Remarks:

The hpp-headers from C++Builder have a using clause at their end. That's why Delphi2Cpp II doesn't insert namespace qualifiers and using clauses for that files. The other way round: If a file has the name of a VCL unit, an according uses clause is inserted.

7.2.5 extern variables

Variables declared in interface parts are qualified as extern in the C++ headers and their instances are included into the implementation cpp-files.

```
TokenList : TList = NIL;
->
extern TList TokenList;  // in the header file
TList* TokenList = NULL; // in the cpp -file
```

7.3 Indexes

While in C++ all arrays are zero based, that means, that they start at the index null, in Delphi arrays with other lower bounds can be defined. For example:

```
TRangeArray = array [3..7] of Integer;
```

The C++ code which is generated from the Delphi source has to correct the indexes accordingly. Because Delphi strings are one based, corrections also have to be done, if the target string type is zero based. Additional corrections have to be done, if the directive ZEROBASEDSTRINGS is set on.

The strategy at the translation wit Delphi2Cpp II is, that all functions that often are used to calculate indexes, will have the same results in C++ that they had in Delphi, but as soon as these values are used to access arrays or strings, the values are corrected.
Functions, which deliver string positions therefore have to be defined differently, depending on the chosen target string. If th target string is one based as in Delphi the function *High* e.g. would become to:

```
UnicodeString::size_type High(const UnicodeString& X)
{
  return X.Length - 1; // 1 based
}
```

If the target string is zero based the following definition has to be used:

```
UnicodeString::size_type High(const UnicodeString& X)
{
  return X.size(); // 0 based
}
```

The zero based High function doesn't deliver the highest index any more, but the same value as in Delphi. The corrections is done at the access of the stings, as can be seen in the example below:

```
var
  arr : TRangeArray;
  s : String;
begin

for index := Low(arr) to High(arr) do
  writeln(arr[index]);

for index := Low(s) to High(s) do
  writeln(s[index]);
```

->

```
TRangeArray arr;
String s;
for(index = 3 /*# Low(arr) */; index <= 7 /*# High(arr) */; index++)
{
  WriteLn(arr[index - 3]);
}
for(index = 1 /*# Low(s)*/; index <= High(s); index++)
{
  WriteLn(s[index - 1]);
}
```

However, if the Delphi code uses hard-coded values as in the following example, the translation will fail:

```
MyString := 'This is a string.';
if Pos('a', myString) = 9 do
   ...
```

->

```
myString = L"This is a string.";
if(myString.find(L"a") == 9)      // bug: myString.find(L"a") == 8
   ...
```

7.3.1 ZEROBASEDSTRINGS

Sometimes a ZEROBASEDSTRINGS directive is used in Delphi code. by which the local string indexing is changed. For example in front of the implementation code of TStringHelper the directie is set on:

```
{$ZEROBASEDSTRINGS ON}
```

The following is a code example from Embarcadero, which demonstrates, how to use the Char-property of TStringHelper:

```
var
  I: Integer;
  MyString: String;

begin
  MyString := 'This is a string.';

  for I:= 0 to MyString.Length - 1 do
```

```
      Write(MyString.Chars[I]);
  end.
```

The individual characters of the string are accessed here via a zero based index.

The automatic translation of the TStringHelper code doesn't regard the ZEROBASEDSTRINGS directive. Therefore - if zero based target strings are chosen - the translator inserts a wrong correction for the String m_Helped member. E.g.

```
result = m_Helped[Index - 1];
```

This correction has to be removed manually:

```
result = m_Helped[Index];
```

If return values of functions, which are used inside of sections of code where ZEROBASEDSTRINGS is ON, depend on the assumption of one based strings, these values have to be corrected too. the function *Low* and *High* are such typical candidates. E.g.

7.4 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;
```

In C++ the typedef keyword has to be used and the definition then goes the other way round:

```
typedef System::WideChar WCHAR;
```

Other types that can be defined in Delphi are:

Records, Classes and Interfaces
Arrays
Enumerated types
Ranges
Sets

Sometimes the order of type definitions has to be rearranged in C++. Also the order of lookup is different.

7.4.1 Records, Classes, Interfaces

Delphi and C++ have is the same concept of classes. Delphi records become to structures in C++. Their concepts are very similar too. The conversion of Delphi interfaces depends on the C++ compiler that shall be used.

7.4.1.1 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.

7.4.1.1.1 Variant parts in records

There is only a makeshift to treat variant parts in records: For every *case* there is created an according *union* in C++.

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

->

  #pragma pack(push, 1)
  struct TRect {
    /*# case Integer */
    union {
      /*# 0 */
      struct {
      int Left, Top, Right, Bottom;
      };
      /*# 1 */
      struct {
      TPoint TopLeft, BottomRight;
      };
    }; //union
  };
  #pragma pack(pop)
```

7.4.1.2 Class

A typical class consists may have following additional elements:

Ancestor
Constructor
Destructor
Class methods
Abstract methods
Visibility of class members
Creation of instances of classes

7.4.1.2.1 Ancestors

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

```
TNewClass = class ...
```

```
->

class TNewClass : public System::TObject ...
```

7.4.1.2.2 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;    ->   __fastcall classname::classname ( )
```

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

7.4.1.2.2.1 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 *Delphi2Cpp II* tries to find this call and puts it into the initialization list:

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

->

__fastcall foo::foo ( TComponent * Owner )
: inherited   ( 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++. *Delphi2Cpp II* preserves the original calls of such constructors as line comments.

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

->

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

7.4.1.2.2.2 Constructor delegation

A constructor can call another constructor in it's body:

```
type
  TFoo = class(TBase)

    constructor Create(s : string); overload;
    constructor Create(b : PChar; l : integer); overload;
...

implementation


constructor TFoo.Create(s : string);
begin
  inherited Create;
  Create(PChar(s), length(s))
end;
```

A direct translation of the constructor definition would look like:

```
TFoo::TFoo(String s)
{
  TFoo(ustr2pwchar(s), s.size()),
}
```

However, this does not the same as in Delphi. In C++ the call of the second constructor in the body of the first only creates a temporary local second *TFoo* object, which has no effect to the current instance. But in C++11 there is the new feature to call the second constructor instead of an initialization list

```
TFoo::TFoo(String s)
 : TFoo(ustr2pwchar(s), s.size())  //# delegation
{

}
```

Though this construction doesn't work for C++98 compiler, Delphi2Cpp nevertheless translates the code in this way too, because there is no gneral alternative solution. For C++98 the code has to be post-processed manually.

7.4.1.2.2.3 Initialization lists

In Delphi member variables like other variables too are initialized automatically with default values. Because this is not the case in C++ Delphi2C++ has to do these initializations explicitly, like in the following example:

.	**Delphi source**	**C++ translation**

```
  Base = class                              class Base: public System::TObject {
  public                                      friend class Derived;
    constructor Create(arg : Integer);        public:
    destructor Destroy;                       __fastcall Base( int arg );
  private                                     __fastcall ~Base( );
    FList : TList;                            private:
    FI : Integer;                             TList* FList;
    FTimeOut: Longint;                        int FI;
  end;                                        int FTimeOut;
                                              public: inline __fastcall Base () {} <- dangerous
                                            };


  constructor Base.Create(arg : Integer);   __fastcall Base::Base( int arg )
  begin                                      : FI(0),
  end;                                         FList(NULL),
                                               FTimeOut(0)
                                            {
                                            }
```

If the members are initialized explicitly in Delphi, *Delphi2Cpp* tries to find the according statements and puts them into the initialization list of the class constructor:

```
constructor Base.Create(arg : Integer);     __fastcall Base::Base( int arg )
begin                                        : FI(arg),
  FList := TList.Create;                       FList(new TList),
  FI := arg;                                   FTimeOut(0)
  if arg <> $00 then                        {
    FTimeOut := arg                           if ( arg != 0x00 )
  else                                          FTimeOut = arg;
   FTimeOut := DefaultTimeout;                else
end;                                            FTimeOut = DefaultTimeout;
                                            }
```

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 *Delphi2Cpp* tries to take care about such cases. But *Delphi2Cpp* cannot find all such hidden dependencies, as in the following example:

```
    constructor Derived.Create;                 __fastcall Derived::Derived( )
    var                                          : inherited( i ),
      i : Integer;                                 FB(false)
    begin                                       {
      i := SomeFunction;                          int i = 0;
      inherited Create(i);                        i = SomeFunction;
    end;                                        }
```

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 Delphi2Cpp 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;
}
```

7.4.1.2.2.4 Addition of missing constructors

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 Delphi2Cpp generates a default constructor for each derived class, even if such a constructor doesn't exist in the original Delphi code. So, resuming the previous example, the additional standard constructor would look like:

```
    __fastcall Base::Base()
     : FI(0),
       FList(NULL),
       FTimeOut(0)
    {
    }
```

Here the member variables are initialized with default values.

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

```
    class MyException: public Sysutils::Exception {
      typedef Sysutils::Exception inherited;
      public: inline __fastcall MyException( const String MSG ) : inherited( MSG ) {}
      public: inline __fastcall MyException( const String MSG, const TVarRec* Args, int Args_maxidx ) : inh
      public: inline __fastcall MyException( int Ident ) : inherited( Ident ) {}
      public: inline __fastcall MyException( PResStringRec ResStringRec ) : inherited( ResStringRec ) {}
      public: inline __fastcall MyException( int Ident, const TVarRec* Args, int Args_maxidx ) : inherited(
      public: inline __fastcall MyException( PResStringRec ResStringRec, const TVarRec* Args, int Args_maxi
      public: inline __fastcall MyException( const String MSG, int AHelpContext ) : inherited( MSG, AHelpCo
      public: inline __fastcall MyException( const String MSG, const TVarRec* Args, int Args_maxidx, int AH
      public: inline __fastcall MyException( int Ident, int AHelpContext ) : inherited( Ident, AHelpContext
      public: inline __fastcall MyException( PResStringRec ResStringRec, int AHelpContext ) : inherited( Re
      public: inline __fastcall MyException( PResStringRec ResStringRec, const TVarRec* Args, int Args_maxi
      public: inline __fastcall MyException( int Ident, const TVarRec* Args, int Args_maxidx, int AHelpCont
    };
```

7.4.1.2.2.5 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 *Delphi2Cpp*cannot recognize this.
3. in Delphi there can be several constructors with the same signature but with different names
4. the call of virtual functions inside of constructors
5. Delphi has the concept of virtual constructors

[****]

A big problem with constructors is, that in Delphi there can be several constructors with the same signature but with different names.. E.g.:

```
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 strict translation the two constructors would become ambiguous:

```
__fastcall TCoordinate::TCoordinate( double AX, double AY )
```

```
 : x(AX),
   y(AY)
{
}

__fastcall TCoordinate::TCoordinate( double Radius, double Angle )
 : x(Radius * cos( Angle )),
   y(Radius * sin( Angle ))
{
}
```

Therefore Delphi2Cpp inserts a second parameter into the alphabetically second declaration to distinguish the constructors in C++.

```
class TCoordinate : public System::TObject
{
public:
  __fastcall TCoordinate(double AX, double AY, void* OverloadSelector0);
  __fastcall TCoordinate(double Radius, double Angle);
private:
  double x;
  double y;
};

__fastcall TCoordinate::TCoordinate(double AX, double AY, void* OverloadSelector0)
 : x(AX),
   y(AY)
{
}

__fastcall TCoordinate::TCoordinate(double Radius, double Angle)
 : x(Radius * cos(Angle)),
   y(Radius * sIn(Angle))
{
}
```

The following code snippet demonstrates how the constructors are called:

```
procedure TestConstructors;
var
  c1, c2 : TCoordinate;
begin
  c1 := TCoordinate.CreateRectangular(1.0, 2.0);
  c1 := TCoordinate.CreatePolar(1.0, 90.0);
```

becomes to:

```
void __fastcall TestConstructors()
{
  TCoordinate* c1 = nullptr;
  TCoordinate* c2 = nullptr;
  c1 = new TCoordinate(1.0, 2.0, nullptr);
  c1 = new TCoordinate(1.0, 90.0);
```

Another problem with constructors is that calls of virtual functions inside of constructors are allowed in Delphi, but in C++ such calls should be avoided

As example the *TPolygon* class and the derived classes *TTriangle* and *TSquare* as defined here

http://www.delphibasics.co.uk/RTL.asp?Name=Abstract can be taken:

```
  TPolygon = class
  ...
  protected
     procedure setArea; Virtual; Abstract;  // Cannot code until sides known
  ...

  TTriangle = class(TPolygon)
  protected
    procedure setArea; override;   // Override the abstract method


  TSquare = class(TPolygon)
  protected
    procedure setArea; override;   // Override the abstract method


constructor TPolygon.Create(sides, length : Integer);
begin
  ...
  setArea;
end;
```

In Delphi the *setAres*-procedure of the derived classes will be called in their constructors. With C++Builder this works well too, but other C++ compilers always try to call the *setArea*-procedure of *TPolygon*. Manual post-processing is necessary then. E.g. *setArea* could be made accessible and called after construction:

```
triangle = new TTriangle(3, 10);
triangle->setArea();
```

A third problem with constructors in Delphi is that constructors can be used there 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.

This behavior can be reproduced, if the option to create meta classes is enabled.

7.4.1.2.3 Destructors

In Delphi a declaration of a destructor 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;    ->   __fastcall classname::~classname ( )
```

Delphi2Cpp 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. That's the case for all classes, which are derived from TObject.

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

In Delphi parameters can be passed to destructors, but this isn't possible in C++.
For destructors analogous problems can occur as for constructors, but they are very rare.

7.4.1.2.4 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.
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).

From C++Builder 2009 on there is the additional keyword *__classmethod* to - partially - reproduce Delphi class method, For other compilers class methods are converted to static methods.

7.4.1.2.4.1 C++Builder __classmethod

For C++Builder Delphi class methods are converted to functions using the keyword *__classmethod*, as described in the Embarcadero documentation:

https://docwiki.embarcadero.com/RADStudio/Sydney/en/Class_Methods

7.4.1.2.4.2 Other compilers cllass methods

For other compilers than C++Builder class methods are converted as static methods;

non virtual class methods
virtual class methods
Self instance

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:

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

...

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

Because there are no virtual static methods in C++ DelpiXE2Cpp11 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 an 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 Delphi2Cpp provides these instances automatically:

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

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

->

```
base->ClassMethod(0);
TBase::ClassRefType::getClassInstance()->ClassVirtual(0);
TDerived::ClassRefType::getClassInstance()->ClassVirtual(0);
```

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 through class references too, because the static function *ClassRefType* delivers the class reference of *TBase* or *TDerived* and the call of *getClassInstance* delivers a singleton instance of a pointer to *TBase or TDerived* respectively.

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;
```

Delphi2Cpp can convert this code adequately only, if the option to create meta classes is enabled. The code then becomes to:

```
/*#static*/ int TBase::ClassMethod(int xi)
{
  int result = 0;
  /*# with Create do */
  {
    auto with0 = SCreate();
    with0->Init();
    with0->Done();
    delete with0;
  }
  result = xi;
  return result;
}
```

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

7.4.1.2.5 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;
->
virtual int __fastcall Get(int Index) = 0;
```

But opposed to Delphi. in C++ no objects can be created from classes with abstract methods. A C++ compiler even complains about the code at compile time. At development time sometimes it's practical, that such code compiles and runs in C++ too. Therefore Delphi2Cpp has the option to create minimal function bodies for abstract functions. The example becomes to:

```
virtual int __fastcall Get(int Index){return 0;} // = 0;
```

Of course, this option should be used temporarily only.

7.4.1.2.6 Visibility of class members

In Delphi Members at the beginning of a VCL class declaration that don’t have a specified visibility are by default published and in other classes they are public. In C++ this is written explicitly. (Delphi2Cpp ignores the {$M+} directive, which would make them public.)

A problem at the translation of Delphi code is, that in Delphi a private or protected member is visible everywhere in the module where its class is declared. In C++ a private or protected member is visible only inside of the class. *Delphi2Cpp* solves this problem by making all classes in the same module to friends of each other. Free routines also are declared as friend.

The following Delphi code is an example, where the direct translation to C++ code would not compile, if *TFriend* isn't declared as a friend of *TLonely*:

```
TFriend = class
private
  FCount: Integer;
end;

TLonely = class(TFriend)
public
  procedure NeedsFriend;
end;

implementation

procedure TLonely.NeedsFriend;
begin
  FCount := 0; // in C++ access to TLonely::FCount is not possible
end;
```

The converted C++ code doesn't compile, because *FCount* cannot be accessed in *TLonely::NeedsFriend*.

```
unit friends;

type
```

```
class TFriend : public TObject
{
private:
    int FCount;
public:
    __fastcall TFriend();
};

class TLonely : public TFriend
{
public:
    void __fastcall NeedsFriend();
    __fastcall TLonely();
};



void __fastcall TLonely::NeedsFriend()
{
  bool result = false;
  FCount = 0; // in C++ access to TLonely::FCount is not possible
  return result;
}
```

Delphi2Cpp therefore lists all possible class- and routine- friend declarations of a unit into a file and includes it into all class declarations. The name of this file is created by appending "_friends" to the file name. The file gets the extension ".inc". The class declarations of the example therefore becomes to:

```
class TFriend : public TObject
{
  #include "friends_friends.inc"
private:
    int FCount;
public:
    __fastcall TFriend();
};

class TLonely : public TFriend
{
  #include "friends_friends.inc"
public:
    bool __fastcall NeedsFriend();
    __fastcall TLonely();
};
```

The content of "friends_friends.inc" is:

```
friend class TFriend;
friend class TLonely;
```

Now TLonely::NeedsFriend compiles without problem. If this function were no member function, but a free function like:

```
procedure NeedsFriend;
var
  f : TFriend;
begin
  f := TFriend.Create;
  f.FCount := 0; // in C++ access to TLonely::FCount is not possible
...
```

Delphi2Cpp adds another line to "friends_friends.inc"

```
friend class TFriend;
friend class TLonely;
friend void __fastcall NeedsFriend();
```

Now

```
void __fastcall NeedsFriend()
{
  TFriend* F = NULL;
  F = new TFriend();
  F->FCount = 0;
```

also compiles without problem.

7.4.1.2.7 Creation of instances of classes

VCL classes have to be created with new in C++. For example:

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

7.4.1.3 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;
```

For **C++Builder** the special macro "__interface" macro:

```
#define __interface struct  // in sysmac.h
```

is used instead of the *class* keyword to mark interfaces:

```
__interface INTERFACE_UUID("{ GUID}" ) IConverter
{
  virtual String __fastcall convert(String Source);
};

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

However there will be a linker error like "unresolved vtable", if such an interface is used. If you create a small pas file for the interface, add it to the C++Builder project and remove the C++ definition for the interface, C++Builder will create a header file for the interface file automatically, which you can include then. Example pas file:

```
unit Recyclable;

interface

 type
    IRecyclable = Interface(IInterface)
      function GetIsRecyclable : Boolean;
      property isRecyclable : Boolean read GetIsRecyclable;
    end;

implementation
end.
```

```
->
```

```
#include "Recyclable.hpp"
```

Now the project will compile without linker error.

Visual C++ also knows this keyword, but the GUID has to be written differently:

```
[ uuid( "GUID" ) ]
__interface IConverter
{
  virtual String convert(String Source);
};

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

At **other compilers**, which have not the interface extension, multiple inheritance can be used instead, As explained here:

http://www.codeproject.com/Articles/10553/Using-Interfaces-in-C

the interface class needs a virtual destructor and the methods should be public and declared abstract:

```
//[ uuid( "GUID" ) ]
class IConverter
{
  public:
  virtual ~IConverter() {}
  virtual String convert(String Source) = 0;
};

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

GUID's cannot be used here. Under Microsoft Windows GUID's are used for COM purposes.

7.4.1.4 Multiple interfaces

In Delphi a class can be derived from multiple interfaces. Interfaces in last instance have to be derived from *IInterface*, which in Delphi is the same as *IUnknown*, C++ however uses *IUnknown* from MS Windows, which has the abstract methods: *AddRef*, *Release* and *QueryInterface*. In Delphi analogous methods are defined automatically, when a class is derived from Delphi's *IInterface* (=*IUnknown*). In C ++ however, the implementations have to be defined explicitly. Therefore a macro is inserted into class declaration:

```
#define INTFOBJECT_IMPL_IUNKNOWN(BASE) \
    ULONG   __stdcall AddRef() { return BASE::AddRef();} \
    ULONG   __stdcall Release(){ return BASE::Release();} \
    HRESULT __stdcall QueryInterface(REFIID iid, void** p){ return
BASE::QueryInterface(iid, p);}
```

An class declaration then for example looks like:

```
class TCar : public System::TInterfacedObject, public IRecyclable
{
public:
  INTFOBJECT_IMPL_IUNKNOWN(System::TInterfacedObject)
...
```

More details to this subject can be found here:

http://docwiki.embarcadero.com/RADStudio/Rio/en/Inheritance_and_Interfaces

7.4.2 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 passes to routines. Open arrays are arrays of unspecified size with elements, that all have the same type.

7.4.2.1 Static arrays

Static arrays in C++ are declared similar as in Delphi:

```
TArray2 = array [1..10] of Char
->
typedef char [ 10 ] TArray2
```

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 *Delphi2Cpp*.

This `MAXIDX` macro is used, when a static array is passed to a function, which accepts an open array.

7.4.2.2 Dynamic arrays

Dynamic arrays are simulated in the C++Builder C++ by the class *DynamicArray*:

```
template <class T> class DELPHIRETURN DynamicArray;
```

If the output is generated for other compilers *std::vector* is used instead of a *DynamicArray*.

```
MyFlexibleArray: array of Real;
->
DynamicArray < double > MyFlexibleArray;  // C++Builder
std::vector < double > MyFlexibleArray;   // other compiler
```

This *DynamicArray* class has the properties *Low*, *High* and *Length*. By the *Length* property, the size of the array can be changed. Dynamic arrays are accepted as parameters only, if the type of the array is defined explicitly and if the function expects this type.

7.4.2.3 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, Delphi2Cpp 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;
```

7.4.2.4 Initializing arrays

The initialization of arrays in Delphi and C++ looks very similar. For example the initialization of an array of *TStyleRecord*'s:

```
TStyleRecord = record
  Name    : string;
  Color   : TColor;
  Style   : TFontStyles;
end;
TStylesArray = array[0 .. 2] of TStyleRecord;
```

might be:

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

With the C++11 std::initializer_list there is a simple equivalent in C++:

```
#define arrayinit__0 TSet<int, 0, 255>()
#define arrayinit__1 TSet<int, 0, 255>()
#define arrayinit__2 (TSet<int, 0, 255>() << fsItalic)


const TStylesArray DefaultStyles = {{_T("tnone"), clBlack, arrayinit__0},
{_T("tstring"), clMaroon, arrayinit__1},
{_T("tcomment"), clNavy, arrayinit__2}};
```

In C++98 this was not possible, because all elements in such lists had to be C built in types.

7.4.2.5 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.

7.4.2.5.1 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 C++ there is no counterpart to the function *High*, which typically is needed to use the open array. Therefore in C++ the value of the upper bound of the open array has to be passed together with a pointer to the first element of the array.

For C++Builder:

```
int __fastcall Sum(const int* Arr, int Arr_maxidx )
{
  int result;
  int i;
  result = 0;
  for ( i = 0 /* Low( Arr )*/; i <= Arr_maxidx /* High( Arr )*/; i++)
    result = result + Arr[i];
  return result;
}
```

For other compilers a vector is passed:

```
int Sum(const std::vector<int> Arr)
{
  int result = 0;
  int i = 0;
  int stop = 0;
  result = 0;
  for(stop = (int) Arr.size() - 1 /*# High(Arr) */, i = 0 /*# Low(Arr) */; i <= stop; i++)
  {
    result = result + Arr[i];
  }
  return result;
}
```

If a temporary *set* of values is passed as open array parameter to a function, a corresponding array is produced in the C++ output, which is put in front of the function call.

The function call

```
Sum([1,2,3]);
```

is converted for C++ Builder by use of the OPENARRAY macro, which is defined in sysopen.h:

```
Sum(OPENARRAY(int, (1, 2, 3)));
```

For other compilers the call is simply converted to

```
Sum({1, 2, 3});
```

A special case of open array parameters is the use as var-parameter

7.4.2.5.2 Open array var parameters

A special case of open array parameters is the use as var-parameter as for example at the CopyTo-method of TStringHelper:

```
procedure TStringHelper.CopyTo(SourceIndex: Integer; var Destination: array of Char; DestinationIndex,
begin
  Move(PChar(PChar(Self)+SourceIndex)^, Destination[DestinationIndex], Count * SizeOf(Char));
end;
```

For this case *d2c_openarray.h* defines a special *OpenArrayRef*-type which can be used to pass dynamic arrays and strings to such methods.*OpenArrayRef* has a similar interface as a std::vector.

```
template <class T>
class OpenArrayRef
{
public:
        OpenArrayRef(std::vector<T>& v);
        OpenArrayRef(DynamicArray<T>& arr);  // C++Builder only
        OpenArrayRef(std::basic_string<T>& s);

        ...

};
```

By use of this helper class the code above is converted to:

```
void TStringHelper::CopyTo(int sourceIndex, OpenArrayRef<Char> Destination, int DestinationIndex, int C
{
  Move(ustr2pwchar(m_Helped) + sourceIndex, Destination.data() + DestinationIndex, Count * sizeof(Char)
}
```

There are an additional *OpenArrayRef2*-type and an *OpenArrayRef3*-type derived form *OpenArrayRef* by which normal fixed arrays respectively ShortString's can be passed to such parameters..

7.4.2.5.3 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);

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

is translated to:

C++Builder	Other
void __fastcall Foo(const Char* Arr, int Arr_maxidx)	void Foo(const std::vector<Char> Arr)

```
{
}

void __fastcall bar()
{
  Char chararray[10/*# range 1..10*/];
  Foo(OpenArrayEx<Char>(chararray, 10), 9);
}
```

```
{
}

  void bar()
{
     Char chararray[10/*# range 1..10*/];

  std::vector<Char> test__0(chararray, chararray + 10 - 1 + 1);
  Foo(test__0);
}
```

7.4.2.5.4 Dynamic array parameter

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

```
type
strarray = Array of String;
procedure CheckDynamicArray(aSources : strarray);
->

C++Builder
typedef DynamicArray<System::String> strarray;
void __fastcall CheckDynamicArray(strarray& aSources);

Other
typedef std::vector<System::String> strarray;
void CheckDynamicArray(strarray& aSources);
```

In this case Delphi2Cpp translates such a parameter as a reference to a dynamic array.

Let's compare this case with the case, where the called function has an open array parameter.

Other

For Other compilers than C++Builder there is no surprise, because the expected parameter is the same for *CheckDynamicArray* and *CheckOpenArray:*

```
procedure CheckOpenArray(const AArray: Array of String);
->
void CheckOpenArray(const std::vector<String>& AArray)

std::vector<String> strarray;
CheckDynamicArray(strarray);
CheckOpenArray(strarray);
```

C++Builder

For C++Builder however, though the calls of these function look similar in Delphi they lo quite different in C++:

```
procedure CheckOpenArray(const AArray: Array of String);
->
void __fascallCheckOpenArray(const String* AArray, int AArray_maxidx)

DynamicArray<String> strarray;
CheckDynamicArray(strarray);
```

```
CheckOpenArray(DynamicArrayPointer(strarray), strarray.High);
```

Instead of only one parameter here a pointer to the array is passed as a first parameter and the upper bound (*High*) of the array is passed as second parameter. The pointer to the array is calculated by the *DynamicArrayPointer* function, which returns the NULL pointer, if the array is empty.

```
template <class T>
const T* DynamicArrayPointer(const DynamicArray<T>& DA, unsigned int Index = 0)
{
    if(DA.Length > 0)
           return &DA[Index];
     else
           return NULL;
}
```

This function has to be used, because passing "&strarray[0]" throws an exception, when *strarray* is empty.

7.4.2.5.5 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 which can contain the single values of different types.

These array's are reproduced in C++ differently for:

C++Builder
Other compilers

Delphi2C++ can distinguish whether set parameters have to be passed as array of const or normal set's.

7.4.2.5.5.1 array of const for C++Builder

For C++Builder the value of an ***array of const*** is represented by two values: a pointer to a *TVarRec* and the index of the last element of the array, which begins at the position which the pointer points to.

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

->

void __fastcall foo (  TVarRec* Args, const int Args_Size );
```

When such a functions is called with a set as argument, the macro ***ARRAYOFCONST*** is used into the C++ output.

```
foo(['hello', 'world']);   ->  foo (  ARRAYOFCONST(( "hello", "world" )) );
```

This macro is defined for the C++Builder as:

```
#define ARRAYOFCONST(values)
OpenArray<TVarRec>values,
OpenArrayCount<TVarRec>values.GetHigh()
```

The class OpenArray<TVarRec> is constructed in a manner, that it's address is equal to the pointer *TVarRec** used in the signature of *foo* above.

There is a small difference between Delphi and C++Builder concerning character pointers like 'abc'. In Delphi 'abc' is stored as *VUnicodeString*, at C++Builder the TVarRec constructor for character pointers is used and therefore the value is stored as *VPWideChar*.

```
procedure foo(const constArray : Array of const)

case constArray[i].VType of
  vtPChar:      pac := constArray[i].VPChar;
  vtPWideChar:  pwc := constArray[i].VPWideChar;
  vtAnsiString: sa := AnsiString(constArray[i].VAnsiString);
  vtWideString: sw := WideString(constArray[i].VWideString);
  vtUnicodeString: su := UnicodeString(constArray[i].VUnicodeString);
end;
// Delphi: foo8['abc') => su = 'abc';
// C++Builder: foo8['abc') => pwc = 'abc';
```

7.4.2.5.5.2 array of const for other compilers

array of const is reproduced for other compilers by an *ArrayOfConst* class defined in d2c_sysvariant.h, *ArrayOfConst* is derived from std::vector<TVarRec>.

```
class ArrayOfConst : public std::vector<TVarRec>
```

A function declaration with such a parameter looks like:

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

->

void foo ( const ArrayofConst& Args );
```

The call of the function therefore converts as:

```
foo(['hello', 'world']);   ->  foo (  ArrayofConst&( "hello", "world" ) );
```

Since the *ArrayOfConst* class has the size method in contrast to C++Builder an additional parameter isn't necessary.

TVarRec

For C++ versions before C++17 *TVarRec* is defined as a union of different C++ types and a *VType* field, which indicates which of that types the actual value has.For C++ 17 *TVarRec* ist defined as a *std::variant*. In both versions the number of different types that can be stored in *TVarRec* is less than the number in Delphi, because in C++ there is no difference between *WideString* and *UnicodeString*. Therefore there are double cases in case-/switch-statement.

7.4.2.5.5.3 array of const vs. set's

Delphi2Cpp 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(['h', 'w']);
bar(['h', 'w']);

 ->

typedef System::Set<unsigned char, 0, 255> TCharSet;
#define test__0 (TCharSet() << L'h' << L'w')

void __fastcall foo (const TVarRec* ASet, int ASet_maxidx);
void __fastcall bar (TCharSet ASet);

foo ( ARRAYOFCONST(( "h", "w" )));
bar ( test__0 );
```

If such an array is passed further to another function, then Delphi2Cpp takes care that the second parameter is also passed in the C++ code.

```
procedure foo2(var arr: array of const);
begin
  bar2( arr );
end;

->

void __fastcall foo2 ( TVarRec* arr, const int arr_high )
{
  bar2 ( arr, arr_high );
}
```

7.4.2.6 Returning arrays

In Delphi arrays can be returned from functions by value, but this is not allowed for C-style arrays in C ++. In C++ arrays are passed to functions by reference instead. That's what Delphi2Cpp*) does too. If *TObjectArray* is defined as:

```
type TObjectArray = array[1..3] of TObject;
```

or in C++:

```
typedef TObject* TObjectArray[3/*# range 1..3*/];
```

the following Delphi function:

```
function CreateArray: TObjectArray;
begin
  Result[1] := TObject.Create;
  Result[2] := TObject.Create;
  Result[3] := TObject.Create;
end;
```

becomes in C++ to:

```
TObjectArray& CreateArray(TObjectArray& result, uniquetype u)
{
  result[1 - 1] = new TObject();
  result[2 - 1] = new TObject();
  result[3 - 1] = new TObject();
  return result;
}
```

This function receives the array as reference parameter, so it can return the reference without danger, There is a second *uniquetype* parameter, which distinguishes the function from a possible overload:

```
procedure CreateArray(var arr: TObjectArray);
void CreateArray(TObjectArray& arr)
```

The function call:

```
procedure Test;
var
  arr2: TObjectArray;
begin
  arr2 := CreateArray;
end;
```

is translated by Delphi2Cpp to

```
void Test()
{
  TObjectArray arr2;
  CreateArray(arr2, uniquetype());
}
```

In this example the returned array reference isn't used at all. It is used however, if *CreateArray* delivers the value for another function:

```
procedure ProcessArray(arr: TObjectArray);


procedure Test2;
begin
  ProcessArray(CreateArray);
end;
```

This becomes in C++:

```
void Test2()
```

```
{
  TObjectArray arrayreturn__0; ProcessArray(CreateArray(arrayreturn__0, uniquetype()));
}
```

At first Delphi2Cpp creates a local TObjectArray, which is passed to the CreateArray function and finally is directly passed as reference parameter to the other function. The treatment of array properties is similar.

*) In contrast to Delphi2Cpp the old Delphi2Cpp in such cases created a helping array in the file scope which is used for an intermediate copy of the array

7.4.3 Enumerated types

The explicit definition of enumeration types is easy to translate.

```
Day = (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
->
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.

```
Day : (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
->
enum test__0 {Mon, Tue, Wed, Thu, Fri, Sat, Sun };
test__0 Day;
```

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;

->

enum TEnum {cm1, cm2, cm3, cm4, cm5, cm6 };
AnsiString foo [ 6 /*TEnum*/ ];
```

7.4.4 Ranges

Numeric ranges for the specification of the size of an array are reduced to a single value at the translation into C++. The original limits are inserted in the translated code as a comment.

```
type foo = array [1..10] of Char
->
typedef char foo [ 10/* 1..10 */ ]
```

Numeric ranges for the definition of the range of a type are left out at the translation.

```
TYearType = 1..12;
->
int TYearType;/* range 1..12*/
```

In other cases the range specifications are copied in the C++ code as they are in Delphi and must be adapted by hand.

7.4.5 Sets

A Delphi set is simulated in the C++ VCL by the class Set:

```
template<class T, unsigned char minEl, unsigned char maxEl>
class __declspec(delphireturn) Set;
```

This set class is part of the C++Builder VCL. Users of other compilers can use the emulation of Delphi set's in "DelphiSets.h" in the *Source* folder of the *Delphi2Cpp*installation. This file is a contribution from Daniel Flower. The set type "System::Set" can be renamed to TSet, be means of the list of substitutions of the translator.

The use of set's is translated as follows:

```
MySet: set of 'a'..'z';

->

System::Set <  char/* range 'a'..'z'*/, 97, 122 > MySet;

or

type TIntSet = set of 1..250;

->

typedef System::Set <  int/* range 1..250*/, 1, 250 > TIntSet;
```

If there is no explicit type-declaration of a set, as e.g. in:

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

a helping macro and a helping type is created:

```
typedef System::Set <  char, 97, 122 > test__0;
#define test__1 ( test__0 ()
   << char ( 97 ) << char ( 98 ) << char ( 99 ) )

MySet = test__1;
```

The names of such helping types can be adjusted to according names in the C++ Builder VCL by means of the list of substitutions of the translator.
If a temporary *set* of values is passed as open array parameter to a function, a corresponding array is produced in the C++ output, which is put in front of the function call.

7.4.6 Order of type definitions

In Delphi types can be defined by other types, which aren't defined yet. In C++ a type only can be defined by another type, which is already defined. So the order of the Delphi type definitions has to be rearranged sometimes.

The following example is taken from the ShellApi.pas:

```
PSHFileOpStructA = ^TSHFileOpStructA;
PSHFileOpStructW = ^TSHFileOpStructW;
PSHFileOpStruct = PSHFileOpStructA;
{$EXTERNALSYM _SHFILEOPSTRUCTA}
_SHFILEOPSTRUCTA = packed record
  Wnd: HWND;
  wFunc: UINT;
  pFrom: PAnsiChar;
  pTo: PAnsiChar;
  fFlags: FILEOP_FLAGS;
  fAnyOperationsAborted: BOOL;
  hNameMappings: Pointer;
  lpszProgressTitle: PAnsiChar; { only used if FOF_SIMPLEPROGRESS }
end;
{$EXTERNALSYM _SHFILEOPSTRUCTW}
_SHFILEOPSTRUCTW = packed record
  Wnd: HWND;
  wFunc: UINT;
  pFrom: PWideChar;
  pTo: PWideChar;
  fFlags: FILEOP_FLAGS;
  fAnyOperationsAborted: BOOL;
  hNameMappings: Pointer;
  lpszProgressTitle: PWideChar; { only used if FOF_SIMPLEPROGRESS }
end;
{$EXTERNALSYM _SHFILEOPSTRUCT}
_SHFILEOPSTRUCT = _SHFILEOPSTRUCTA;
TSHFileOpStructA = _SHFILEOPSTRUCTA;
TSHFileOpStructW = _SHFILEOPSTRUCTW;
TSHFileOpStruct = TSHFileOpStructA;
{$EXTERNALSYM SHFILEOPSTRUCTA}
SHFILEOPSTRUCTA = _SHFILEOPSTRUCTA;
{$EXTERNALSYM SHFILEOPSTRUCTW}
SHFILEOPSTRUCTW = _SHFILEOPSTRUCTW;
{$EXTERNALSYM SHFILEOPSTRUCT}
SHFILEOPSTRUCT = SHFILEOPSTRUCTA;
```

This is translated to

```
/*# waiting for definiens
typedef TSHFileOpStructA *PSHFileOpStructA;
*/ /*# waiting for definiens
typedef TSHFileOpStructW *PSHFileOpStructW;
*/ /*# waiting for definiens
typedef PSHFileOpStructA PSHFileOpStruct;
*/
  /*$EXTERNALSYM _SHFILEOPSTRUCTA*/

#pragma pack(push, 1)
struct _SHFILEOPSTRUCTA {
  HWND Wnd;
  UINT wFunc;
  PAnsiChar pFrom;
  PAnsiChar pTo;
  FILEOP_FLAGS fFlags;
  BOOL fAnyOperationsAborted;
  void* hNameMappings;
  PAnsiChar lpszProgressTitle; /* only used if FOF_SIMPLEPROGRESS */
};
#pragma pack(pop);
```

```
  /*$EXTERNALSYM _SHFILEOPSTRUCTW*/


#pragma pack(push, 1)
struct _SHFILEOPSTRUCTW {
  HWND Wnd;
  UINT wFunc;
  PWideChar pFrom;
  PWideChar pTo;
  FILEOP_FLAGS fFlags;
  BOOL fAnyOperationsAborted;
  void* hNameMappings;
  PWideChar lpszProgressTitle; /* only used if FOF_SIMPLEPROGRESS */
};
#pragma pack(pop);
  /*$EXTERNALSYM _SHFILEOPSTRUCT*/


typedef _SHFILEOPSTRUCTA _SHFILEOPSTRUCT;
typedef TSHFileOpStructA *PSHFileOpStructA;
typedef PSHFileOpStructA PSHFileOpStruct;
typedef _SHFILEOPSTRUCTA TSHFileOpStructA;
typedef TSHFileOpStructW *PSHFileOpStructW;
typedef _SHFILEOPSTRUCTW TSHFileOpStructW;
typedef TSHFileOpStructA TSHFileOpStruct;
  /*$EXTERNALSYM SHFILEOPSTRUCTA*/
typedef _SHFILEOPSTRUCTA SHFILEOPSTRUCTA;
  /*$EXTERNALSYM SHFILEOPSTRUCTW*/
typedef _SHFILEOPSTRUCTW SHFILEOPSTRUCTW;
  /*$EXTERNALSYM SHFILEOPSTRUCT*/
typedef SHFILEOPSTRUCTA SHFILEOPSTRUCT;
```

7.4.7 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 it 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 a pointer to an integer:

```
uses aunit,  //  PType = ^TestRecord;
     bunit;  //  PType = ^Integer;


Type MyType = PType;
```

In C++ however both definitions of *PType* will be looked up and the code will not compile, because of the ambiguity. Even worse, if for example *bunit* would include *cunit* with another definition of *PType*, C ++ would lookup this definition too. In C++ therefore the ambiguity has to be resolved with the correct namespace:

```
#include "aunit.h"    //  PType = ^TestRecord;
#include "bunit.h"  //  PType = ^Integer;

typedef bunit::PType MyType;
```

DelphiXE2Cpp11 inserts the correct scope expression automatically.

7.4.8 API Integration

API (= Application-Programming-Interface) commands and types are often used in Delphi code. API's are always written in programming languages other than Delphi and different types and routines are defined in different API's. Type definitions in different languages and API's are standardized by conditional compilation and by use of the additional directives $HPPEMIT, $EXTERNALSYM, $NODEFINE and $NOINCLUDE. This often makes it difficult for the human reader to see what actually defines a type. Some examples shall be presented here.

BOOL
DWORD
PByte
THandle

7.4.8.1 BOOL

BOOL is an example for the API integration. *BOOL* is defined in *Winapi,Windows.pas* as:

```
{$EXTERNALSYM DWORD}
BOOL = LongBool;
```

Because of the EXTERNALSYM directive, the definition is omitted in the C++ output (in the standard case, that the EXTERNALSYM directive is applied)..When BOOL is used anywhere in the code, the original definition of the Windows API is used.

It is recommended to suppress the namespace for API files. Otherwise BOOL would be qualified in the created C++ headers as

```
  Wainapi::Windows::BOOL
```

7.4.8.2 DWORD

DWORD is an example for the API integration. *DWORD* is defined in *Winapi,Windows.pas* as:

```
DWORD = System.Types.DWORD;
{$EXTERNALSYM DWORD}
```

and in System.Types.pas it is defined as:

```
DWORD = FixedUInt;
{$EXTERNALSYM DWORD}
```

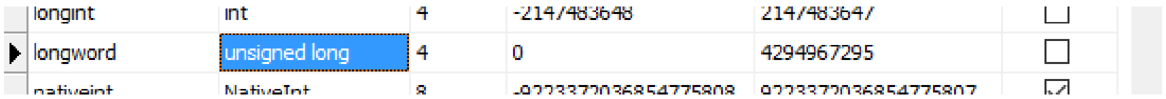
Because in both cases the EXTERNALSYM directive is applied, these definitions are ignored and *DWORD* simply remains *DWORD* at the translation to C++. Indeed *DWORD* is defined in the Windows API in *minwindef.h* as:

```
typedef unsigned long       DWORD;
```

It's not necessary to know, that *FixedUInt* is defined in System.pas for Windows as

```
FixedUInt = LongWord;
```

In the type-map you can see:



longint	int	4	-2147483648	2147483647	☐
longword	unsigned long	4	0	4294967295	☐
nativeint	NativeInt	8	-9223372036854775808	9223372036854775807	☑

7.4.8.3 PByte

PByte is an example for the API integration. PByte is defined in *Winapi,Windows.pas* as:

```
PByte = System.Types.PByte;
```

and in System.Types.pas it is defined as:

```
PByte = System.PByte;
{$EXTERNALSYM PByte}
```

In System.pas it is defined as:

```
PByte         = ^Byte;        {NODEFINE PByte}       { defined in sysmac.h }
```

Here the NODEFINE directive applies. For C++Builder *PByte* is defined in *symac.h* as

```
  typedef Byte*                PByte;
```

For other compilers than C++Builder the definitions are incongruent. The definition in Winapi.Windows.pas becomes in C++ to:

```
namespace Winapi {
  namespace Windows {

typedef System::Types::PByte PByte;
```

But the automated translation would ignore the EXTERNALSYM in System.Types.pas. Therefore this definition is inserted manually into System.Types.h,

```
   typedef System::PByte PByte;
```

In System.h, the definition has to exist:

```
typedef unsigned char* PByte;
```

Indeed the last definition is inserted two times in System.h: one time inside of the namespace System and one time in the global namespace. The latter definition is needed, because it is recommended to suppress namespaces for API headers. When the namespace "Winapi.Windows" is ignored, PByte has to exist

Remark:

FFor C++Builder Winapi.Windows.hpp defines:

```
using System::PByte;
```

But an explicit reference to System::Types::PByte then fails.

7.4.8.4 THandle

THandle is an example for the API integration. *THandle* is defined in *Winapi,Windows.pas* as:

```
    THandle = System.THandle;
```

System.pas defines:

```
  THandle = NativeUInt;
  {$NODEFINE THandle}
```

The *NODEFINE* directive is applied here, because for C++Builder there is a definition in *Winapi. Windows.hpp*:

```
    typedef NativeUInt THandle;
```

For other compilers in *Winapi.Windows.h* this definition exists.

```
   typedef System::THandle THandle;
```

But instead of defining it in Windows.h as NativeUInt, it si defined there as:

```
   typedef HANDLE THandle;
```

Handle is defined in winnt.h as "void*". (If *THandle* were defined as NativeUInt = uint64_t.

HINSTANCE could not be assigned to it without cast.)

Like *PByte* also *THandle* is defined twice: one time inside of the System namespace and a second time as a global type. Therefore the translation works as well with API namespaces as with suppressed API namespaces;

7.5 Variables

In Delphi declarations of variables in 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

7.5.1 threadvars

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

```
threadvar
 x: Integer;
```

C++Builder as well as gcc have an according keyword *__thread:*:

```
int __thread x;
```

Visual C++ uses:

```
declspec(thread) int x;
```

7.5.2 Resource strings

Delphi compiler has built-in support for resource strings whereas in C++ you have to edit resource files manually and insert them into your project. If a project is prepared in that manner the resource strings can be loaded either by the functions *LoadStr* and *FmtLoadStr* of the unit *Sysutils* or by the function *LoadResourceString* in the *System* unit. The latter function is used in C++Builder, when it includes Delphi files with resource strings. The first approach of Delphi2Cpp was, to use this method too. But it has proved to be too complicated, because it needs instances of *ResourceString* structures with a

pointers to the module handles of the modules, where the strings are included.
DelphiXE2Cpp11 simply declares resource strings as normal strings:

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

then the translated code will be:

```
    const System::Char SIndexError[] = L"Index out of bounds: %d";
```

7.6 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 is-operator and the in-operator have to be substituted in special ways.

Also operator overloading has a different syntax.

7.6.1 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 " &".

7.6.2 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++ on 11 levels.

level **operators**

1.	(address) & ! ~	// dereference *, unarary + -
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 )
```

7.6.3 is-operator

In C++ test with *dynamic_cast* corresponds to the is operator for the dynamic type check in Delphi.

```
ActiveControl is TEdit
->
std::dynamic_cast<TEdit*>(ActiveControl)
```

If the overwritten System.pas is used, the is-operator is substituted by the macro, *ObjectIs* :

```
ObjectIs( ActiveControl, TEdit* )
```

ObjectIs is defines as:

```
#define ObjectIs(xObj, xIs) dynamic_cast< xIs >( xObj )
```

If a VCL class is tested for a Meta-class, the translated code looks like:

```
Obj->ClassNameIs( targetClass->ClassName() )
```

7.6.4 in-operator

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

7.7 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, especially for void pointers or
special assignment routines

are necessary in C++.

7.7.1 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:

```
  signed char si = 0;
  int i = 0;
  i = si;
  si = (signed char) i;
```

DelphiXE2Cpp11 always inserts the according casts, also when such casts are necessary to pass parameters to functions. Especially such casts often are necessary for void pointers.

7.7.2 void pointer casts

In Delphi frequently void pointers are casted to specific pointer types. C++ compilers produce error messages here, if the cast isn't made explicitly. *DelphiXE2Cpp11* automatically inserts according cast's to avoid such error messages. E.g.

```
var
  a : Pointer;
  b : PInteger;
begin
  b := a;
```

->

```
void *a;
PInteger b;
```

```
b = (PInteger) a;
```

An according cast takes place, if a pointer to another type is expected as parameter in a function call.

```
List.Add(Item, Pointer(1));
```

->

```
List->Add( Item, (TObject*) ((void*) 1 ) );
```

7.7.3 Special assignments

In Delphi the contents of array variables of the same type can be assigned directly. In C++ the assignment has to be done via pointers to the first array element by means of the functions "strcpy" or "memcpy":

Assignments to character arrays is done with "strcpy".

```
var
 chr10 : array[1..10] of char;
begin
 chr10 := 'abcdefghij';
```

->

```
char chr10[ 10/*# range 1..10*/ ];
strcpy( chr10, "abcdefghij" );
```

Assignments of other static arrays are done with "memcpy".

```
procedure test(xArr: TObjectArray);
var
  arr: TObjectArray;
begin
  arr := xArr;
end;
```

->

```
void __fastcall test( const TObjectArray& xArr )
{
  TObjectArray arr;
  memcpy( arr, xArr, sizeof( TObjectArray ) );
}
```

7.8 Routines

There are two kinds of routines in Delphi: procedures and functions.

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 are reproduced in C++11.

7.8.1 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 implicitely 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;                               bar result;
  if i < 0  then                             result = 0;
    EXIT                                     if ( i < 0 )
  else                                         return result;
    Result := 1;                             else
end;                                           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;
```

7.8.2 Parameter types

Parameters either are passed to routines by value or be 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: Different cases of single parameters and how they are converted are listed below. Array parameters are discussed in the array section.

```
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);
```

->

->

```
C++Builder                                          Other compilers

 void __fastcall Foo(int param);                     void Foo(int param);
 void __fastcall Foo(int param);                     void Foo(int param);
 void __fastcall Foo(int& param);                    void Foo(int& param);
 void __fastcall Foo(int& param);                    void Foo(int& param);
 void __fastcall Foo(System::String param);          void Foo(System::String param);
 void __fastcall Foo(const System::String& param);   void Foo(const System::String& param);
 void __fastcall Foo(System::String& param);         void Foo(System::String& param);
 void __fastcall Foo(System::String& param);         void Foo(System::String& param);
 void __fastcall Foo(void* param);                   void Foo(void* param);
 void __fastcall Foo(const void* param);             void Foo(const void* param);
 void __fastcall Foo(void*& param);                  void Foo(void*& param);
 void __fastcall Foo(void*& param);                  void Foo(void*& param);
 void __fastcall Foo(System::PChar param);           void Foo(System::PChar param);
 void __fastcall Foo(const System::PChar param);     void Foo(const System::PChar param);
 void __fastcall Foo(System::PChar& param);          void Foo(System::PChar& param);
 void __fastcall Foo(System::PChar& param);          void Foo(System::PChar& param);
 void __fastcall Foo(PInteger param);                void Foo(PInteger param);
 void __fastcall Foo(const PInteger param);          void Foo(const PInteger param);
 void __fastcall Foo(PInteger& param);               void Foo(PInteger& param);
 void __fastcall Foo(PInteger& param);               void Foo(PInteger& param);
 void __fastcall Foo(const MyRecord& param);         void Foo(const MyRecord& param);
 void __fastcall Foo(const MyRecord& param);         void Foo(const MyRecord& param);
 void __fastcall Foo(MyRecord& param);               void Foo(MyRecord& param);
 void __fastcall Foo(MyRecord& param);               void Foo(MyRecord& param);

 // untyped parameters                               // untyped parameters
 void __fastcall Foo(const void* param);             void Foo(const void* param);
 void __fastcall Foo(void** param);                  void Foo(void** param);
 void __fastcall Foo(void** param);                  void Foo(void** param);
```

Passing a record by value is a special case. The record may be changed inside of the routine, but the change may not have an effect outside of the routine. Therefore Delphi2Cpp passes the parameter as a constant reference, but automatically creates a copy of the parameter inside of the routine. The original parameter is renames to avoid a conflict. E.g.:

```
void Foo(const MyRecord& cparam)
{
  MyRecord param = cparam;
```

There is a problem with var pointer parameters. If a pointer of a special type is passed the C++ compiler will produce an error. For example:

```
void* ReallocMem(void*& P, size_t Size);

char* buf;

ReallocMem(buf, 10); // error: conversion from char* into "void *&" isn't possible
```

That's the reason, why C++Builder doesn't knows *ReallocMem*, but only *ReallocMemory*:

```
void * __cdecl ReallocMemory(void * P, NativeInt Size);
```

A good solution for *Delphi2Cpp* would be to define ReallocMem as a template function like:

```
template <typename T>
void ReallocMem(T*& P, size_t Size)
{
  ...
```

But this could be a solution for special functions of the RTL/VCL only. Non-template user routines hardly can be converted into routines with templates, because this would require to move them together with their implementations into the header. Therefore the solution above has been chosen for *Delphi2Cpp*. In cases where such routines are used, *Delphi2Cpp* automatically inserts a typecast for the parameter:

```
ReallocMem((void*&) Buf, 10);
```

Untyped var-parameters are converted to void** parameters. An address is passed as argument and inside of the routine the parameter is dereferenced.

7.8.3 Adaption of parameters

When parameters are passed to functions in the Delphi source code, the translator tries to match the signature of the function with the type of the variable which is passed. The function call:

```
Print(a);
```

might be translated as one of the following alternatives:

```
Print( a );
Print( &a );
Print( a.c_str() );
```

E.g. the signature of *Print* might be:

```
procedure Print(const Buffer);
```

and the parameters might be of the type *Integer* or *void** or *String*.

7.8.4 Temporary variables

In Delphi it is possible to pass combinations of string literals with strings as parameters like in the following example:

```
function Greet(Msg : PChar): Boolean;
begin
  // doing something with Msg
end;

procedure GreetSomeone(Name : String);
begin
 if Greet(PChar('hello ' + Name + '!')) then
   Exit;
 ...
end;
```

In C++ a string literal can be added to a string, but not the other way round. In such cases Delphi2Cpp automatically creates a temporary string from the string literal to which the following strings and string literals can be added, like:

```
String( "hello " ) + Name + "!";
```

To make a character pointer from this construct, another temporary string would have to be created, like:

```
String(String( "hello " ) + Name + "!").c_str();
```

But, if such a construct would be passed to a function like:

```
bool __fastcall Greet( char* Msg )
{
  // doing something with Msg
}
```

the resulting character pointer is destroyed as soon as the destructors of the temporary strings is executed. So, inside of the body of the called function, the character pointer isn't valid any more. Therefore a temporary variable is created and enclosed into a block together with the statement of the function call:

```
void __fastcall GreetSomeone( String Name )
{
  {
    AnsiString Str__0 = AnsiString( "hello " ) + Name + "!";
    if ( Greet( Str__0.c_str( ) ) )
      return;;
  }
  ...
}
```

In a similar way temporary variables are constructed for temporary array parameters:

```
procedure Log(strings : array of String);

Log(['one', 'two', 'three']);
```

This becomes to:

```
void __fastcall Log( const String* strings, int strings_maxidx )

{
  String tmp__0[ 3 ];
  tmp__0[ 0 ] = "one";
  tmp__0[ 1 ] = "two";
  tmp__0[ 2 ] = "three";
  Log( tmp__0, 3 );
}
```

A special case is "array of const". This case is handled by a macro.
If a function has a set-Parameter, temporary sets are constructed in the C++ translation by means of a definition.

7.8.5 Calls of inherited procedures and functions

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

```
class foo: public bar {
  typedef bar inherited;
```

So, if in the Delphi code an inherited routine is called by the identifier "inherited" followed by the name of the routine, it can be translated easily to C++ accordingly.

```
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 );
}
```

7.8.6 Nested routines

There aren't nested functions in C++.

with C++11 they can be simulated elegantly by use of lambda-functions.
with C++98 inner functions are replace by new memberfunctions

7.8.6.1 Nested routines with C++11

There aren't nested functions in C++, but they can be simulated by use of C++11 lambda-functions.

```
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;
 NestedTest1(3, 4);
 result := result + iTwiceParam;
end;
```

->

```
class TNested : public System::TObject
{
  typedef System::TObject inherited;
public:
  int iClassVar;
  int Test(int iOuterParam, int iTwiceParam, System::String s);
  void InitMembers(){iClassVar = 0;}
public:
  TNested() {InitMembers();}
};

//---------------------------------------------------------------------------
int TNested::Test(int iOuterParam, int iTwiceParam, String s)
{
  int result = 0;
  const DWideChar cSeparate = _T(':');
  int iFunctionVar = 0;
//---------------------------------------------------------------------------
  auto NestedTest = [&](int iInnerParam, int iTwiceParam) -> void
  {
    result = iClassVar + iOuterParam + iFunctionVar + iInnerParam + iTwiceParam;
  };
  iClassVar = 1;
  iFunctionVar = 2;
  NestedTest1(3, 4);
  result = result + iTwiceParam;
  return result;
}
```

Like nested routines in Delphi lambda functions can access variables from the outer scope. The capture clause [&] ensures that access is via reference.

In the special case, that a sub-routine is called reflexively, the auto variable cannot be used. (VisualC produces the error C2064: term does not evaluate to a function taking N arguments. "The expression does not evaluate to a pointer to a function"). DelphiXE2C11 generates an explicit forward declaration in such cases. For example:

```
function nested : boolean;

  function nested_reflexive(depth :Integer) : boolean;
  begin
```

```
        if depth = 2 then
          result := true
         else
          result := nested_reflexive(depth + 1);
      end;


    begin
      result := nested_reflexive(0);
    end;
```

->

```
function<bool (int)> nested_reflexive;


bool __fastcall nested()
{
  bool result = false;

  nested_reflexive = [&](int depth) -> bool
  {
    bool result = false;
    if(depth == 2)
      result = true;
    else
      result = nested_reflexive(depth + 1);
    return result;
  };
  result = nested_reflexive(0);
  return result;
}
```

This compiles and works well.

7.8.6.2 Nested routines with C++98

There aren't nested functions in C++. For C++98 in contrast to C++11 the automatic translation of nested Delphi functions replaces the inner functions by new free functions or member functions. The parameters and the declared variables of the outer function are passed to these new functions.

```
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;
 NestedTest1(3, 4);
 result := result + iTwiceParam;
end;
```

->

```
class TNested : public System::TObject
{
  #include "Test_friends.inc"
public:
  int iClassVar;
  void __fastcall NestedTest(int iInnerParam, int iTwiceParam, int& result, int& iOuterParam, int& iFun
  int __fastcall Test(int iOuterParam, int iTwiceParam, System::String s);
};


void __fastcall TNested::NestedTest(int iInnerParam, int iTwiceParam, int& result, int& iOuterParam, in
{
  result = iClassVar + iOuterParam + iFunctionVar + iInnerParam + iTwiceParam;
};

int __fastcall TNested::Test(int iOuterParam, int iTwiceParam, String s)
{
  int result = 0;
  const WideChar cSeparate = L':';
  int iFunctionVar = 0;
  iClassVar = 1;
  iFunctionVar = 2;
  NestedTest(3, 4, result, iOuterParam, iFunctionVar);
  result = result + iTwiceParam;
  return result;
}
```

It is taken into account that at multiple nesting possible parameters which aren't needed in a function itself but in a function subordinated to it are passed through.

In contrast to the solution with lambda functions however, there may be difficulties with undeclared types at the declaration of the inner functions.

Other possibilities to translate nested functions are discussed here:

http://www.gotw.ca/gotw/058.htm

7.9 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 *Delphi2Cpp*. 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 leas
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..

see also: RegisterComponents

7.9.1 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 f*ree pascal FCL*. It is contained in the files *d2c_sysfile.h and d2c_sysfile.cpp* in the source folder of the Delphi2Cpp 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.

7.9.2 Read(Ln)/Write(Ln) routines

The *Read(Ln)/Write(Ln)* routines can be called in *Delphi* with an arbitrary number of arguments. Delphi2Cpp divides them into a series of function calls:

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

becomes:

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

7.9.3 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 *Delphi2Cpp*.

7.9.4 RegisterComponents

Since components are an important feature of Delphi, a special translation routine was made for their registration in C++Builder too.

```
RegisterComponents('NewPage',[TCustom1, TCustom2]);
->
TComponentClass classes[2] = {__classid(TCustom1), __classid(TCustom2)};
RegisterComponents("NewPage", classes, 1);
```

For other compilers this function is useless.

7.10 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.

7.10.1 Field properties

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;
```

C++Builder

For C++Builder the"__property" key word is a counterpart to the Delphi properties. With that the code snippet above becomes to:

```
typedef int /*0..359*/ THeading;

class TCompass : public TControl
{
  typedef TControl inherited;

private:
  THeading FHeading;
  void __fastcall SetHeading(THeading Value);
__published:
  __property THeading Heading = { read = FHeading, write = SetHeading };
       // ...
};
```

Visual C++

Visual C++ also has compiler specific properties:

```
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

While in Delphi and for C++Builder the field can be set simply after the read or write specifier, Visual C++ needs functions for for the corresponding get and put specifiers. If in the original Delphi code a field is used, Delphi2Cpp creates an according function for it, as described for other compilers below. Such functions also are created, if the original access function is private - as in the example - or protected or if the type of the property is an array and for indexed properties.

```
class TCompass : public TControl
{
  typedef TControl inherited;

private:
  THeading FHeading;
  void SetHeading(THeading Value);
public:
  /*property Heading : THeading read FHeading write SetHeading;*/
  THeading ReadPropertyHeading() { return FHeading;}
  void WritePropertyHeading(THeading Value){SetHeading(Value);}
  __declspec(property(get = ReadPropertyHeading, put = WritePropertyHeading)) THeading Heading;
       // ...
};
```

In Visual C++ base class properties can be used in derived classes too, which in Delphi has to be declared explicitly.

Other compilers

For other compilers.properties are eliminated. The read and write specifications are replaced by two functions whose names are derived from the name of the original property. As default the expression *ReadProperty* or *WriteProperty* is put in front of this name respectively. You can change these prefixes in the option dialog.

```
class TCompass : public TControl
{
  typedef TControl inherited;

private:
  THeading FHeading;
  void SetHeading(THeading Value);
public:
  /*property Heading : THeading read FHeading write SetHeading;*/
  THeading ReadPropertyHeading() { return FHeading;}
  void WritePropertyHeading(THeading Value){SetHeading(Value);}
       // ...
};
```

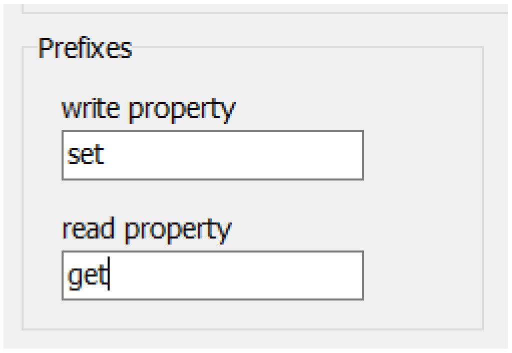
The fields or methods, which originally were set in the property are now accessed via these functions. While the visibility of these fields or methods usually is private or protected, the access functions which are created by Delphi2Cpp 2.x are public. In the *ReadProperty* function the originally field is returned or a call of the original return function is carried out. In the *WriteProperty* function the assignment to the original field is carried out and the parameters are passed to the originally method.

At all places in the remaining code where a property is read, the *ReadProperty* function is used and the *WriteProperty* function is called in all places, where originally a value is assigned to a property.

```
  if Compass.Heading = 180 then GoingSouth;
    Compass.Heading := 135;
->
  if(Compass->ReadPropertyHeading() == 180)
    GoingSouth();
  Compass->WritePropertyHeading(135);
```

7.10.1.1 Changing the property prefixes

For other compilers than C++Builder Delphi properties are replaced by a pair methods. If the default prefixes *ReadProperty* and *WriteProperty* are left, then it is very unlikely that there will be conflicts with existing names in the code. The situation is different when these prefixes are changed to preferred prefixes such as "get" and "set".


Prefixes

write property

set

read property

get

The first thing to note, however, is that the supplied C++ code for the Delphi RTL must also be adapted. This can be done simply by searching and replacing.

For the generation of the property code several cases have to be distinguished.

Let's assume that in the call the following methods are defined already:

```
      function getName: String;
      procedure setIdent(AIdent: String);
      function getValue: String;
      procedure setValue(AValue: String);
      function getNote: String;
      procedure setNote(ANote: String);
```

1. There is no problem for for following property:

```
      property Caption : String read FCaption write FCaption;
    ->
      System::String& getCaption() {return FCaption;}
      void setCaption(const System::String& Value){FCaption = Value;}
```

the newly created methods *getCaption* and *setCaption* didn't exist yet.

2. For the next property Name Delphi2Cpp will reuse the existing method *getName*, which also was used in the original Delphi code. Only the *setName* method has to be created newly.

```
      property Name : String read getName write FName;
    ->
      void setName(const System::String& Value){FName = Value;}
```

Accordingly for:

```
     property Ident : String read FIdent write setIdent;
    ->
     System::String& getIdent() {return FIdent;}
```

3: No new method has to be created, if all access methods exist already and they were used in the original Delphi code too:

```
     property Value : String read getValue write setValue;
    ->
```

4. A problem arises, when the getter and setter methods that Delphi2Cpp generates exist already, but are not used in the original Delphi code:

```
      property Note : String read FNote write FNote;
    ->
      System::String& getNote() {return FNote;}
      void setNote(const System::String& Value){FNote = Value;}
```

In this case Delphi2Cpp writes a warnings into the output, like:

```
//# conflict with existing procedure name
or
//# conflict with existing function name
```

By means of the list of identifiers either the case of the name of the existing method can be changed or the case of the property might be change, so that there will be no naming conflict any more.

7.10.2 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;

->
```

C++Builder

```
class TRectangle : public TObject
{
  typedef TObject inherited;

private:
  int fCoords[4/*# range 0..3*/];
  int __fastcall GetCoord(int Index);
  void __fastcall SetCoord(int Index, int Value);
public:
  __property int Left = { index = 0, read = GetCoord, write = SetCoord };
  __property int Top = { index = 1, read = GetCoord, write = SetCoord };
  __property int Right = { index = 2, read = GetCoord, write = SetCoord };
  __property int Bottom = { index = 3, read = GetCoord, write = SetCoord };
public:
  __fastcall TRectangle() {}
};
```

Other compilers

```
class TRectangle : public TObject
```

```
{
  typedef TObject inherited;

private:
  int fCoords[4/*# range 0..3*/];
  int GetCoord(int Index);
  void SetCoord(int Index, int Value);
public:
  /*property Left : int read GetCoord write SetCoord;*/
  int ReadPropertyLeft() { return GetCoord(0);}
  void WritePropertyLeft(int Value){SetCoord(0, Value);}
  /*property Top : int read GetCoord write SetCoord;*/
  int ReadPropertyTop() { return GetCoord(1);}
  void WritePropertyTop(int Value){SetCoord(1, Value);}
  /*property Right : int read GetCoord write SetCoord;*/
  int ReadPropertyRight() { return GetCoord(2);}
  void WritePropertyRight(int Value){SetCoord(2, Value);}
  /*property Bottom : int read GetCoord write SetCoord;*/
  int ReadPropertyBottom() { return GetCoord(3);}
  void WritePropertyBottom(int Value){SetCoord(3, Value);}
public:
  TRectangle() {}
};
```

Again there is a simplified notation for C++Builder, while for other compilers only published Access methods can be created

```
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;


->
```

C++Builder

```
class TRectangle : public TObject
{
  typedef TObject inherited;

private:
  int fCoords[4/*# range 0..3*/];
  int __fastcall GetCoord(int Index);
  void __fastcall SetCoord(int Index, int Value);
public:
  __property int Coords[int Index] = { read = GetCoord, write = SetCoord };
public:
  __fastcall TRectangle() {}
};
```

Other compilers

```
class TRectangle : public TObject
{
  typedef TObject inherited;

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

7.10.3 Default array-property

If a class has a default property, you can access that property in Object-Pascal with the abbreviation object[index], which is equivalent to object.property[index]. For C++Builder the translated code looks like:

```
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;

->
```

C++Builder

```
class TRectangle : public TObject
{
  typedef TObject inherited;

private:
  int fCoords[4/*# range 0..3*/];
  int __fastcall GetCoord(int Index);
  void __fastcall SetCoord(int Index, int Value);
public:
  __property int Coords[int Index] = { read = GetCoord, write = SetCoord/*# default */ };
public:
  __fastcall TRectangle() {}
};
```

Other compilers

```
class TRectangle : public TObject
{
  typedef TObject inherited;

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

If there is an instance ot TRectangle the array can be accessed in Delphi simply by rect [i]. For C++Builder this becomes to:

```
   rect->Coords[i]
```

and for other compilers:

```
     rect->WritePropertyCoords(i, 0);
     ... = rect->ReadPropertyCoords(i);
```

7.10.4 Array property

As arrays cannot be returned by functions in C++ in contrast to Delphi, arrays may not be properties in C++ in contrast to Delphi. If there is such a property in Delphi it will be converted to according getter or setter functions in C++. The following code uses the same array *TObjectArray* and the same function *CreateArray* which are defined for the previous example for returned arrays.

```
TArrayClass = class
private
  FArray : TObjectArray;
public

  property Arr : TObjectArray read FArray write FArray;
end;

->

class TArrayClass : public TObject
{
  typedef TObject inherited;
private:
  TObjectArray FArray;
public:
  /*property arr : TObjectArray read FArray write FArray;*/
  TObjectArray& ReadPropertyarr(TObjectArray& result) const {ArrAssign<3>(result, FArray); return result;}
  void WritePropertyarr(TObjectArray& Value){ArrAssign<3>(FArray, Value);}
};
```

ArrAssign is the common name of some template functions, which assign arrays to each other. The template parameter <3> specifies, that the arrays have three elements in one dimension.

In the following Test3 function the array of the class is initialized by means of the *CreateArray* function:

```
procedure Test3;
var
  C : TArrayClass;
begin
  C := TArrayClass.Create;
  C.arr := CreateArray;
end;
```

In the C++ translation an additional `TObjectArray` is needed,wich is passed to the *CreateArray* funtion at first. There the elements of the array are initialized. Finally the array is is returned from *CreateArray* and becomes the array parameter of the writer function of the property.

```
void Test3()
{
  TArrayClass* C = NULL;
  C = new TArrayClass();
  TObjectArray arrayproperty__0; C->WritePropertyarr(CreateArray(arrayproperty__0, uniquetype()));
}
```

7.11 Statements

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

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

7.11.1 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;
}
```

Delphi2Cpp can produce both code variants, depending on the option to *Use "stop" variable in for-loop* or not..

Delphi2Cpp also checks the type of the loop variable to avoid a sublime error.

7.11.1.1 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'. These cases mostly are translated to a C++11 range-based for loop:

```
typename a;

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

For C++Builder, in the special case, that 'B' is an open array, a cast is necessary. That may for example look like:

```
void __fastcall ArrayOfConstLoop(const T* B, int B_maxidx)
{
  T a;

  for(auto element_0 : *(T(*)[B_maxidx])B)
  {
    a = element_0;
    DoSomething(a);
  }
}
```

The necessary iterators for sets and open arrays are defined in d2c_systypes.h.

For container types that implement a GetEnumerator() method the for loop looks like:

```
    while(B->GetEnumerator()->MoveNext())
    {
            T a = B->GetEnumerator()->Current
            DoSomething(a);
    }
```

7.11.1.2 loop variable

The variable that is incremented in a for loop is declared like any other variable (before Delphi 10.4) at the beginning of the function body. However, converting this declaration to C++, like the other variables, can lead to a sublime error. This is demonstrated in the following example:

```
function ToHigh: boolean;
```

```
var
  C: WideChar;
  I: Integer;
begin
  I := 0;
  for C := Low(WideChar) to High(WideChar) do
    I := Integer(C);
  result := I = Integer(High(WideChar));   // 65535
end;
```

The straight forward translation of this code - without use of a stop variable - results in:

```
bool __fastcall ToHigh()
{
  bool result = false;
  WideChar C = L'\0';       //  <=  wrong type
  int I = 0;
  I = 0;
  for(C = 0 /*# Low(WideChar) */; C <= 65535 /*# High(WideChar) */; C++)
  {
    I = ((int) C);
  }
  result = I == 65535 /*# High(WideChar) */;   // 65535
  return result;
}
```

However, executing this code will result in an infinite loop, because in C++ the loop is only broken after the loop variable has a value higher than the maximum value of a wide character variable "High (WideChar)". Therefore the type of the loop variable must be changed so that it can get this value.

```
bool __fastcall ToHigh()
{
  bool result = false;
  int C = 0;               //  <=  corrected type
  int I = 0;
  int stop = 0;
  I = 0;
  for(stop = 65535 /*# High(WideChar) */, C = 0 /*# Low(WideChar) */; C <= stop; C++)
  {
    I = C;
  }
  result = I == 65535 /*# High(WideChar) */;   // 65535
  return result;
}
```

Delphi2Cpp checks this case and automatically changes the variable type during code translation; also accordingly for the "downto" case.

7.11.2 case statements

The translation of Delphi case statements to C++ switch statements mostly is straightforward like:

```
Case colour of
     Red : result := 1;
   Green : result := 2;
    Blue : result := 3;
  Yellow : result := 4;
else result := 0;
end;
```

->

```
  switch(colour)
  {
    case Red:
    result = 1;
    break;
    case Green:
    result = 2;
    break;
    case Blue:
    result = 3;
    break;
    case Yellow:
    result = 4;
    break;
    default:
    result = 0;
    break;
  }
```

In Delphi, not only single constant expressions can be used to define a case, but also lists and subranges of such constants.
Their elements must then be output as separate cases for C++.

```
  Case Key of
    #13, #27:
```

->

```
  switch(Key)
  {
    case L'\x0d':
    case L'\x1b':
```

However, if such subranges are very large (more than 256 elements) Delphi2Cpp moves them into the default section of the C++ switch-statement:

```
  Case Key of
    #32..High(WideChar):
      begin
        ...
      end;
    #8:
      ...
   else
     ...
```

->

```
  switch(Key)
  {
    case L'\x08':
    ...
    break;
    default:
    if(Key >= L'\x20' && Key <= 65535 /*# High(WideChar) */)
    {
      ...
    }
    else
    {
      ...
    }
```

7.11.3 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. C++Builder has an according key word ***__finally*** , which does the same in C++, but this is not a standard keyword. For other compilers finally statements have to be simulated. *Delphi2Cpp* takes a solution which is presented by Craig Scott::

https://crascit.com/2015/06/03/on-leaving-scope-part-2/

By use of the presented **OnLeavingScope** class the translation of a try-finally statement looks as follows:

```
var
  obj : TObject;
begin

try
  obj := TObject.Create(NIL);
  ...
finally
  obj.free;
end;
```

->

```
#include "OnLeavingScope.h"

TObject* Obj = NULL;
{
  auto olsLambda = onLeavingScope([&]
  {
    delete Obj;
  });
  Obj = new TObject(NULL);
}
```

olsLambda is a class, which gets a lambda function as parameter to its constructor. This function is stored internally and gets executed in the destructor of the class. The include `"OnLeavingScope.h" is inserted automatically.`

7.11.4 with-statements

In C++ there are no with-statements. Therefore Delphi2Cpp 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 */
    if Month = 12 then                  {
    begin                                 auto& with0 = OrderDate;
      Month := 1;                         if(with0.Month == 12)
      Year := Year + 1;                   {
    end                                     with0.Month = 1;
    else                                    with0.Year = with0.Year + 1;
      Month := Month + 1;                 }
end;                                      else
                                           with0.Month = with0.Month + 1;
                                        }
                                      }
```

7.11.5 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. When the program is started, the global variables are created at first and therefore also the units are initialized.

```
initialization

pTest := CTest.Create;

finalization

pTest.Free();
```

->

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

void Tests_finalization()
```

```
{
  delete pTest;
}

class test_unit
{
public:
  test_unit()
  {
    test_initialization();
  }
  ~  test_unit(){test_finalization(); }
};
test_unit _test_unit;
```

This however is only the short version. Sometimes the initialization have to be executed in a special order: one initialization can require another. Such cases have to be corrected by the developer, but Delphi2Cpp already generates code that allows these corrections to be made easily. The declarations:

```
void Tests_initialization();
void Tests_finalization();
```

are put into the header and there is an additional static boolean variable, that prevents that the initialization or finalization is executed twice. E.g.:

```
static bool Test_Initialized = false;

void Tests_initialization()
{
  if(Test_Initialized)
    return;
  Test_Initialized = true;
...
```

So in the normal case the initializations and finalizations keep being executed automatically, but the developer also may easily call the according functions from another unit, to force a special order of initialization or finalization.

7.12 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.C++Builder has a special extension, which allows the creation of class references, but the creation of class instances from them isn't possible, only some other class functions can be called from them.

With Delphi2Cpp the code above can be translated. The way of translation is different for C++Builder and other compilers.
A creation of class instances from class references is possible only, if the class has a standard constructor.

Alternatively also the macros *DECLARE_DYNAMIC* and *IMPLEMENT_DYNAMIC might* help.

7.12.1 C++Builder __classid

C++Builder has as counterpart to Delphi's *TClass*:

```
typedef TMetaClass* TClass
```

A Delphi class reference type is defined as *TClass* in C++Builder:

```
type TBaseClass = class of TBase;
```

->

```
typedef System::TMetaClass TBaseClass;
```

Variables of this type can be defined and classes can be assigned to them. For C++Builder the *__classid* function is a special extension, to get class references.

```
var
 p : TBase;
 bc :TBaseClass;
begin
 bc := p;
```

->

```
TBase* p = nullptr;
TBaseClass bc = nullptr;
bc = __classid(p);
```

With such class references code such as:

```
ClassRef := Sender.ClassType;

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

can be translated pretty well as:

```
TClass ClassRef = Sender->ClassType();

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

It is not possible however, to create an instance of a class from such a *TClass*. To do that, a small Delphi unit has to be added to the C++Builder project. The unit CreateClass.pas, which is delivered with Delphi2Cpp contains the simple function:

```
function CreateObject(C: TClass) : TObject;
begin
  Result := C.Create();
end;
```

When this unit is added to a C++Builder project, automatically a C++ header file "CreateClass.hpp" is created with the declaration:

```
extern DELPHI_PACKAGE System::TObject* __fastcall CreateObject(System::TClass C);
```

That function can be used now in the C++ code to create class instances from class references:

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

->

```
TBase* make(TBaseClass* Base)
{
  TBase* result = nullptr;
  result = (TFBase*) CreateObject(Base);
  return result;
}
```

If the class referenc of a class which is derived from TBase is passed to the make-function an instance of that class will be created.

```
p := make(TDerived);:
```

->

```
P = make(__classid(TDerived));
```

.

7.12.2 Other compiler ClassRef

As for C++Builder where a class*TMetaClass* is defined, for other compilers such a class is defined too in the code delivered with Delphi2Cpp. In addition the type *TClass is defined as a pointer to TMetaClass*:

```
typedef TMetaClass* TClass
```

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:

```
template <typename Class>
class ClassRef
```

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. To avoid unwanted side effects at the creation of these classes the variable _CreatingClassInstance can be used. The exact definition of the ClassRef class is tricky and works only, because Delphi2Cpp 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(TDerived);
 result := p.GetName = 'TDerived';
end;
```

->

```
class TBase : public System::TObject
{
public:
  typedef System::ClassRef<TBase> ClassRefType;
  ClassRefType* ClassType() const {return  System::class_id<TBase>();}
  TBase* Create() {return new TBase();}
  static TBase* SCreate() {return new TBase();}
  System::String ClassName() {return L"TBase";}
  static System::String SClassName() {return L"TBase";}

  TBase();
};

typedef TBase::ClassRefType TBaseClass;

class TDerived : public TBase
{
public:
  typedef System::ClassRef<TDerived> ClassRefType;
  ClassRefType* ClassType() const {return  System::class_id<TDerived>();}
  TDerived* Create() {return new TDerived();}
  static TDerived* SCreate() {return new TDerived1();}
  System::String ClassName() {return L"TDerived";}
  static System::String SClassName() {return L"TDerived";}
  TDerived();
};

typedef TDerived::ClassRefType TDerivedClass;
```

```
  TBase* make(TBaseClass* Base)
  {
    TBase* result = nullptr;
    result = Base->Create();
    return result;
  }

  bool testfactory()
  {
    bool result = false;
    String s;
    TBase* P = nullptr;
    P = make(class_id<TDerived>());
    result = P->GetName() == L"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 fulfills the same purpose as the __classid-function in C++Builder code: it 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.

7.12.2.1 _CreatingClassInstance

To simulate class references in C++, a hierarchy of default constructed class instances can be used. However, the construction of these instances may have unwanted side effects or the construction can fail, if certain conditions are not yet met, such as the existence of certain global variables. In such cases the global boolean variable

```
bool _CreatingClassInstance;
```

which is defined in d2c_sysmeta can be used. Before the class reference simulating class instance is constructed, this variable will be set to true automatically and afterwards it will be set to false again. You may modifiy the code for the constructor like

```
XXX::XXX
{
  if (_CreatingClassInstance)
    return;

  ...
}
```

7.13 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; 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).

Delphi2Cpp provides the files d2c_sysfile.h/d2c_sysfile.cpp where these three file types are converted to C++ structures. d2c_sysfile also contains all the Delphi routines converted to C++ that are used to read and write to the console and to files by use of these file types.

d2c_sysfile is derived from the *FreePascal* library:

http://www.freepascal.org/

FreePascal is published under the terms of GNU Lesser General Public License and therefore the same terms apply to *d2c_sysfile*..

7.14 Message handlers

Message handlers are methods that implement responses to dynamically dispatched messages. Delphi’s VCL uses message handlers to respond to Windows messages.

In Delphi a message handler is created by including the message directive in a method declaration, followed by an integer constant between 1 and 49151 which specifies the message ID.

The routine for handling the message can be declared as a macro:

```
#define VCL_MESSAGE_HANDLER(msg,type,meth)          \

          case    msg:                              \
            meth(*((type *)Message));               \
            break;
```

This macros has to be embedded into two other macros:

```
#define BEGIN_MESSAGE_MAP virtual void __fastcall Dispatch(void *Message) \

        {                                           \
          switch  (((PMessage)Message)->Msg)        \
          {


#define END_MESSAGE_MAP(base)    default:    \

                        base::Dispatch(Message);\
                        break;                  \
          }                                     \
        }
```

For example the two message handlers:

```
procedure WMVScroll(var Message: TWMVScroll);
                    Message WM_VSCROLL;
procedure WMHScroll(var Message: TWMHScroll);
                    Message WM_HSCROLL;
```

are translated to C++Builder C++:

```
MESSAGE void __fastcall WMVScroll( TWMVScroll& Message )
             /*# WM_VSCROLL */;
MESSAGE void __fastcall WMHScroll( TWMHScroll& Message )
             /*# WM_HSCROLL */;


BEGIN_MESSAGE_MAP
  VCL_MESSAGE_HANDLER(WM_VSCROLL, TWMVScroll, WMVScroll)
  VCL_MESSAGE_HANDLER(WM_HSCROLL, TWMHScroll, WMHScroll)
END_MESSAGE_MAP( TPanel )
```

7.15 Absolute address

By the word *absolute* a variable can be declared in Delphi that resides at the same address as an existing variable. This behavior is reproduced in C++ by declaring the new variable as a reference to the existing variable. If necessary according typecast's are inserted.

```
var
  Size: Int64;
  SizeRec: TInt64Rec absolute Size;
```

->

```
__int64 Size = 0;
```

```
TInt64Rec& SizeRec = *(TInt64Rec*) &Size;
```

7.16 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;
```

According to the *Delphi* help "a method pointer is really a pair of pointers; the first stores the address of a method, and the second stores a reference to the object the method belongs to". Such method pointers can point to any member functions in any class. For example by means of a method pointer the event handling of a special instance of a control - e.g. *TButton* - can be delegated to the instance of another class - e.g. *TForm* .

Delphi's method pointers cannot be translated as standard C++ member function pointers, because they can point to other member functions of the same inheritance hierarchy only. That's why Borland has extended the standard C++ syntax by the keyword *__closure*. With this keyword method pointers with the same properties as Delphi's method pointers can be declared in Borland C++. E.g. the event above is:

```
typedef void __fastcall (__closure *TNotifyEvent)(TObject* Sender);
```

For other compilers C++Builder closures can be substituted by means of the new standard functions in C++11. The definition of the *`TNotifyEvent`* above then becomes to:

```
typedef std::function<void (TObject*)> TNotifyEvent;
```

A class instance - e.g. TButton* pButton - can be bound to a member function of this signature - e.g. TButton::OnClick - by means of std::bind1st and std::mem_fun:

```
TNotifyEvent ev = std::bind1st(std::mem_fun(&TMyButton::OnClickHandle), pButton);
```

Once a handler is assigned, further operations with the event are looking as simple as in the original *Delphi* code. E.g.:

```
// calling the event
Button1.OnClick(Button1); ->  Button1->OnClick(Button1);

// assigning the event handler to another button
Button2.OnClick = Button1.OnClick; -> Button2->OnClick = Button1->OnClick;
```

Remark: In contrast to Delphi2Cpp the first version of Delphi2Cpp used a similar solution from Tamas Demjen :

http://tweakbits.com/articles/events/index.html

7.17 Libraries

Delphi2Cpp can translate library files for Dll's like the following example from the Delphi help. It shows a DLL with two exported functions, Min and Max.

```
library MinMax;

function min(X, Y: integer): integer; stdcall;
begin
  if X < Y then min := X else min := Y;
end;

function max(X, Y: integer): integer; stdcall;
begin
  if X > Y then max := X else max := Y;
end;

exports
  min,
  max;

begin
end.
```

->

```
extern "C" __declspec(dllexport) int __stdcall max( int X, int Y );
extern "C" __declspec(dllexport) int __stdcall min( int X, int Y );

int __stdcall min( int X, int Y )
{
  int result = 0;
  if ( X < Y )
    result = X;
  else
    result = Y;
  return result;
}

int __stdcall max( int X, int Y )
{
  int result = 0;
  if ( X > Y )
    result = X;
  else
    result = Y;
  return result;
}
```

The Delphi help recommends: "If you want your DLL to be available to applications written in other languages, it’s safest to specify **stdcall** in the declarations of exported functions." However, the names of such exported functions get a special "decorated" signature in order to facilitate language features like overloading. To avoid such name mangling a module definition (.def-) file can be used in the Dll project. Delphi2Cpp creates module definition files automatically.

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
for-in loops
Inline variable declarations

.

8.1 Unicode

Delphi2Cpp is able to process Delphi files which uses non ANSI characters for identifiers or in comments. For example:

```
unit Unicode;

interface


(* Delphi2Cpp □ □ Unicode *)

type

  □□ = record
    □□: string;
    □□ : string;
  end;


implementation

   //□ □□ □ □□ (xìngzhì)
   procedure□ (A□□ : □□);
  begin
     WriteLn(A□ .□□);
     WriteLn(A□ .□□);
  end;



end.
```

becomes to:

```
/* Delphi2Cpp □ □ Unicode */

struct □□
{
  System::String □□;
  System::String □□;
};
```

```
---------

  //□ □□ □ □□ (xìngzhì)

void □ (□□ A□□ )
{
  WriteLn(A□□ .□□ );
  WriteLn(A□□ .□□ );
}
```

8.2 Unit scope names

Delphi2Cpp 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*. Delphi2Cpp 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)

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 typecast`	`type(int A); / operator resultType () const`
`Explicit(a: type) : resultType;`	`explicit typecast`	`explicit type(int A); / explicit operator resultType () con`
`Negative(a: type) : resultType;`	`-`	`type operator - () const;`
`Positive(a: type): resultType;`	`+`	`type operator + () const;`
`Inc(a: type) : resultType;`	`Inc`	`type & operator ++ ();`
`Dec(a: type): resultType;`	`Dec`	`type & operator -- ();`
`LogicalNot(a: type): resultType;`	`not`	`type operator ! () const;`
`Trunc(a: type): resultType;`	`Trunc`	`static __int64 Trunc(const TMyClass& Value);`
`Round(a: type): resultType;`	`Round`	`static __int64 Round(const type Value);`
`In(a: type; b: type) : Boolean;`	`in`	`friend bool IsContained(const type A, const type B);`

`Equal(a: type; b: type) : Boolean;`	`=`	`friend bool operator == (const type A, const type B);`
`NotEqual(a: type; b: type): Boolean;`	`<>`	`friend bool operator != (const type A, const type B);`
`GreaterThan(a: type; b: type) Boolean;`	`>`	`friend bool operator > (const type A, const type B);`
`GreaterThanOrEqual(a: type; b: type): Boolean;`	`>=`	`friend bool operator >= (const type A, const type B);`
`LessThan(a: type; b: type): Boolean;`	`<`	`friend bool operator < (const type A, const type B);`
`LessThanOrEqual(a: type; b: type): Boolean;`	`<=`	`friend bool operator <= (const type A, const type B);`
`Add(a: type; b: type): resultType;`	`+`	`friend resultType operator + (const type A, const type B);`
`Subtract(a: type; b: type) : resultType;`	`-`	`friend resultType operator - (const type A, const type B);`
`Multiply(a: type; b: type) : resultType;`	`*`	`friend resultType operator * (const type A, const type B);`
`Divide(a: type; b: type) : resultType;`	`/`	`friend resultType operator / (const type A, const type B);`
`IntDivide(a: type; b: type): resultType;`	`div`	`friend resultType operator / (const type A, const type B);`
`Modulus(a: type; b: type): resultType;`	`mod`	`friend resultType operator % (const type A, const type B);`
`LeftShift(a: type; b: type): resultType;`	`shl`	`friend resultType operator << (const type A, const type B)`
`RightShift(a: type; b: type): resultType;`	`shr`	`friend resultType operator >> (const type A, const type B)`
`LogicalAnd(a: type; b: type): resultType;`	`and`	`friend resultType operator && (const type A, bool B);`
`LogicalOr(a: type; b: type): resultType;`	`or`	`friend resultType operator || (const type A, bool B);`
`LogicalXor(a: type; b: type): resultType;`	`xor`	`friend resultType operator XOR (const type A, bool B); //`
`BitwiseAnd(a: type; b: type): resultType;`	`and`	`friend resultType operator & (const type A, bool B);`
`BitwiseOr(a: type; b: type): resultType;`	`or`	`friend resultType operator | (const type A, bool B);`
`BitwiseXor(a: type; b: type): resultType;`	`xor`	

All Delphi declarations have the signature of functions with parameters and a return type. In C++ however, some operators don't return a value, but operate on the class instance itself.

- The binary operators in C++ are formed like their counterparts in C++ and therefore the translation is straightforward.
- The code for the unary operators *Negative*, *Positive, LogicalNot*, *Inc* and *Dec* has to be remodeled at the C++ translation.
- The conversion operators have to be remodeled too. Dependant on the direction of the conversion - to the class or from the class - the translation has to be done differently.
- 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:

```
TMyClass operator + (const TMyClass& A, const TMyClass& B)
{
  TMyClass result = {0};
  TMyClass returnrec = {0};
  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++. Delphi2Cpp 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

While the operators *Negative*, *Positive, LogicalNot*, *Inc* and *Dec* have a parameter and a return value, in Delphi, the counterparts in C++ don't have a parameter, but return a modified copy of themselves. The code for the operator implementation has to be remodeled accordingly. This is demonstrated at the example of the *Negative* operator:

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

Delphi2Cpp converts this to:

```
TOperatorClass TOperatorClass::operator - () const
{
  TOperatorClass result = {0};
  TOperatorClass B = {0};
  B = -this->payload;  // Use the implicit conv here?
  result = B;
  return result;
}
```

All occurrences of the parameter are substituted by *this* in C++.

8.3.3 conversion operators

The translation of the Delphi conversion operators depends on the direction of the conversion. The case, that a class instance is converted to another type is similar to the unary operators.

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

In C++ there is no parameter. A modified copy of the class instance itself is returned instead. All occurrences of the parameter are substituted by *this* in C++. So the code above becomes to:

```
TMyClass::operator int () const
{
  int result = 0;
  int myint = 0;
  myint = this->payload;
  result = myint;
  return result;
}
```

If the other way round another type is converted to the class, the operator has to be converted to a class constructor in C++. For example:

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

Becomes to:

```
TMyClass::TMyClass(int A)
{
  //#   TMyClass returnrec = {0};
  this->payload = A;
  *this = *this;
}
```

Therefore there isn't a return value in C++ and all occurrences of *result* in the Delphi code have to be substituted by *this* in C++.

For explicit operators simply the keyword explicit has to be added to the C++ declarations.

```
explicit operator int () const
explicit TMyClass(int A)
```

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++.

```
/*#static*/ __int64 TMyClass::Round(const TMyClass& Value)
{
  __int64 result = 0;
    result = d2c_system::Round(((double) Value)); // cast to double prevents from cycle
  return result;
}
```

At positions, where these operators are used, Delphi2Cpp 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);
```

New in Delphi 10.4 Sydney are the *Initialize* and the *Finalize* operators.

8.4 Class helpers

There is no counterpart to class/record helpers in C++.

For *C++Builder* for *TCharHelper* there are library functions declared in *System.Character.hpp*, which have to be used instead. For example

```
    s[1] := s[1].ToUpper;
```

->

```
    s[1] = ToUpper(s[1]);
```

How code that uses helpers can be translated to C++ in other cases for C++Builder and for other compilers is demonstrated at the example from here:

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 Delphi2Cpp for C++Builder to

```
    class TStringsHelper
    {
      public:
      TStringsHelper(TBase* xpClass) : m_pClass(xpClass) {}

    private:
      TObject* __fastcall GetTheObject(const String& AString);
      void __fastcall SetTheObject(String& AString, TObject* Value);
    public:
      __property TObject* ObjectFor[const String& AString] = { read = GetTheObject, write = SetTheObject };

      private:

      TBase* m_pClass;

    };
```

Of course, for other compilers than C++Builder the properties become setter and getter functions. If *S* is an instance of *TBase*, an assignment of a *TObject* like:

```
    S.ObjectFor['a'] := Object;
```

becomes to:

```
    TStringsHelper(s).ObjectFor[L"a"] = Object;
```

The trick is, that functions calls of the helper class are redirected to calls of the helped class inside of a local instance of the helper class. For example the setter method of *TStringHelper* might look like:

```
    void __fastcall TStringsHelper::SetTheObject(String& AString, TObject* Value)
    {
      int idx = 0;
      idx = m_pClass->IndexOf(AString);
      if(idx >- 1)
        m_pClass->Objects[idx] = Value;
    }
```

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

8.5 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)#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;
```

Delphi2Cpp converts these new features for C++Builder to:

```
struct TMyRecord
{
  typedef int TInnerColorType;
  int Red;
  static int Blue;
  void __fastcall printRed();
  __fastcall TMyRecord(int val);
  __property TInnerColorType RedProperty = { read = Red, write = Red };
  /*static */__property TInnerColorType BlueProp = { read = Blue, write = Blue };

  TMyRecord() {}
};


---------------


int TMyRecord::Blue = 0;

__fastcall TMyRecord::TMyRecord(int val)
```

```
 : Red(val)
{
}

void __fastcall TMyRecord::printRed()
{
  { Write(L"Red: "); WriteLn(Red); };
}
```

And for other compilers it becomes:

```
struct TMyRecord
{
  typedef int TInnerColorType;
  int Red;
  static int Blue;
  void printRed();
  TMyRecord(int val);
  /*property RedProperty : TInnerColorType read Red write Red;*/
  TInnerColorType ReadPropertyRedProperty() { return Red;}
  void WritePropertyRedProperty(int Value){Red = Value;}
  /*property BlueProp : TInnerColorType read Blue write Blue;*/
  static TInnerColorType ReadPropertyBlueProp() { return Blue;}
  static void WritePropertyBlueProp(int Value){Blue = Value;}
  void InitMembers(){Red = 0;}

  TMyRecord() {InitMembers();}
};

---------------------


 int TMyRecord::Blue = 0;

TMyRecord::TMyRecord(int val)
 : Red(val)
{
}

void TMyRecord::printRed()
{
  { Write(L"Red: "); WriteLn(Red); };
}
```

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

```
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:

```
    class TOuterClass : public System::TObject
    {
      typedef System::TObject inherited;

    private:
      int myField;
    public:

      class TInnerClass : public System::TObject
      {
        typedef System::TObject inherited;

      public:
        int myInnerField;
        void innerProc();
        void InitMembers(){myInnerField = 0;}
    public:
        TInnerClass() {InitMembers();}
      };
      void outerProc();
      void InitMembers(){myField = 0;}
    public:
      TOuterClass();
    };

    TOuterClass::TOuterClass() {InitMembers();}



    void TOuterClass::TInnerClass::innerProc()
    {

      // ...
    }

    void foo()
    {
      TOuterClass* x = nullptr;
      TOuterClass::TInnerClass* y = nullptr;
      x = new TOuterClass();
      x->outerProc();
       //...
      y = new TOuterClass::TInnerClass();
      y->innerProc();
    }
```

8.7 Anonymous Methods

The corresponding C++ feature to Delphi's anonymous methods are lambda expressions. 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

- 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 std::function type:

```
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
```

->

```
typedef std::function<int (int)> TFuncOfInt;

 TFuncOfInt adder;
 adder = [&](int X) -> int {
  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;

class TAn3Class : public System::TObject
{
  typedef System::TObject inherited;
public:
  void method(int X);
public:
  TAn3Class() {}
};

void Test()
{
  TMethRef m;
  TAn3Class* i = nullptr;
  // ...
  m = std::bind(&TAn3Class::method, i, std::placeholders::_1);
}
```

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;
```

->

```
    typedef std::function<System::String (int)> TFuncOfIntToString;


    void AnalyzeFunction(TFuncOfIntToString Proc)
    {
      Proc(3);
    }
```

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;
```

->

```
    typedef std::function<int (int)> TFuncOfInt;

    TFuncOfInt MakeAdder(int Y)
    {
      TFuncOfInt result;
      result = [&](int X) -> int {  // => error
      int result = 0;
      result = X + Y;
      return result;
      };
      return result;
    }
```

```
//---------------------------------------------------------------------------
void Test()
{
  TFuncOfInt adder;
  adder = MakeAdder(20);
  WriteLn(adder(22));
}
```

Lambda expression capture variables either by reference or as copies. The C++ code that Delphi2Cpp generates, always uses the most general capture [&], which binds all used variables as references. But in the example above the lifetime of *y* isn't extended. Therefore *y* has an accidental value, when adder is called. In this case the code can be corrected easily, by use of a copying capture:

```
result = [y](int X) -> int {
```

binding just *y* or

```
result = [=}(int X) -> int {
```

binding all used variables, here just *y* too.

8.7.5 Use as events

Method reference types can be used as a kind of event in Delphi and become std::function's 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;
```

->

```
typedef std::function<void ()> TAnProc;

class TAn4Component : public System::TComponent
{
  typedef System::TObject inherited;
private:
  TAnProc FMyEvent;
public:
  /*property MyEvent : TAnProc read FMyEvent write FMyEvent;*/
  TAnProc ReadPropertyMyEvent() { return FMyEvent;}
  void WritePropertyMyEvent(TAnProc Value){FMyEvent = Value;}
public:
  TAn4Component() {}

};

void TestAnonymous4()
{
  TAn4Component* C = nullptr;
  C = new TAn4Component();
  C->WritePropertyMyEvent([&]() -> void {
  ;
});
}
```

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

Declaration
Nested types
Base types
Procedural types
Parameterized methods
Redeclared types

8.8.1 Declaration

The following code demonstrated the conversion of a generic Delphi type to C++.

```
type
   TPair<TKey,TValue> = class   // declares TPair type with two type parameters

   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;

 type
   TSIPair = TPair<String,Integer>; // declares instantiated type
   TSSPair = TPair<String,String>;  // declares with other data types
   TISPair = TPair<Integer,String>;
   TIIPair = TPair<Integer,Integer>;
   TSXPair = TPair<String,TXMLNode>;

implementation

function TPair<TKey,TValue>.GetValue: TValue;
 begin
   Result := FValue;
 end;
```

->

```
// declares TPair type with two type parameters
template<typename TKey, typename TValue>
class TPair : public System::TObject
{
public:
  typedef System::TObject inherited;
private:
  TKey FKey;
  TValue FValue;
public:
  TKey GetKey() const;
  void SetKey(TKey key);
  TValue GetValue() const;
  void SetValue(TValue Value);
  /*property key : TKey read GetKey write SetKey;*/
  TKey ReadPropertykey() const { return GetKey();}
  void WritePropertykey(TKey key){SetKey(key);}
  /*property Value : TValue read GetValue write SetValue;*/
  TValue ReadPropertyValue() const { return GetValue();}
  void WritePropertyValue(TValue Value){SetValue(Value);}
  TPair() {}
};
typedef TPair<System::String, int> TSIPair; // declares instantiated type
typedef TPair<System::String, System::String> TSSPair;  // declares with other data types
typedef TPair<int, System::String> TISPair;
typedef TPair<int, int> TIIPair;
typedef TPair<System::String, TXMLNode> TSXPair;




template<typename TKey, typename TValue>
TValue TPair<TKey, TValue>::GetValue() const
{
  TValue result;
  result = FValue;
  return result;
}
```

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;
```

->

```
//---------------------------------------------------------------------------
template<typename T>
class TFoo : public System::TObject
{
  typedef System::TObject inherited;
  friend class TBaz;
public:
      // ...
//---------------------------------------------------------------------------
  class TBar : public System::TObject
  {
    typedef System::TObject inherited;
  public:
    int X;
    void InitMembers(){X = 0;}
public:
    TBar() {InitMembers();}
  };

public:
  TFoo() {}
};


  // ...
//---------------------------------------------------------------------------
class TBaz : public System::TObject
{
  typedef System::TObject inherited;
  //#   template<typename T> friend class TFoo;
public:
      // ...
//---------------------------------------------------------------------------
  template<typename T>
  class TQux : public System::TObject
  {
    typedef System::TObject inherited;
  public:
    int X;
    void InitMembers(){X = 0;}
public:
    TQux() {InitMembers();}
  };

    // ...
public:
  TBaz() {}
};

extern TFoo<double>::TBar* n;
```

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;
```

->

```
//---------------------------------------------------------------------------
class TOuter : public System::TObject
{
  typedef System::TObject inherited;
public:
//---------------------------------------------------------------------------
  template<typename T>
  class TData : public System::TObject
  {
    typedef System::TObject inherited;
  public:
    TFoo<int>* FFoo1;
    TFoo<T>* FFoo2;
    TFoo<int>::TBar* FFooBar1;
// doesn't compile: TFoo<T>::TBar* FFooBar2;
    TBaz::TQux<int>* FBazQux1;
    TBaz::TQux<T>* FBazQux2;
       //...
public:
    TData() {}
  };
  TData<int>* FIntegerData;
  TData<System::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;
```

->

```
            // Actual type
//----------------------------------------------------------------------------
template<typename T>
class TFoo1 : public TBar
{
  typedef TBar inherited;

};
       // Open constructed type
//----------------------------------------------------------------------------
template<typename T>
class TFoo2 : public TBar2<T>
{
  typedef TBar2<T> inherited;

};
  // Closed constructed type
//----------------------------------------------------------------------------
template<typename T>
class TFoo3 : public TBar3<int>
{
  typedef TBar3<int> inherited;

};
```

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;

 type
   anArray<T>= array of T;
   intArray= anArray<Integer>;
```

->

```
//-----------------------------------------------------------------------------
template<typename T>
struct TRecord
{
  T FData;
};
template<typename T>
class IAncestor
{
  public:
  virtual ~IAncestor() {}
  virtual TRecord<T> GetRecord() = 0;
};
template<typename T>
class IFoo : public IAncestor<T>
{
  public:
  virtual ~IFoo() {}
  virtual void AMethod(T Param) = 0;
};
//-----------------------------------------------------------------------------
template<typename T>
class TFoo : public System::TObject, IFoo<T>
{
  typedef System::TObject inherited;
public:
  TRecord<T> FField;
  void AMethod(T Param);
  TRecord<T> GetRecord();
public:
  TFoo() {}
};
template<typename T> using anArray = std::vector<T>;
typedef anArray<int> intArray;
```

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
   WriteLn(Param);
 end;

 procedure TFoo.MyProc(X, Y: Integer);
 begin
   WriteLn('X:', X, ', Y:', Y);
 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;
```

->

```
template<typename T> using TMyProc = std::function<void (T)>;
template<typename Y> using TMyProc2 = std::function<void (Y, Y)>;
//---------------------------------------------------------------------------
class TFoo : public System::TObject
{
  typedef System::TObject inherited;
public:
  void Test();
  void MyProc(int X, int Y);
public:
  TFoo() {}
  System::TClass ClassType() const;
};

//---------------------------------------------------------------------------
void sample(int Param)
{
  WriteLn(Param);
}
//---------------------------------------------------------------------------
void TFoo::MyProc(int X, int Y)
{
  { Write(L"X:"); Write(X); Write(L", Y:"); WriteLn(Y); };
}
//---------------------------------------------------------------------------
void TFoo::Test()
{
  TMyProc<int> X;
  TMyProc2<int> Y;
  X = sample;
  X(10);
  Y = std::bind(&TFoo::MyProc, this, std::placeholders::_1, std::placeholders::_2);
  Y(20, 30);
}
```

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);
var
  Comparer : IComparer<T>;
begin
  Comparer := TComparer<T>.Default;
  if Comparer.Compare(X, Y) = 0 then
    WriteLn('Both members compare as equal')
  else
    WriteLn('Members do not compare as equal');
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;
```

->

```
//---------------------------------------------------------------------------
class TFoo : public System::TObject
{
  typedef System::TObject inherited;
public:
  void Test();
    template<typename T> void CompareAndPrintResult(T X, T Y);
public:
  TFoo() {}
  System::TClass ClassType() const;
};
//---------------------------------------------------------------------------
class TFoo : public TObject
{
  typedef TObject inherited;
public:
  void Test();
    template<typename T> void CompareAndPrintResult(T X, T Y);
public:
  TFoo() {}
  System::TClass ClassType() const;
};
//---------------------------------------------------------------------------
class TFooClassRef : public ClassRef<TFoo, TObjectClassRef> { };
TFooClassRef TFooClassRefInstance;
System::TClass TFoo::ClassType() const
{
  return &TFooClassRefInstance;
}
//---------------------------------------------------------------------------
template<typename T> void TFoo::CompareAndPrintResult(T X, T Y)
{
  IComparer<T> Comparer;
  Comparer = TComparer<T>.Default;   //  error
  if(Comparer.Compare(X, Y) == 0)
    WriteLn(L"Both members compare as equal");
  else
    WriteLn(L"Members do not compare as equal");
}
//---------------------------------------------------------------------------
void TFoo::Test()
{
  CompareAndPrintResult<String>(L"Hello", L"World");
  CompareAndPrintResult(L"Hello", L"Hello");
  CompareAndPrintResult<int>(20, 20);
  CompareAndPrintResult(10, 20);
}
//---------------------------------------------------------------------------
void Test()
{
  TFoo* F = nullptr;
  F = new TFoo();
  F->Test();
  System::ReadLn();
  delete F;
}
```

The function "`TFoo::CompareAndPrintResult doesn't compile. It can be improved in C++Builder with:`

```
System::Generics::Defaults::IComparer__1<T> Comparer; // abstract
Comparer = TComparer__1<T>().Default();
```

But the code still doesn't compile, because Comparer is abstract.

8.8.6 Redeclaration

Sometimes types are declared with the same name, which only differ in their template parameters. For example, there is a *TList* in System.pas:

```
TList = class(TObject)
```

and another *TList* in System.Generics.Collections:

```
TList<T> = class(TEnumerable<T>)
```

There is no problem to convert these types to C++ as they belong to different units. But sometimes such similar types are defined in the same scope, as for example *IEnumerator* in System.pas:

```
IEnumerator = interface(IInterface)
...
IEnumerator<T> = interface(IEnumerator)
```

In C++ it is not allowed to declare both in the same scope. Therefore the re-declared type will be renamed by appending two underscores and the number of template parameters. In this case the automatic conversion results in:

```
class IEnumerator : public IInterface
...
template<typename T>class IEnumerator__1 : public IEnumerator
```

A more complet example:

```
type

  TGenRedecl = class                              class TGenRedecl : public System::TObject
  public                                          {
    class function Lookup: Pointer;               public:
    class constructor Create;                       static void* Lookup();
  end;                                              TGenRedecl();
                                                  };



                                                  template<typename T>
  TGenRedecl<T> = class(TGenRedecl)               class TGenRedecl__1 : public TGenRedecl
  public                                          {
    class function Compare: Pointer;              public:
    class constructor Create;                       static void* Compare()
  end;                                              {
                                                      void* result;
                                                      return result;
                                                    }
implementation                                      TGenRedecl__1() {}
                                                  };

                                                  /*#static*/
class function TGenRedecl.Lookup: Pointer;        void* TGenRedecl::Lookup()
begin                                             {
end;                                                void* result = nullptr;
                                                    return result;
class constructor TGenRedecl.Create;              }
begin
end;                                              TGenRedecl::TGenRedecl()
                                                  {
                                                  }
class function TGenRedecl<T>.Compare: Pointer;
begin
end;

class constructor TGenRedecl<T>.Create;
begin
end;

function test: boolean;                           bool test()
var                                               {
  t1 : TGenRedecl;                                  bool result = false;
  t2 : TGenRedecl<integer>;                         TGenRedecl* t1 = nullptr;
  t3 : TGenRedecl<string>;                          TGenRedecl__1<int>* t2 = nullptr;
begin                                               TGenRedecl__1<string>* t3 = nullptr;
  t1 := TGenRedecl.Create;                          t1 = new TGenRedecl();
  t2 := TGenRedecl<integer>.Create;                 t2 = new TGenRedecl__1<int>();
  t3 := TGenRedecl<string>.Create;                  t3 = new TGenRedecl__1<string>();
...                                                ...
```

8.9 Inline variable declarations

Since Delphi 10.3 variables can be declared not only at the beginning of a routine, but inline too.

Previously, variables always had to be declared in a var block, either globally or at the head of a routine. . For example:

```
procedure Test;
var
  I: Integer;
begin
  I := 22;
  ...
```

Since Delphi 10.3. variables also may be declared inside of a code block, either with an explicit type or with an implicit type:

```
procedure Test;           procedure Test;
begin                     begin
  var I: Integer;           var I := 22;
  ...                       ...
```

Delphi2Cpp converts this to:

```
void Test()               void TestAuto()
{                         {
  int I = 22;               auto I = 22;
  ...                       ...
```

Similarly, constants can now also be defined inline:

```
begin
  const C1: Integer = 10;
  const C2 = 10;
```

->

```
{
  const int C1 = 10;
  auto C2 = 10;
```

No distinction is made here between C++98 and C++11. Auto variable and constant declarations have to be manually corrected for C++98.

The rules for scoping local variables in C++ are exactly the same as inline variables in Delphi, so Delphi2cpp doesn't need to use any special tricks here. This also applies to variables in for loops:

```
var total := 0;
for var I: Integer := 1 to 10 do
```

```
    Inc (total, I);
```

->

```
  auto total = 0;
  for(int I = 1; I <= 10; I++)
  {
    total += I;
  }
```

9 DFM-Translator

Delphi units, which either define Delphi *VCL* forms or frames are associated with form modules. These form modules are files with the extension *dfm*. The code in a *DFM*-files determines how forms or frames are constructed by means of the members of the visual component library (VCL). The code in DFM-files is not Object Pascal, but describes graphical interfaces in an abbreviated way. Delphi2Cpp can parse the *DFM*-code and and translate it into C++ code to create the form at runtime. Therefore the option to convert the DFM code has to be enabled.

Per default all lines of the *DFM* code are converted to C++ assignment statements. However, when the Delphi compiler reads the *DFM* code, more can actually happen than simple assignments. Delphi2Cpp tries to reproduce the complex loading process in the most intuitively simple way possible when dynamically creating the components at runtime. To reproduce the additional effects just mentioned, Delphi2Cpp uses some special routines. Instead of assigning values directly to properties, they are passed as arguments to the routines in which additional actions can then be carried out. Based on some exemplary examples, there are a number of such predefined routines. More routines can be defined by the user if necessary.

The names of these routines are formed in a systematic manner from the types and properties involved. The routines themselves should be declared and defined in the file d2c_dfm.h/d2c_dfm.cpp. In order to trigger the output of these routines into the generated code, Delphi2Cpp must be configured accordingly..There is a dialog with the label DFM Conversion, in which the list of types and properties for which such special assignment routines should be issued will be defined.

The code will be written into the constructor of the form or into the constructor of a frame. If the components are created dynamically at runtime the form file is not needed any more.

9.1 Normal assignments

By default, when DFM translation is enabled in the processor options, all lines of DFM code are converted to C++ assignment statements. As an example of such a conversion, the original code and the resulting code in the constructor of a *TForm* are compared line by line:below:

```
DFM code                              C++ code
-------------------------------------------------------------------------------
Object AboutBox: TAboutBox
  Left = 229                          Left = 229;
  Top = 166                           Top = 166;
  BorderStyle = bsDialog              BorderStyle = TFormBorderStyle::bsDialog;
  Caption = 'About RichEdit'          Caption = L"About RichEdit";
  ...                                 ...

  Object OKButton: TButton            OKButton = new TButton(this);
    Left = 269                        OKButton->Left = 269;
    Top = 208                         OKButton->Top = 208;
    Width = 75                        OKButton->Width = 75;
    Height = 25                       OKButton->Height = 25;
    Cancel = True                     OKButton->Cancel = true;
    Caption = 'OK'                    OKButton->Caption = L"OK";
    Default = True                    OKButton->Default = true;
    ModalResult = 2                   OKButton->ModalResult = 2;
    TabOrder = 0                      OKButton->TabOrder = 0;
  end                                 OKButton->Parent = this;
end
```

Only the creation of the button goes beyond a simple assignment: in C++ the *OKButton* is explicitely created at runtime and the form is set as its parent.

9.2 Special assignments

Some cases where the assignments in the *DFM* file cannot be directly translated into C++ statements. This is due to the dependencies between properties and because the DefineProperties procedure, which is called when the component is loaded, can be overridden individually for each component. Some special cases are listed below:

Internally used properties
Design time only properties
Binary data
Protected properties
TDataSet
TSplitter
TToolBar
Sets
Lists
Collections
Setting the parent
List of predefined DFM routines

In these cases, DFM conversion routines can be executed instead. Such procedures can be defined by the user, others are predefined.

9.2.1 DefineProperties

Unpublished data of a component can be written into a form file and read from a form file by means of the overwritten function *DefineProperties*. These data often concern the presentation of non-visible components at design time. Such data are not used, when the code of a DFM-file is translated to C++. Other effects of *DefineProperties* can be simulated in Delphi2Cpp using special assignment procedures

Here is a short sketch of how "DefineProperties" comes into play when creating the components: when Delphi creates a new form (*TApplication.CreateForm*), a *TResourceStream* is created in which the *DFM* content is read with a *TReader* to create the complete tree of the components on the from. Individual components are read and generated by the function:

```
function TReader.ReadComponent(Component: TComponent): TComponent;
```

In the procedure

```
procedure TReader.ReadProperty(AInstance: TPersistent);
```

first the name of the property is read and then its value. Normally, a setter method is determined here via property information (TProperty) for the property, which can be used to set the value. If this is not possible, the component method *DefineProperties* is called to set the value. *DefineProperties* has access to the internal state of the component and can manipulate it. This access cannot be fully reproduced with Delphi2cpp's simple approach of dynamically creating the components.

9.2.2 DFM conversion routines

The names for the *DFM* conversion routines are constructed in a systematically way: depending on whether the procedure is used for a simple assignment or whether a function is used to assign a new variable instance, its name starts with *Assign* or *Create*. For collections functions with the prefix *Get* can be created. As fourth kind of procedures events can be defined.

The second part of the name of the assignment function specifies a type. This type is either assigned a property, or an element is created or modified in it, or an event occurs on it.

The type is either the concrete type of the current element - e.g. *TButton* - or one of the following base classes. In the latter case, the initial 'T' is omitted from the name.

Type	Name part
TControl	Control
TForm	Form
TFrame	Frame
TDataModule	DataModule
TDataSet	DataSet
TConcreteType	TConcreteType

Assignment procedures for these base classes are applied to all elements with types, which are derived from the base class.

1. Assignment

In this case the name for the assignment procedure is constructed by the word *Assign* followed the name of the type that owns the property and finally the name of the property. Some predefined procedures can demonstrate this naming convention:

Type	Property	Name
TBitmap	Data	AssignTBitmapData
TDataModule	Height	AssignTDataModuleHeight
TDataM	PixelsP	AssignTDataModulePixelsPerInch

odule	erInch	
TDataModule	Width	AssignTDataModuleWidth
TForm	PixelsPerInch	AssignTFormPixelsPerInch
TForm	TextHeight	AssignTFormTextHeight
TIcon	Data	AssignTIconData
TImageList	Bitmap	AssignTImageListBitmap
TPicture	Data	AssignTPictureData

For example bitmap data can be assigned to the *Glyph* property of a *TSpeedButton* with *AssignTBitmapData*.

```
void AssignTBitmapData(TSpeedButton* xp, const System::DynamicArray<System::Byte>& xBytes)
```

Remark: Not the property itself is passed to the procedure, but it's parent, because the property might not exist at runtime.

By use of this function the following *DFM* code:

```
    Object LineButton: TSpeedButton
...
      Glyph.Data = {
        66010000424D6601000000000000760000002800000014000000140000000100
...
```

will be converted to:

```
  LineButton = new TSpeedButton(Panel1);
...
  AssignTBitmapData(LineButton, {
    0x66,0x01,0x00,0x00,0x42,0x4D,0x66,0x01,0x00,0x00
...
```

2. Creation / Modification

A creation function is constructed by the word *Create* followed by the name of the type of the parent of the new variable finally followed by the name of the child variable type..Again, some predefined functions can demonstrate this naming convention:

Parent	Child	Name
TMainMenu	TMenuItem	CreateTMainMenuTMenuItem
TMenuItem	TMenuItem	CreateTMenuItemTMenuItem

These creation functions return the type of the new child. For example a new menu item in the main menu will be created with:

```
TMenuItem* CreateTMainMenuTMenuItem(TMainMenu* xp);
```

By use of this function the following dfm code:

```
  Object MainMenu1: TMainMenu
...
    Object File1: TMenuItem
...
```

will be converted to:

```
  MainMenu1 = new TMainMenu(this);
...
  File1 = CreateTMainMenuTMenuItem(MainMenu1);
...
```

3. Item

Collection items can be assigned by means of functions, whose names are constructed from the prefix *Get*, followed by the name of the collection and the word *item*. Such a predefined getter function is:

```
TFieldDef* GetTFieldDefsitem(TFieldDefs* xp, int xiIndex)
```

4. Event

An event procedure is constructed by the prefix *On* followed by the name of the type for which this event shall be executed, finally followed by the type of event. A variable of that type will be passed as parameter.Four event procedures are predefined: Two kinds of events can be defined

1. the *Begin*-event will be executed when a new Control will be created or modified
1. the *End*-event will be executed after the properties of the Contorl are set.

Four such event procedures are predefined:

Type	**Event**	**Name**
TDataSet	Begin	OnTDataSetBegin
TDataSet	End	OnTDataSetEnd
TSplitter	Begin	OnTSplitterBegin
TSplitter	End	OnTSplitterEnd

9.2.3 Internally used properties

A special case when converting *DFM* files are internally used properties:

```
ExplicitLeft
ExplicitRight
ExplicitTop
```

```
ExplicitBottom
```

Because they are used internally by *Delphi* only *Delphi2cpp* only outputs them as comments.

9.2.4 Design time only properties

A special case when converting *DFM* files are design time properties Such properties are in *TCustomForm*

PixelsPerInch
TextHeight
IgnoreFontProperty

They are useful at design time only and cannot be set at runtime at all. Delphi2Cpp suppresses the output of these frequent properties by means of the pre-defined procedures:

```
void AssignFormTextHeight(TForm* xpForm, int xi);
void AssignFormPixelsPerInch(TForm* xp, int xi);
void AssignDataModulePixelsPerInch(TDataModule* xp, int xi);
```

9.2.5 Binary data

A special case when converting *DFM* files are binary data. Binary data are assigned for example to *TIcon's* or *TBitmaps* as in the following example:

```
object EllipseButton: TSpeedButton
...
  Glyph.Data = {
    4E010000424D4E01000000000000760000002800000012000000120000000100
    040000000000D800000000000000000000000000100000001000000000000000
    80000080000000808000800000008000800080800000C0C0C000808080000000
    FF0000FF000000FFFF00FF000000FF00FF00FFFF0000FFFFFF00333333000000
    3333330000003333003333330033330000000333033333333333303330000003303
    3333333333303330000000303333333333333330300000003033333333333330300
    0000333333333333333330000000003333333333333333330000000003333333333
    333330000000000333333333333333000000000033333333333333330000000000333
    3333333333330000000303333333333333303000000030333333333333330300
    0000330333333333333033000000033303333333333033300000000333300333333
    003333000000033333300000003333330000000}
```

Glyph is a *TBitmap*. But a *TBitmap* doesn't really has a *Data* property.Rather, Delphi reads and writes this data by means of the functions

```
ReadData(System::Classes::TStream* Stream)
WriteData(System::Classes::TStream* Stream);
```

Delphi2Cpp converts this into:

```
  AssignTBitmapData(EllipseButton, {
     0x4E,0x01,0x00,0x00,0x42,0x4D,0x4E,0x01,0x00,0x00
     ,0x00,0x00,0x00,0x00,0x76,0x00,0x00,0x00,0x28,0x00
     ,0x00,0x00,0x12,0x00,0x00,0x00,0x12,0x00,0x00,0x00
     ,0x01,0x00,0x04,0x00,0x00,0x00,0x00,0x00,0xD8,0x00
     ,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00
     ,0x10,0x00,0x00,0x00,0x10,0x00,0x00,0x00,0x00,0x00
     ,0x00,0x00,0x00,0x00,0x80,0x00,0x00,0x80,0x00,0x00
     ,0x00,0x80,0x80,0x00,0x80,0x00,0x00,0x00,0x80,0x00
     ,0x80,0x00,0x80,0x80,0x00,0x00,0xC0,0xC0,0xC0,0x00
     ,0x80,0x80,0x80,0x00,0x00,0x00,0xFF,0x00,0x00,0xFF
     ,0x00,0x00,0x00,0xFF,0xFF,0x00,0xFF,0x00,0x00,0x00
     ,0xFF,0x00,0xFF,0x00,0xFF,0xFF,0x00,0x00,0xFF,0xFF
```

```
,0xFF,0x00,0x33,0x33,0x33,0x00,0x00,0x00,0x33,0x33
,0x33,0x00,0x00,0x00,0x33,0x33,0x00,0x33,0x33,0x33
,0x00,0x33,0x33,0x00,0x00,0x00,0x33,0x30,0x33,0x33
,0x33,0x33,0x33,0x03,0x33,0x00,0x00,0x00,0x33,0x03
,0x33,0x33,0x33,0x33,0x33,0x30,0x33,0x00,0x00,0x00
,0x30,0x33,0x33,0x33,0x33,0x33,0x33,0x33,0x03,0x00
,0x00,0x00,0x30,0x33,0x33,0x33,0x33,0x33,0x33,0x33
,0x03,0x00,0x00,0x00,0x03,0x33,0x33,0x33,0x33,0x33
,0x33,0x33,0x30,0x00,0x00,0x00,0x03,0x33,0x33,0x33
,0x33,0x33,0x33,0x33,0x30,0x00,0x00,0x00,0x03,0x33
,0x33,0x33,0x33,0x33,0x33,0x33,0x30,0x00,0x00,0x00
,0x03,0x33,0x33,0x33,0x33,0x33,0x33,0x33,0x30,0x00
,0x00,0x00,0x03,0x33,0x33,0x33,0x33,0x33,0x33,0x33
,0x30,0x00,0x00,0x00,0x03,0x33,0x33,0x33,0x33,0x33
,0x33,0x33,0x30,0x00,0x00,0x00,0x30,0x33,0x33,0x33
,0x33,0x33,0x33,0x33,0x03,0x00,0x00,0x00,0x30,0x33
,0x33,0x33,0x33,0x33,0x33,0x33,0x03,0x00,0x00,0x00
,0x33,0x03,0x33,0x33,0x33,0x33,0x33,0x30,0x33,0x00
,0x00,0x00,0x33,0x30,0x33,0x33,0x33,0x33,0x33,0x03
,0x33,0x00,0x00,0x00,0x33,0x33,0x00,0x33,0x33,0x33
,0x00,0x33,0x33,0x00,0x00,0x00,0x33,0x33,0x33,0x00
,0x00,0x00,0x33,0x33,0x33,0x00,0x00,0x00
});
```

AssignTBitmapData belongs to the predefined DFM conversion routines and is declared in the helper file *d2c_vcl.h* which, will be included automatically.

```
void AssignTBitmapData(TSpeedButton* xp, const System::DynamicArray<System::Byte>& xBytes)
```

The creation of the *DynamicArray* only works if the use of the clang compiler is enabled in C++Builder. For the classic compiler an error "E2188: expression syntax" will be produced:

The dats of TClientDataSet are to large to be treated this way and can be suppressed.

9.2.5.1 TClientDataSet

If treated in the same way as other binary data, the data in a *TClientdataset* cause a stack overflow, because these data can be very large.

The data of a TClientdataset are used only at design time. In fact, these data are only needed for display at design time. Therefore, the output of this data is suppressed in Delphi2Cpp and only a small comment is output

```
// The data of a TClientdataset are only used at design time
```

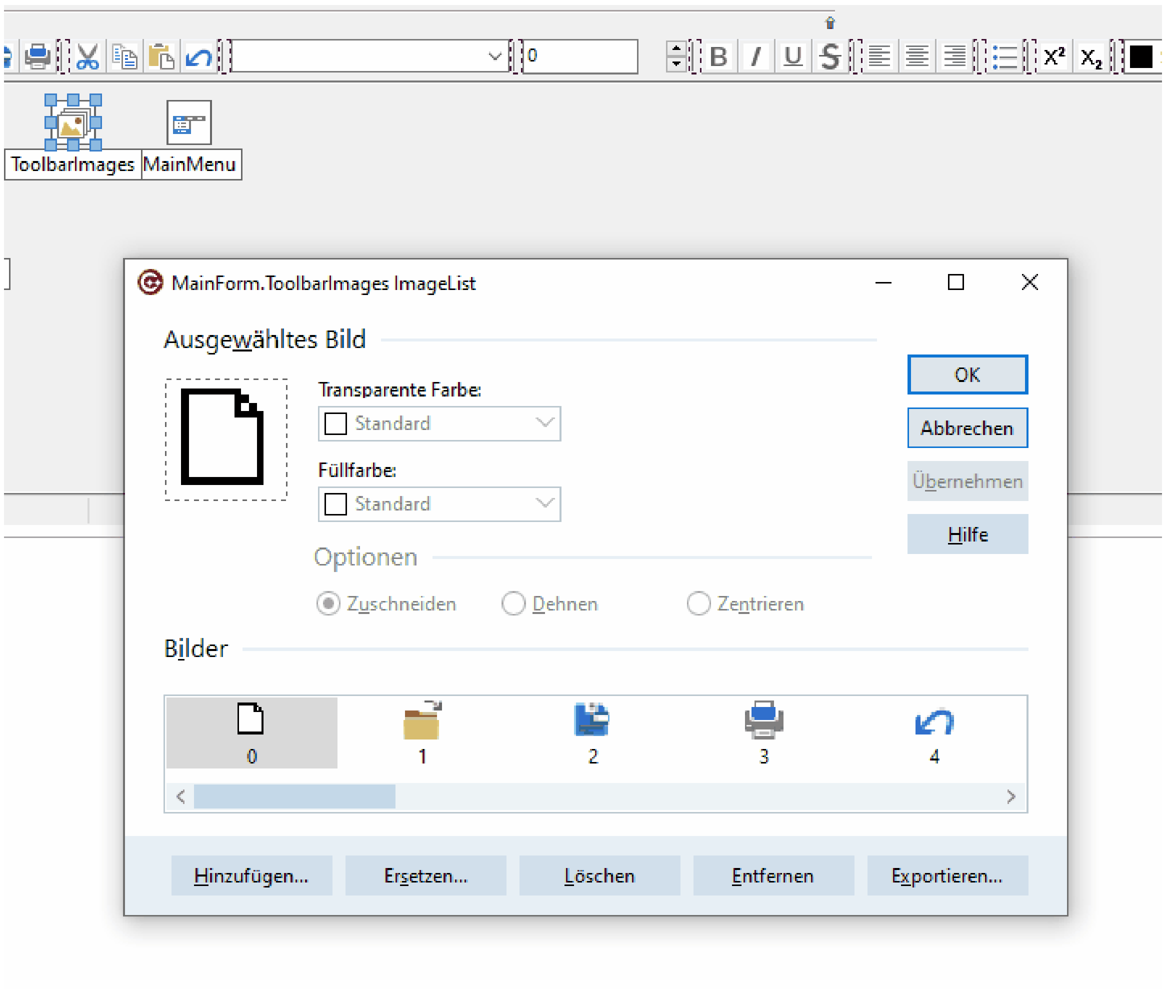
However, if you like, you can try to construct the assignment procedure *AssignTClientDataSetData* by use of the type *TClientDataSet* and the property *Data*.

The field definition of a client data set are a special case of collection assignments.

9.2.5.2 TImageList

Currently, the *AssignTImageListBitmap* routine for assigning the binary data to the instance of a *TImageList* at runtime is a dummy routine that makes the converted code compile, but does not work. As a workaround, the images in the list can be saved to the hard drive and loaded one after the other and added to the list.

To do this, the Delphi file can first be opened in C++Builder. Double-clicking on the TImagelist icon will then display a dialog that offers the option of saving the individual images.



```
void AssignTImageListBitmap2(TImageList* xp, const System::DynamicArray<System::Byte>& Bytes)
{
//D:\develop\Delphi2Cpp\Tests\CBTestDone\RichEdit\ToolbarImages.bmp
//		xp->FileLoad(rtBitmap, L"D:\\develop\\Delphi2Cpp\\Tests\\CBTestDone\\RichEdit\
\ToolbarImages.bmp", clFuchsia);
		xp->Masked = true;
		TBitmap* pBitmap = new TBitmap;
		pBitmap->LoadFromFile(L"D:\\develop\\Delphi2Cpp\\Tests\\CBTestDone\\TListViewItems\
\Images0.bmp");
		xp->AddMasked(pBitmap, clFuchsia);
		pBitmap->LoadFromFile(L"D:\\develop\\Delphi2Cpp\\Tests\\CBTestDone\\TListViewItems\
\Images1.bmp");
		xp->AddMasked(pBitmap, clFuchsia);
		pBitmap->LoadFromFile(L"D:\\develop\\Delphi2Cpp\\Tests\\CBTestDone\\TListViewItems\
\Images2.bmp");
        xp->AddMasked(pBitmap, clFuchsia);


        delete pBitmap;
}
```

9.2.6 Protected properties

A special case when converting *DFM* files are protected properties as for example *IsControl* in *BMPDlg.dfm* of the *Graphex* example:

```
object Bevel1: TBevel
...
  IsControl = True
```

The standard translation would produce an error:

```
Bevel1->IsControl = true; // 'IsControl' is a protected member of 'Vcl::Controls::TControl'
```

There is a pre-defined procedure, which takes care for the correct assignment.

```
void AssignControlIsControl(TControl* xp, bool xb)
```

9.2.7 TDataSet

A special case when converting *DFM* files concerns *TDataSets*. Properties that affect the status of a database or the display of its data can only be changed if the corresponding dataset is deactivated. If a dataset is set to active in a DFM file, this assignment will be shifted to the end of the generated C++ code.

If properties of a dataset in a frame are changed the *Active* property is temporarily set to false and reset to it's original value at the begin and at the end of the changes by means of the procedures

```
void OnTDataSetBegin(TDataSet* xp);
void OnTDataSetEnd(TDataSet* xp);
```

9.2.8 TSplitter

A special case when converting *DFM* files is *TSplitter*. To ensures that *TSplitter* controls are placed correctly without moving the positions of other controls, following two event procedures are predefined:

```
void OnTSplitterBegin(TSplitter* xp);
void OnTSplitterEnd(TSplitter* xp);
```

9.2.9 TToolBar

A special case when converting *DFM* files is *TToolBar*. If the toolbar is first created with its parent and then components are placed in it, it may happen that the components do not appear in the desired positions. However, the correct positions arise if the parent of the toolbar is only set after the components have been placed.

Unfortunately, the delayed setting of the parent of the toolbar also requires a delayed setting of some properties of its components. Property assignments that require a parent can be configured to only occur if the parent exists.

To ensure that components placed from a toolbar appear in the desired positions, the parent of the toolbar must only be set after the components have been placed.

9.2.9.1 Requires parent

The delayed setting of the parent of a toolbar also requires a delayed setting of some properties of its components. If for example the *Style* property of a *TColorBox* is seton a TToolBar, which hasn't a parent yet, the following exception is thrown:

```
EInvalidOperation: Element has no parent window
```

Therefore there is the predefined procedure *AssignTColorBoxStyle*, which will only be executed after the parent is set. If a user want to define such a procedure manually, he has to check the according box in the dialog for the definition of *DFM* assignment routines.

9.2.10 Sets

Set values, which shall be assigned to a property are listed in a DFM file inside of brackets "[" ... "]". For example:

```
 Object FGColorBox: TColorBox
...
   Style = [cbStandardColors, cbExtendedColors, cbSystemColors, cbIncludeNone, cbIncludeDefault, cbPrett
```

Because changing the style of a *TColorBox* requires a parent for that box, there is a predefined assignment procedure, which might be executed with delay.

```
AssignTColorBoxStyle(FGColorBox, (System::Set<TColorBoxStyles, cbStandardColors, cbCustomColors>() <<
```

9.2.11 Lists

Not only individual values can be assigned, but entire lists of values can also be passed to list boxes, The values either directly passed into an array as in simple lists or they are passed as items to a list member. In both cases the values are listed inside of parenthesis "(" ... ")":

9.2.11.1 Simple lists

Examples of a simple lists are *ColWidths* or *RowHeights* in a *TStringGrid* The *DFM* code which assign some column widths might look like:

```
object StringGrid1: TStringGrid
...
  ColWidths = (
    10
    100
    ...
    )
```

For such lists assignment procedures can be defined, simitar as to other DFM assignments. For *TStringGrid* the following two procedures are predefined:

```
void AssignTStringGridColWidths(TStringGrid* xp, int xi, int xiIndex);
void AssignTStringGridRowHeights(TStringGrid* xp, int xi, int xiIndex);
```

Here the assignment procedure a third parameter is passed, which is the number of the list value.

9.2.11.2 List items

Examples of components with list properties are TListBox, TComboBox, etc. The elements that are assigned to a combo box, for example, are put in parenthesis in a DFM file, like:

```
object ComboBox1: TComboBox
 ...
  Items.Strings = (
    'first'
    'second'
  )
end
```

By default, this is translated by Delphi2Cpp in the following way:

```
ComboBox1 = new TComboBox(this);
...
ComboBox1->Items->Add(L"first");
ComboBox1->Items->Add(L"second");
```

Here too, Delphi2Cpp can be configured to output special assignment functions. With *TComboBox* as type and *Items* as name part results:

```
ComboBox1 = new TComboBox(this);
...
AssignTComboBoxItems(ComboBox1, L"first", 0);
AssignTComboBoxItems(ComboBox1, L"second", 1);
```

Here the assignment procedure a third parameter is passed, which is the number of the list value.

The type of the Items property is *TStrings* and *TStrings* has the property:

```
property Strings[Index: Integer]: string read Get write Put; default;
```

So also an assignment procedure for all *TStrings* can be defined:

```
ComboBox1 = new TComboBox(this);
...
AssignTStringsStrings(ComboBox1->Items, L"first", 0);
AssignTStringsStrings(ComboBox1->Items, L"second", 1);
```

However, this procedure would be applied to all TStrings, not just those from TComboBox, and it would not be applied if AssignTComboBoxItems also existed.

9.2.12 Collections

Another special kind of assignments are collections. Their values are listed as items in angle brackets "<" ... ">". Each item consists of a series of assignments within an item structure that begins with the keyword *item* and ends with the keyword *end*. For example, a status bar consists of such items

```
object StatusBar1: TStatusBar
...
  Panels = <
    item
      Width = 150
    end
    item
      Width = 50
    end>
end
```

By default, this example is translated as follows:

```
StatusBar1 = new TStatusBar(this);
...
StatusBar1->Panels->Add();
StatusBar1->Panels->Items[0]->Width = 150;
StatusBar1->Panels->Add();
StatusBar1->Panels->Items[1]->Width = 50;
```

However, not all collection classes have an *Add*-member function. For example *TFieldDefs*, which is used in a *TClientDataSet* doesn't have this method. It has the member AddFieldDef instead. For such cases special assignments can be defined again. By user of the type *TFieldDef* and the name part item a function can be defined, by which the following code can be translated:

```
Object ClientDataSet1: TClientDataSet
 ...
  FieldDefs = <
    Item
      Name = 'Species No'
      DataType = ftFloat
    end
    Item
      Name = 'Category'
      DataType = ftString
      Size = 15
    end
```

The translated code then is:

```
ClientDataSet1 = new TClientDataSet(this);
...
GetTFieldDefsitem(ClientDataSet1->FieldDefs, 0)->Name = L"Species No";
GetTFieldDefsitem(ClientDataSet1->FieldDefs, 0)->DataType = ftFloat;
GetTFieldDefsitem(ClientDataSet1->FieldDefs, 1)->Name = L"Category";
GetTFieldDefsitem(ClientDataSet1->FieldDefs, 1)->DataType = ftString;
GetTFieldDefsitem(ClientDataSet1->FieldDefs, 1)->Size = 15;
```

The function *GetTFieldDefsitem* will create e new TFielDef if there is no for the current index, otherwise it will return the existing one.

For other compilers then C++Builder the output will be:

```
ClientDataSet1 = new TClientDataSet(this);
...
GetTFieldDefsitem(ClientDataSet1->FieldDefs, 0)->WritePropertyName(L"Species No");
GetTFieldDefsitem(ClientDataSet1->FieldDefs, 0)->WritePropertyDataType(ftFloat);
GetTFieldDefsitem(ClientDataSet1->FieldDefs, 1)->WritePropertyName(L"Category");
GetTFieldDefsitem(ClientDataSet1->FieldDefs, 1)->WritePropertyDataType(ftString);
GetTFieldDefsitem(ClientDataSet1->FieldDefs, 1)->WritePropertySize(15);
```

9.2.13 Setting the parent

A special case when converting *DFM* files are parents. The Parent property of a control is not set in the *DFM* file expilcitely, nevertheless *Delphi2Cpp* automatically sets the parent for all newly created controls. It is necessary, because all displayable components, i.e. all controls, must have a parent in order to be visible. Position properties of these controls then apply relative to the position of the parent.

Assignment procedures can also be defined for setting the parent property, like

```
void AssignTSplitterParent(TSplitter* xp, TControl* xpParent);
```

This procedure is predefined.

9.2.14 List of predefined DFM routines

Some procedures for the conversion of DFM files are predefined in d2c_dfm.h/.cpp, which is provided by the Delphi2Cpp installer . Their names are constructed in the same way as other user defined *DFM* assignment routines.. The *DfmRoutines.txt* file in the project folder contains the name parts for *DFM* conversion routines which were used in applications from which the *DFM* feature of *Delphi2Cpp* was developed. *DfmRoutines.txt* can be opened and inserted into project files directly in the according dialog.

1. Assignment

```
void AssignTIconData(TForm* xp, const System::DynamicArray<System::Byte>& Bytes);
void AssignTImageListBitmap(TImageList* xp, const System::DynamicArray<System::Byte>& Bytes);
void AssignTPictureData(TImage* pImage, const System::DynamicArray<System::Byte>& Bytes);

void AssignFormTextHeight(TForm* xpForm, int xi);
void AssignTBitmapData(TSpeedButton* xp, const System::DynamicArray<System::Byte>& xBytes);
void AssignFormPixelsPerInch(TForm* xp, int xi);
void AssignDataModuleHeight(TDataModule* xp, int xi);
void AssignDataModulePixelsPerInch(TDataModule* xp, int xi);
void AssignDataModuleWidth(TDataModule* xp, int xi);

void AssignTStringGridColWidths(TStringGrid* xp, int xi, int xiIndex);
void AssignTStringGridRowHeights(TStringGrid* xp, int xi, int xiIndex);


void AssignControlIsControl(TControl* xp, bool xb);
```

// requires parent

```
void AssignTRichEditHideSelection(TRichEdit* xp, bool xb);
void AssignTColorBoxStyle(TColorBox* xp, const System::Set<TColorBoxStyles, cbStandardColors, cbCustomC
```

2. Creation

```
TMenuItem* CreateTMainMenuTMenuItem(TMainMenu* xp);
TMenuItem* CreateTMenuItemTMenuItem(TMenuItem* xp);
```

3. Items

```
TFieldDef* GetTFieldDefsitem(TFieldDefs* xp, int xiIndex)
```

4. Events

```
void OnTDataSetBegin(TDataSet* xp);
void OnTDataSetEnd(TDataSet* xp);
void OnTSplitterBegin(TSplitter* xp);
void OnTSplitterEnd(TSplitter* xp);
```

An example for the implementation of such a routine is:

```
TMenuItem* CreateTMainMenuTMenuItem(TMainMenu* xp)
{
    TMenuItem* pItem = new TMenuItem(xp);
    xp->Items->Add(pItem);
    return pItem;
}
```

9.3 Creating Forms dynamically

Compilers other than the C++Builder don't know *DFM* files and therefore forms have to be created dynamically. In Cpp-Builder projects form can be created by means of DFM files like in Delphi, But forms may be created dynamicall too. I fhte option to convert *DFM* files is enabled, the dpr file will be changed accordingly.

The two options are explained using the example of the GraphEx demo, which is one of the Embaradero example applications. graphex.dpr looks like:

```
program GraphEx;

uses
  Forms,
  GraphWin in 'GraphWin.pas' {Form1},
  BMPDlg in 'BMPDlg.pas' {NewBMPForm},
  Vcl.Themes in 'Vcl.Themes.pas';

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.CreateForm(TNewBMPForm, NewBMPForm);
  Application.Run;
end.
```

This becomes to:

```
using graphex.dfm                                          dybnamic creation at runtime

#include <vcl.h>                                           #include <vcl.h>
#pragma hdrstop                                            #pragma hdrstop
#include <tchar.h>                                         #include <tchar.h>
//-----------------------------------------------------    //------------------------------------------
USEFORM("GraphWin.cpp", Form1);                            #include "GraphWin.h"
USEFORM("BMPDlg.cpp", NewBMPForm);                         #include "BMPDlg.h"
//-----------------------------------------------------    //------------------------------------------
```

```
int WINAPI _tWinMain(HINSTANCE, HINSTANCE, LPTSTR, int)
{
    try
    {
            Application->Initialize();
            Application->MainFormOnTaskBar = true;
            Application->CreateForm(__classid(TForm1), &Form1);
            Application->Run();
    }
    catch (Exception &exception)
    {
            Application->ShowException(&exception);
    }
    catch (...)
    {
            try
            {
                    throw Exception("");
            }
            catch (Exception &exception)
            {
                    Application->ShowException(&exception);
            }
    }
    return 0;
}
```

```
int WINAPI _tWinMain(HINSTANCE, HINSTANCE, L
{
     try
     {
            Application->Initialize();
            Form1 = new TForm1(Application,
            NewBMPForm = new TNewBMPForm(App
            Application->Run();
                    Form1->ShowModal();
      }
      catch (Exception& exception)
         {
                    Application->ShowExcepti
      }
      catch (...)
      {
            try
            {
                            throw Exception('
            }
            catch (Exception& exception)
                    {
                    Application->ShowExcepti
                    }
      }
      return 0;
}
```

The differences are:

1. for pure C++ there will be no *DFM* file (otherwise components could be duplicated)
2. the pure C++ code doesn't need the *USEFORM* macro.
3. the pure C++ code doesn't use the Application->CreateForm method, but create the forms with new directly.
4. the pure C++ code needs an additional call of ShowModal for the main form of the application

The first form automatically created with *Application.CreateForm* becomes the main form of the application.

The call of *Application->Run* in the pure C++ code sometimes takes care that the code is cleaned correctly, when the application is closed.

It is very important, that forms have to be constructed like

```
Form1 = new TForm(this, 0);  // in Delphi: CreateNew
```

with the second parameter being a dummy parameter that distinguishes it from the normally used constructor with one parameter only. In Vcl.Forms.pas two constructors for *TForm* are defined:

```
constructor TCustomForm.Create(AOwner: TComponent);
constructor TCustomForm.CreateNew(AOwner: TComponent; Dummy: Integer = 0);
```

to which the constructors in Vcl.Forms.hpp correspond:

```
/* TCustomForm.Create */ inline __fastcall virtual TForm(System::Classes::TComponent* AOwner) : TCustom
/* TCustomForm.CreateNew */ inline __fastcall virtual TForm(System::Classes::TComponent* AOwner, int Du
```

When the first constructor is called, it always tries to read the DFM file, which then results in an error: *EResNotFound* exception will be thrown.

9.3.1 EResNotFound

If all components of a form are generated by code in C++Builder without using the DFM file, the constructor like

```
Form1 = new TForm(this);
```

fails with the meessage:

```
In the project .. an exception of the class EResNotFound with the message 'Resource T... not found' has
```

When using this constructor, the *InternalReadComponentRes* method is called indirectly. Since the resource is not found, the exception is thrown. Therefore the constructor

```
Form1 = new TForm(this, 0);
```

has to be used, with the second parameter being a dummy parameter that distinguishes it from the other constructor (in Delphi it is the CreateNew constructor).

The same problem arises when generating frames dynamically. Unfortunately, the alternative constructor is not defined here. Therefore, a dynamically generated frame must be derived from a specially defined *TCustomDynFrame*.
TCustomDynFrame is a copy of *TCustomFrame* that defines the second constructor instead of the first.

9.3.2 Main form

The first form automatically created with *Application.CreateForm* becomes the main form of the application. However, if the form is created dynamically with new and shown with *ShowModal*, it does not have this status. As a result, no icon is displayed in the taskbar for the application and it is no longer possible to switch back to the application once it has been minimized.

The read-only property Application.*MainForm* cannot be easily changed, but overriding the *CreateParams* function helps here. Since Delphi2Cpp translates Delphi files independently without project information, it does not know which form files are used as main forms. The *CreateParams* function is therefore only output as a comment. If a form is used as the main form, the comment characters have to be be removed manually.

```
//# please uncomment for main form
//# void __fastcall CreateParams(Vcl::Controls::TCreateParams &Params)
  //# {inherited::CreateParams(Params); Params.ExStyle = Params.ExStyle | WS_EX_APPWINDOW;}
```

->

```
void __fastcall CreateParams(Vcl::Controls::TCreateParams &Params)
  {inherited::CreateParams(Params); Params.ExStyle = Params.ExStyle | WS_EX_APPWINDOW;}
```

9.4 Creating Frames dynamically

As already mentioned for the creation of forms, these must not be created with the usual constructor (with only one parameter) if there is no longer an associated DFM file, as this constructor always tries to read this file. But while with forms you can simply use the other constructor (with the additional dummy parameter) in this case, this is not possible with frames because there is no such second constructor there. The only way out in this case is to derive frames to be created dynamically at runtime not from *TFrame*, but from a newly created class that contains the required constructor. *TCustomDynFrame* consists of the code of the *TCustomFrame* class translated into C++, whereby its constructor is replaced by the required constructor.

For example the class *TFancyFrame* from the Embarcadero frames demo then becomes to:

```
class TFancyFrame : public TCustomDynFrame
{
__published:
public:
  TDBMemo* DBMemo1;
  TDBImage* DBImage1;
  TSplitter* Splitter1;
private:
    /* Private declarations */
public:
    /* Public declarations */
  typedef TCustomDynFrame inherited;
  __fastcall TFancyFrame(TComponent* AOwner, int Dummy);
};
```

In the constructor of *TDataFrame*, which is part of the same demo, a *TFancyFrame* will be constructed with:

```
 FancyFrame1 = new TFancyFrame(this, 0);
```

10 Recursive translation

A recursive translation starts the translation of a selected file and looks up all the files from which it depends and translates them too. The start file and a target folder are selected by the

Start parameter dialog.

When the recursive translation is started, at first the start file is processed like described for the translation of a single file. But in contrast to that case, all files that are used in the first file are remembered and as soon as the conversion of start file is completed the translation of the first remembered file is started. Only files that are found in the set of folders of files, which might be be translated are remembered. All files from which this second file depends are remembered too, and so on. The result of the translation of the start file is written into the target folder and the results of the other translations are written into folders, with retention of the original relative file structure.

11 What is partially translated

Some features of Delphi can be translated partly only.

Variant parts in records
Visibility of class members

Virtual class methods
Abstract classes cannot be created, they have to made non-abstract before
A creation of class instances from class references is possible only, if the class has a standard constructor
API functions often are specified too vaguely in Delphi

11.1 API parameter casts

The Delphi files, which bridge the gap between the Delphi code and the API of the operation system, sometimes are too vague to allow a precise back translation. For example the third parameter of the function *SetFilePointer* in Winapi.Windows.pas is specified as *Pointer*:

```
function SetFilePointer(hFile: THandle; lDistanceToMove: Longint;
  lpDistanceToMoveHigh: Pointer; dwMoveMethod: DWORD): DWORD; stdcall;
```

The original specification is:

```
WINBASEAPI
DWORD
WINAPI
SetFilePointer(
    _In_ HANDLE hFile,
    _In_ LONG lDistanceToMove,
    _Inout_opt_ PLONG lpDistanceToMoveHigh,
    _In_ DWORD dwMoveMethod
    );
```

The type of the third parameter is specified here as *PLONG*. If a void Pointer is passed instead of a *PLONG* Visual Studio produces the error message:

```
Conversion of argument 3 from "void *" to "PLONG" is not possible
```

Another example:

```
type DWORD = Cardinal;
```

Delphi2Cpp converts a Cardinal to unsigned int. But it's not possible to assign an unsigned int* to PDWORD or to LPDWORD in C++,

12 What is not translated

There are some principle problems at the conversion of Delphi code to C++ which cannot be resolved by an automatic translator. But even things which *Delphi2Cpp*, normally can handle may fail in complex nested cases. Sometimes Delphi2Cpp generates explicit "todo"-comments where something has to be completed manually.

The conversion to C++Builder code seamlessly works together with the existing adoption to the Delphi RTL/VCL, but manual justifications to some helper names might be necessary. *Delphi2Cpp* makes little effort to cooperate with own Delphi code. An example of using a Delphi interface is here. If you need more, please contact me.

Some Delphi constructs, which aren't, automatically translated yet are:

- Inline assembler code in Delphi and C++ almost are identically. Delphi2Cpp doesn't translate these parts.but only copies them.

- Code that relies on the internal members or memory layout of Delphi types cannot be converted automatically.
- *Delphi2Cpp* always assumes unique names.But e.g. there might be symbols from the operation system, which differ in notation..
- parameters for destructors are ignored
- In Delphi everything inside of a unit is accessible to each other. As s compromise only classes are made friends to each others
- Manual post-processing to achieve const-correctness is necessary.
- Multithreading classes and routines are formally translated, but not checked by an expert
- Resource strings simply are treated as non-resource strings
- In C++ classes with abstract methods cannot be created
- The consequences of the ZEROBASEDSTRING directive are not corrected automatically.
- 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.
- Advanced techniques such as ActiveX, COM, CORBA etc. are not specifically supported
- For C++Builder Variant is supported for other compilers not, but TVarRec
- At the current state Delphi2Cpp doesn't deal with method resolution clauses
- Delphi2Cpp has no solution to simulate the results of overwritten *DefineProperty* functions at the conversion of dfm-files.

Special problems:

lifetime extension of bound variables
https://isocpp.org/wiki/faq/strange-inheritance#calling-virtuals-from-ctors

12.1 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 first 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 *Delphi2Cpp 2*, 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,

12.2 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 Delphi2Cpp, 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 Delphi2Cpp 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.

12.3 Low level code

It is not possible to convert code automatically, that uses low level tricky pointer manipulation, which in addition may rely on the memory layout of the intrinsic Delphi types as in the following example:

```
procedure SetTBytesLength(var b : TBytes; len : integer);
```

```
type
  PDynArrayRec = ^TDynArrayRec;
  TDynArrayRec = packed record
    RefCnt: LongInt;
    Length: NativeInt;
  end;
var
  p : Pointer;
  oldL, minL : NativeInt;
begin
  if len = 0 then begin
    b := nil
  end
  else begin
    p := Pointer(b);
    oldL := 0;
    if p <> nil then begin
      dec(PByte(p), SizeOf(TDynArrayRec));
      oldL := PDynArrayRec(p).Length
    end;

    if (p = nil) or (PDynArrayRec(p).RefCnt = 1) then begin
      ReallocMem(p, SizeOf(TDynArrayRec) + len)
    end
    else begin
     ...
```

For other compilers then C++Builder Delphi2Cpp uses a std::vector as substitute for a dynamic array. std::vector has no *RefCnt* ond no *Length* element. The translation of the example case is fortunately easy because an existing method can be used.

```
void SetTBytesLength(TBytes& B, int Len)
{
  B.resize(Len);
}
```

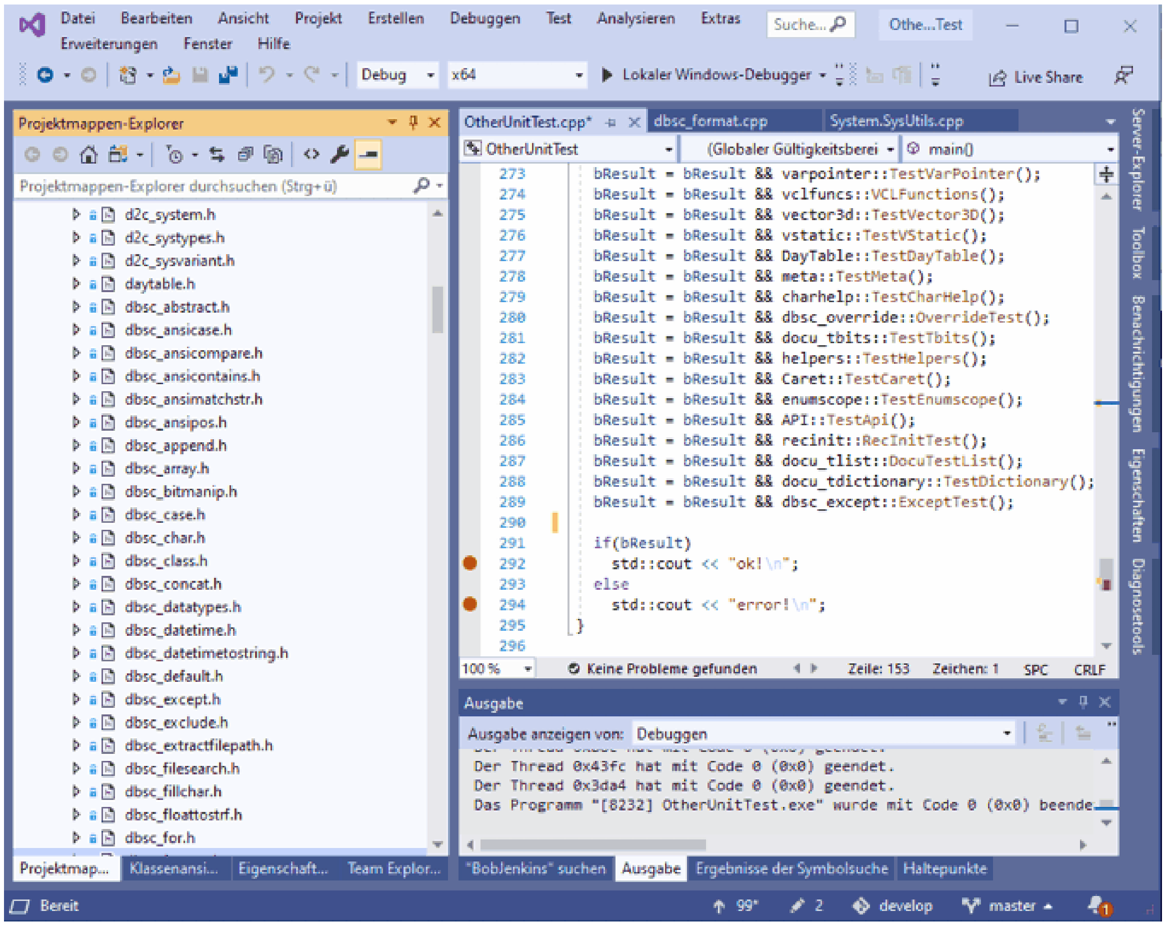
13 Unit tests

The quality of the translation results of Delphi code to C++ with Delphi2Cpp 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 Delphi2Cpp to C++. The translated test files then are inserted into a C++ test application (C++-Builder or Visual C++ respectively). There the tests are repeated then in C++.



```
    bResult = bResult && varpointer::TestVarPointer();
    bResult = bResult && vclfuncs::VCLFunctions();
    bResult = bResult && vector3d::TestVector3D();
    bResult = bResult && vstatic::TestVStatic();
    bResult = bResult && DayTable::TestDayTable();
    bResult = bResult && meta::TestMeta();
    bResult = bResult && charhelp::TestCharHelp();
    bResult = bResult && dbsc_override::OverrideTest();
    bResult = bResult && docu_tbits::TestTbits();
    bResult = bResult && helpers::TestHelpers();
    bResult = bResult && Caret::TestCaret();
    bResult = bResult && enumscope::TestEnumscope();
    bResult = bResult && API::TestApi();
    bResult = bResult && recinit::RecInitTest();
    bResult = bResult && docu_tlist::DocuTestList();
    bResult = bResult && docu_tdictionary::TestDictionary();
    bResult = bResult && dbsc_except::ExceptTest();

    if(bResult)
      std::cout << "ok!\n";
    else
      std::cout << "error!\n";
}
```

```
Der Thread 0x43fc hat mit Code 0 (0x0) geendet.
Der Thread 0x3da4 hat mit Code 0 (0x0) geendet.
Das Programm "[8232] OtherUnitTest.exe" wurde mit Code 0 (0x0) beende
```

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

13.1 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 their translation with Delphi2Cpp is 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 Delphi2Cpp compiles and works without additional manual processing without faults.

```
bool FormatTest1()
```

```
{
  bool result = false;
  result = true;
  // Just 1 data item
  result = result && (Format(L"%s", OpenArray<TVarRec>(String(L"Hello"))) == L"Hello");

  // A mix of literal text and a data item
  result = result && (Format(L"String = %s", OpenArray<TVarRec>(String(L"Hello"))) == L"String = Hello"
   //ShowMessage('');

  // Examples of each of the data types
  result = result && (Format(L"Decimal          = %d", OpenArray<TVarRec>(-123)) == L"Decimal
  result = result && (Format(L"Exponent         = %e", OpenArray<TVarRec>(12345.678L)) == L"Exponent
  result = result && (Format(L"Fixed            = %f", OpenArray<TVarRec>(12345.678L)) == L"Fixed
  result = result && (Format(L"General          = %g", OpenArray<TVarRec>(12345.678L)) == L"General
  result = result && (Format(L"Number           = %n", OpenArray<TVarRec>(12345.678L)) == L"Number
  result = result && (Format(L"Money            = %m", OpenArray<TVarRec>(12345.678L)) == L"Money
   // makes no sense under C#
   // result := result and (Format('Pointer          = %p', [addr(text)]) = 'Pointer          = 0069FC9
  result = result && (Format(L"String           = %s", OpenArray<TVarRec>(String(L"Hello"))) == L"Strin
  result = result && (Format(L"Unsigned decimal = %u", OpenArray<TVarRec>(123)) == L"Unsigned decimal =
  result = result && (Format(L"Hexadecimal      = %x", OpenArray<TVarRec>(140)) == L"Hexadecimal      =
  return result;
}

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(L"Padded decimal    = <%7d>", OpenArray<TVarRec>(1234)) == L"Padded decima

  // With the '-' operator, the data is left justified
  result = result && (Format(L"Justified decimal = <%-7d>", OpenArray<TVarRec>(1234)) == L"Justified de

  // The precision value forces 0 padding to the desired size
  result = result && (Format(L"0 padded decimal  = <%.6d>", OpenArray<TVarRec>(1234)) == L"0 padded dec

  // A combination of width and precision
  // Note that width value precedes the precision value
  result = result && (Format(L"Width + precision = <%8.6d>", OpenArray<TVarRec>(1234)) == L"Width + pre

  // The index value allows the next value in the data array
  // to be changed
  result = result && (Format(L"Reposition after 3 strings = %s %s %s %1:s %s", OpenArray<TVarRec>(Strin

  // 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(L"In line           = <%10.4d>", OpenArray<TVarRec>(1234)) == L"In line
  result = result && (Format(L"Part data driven  = <%*.4d>", OpenArray<TVarRec>(10, 1234)) == L"Part da
  result = result && (Format(L"Data driven       = <%*.*d>", OpenArray<TVarRec>(10, 4, 1234)) == L"Data
  return result;
}

bool FloatToStrTest1()
{
  bool result = false;
  long double amount1 = 0.0L;
  long double amount2 = 0.0L;
  long double amount3 = 0.0L;
  result = true;
  amount1 = 1234567890.123456789L;  // High precision number
  amount2 = 1234567890123456.123L;  // High mantissa digits
  amount3 = 1E100L;                 // High value number
  result = result && (FloatToStr(amount1) == L"1234567890,12346");
  result = result && (FloatToStr(amount2) == L"1,23456789012346E15");
  result = result && (FloatToStr(amount3) == L"1E100");
  return result;
}

bool FormatFloatTest1()
{
  bool result = false;
  long double flt = 0.0L;
  result = true;
  // Set up our floating point number
  flt = 1234.567L;

  // Display a sample value using all of the format options

  // Round out the decimal value
  result = result && (FormatFloat(L"#####", flt) == L"1235");
  result = result && (FormatFloat(L"00000", flt) == L"01235");
  result = result && (FormatFloat(L"0", flt) == L"1235");
  result = result && (FormatFloat(L"#,##0", flt) == L"1.235");
  result = result && (FormatFloat(L",0", flt) == L"1.235");

  // Include the decimal value
  result = result && (FormatFloat(L"0.####", flt) == L"1234,567");
  result = result && (FormatFloat(L"0.0000", flt) == L"1234,5670");


  // Scientific format
  result = result && (FormatFloat(L"0.0000000E+00", flt) == L"1,2345670E+03");
  result = result && (FormatFloat(L"0.0000000E-00", flt) == L"1,2345670E03");
  result = result && (FormatFloat(L"#.#######E-##", flt) == L"1,234567E3");

  // Include freeform text
  result = result && (FormatFloat(L"\"Value = \"0.0", flt) == L"Value = 1234,6");


  // Different formatting for negative numbers
  result = result && (FormatFloat(L"0.0", -1234.567) == L"-1234,6");
  result = result && (FormatFloat(L"0.0 \"CR\";0.0 \"DB\"", -1234.567) == L"1234,6 DB");
  result = result && (FormatFloat(L"0.0 \"CR\";0.0 \"DB\"", 1234.567L) == L"1234,6 CR");

  // Different format for zero value
  result = result && (FormatFloat(L"0.0", 0.0L) == L"0,0");
  result = result && (FormatFloat(L"0.0;-0.0;\"Nothing\"", 0.0L) == L"Nothing");
  return result;
}

bool FormatTest()
{
  bool result = false;
  result = true;
  result = result && FormatTest1();
  result = result && FormatTest2();
  result = result && FloatToStrTest1();
  result = result && FormatFloatTest1();
  return result;
}
```

13.2 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.

http://docwiki.embarcadero.com/CodeExamples/Rio/en/Generics_Collections_TDictionary_(Delphi)">Generics.Collections.TDictionary

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 Delphi2Cpp doesn't require any further manual post-processing and is shown below. A couple of explanations follow.

```
#include "System.Types.h"
#include "System.Sysutils.h"
#include "System.Math.h"
#include "System.Generics.Collections.h"
```

```
#include "d2c_sysiter.h"

using namespace std;
using namespace System;
using namespace System::Generics::Collections;
using namespace System::Math;
using namespace System::Sysutils;
using namespace System::Types;

namespace docu_tdictionary
{



class TCity : public TObject
{
public:
  typedef TObject inherited;
  String Country;
  double Latitude;
  double Longitude;
  void InitMembers(){Latitude = 0.0; Longitude = 0.0;}
  TCity() {InitMembers();}
};
const double epsilon = 0.0000001;

bool TestDictionary1()
{
  bool result = false;
  TDictionary<UnicodeString, TCity*>* Dictionary = nullptr;
  TCity* City = nullptr;
  TCity* Value = nullptr;
  String key;
  bool bTest = false;
  String s;
  result = true;
  /* Create the dictionary. */
  Dictionary = new TDictionary<UnicodeString, TCity*>();
  City = new TCity();
  /* Add some key-value pairs to the dictionary. */
  City->Country = L"Romania";
  City->Latitude = 47.16;
  City->Longitude = 27.58;
  Dictionary->Add(L"Iasi", City);
  City = new TCity();
  City->Country = L"United Kingdom";
  City->Latitude = 51.5;
  City->Longitude = -0.17;
  Dictionary->Add(L"London", City);
  City = new TCity();
  City->Country = L"Argentina";
  /* Notice the wrong coordinates */
  City->Latitude = 0;
  City->Longitude = 0;
  Dictionary->Add(L"Buenos Aires", City);

  /* Display the current number of key-value entries. */
  result = result && (Dictionary->ReadPropertyCount() == 3);

  // Try looking up "Iasi".
  if(Dictionary->TryGetValue(L"Iasi", City) == true)
  {
    result = result && (City->Country == L"Romania");
  }
  else
  result = false;

  /* Remove the "Iasi" key from dictionary. */
  Dictionary->Remove(L"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(L"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(L"London"))
    {
      Dictionary->TryGetValue(L"London", City);
      if((City->Country == L"United Kingdom") && (CompareValue(City->Latitude, 51.5, epsilon) == EqualsVa
        result = result && (City->Country == L"United Kingdom");
      else
        result = false;
      City = new TCity();
      City->Country = L"United Kingdom";
      City->Latitude = 51.5;
      City->Longitude = -0.17;
      if(Dictionary->containsValue(City))
        result = false;
      delete City;
    }
    else
    result = false;

    /* Update the coordinates to the correct ones. */
    City = new TCity();
    City->Country = L"Argentina";
    City->Latitude = -34.6;
    City->Longitude = -58.45;
    Dictionary->AddOrSetValue(L"Buenos Aires", City);

    /* Generate the exception "Duplicates not allowed". */
    try
    {
      bTest = false;
      Dictionary->Add(L"Buenos Aires", City);
    }
    catch(Exception*)
    {
      bTest = true;
    }
    result = result && (bTest == true);
    bTest = false;
    /* Display all countries. */
    for(TCity* element_0 : *Dictionary->ReadPropertyValues())
    {
      Value = element_0;
      if(Value->Country == L"Argentina")
        bTest = true;
    }
    result = result && (bTest == true);
    bTest = false;
    /* Iterate through all keys in the dictionary and display their coordinates. */
    for(UnicodeString element_0 : *Dictionary->ReadPropertyKeys())
    {
      key = element_0;
      s = FloatToStrF(Dictionary->ReadPropertyItems(key)->Longitude, ffFixed, 4, 2);
      if(s == L"-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->ReadPropertyCount() == 0);

    /* Free the memory allocated for the dictionary. */
    delete Dictionary;
    delete City;
    return result;
  }

  }  // namespace docu_tdictionary
```

Though the C++ version of TDictionary has the same interfaces as the Delphi version, it isn't a direct translation of the Delphi code. To guarantee a smooth integration of TDictionary into other C++ code, the class is derived from std::unordered_map. Therefore class doesn't only dispose the Delphi enumerators but also of C++ iterators. The latter are used implecitely also at the range based for-loop:

for Key in Dictionary.Keys do

->

for(TCity* element_0 : *Dictionary->ReadPropertyValues())

The C++ translation from BobJenkinsHash in System.Generics.Defaults is used as hash function for the unordered map.

13.3 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 TPersitent, 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.)

```
#include "dbsc_tstringlist.h"
#include "d2c_convert.h"

using namespace std;
using namespace System;
using namespace System::Classes;

namespace dbsc_tstringlist
{
bool TStringListTest1()
{
  bool result = false;
  TStringList* animals = nullptr;            // 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(L"Cat");
  animals->Add(L"Mouse");
  animals->Add(L"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->ReadPropertyStrings(0) == L"Cat");
  result = result && (animals->ReadPropertyStrings(1) == L"Mouse");
  result = result && (animals->ReadPropertyStrings(2) == L"Giraffe");

  // Free up the list object
  delete animals;
  return result;
}
```

```
bool TStringListTest2()
{
  bool result = false;
  TStringList* Names = nullptr;            // Define our string list variable
  String ageStr;
  int i = 0;
  int stop = 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->WritePropertyCommaText(L"Neil=45, Brian=63, Jim=22");

  // And now find Brian's age
  ageStr = Names->ReadPropertyValues(L"Brian");

  // Display this value
  // ShowMessage('Brians age = '+ageStr);
  result = result && (ageStr == L"63");

  // Now display all name and age pair values
  for(stop = Names->ReadPropertyCount() - 1, i = 0; i <= stop; i++)
  {
     //ShowMessage(names.Names[i]+' is '+names.ValueFromIndex[i]);
    if(i == 0)
      result = result && (String(ustr2pwchar(Names->ReadPropertyNames(i))) == L"Neil") && (String(ustr2
    if(i == 1)
      result = result && (String(ustr2pwchar(Names->ReadPropertyNames(i))) == L"Brian") && (String(ustr
    if(i == 2)
      result = result && (String(ustr2pwchar(Names->ReadPropertyNames(i))) == L"Jim") && (String(ustr2p
  }

  // Free up the list object
  delete Names;
  return result;
}

bool TStringListTest3()
{
  bool result = false;
  TStringList* cars = nullptr;            // 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 overriden control variables
  cars->WritePropertyDelimiter(L' ');        // Each list item will be blank separated
  cars->WritePropertyQuoteChar(L'|');        // And each item will be quoted with |'s
  cars->WritePropertyDelimitedText(L"|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->ReadPropertyStrings(0) == L"Honda Jazz");
  result = result && (cars->ReadPropertyStrings(1) == L"Ford Mondeo");
  result = result && (cars->ReadPropertyStrings(2) == L"Jaguar \"E-type\"");

  // Free up the list object
  delete cars;
  return result;
}


bool TStringListTest()
{
  bool result = false;
  result = true;
  result = result && TStringListTest1();
  result = result && TStringListTest2();
  result = result && TStringListTest3();
  return result;
```

```
    }
    }  // namespace dbsc_tstringlist
```

14 Pretranslated C++ code

Delphi2Cpp 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.

14.1 Delphi RTL/VCL

The user's Delphi code is based on the Delphi RTL and the VCL The translations of the user's code therefore also need translations of these Delphi libraries.

C++ Builder

The C++ Builder already has its own version of the Delphi RTL/VCL with C++ interface files. *Delphi2Cpp* provides some additional helper files.

Other Compilers

For other compilers one could think this isn't a problem, since this code can be translated by *Delphi2Cpp* 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 function are built into the 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 C++ API functions and types of the operating system such that they are conform to Delphi. In C++ this conversion isn't necessary, you better use the original API instead.. In addition some parts of System.pas aren't needed in C++ at all.

Therefore some parts of the Delphi RTL are pre-translated and prepared to use with the code translated by *Delphi2Cpp*. Because Embarcadero has the copyright of the Delphi RTL/VCL the translated parts cannot be shipped with the *Delphi2Cpp* installer. However as customer of *Delphi2Cpp* 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 provide a proof of your Delphi ownership.

Some helping code is already delivered with the *Delphi2Cpp* installer:
It is recommended to prepare some files of the Delphi RTL

14.1.1 C++ code for C++Builder

The C++ Builder already has its own version of the Delphi RTL/VCL with C++ interface files. If the option to produce C++ for C++ Builder is enabled Delphi2Cpp tries to optimize the translated code to work together with these libraries. For the parts which are missing in System.pas, there is pre-translated code in the folder:

..\Delphi2Cpp\Source\C++Builder

You also should use the extended *System.pas* extension in

d2c_convert
d2c_openarray

d2c_smallstringconvert
d2c_sysexcept
d2c_sysfile
d2c_syshelper
d2c_sysiter
d2c_sysmath
d2c_sysstring
d2c_system
d2c_systypes

14.1.1.1 d2c_convert

d2c_convert contains Delphi2Cpp helper functions to convert different string and array types into each other.

```
AnsiString wstr2str(const System::WideString& xs);
WideString str2wstr(const System::AnsiString& xs);
AnsiChar wchar2char(WideChar xc);
WideChar char2wchar(AnsiChar xc);

System::AnsiString wstr2astr(const System::WideString& xs);
System::AnsiString ustr2astr(const System::UnicodeString& xs);
System::AnsiString sstr2astr(const System::SmallString<255>& xs);

System::WideString astr2wstr(const System::AnsiString& xs);
System::WideString ustr2wstr(const System::UnicodeString& xs);  // see WStrFromUStr
System::WideString sstr2wstr(const System::SmallString<255>& xs);

System::UnicodeString astr2ustr(const System::AnsiString& xs);
System::UnicodeString wstr2ustr(const System::WideString& xs);  // see UStrFromWStr
System::UnicodeString sstr2ustr(const System::SmallString<255>& xs);

void* astr2address(const System::AnsiString& xs, int index = 0);
void* wstr2address(const System::WideString& xs, int index = 0);
void* ustr2address(const System::UnicodeString& xs, int index = 0);
void* sstr2address(const System::SmallString<255>& xs, int index = 0);

System::PWideChar address2pwchar(void* p);

System::PAnsiChar astr2pchar(const System::AnsiString& xs, int index = 0);
System::PAnsiChar wstr2pchar(const System::WideString& xs, int index = 0);
System::PAnsiChar ustr2pchar(const System::UnicodeString& xs, int index = 0);
System::PAnsiChar sstr2pchar(const System::SmallString<255>& xs, int index = 0);

System::PWideChar astr2pwchar(const System::AnsiString& xs, int index = 0);
System::PWideChar wstr2pwchar(const System::WideString& xs, int index = 0);
System::PWideChar ustr2pwchar(const System::UnicodeString& xs, int index = 0);
System::PWideChar sstr2pwchar(const System::SmallString<255>& xs, int index = 0);
System::PWideChar ustr2punichar(const System::UnicodeString& xs, int index = 0);


inline System::ShortString ustr2sstr(const System::UnicodeString& xs) ...
```

inline System::ShortString astr2sstr(const System::AnsiString& xs) ...
template <class T> T* array2ptr(const DynamicArray<T>& s, int offset = 0) ...
inline unsigned char* bytearray2pbyte(const DynamicArray<unsigned char>& s, int offset = 0) ...
inline PAnsiChar bytearray2pchar(const DynamicArray<unsigned char>& s, int offset = 0) ...
inline void* bytearray2pvoid(const DynamicArray<unsigned char>& s, int offset = 0) ...
inline System::PAnsiChar wchararray2pchar(const DynamicArray<WideChar>& s, int offset = 0) ...
inline System::PWideChar wchararray2pwchar(const DynamicArray<WideChar>& s, int offset = 0) ...

template <typename Type, Type Low, Type High> System::Set<Type, Low, High> IntToSet(int xi) ...
template <typename Type, Type Low, Type High> int SetToInt(const System::Set<Type, Low, High>& xsi) ...
template <typename Type, Type Low, Type High> unsigned char ToByte(const System::Set<Type, Low, High>& xset) ...

template <unsigned char sz = 255> SmallString<sz> astr2sstr(const AnsiString xs) ...
template <unsigned char sz> SmallString<sz> wstr2sstr(const WideString xs) ...
template <unsigned char sz> SmallString<sz> sstr2ustr(const UnicodeString xs) ...
template <unsigned char sz> SmallString<255> sstr2sstr(const SmallString<sz>& xs) ...

template <class T> std::vector<T> DynArrayToVector(const DynamicArray<T> &arr) ...
template <class T> std::vector<T> move(DynamicArray<T>& source) ...
template <class T> DynamicArray<T> VecorToDynArray(const std::vector<T> &arr) ...

14.1.1.2 d2c_openarray

d2c_openarray contains Delphi2Cpp helper code for open array parameters. The C++Builder OPENARRAY macro is used to pass an array of values on the fly. In addition *Delphi2Cpp* provides an extended class OpenArrayEx by which dynamic and fixed arrays can be passed where a function expects an open array parameter. In addition there are cases, where open arrays are passed as var-parameters. Strings, SmallStrings, and fixed arrays con be passed to such parameters as well as dynamic arrays. For that case *Delphi2Cpp* uses a special template type *OpenArrayRef*, which is defined in *d2c_openarray.*

```
template <class T>
class OpenArrayEx
{
public:

        __fastcall OpenArrayEx(const DynamicArray<T>& src);
        __fastcall OpenArrayEx(const T* pArr, int Count);
        template <class InputIterator> __fastcallOpenArrayEx(InputIterator first,
InputIterator last);

        ...
};



template <class T>
class OpenArrayRef
{
public:
        __fastcall OpenArrayRef(DynamicArray<T>& arr);
```

```
            __fastcall OpenArrayRef(std::basic_string<T>& s);

            ...
        };
```

14.1.1.3 d2c_sysexcept

d2c_sysexcept contains a Delphi2Cpp helper enumeration of runtime errors.

14.1.1.4 d2c_sysfile

d2c_sysfile contains Delphi2Cpp helper code for basic file reading and writing routines. This version for C++Builder is analogous to the version for other compilers but with C++Builder string types.

14.1.1.5 d2c_syshelper

d2c_syshelper is a translation of the helper classes from *System.SysUtils* for C++Builder. These classes are needed sometimes to be able to translate other Delphi code, where functions from the helper classes are used.

```
struct TStringHelper
{
    TStringHelper(UnicodeString& Helped) : m_Helped(Helped) {}
    TStringHelper(const UnicodeString& Helped) : m_Helped(const_cast<UnicodeString&>(Helped)) {}
private:
    enum TSplitKind {StringSeparatorNoQuoted,
                                                        StringSeparatorQuoted,
                                                        CharSeparatorNoQuoted,
                                                        CharSeparatorQuoted };
    Char __fastcall GetChars(int Index) const;
    int __fastcall GetLength() const;
    static bool __fastcall CharInArray(const Char C, const Char* InArray, int iMaxIndex);
    TArray<String> __fastcall InternalSplit(TSplitKind SplitType, const Char* SeparatorC, int Separator
    int __fastcall IndexOfAny(const String* Values, int Values_maxidx, int& Index, int StartIndex);
    int __fastcall IndexOfAnyUnquoted(const String* Values, int Values_maxidx, Char StartQuote, Char En
    int __fastcall IndexOfQuoted(const String Value, Char StartQuote, Char EndQuote, int StartIndex);
    static int __fastcall InternalCompare(const String StrA, int IndexA, const String StrB, int IndexB,
    static int __fastcall InternalCompare(const String StrA, int IndexA, const String StrB, int IndexB,
    static unsigned long __fastcall InternalMapOptionsToFlags(TCompareOptions AOptions);
public:
    static const WideChar Empty[]; //  = L"";
            // Methods
    static String __fastcall Create(Char C, int Count);
    static String __fastcall Create(const Char* Value, int Value_maxidx, int StartIndex, int Length);
    static String __fastcall Create(const Char* Value, int Value_maxidx);
    static int __fastcall Compare(const String StrA, const String StrB);
    static int __fastcall Compare(const String StrA, const String StrB, TLocaleID LocaleID);
    static int __fastcall Compare(const String StrA, const String StrB, bool IgnoreCase); //deprecated
    static int __fastcall Compare(const String StrA, const String StrB, bool IgnoreCase, TLocaleID Loca
    static int __fastcall Compare(const String StrA, const String StrB, TCompareOptions Options);
    static int __fastcall Compare(const String StrA, const String StrB, TCompareOptions Options, TLocal
    static int __fastcall Compare(const String StrA, int IndexA, const String StrB, int IndexB, int Len
    static int __fastcall Compare(const String StrA, int IndexA, const String StrB, int IndexB, int Len
    static int __fastcall Compare(const String StrA, int IndexA, const String StrB, int IndexB, int Len
    static int __fastcall Compare(const String StrA, int IndexA, const String StrB, int IndexB, int Len
    static int __fastcall Compare(const String StrA, int IndexA, const String StrB, int IndexB, int Len
    static int __fastcall Compare(const String StrA, int IndexA, const String StrB, int IndexB, int Len
    static int __fastcall CompareOrdinal(const String StrA, const String StrB);
    static int __fastcall CompareOrdinal(const String StrA, int IndexA, const String StrB, int IndexB, 
    static int __fastcall CompareText(const String StrA, const String StrB);
    static String __fastcall parse(const int Value);
    static String __fastcall parse(const __int64 Value);
    static String __fastcall parse(const bool Value);
    static String __fastcall parse(const long double Value);
    static bool __fastcall ToBoolean(const String s);
    static int __fastcall toInteger(const String s);
           /// <summary>Class function to Convert a string to an Int64 value</summary>
    static __int64 __fastcall ToInt64(const String s);
    static float __fastcall ToSingle(const String s);
    static double __fastcall ToDouble(const String s);
    static long double __fastcall ToExtended(const String s);
    static String __fastcall LowerCase(const String s);
    static String __fastcall LowerCase(const String s, TLocaleOptions LocaleOptions);
    static String __fastcall UpperCase(const String s);
    static String __fastcall UpperCase(const String s, TLocaleOptions LocaleOptions);
    int __fastcall compareTo(const String StrB);
    bool __fastcall contains(const String Value);
    static String __fastcall Copy(const String Str);
    //void __fastcall CopyTo(int sourceIndex, Char* Destination, int Destination_maxidx, int Destinatio
    void __fastcall CopyTo(int sourceIndex, OpenArrayRef<WideChar> Destination, int DestinationIndex, i
    int __fastcall CountChar(const Char C);
    String __fastcall DeQuotedString();
    String __fastcall DeQuotedString(const Char QuoteChar);
    static bool __fastcall EndsText(const String ASubText, const String AText);
    bool __fastcall endsWith(const String Value);
    bool __fastcall endsWith(const String Value, bool IgnoreCase);
    bool __fastcall Equals(const String Value);
    static bool __fastcall Equals(const String A, const String B);
    static String __fastcall Format(const String Format, const TVarRec* Args, int Args_maxidx);
    int __fastcall GetHashCode();
    int __fastcall IndexOf(Char Value);
    int __fastcall IndexOf(const String Value);
    int __fastcall IndexOf(Char Value, int StartIndex);
    int __fastcall IndexOf(const String Value, int StartIndex);
    int __fastcall IndexOf(Char Value, int StartIndex, int Count);
    int __fastcall IndexOf(const String Value, int StartIndex, int Count);
    int __fastcall IndexOfAny(const Char* AnyOf, int AnyOf_maxidx);
    int __fastcall IndexOfAny(const Char* AnyOf, int AnyOf_maxidx, int StartIndex);
    int __fastcall IndexOfAny(const Char* AnyOf, int AnyOf_maxidx, int StartIndex, int Count);
           /// <summary>Index of any given chars, excluding those that are between quotes</summary>
    int __fastcall IndexOfAnyUnquoted(const Char* AnyOf, int AnyOf_maxidx, Char StartQuote, Char EndQuo
    int __fastcall IndexOfAnyUnquoted(const Char* AnyOf, int AnyOf_maxidx, Char StartQuote, Char EndQuo
    int __fastcall IndexOfAnyUnquoted(const Char* AnyOf, int AnyOf_maxidx, Char StartQuote, Char EndQuo
    String __fastcall Insert(int StartIndex, const String Value);
    bool __fastcall IsDelimiter(const String Delimiters, int Index);
    bool __fastcall IsEmpty();
    static bool __fastcall IsNullOrEmpty(const String Value);
    static bool __fastcall IsNullOrWhiteSpace(const String Value);
    static String __fastcall JOIN(const String separator, const TVarRec* Values, int Values_maxidx);
    static String __fastcall JOIN(const String separator, const String* Values, int Values_maxidx);
// todo  static String __fastcall JOIN(const String separator, IEnumerator<String>* const Values);
// todo  static String __fastcall JOIN(const String separator, IEnumerable<String>* const Values);
    static String __fastcall JOIN(const String separator, const String* Values, int Values_maxidx, int 
    int __fastcall LastDelimiter(const String delims);
    int __fastcall LastIndexOf(Char Value);
    int __fastcall LastIndexOf(const String Value);
    int __fastcall LastIndexOf(Char Value, int StartIndex);
    int __fastcall LastIndexOf(const String Value, int StartIndex);
    int __fastcall LastIndexOf(Char Value, int StartIndex, int Count);
    int __fastcall LastIndexOf(const String Value, int StartIndex, int Count);
    int __fastcall LastIndexOfAny(const Char* AnyOf, int AnyOf_maxidx);
    int __fastcall LastIndexOfAny(const Char* AnyOf, int AnyOf_maxidx, int StartIndex);
    int __fastcall LastIndexOfAny(const Char* AnyOf, int AnyOf_maxidx, int StartIndex, int Count);
    String __fastcall PadLeft(int TotalWidth);
    String __fastcall PadLeft(int TotalWidth, Char PaddingChar);
    String __fastcall PadRight(int TotalWidth);
    String __fastcall PadRight(int TotalWidth, Char PaddingChar);
    String __fastcall QuotedString();
    String __fastcall QuotedString(const Char QuoteChar);
    String __fastcall Remove(int StartIndex);
    String __fastcall Remove(int StartIndex, int Count);
    String __fastcall replace(Char OldChar, Char NewChar);
    String __fastcall replace(Char OldChar, Char NewChar, TReplaceFlags ReplaceFlags);
    String __fastcall replace(const String OldValue, const String NewValue);
    String __fastcall replace(const String OldValue, const String NewValue, TReplaceFlags ReplaceFlags)
    TArray<String> __fastcall split(const Char* separator, int separator_maxidx);
    TArray<String> __fastcall split(const Char* separator, int separator_maxidx, int Count);
    TArray<String> __fastcall split(const Char* separator, int separator_maxidx, TStringSplitOptions Op
    TArray<String> __fastcall split(const Char* separator, int separator_maxidx, int Count, TStringSpli
    TArray<String> __fastcall split(const String* separator, int separator_maxidx);
    TArray<String> __fastcall split(const String* separator, int separator_maxidx, int Count);
    TArray<String> __fastcall split(const String* separator, int separator_maxidx, TStringSplitOptions 
    TArray<String> __fastcall split(const String* separator, int separator_maxidx, int Count, TStringSp
    TArray<String> __fastcall split(const Char* separator, int separator_maxidx, Char Quote);
    TArray<String> __fastcall split(const Char* separator, int separator_maxidx, Char QuoteStart, Char 
    TArray<String> __fastcall split(const Char* separator, int separator_maxidx, Char QuoteStart, Char 
    TArray<String> __fastcall split(const Char* separator, int separator_maxidx, Char QuoteStart, Char 
    TArray<String> __fastcall split(const Char* separator, int separator_maxidx, Char QuoteStart, Char 
    TArray<String> __fastcall split(const String* separator, int separator_maxidx, Char Quote);
    TArray<String> __fastcall split(const String* separator, int separator_maxidx, Char QuoteStart, Cha
    TArray<String> __fastcall split(const String* separator, int separator_maxidx, Char QuoteStart, Cha
    TArray<String> __fastcall split(const String* separator, int separator_maxidx, Char QuoteStart, Cha
    TArray<String> __fastcall split(const String* separator, int separator_maxidx, Char QuoteStart, Cha
    bool __fastcall startsWith(const String Value);
    bool __fastcall startsWith(const String Value, bool IgnoreCase);
    String __fastcall SubString(int StartIndex);
    String __fastcall SubString(int StartIndex, int Length);
    bool __fastcall ToBoolean();
    int __fastcall toInteger();
            /// <summary>Converts the string to an Int64 value</summary>
    __int64 __fastcall ToInt64();
    float __fastcall ToSingle();
    double __fastcall ToDouble();
    long double __fastcall ToExtended();
    TArray<Char> __fastcall ToCharArray();
    TArray<Char> __fastcall ToCharArray(int StartIndex, int Length);
    String __fastcall ToLower();
    String __fastcall ToLower(TLocaleID LocaleID);
    String __fastcall ToLowerInvariant();
    String __fastcall toupper();
    String __fastcall toupper(TLocaleID LocaleID);
    String __fastcall ToUpperInvariant();
    String __fastcall Trim();
    String __fastcall TrimLeft();
    String __fastcall TrimRight();
    String __fastcall Trim(const Char* TrimChars, int TrimChars_maxidx);
    String __fastcall TrimLeft(const Char* TrimChars, int TrimChars_maxidx);
    String __fastcall TrimRight(const Char* TrimChars, int TrimChars_maxidx);
    String __fastcall TrimEnd(const Char* TrimChars, int TrimChars_maxidx)/*# 'use trimright' */;
    String __fastcall TrimStart(const Char* TrimChars, int TrimChars_maxidx)/*# 'use trimleft' */;
    __property Char Chars[int Index] = { read = GetChars };
    __property int Length = { read = GetLength };
private:
    UnicodeString& m_Helped;

};

struct TSingleHelper
{

    TSingleHelper(float& Helped) : m_Helped(Helped) {}
    TSingleHelper(const float& Helped) : m_Helped(const_cast<float&>(Helped)) {}
```

```
private:
    UInt8 __fastcall InternalGetBytes(unsigned int Index) const;
    UInt16 __fastcall InternalGetWords(unsigned int Index) const;
    void __fastcall InternalSetBytes(unsigned int Index, const UInt8 Value);
    void __fastcall InternalSetWords(unsigned int Index, const UInt16 Value);
    UInt8 __fastcall GetBytes(unsigned int Index) const;
    UInt16 __fastcall GetWords(unsigned int Index) const;
    unsigned __int64 __fastcall GetExp() const;
    unsigned __int64 __fastcall GetFrac() const;
    bool __fastcall GetSign() const;
  void __fastcall setbytes(unsigned int Index, const UInt8 Value);
  void __fastcall SetWords(unsigned int Index, const UInt16 Value);
  void __fastcall SetExp(unsigned __int64 NewExp);
  void __fastcall SetFrac(unsigned __int64 NewFrac);
  void __fastcall SetSign(bool NewSign);
public:
  static const float epsilon; //  = 1.4012984643248170709e-45F;
  static const float MaxValue; //  = 340282346638528859811704183484516925440.0F;
  static const float MinValue; //  = -340282346638528859811704183484516925440.0;
  static const float PositiveInfinity; //  = 1.0F / 0.0F;
  static const float NegativeInfinity; //  = -1.0F / 0.0F;
  static const float NaN; //  = 0.0F / 0.0F;
  int __fastcall Exponent();
  long double __fastcall Fraction();
  unsigned __int64 __fastcall mantissa();
  __property bool sign = { read = GetSign, write = SetSign };
  __property unsigned __int64 Exp = { read = GetExp, write = SetExp };
  __property unsigned __int64 Frac = { read = GetFrac, write = SetFrac };
    TFloatSpecial __fastcall SpecialType();
  void __fastcall BuildUp(const bool SignFlag, const unsigned __int64 mantissa, const int Exponent);
  String __fastcall toString();
  String __fastcall toString(const TFormatSettings& AFormatSettings);
  String __fastcall toString(const TFloatFormat Format, const int Precision, const int Digits);
  String __fastcall toString(const TFloatFormat Format, const int Precision, const int Digits, const TF
  bool __fastcall IsNan();
  bool __fastcall IsInfinity();
  bool __fastcall IsNegativeInfinity();
  bool __fastcall IsPositiveInfinity();
  __property UInt8 Bytes[unsigned int Index] = { read = GetBytes, write = setbytes };  // 0..3
  __property UInt16 Words[unsigned int Index] = { read = GetWords, write = SetWords }; // 0..1
  static String __fastcall toString(const float Value);
  static String __fastcall toString(const float Value, const TFormatSettings& AFormatSettings);
  static String __fastcall toString(const float Value, const TFloatFormat Format, const int Precision,
  static String __fastcall toString(const float Value, const TFloatFormat Format, const int Precision,
  static float __fastcall parse(const String s);
  static float __fastcall parse(const String s, const TFormatSettings& AFormatSettings);
  static bool __fastcall TryParse(const String s, float& Value);
  static bool __fastcall TryParse(const String s, float& Value, const TFormatSettings& AFormatSettings)
  static bool __fastcall IsNan(const float Value);
  static bool __fastcall IsInfinity(const float Value);
    static bool __fastcall IsNegativeInfinity(const float Value);
  static bool __fastcall IsPositiveInfinity(const float Value);
  static int __fastcall Size();
private:
  float& m_Helped;

};

struct TDoubleHelper
{
    TDoubleHelper(double& Helped) : m_Helped(Helped) {}
    TDoubleHelper(const double& Helped) : m_Helped(const_cast<double&>(Helped)) {}
private:
    UInt8 __fastcall InternalGetBytes(unsigned int Index) const;
    UInt16 __fastcall InternalGetWords(unsigned int Index) const;
    void __fastcall InternalSetBytes(unsigned int Index, const UInt8 Value);
    void __fastcall InternalSetWords(unsigned int Index, const UInt16 Value);
    UInt8 __fastcall GetBytes(unsigned int Index) const;
    UInt16 __fastcall GetWords(unsigned int Index) const;
    unsigned __int64 __fastcall GetExp() const;
    unsigned __int64 __fastcall GetFrac() const;
    bool __fastcall GetSign() const;
    void __fastcall setbytes(unsigned int Index, const UInt8 Value);
    void __fastcall SetWords(unsigned int Index, const UInt16 Value);
    void __fastcall SetExp(unsigned __int64 NewExp);
    void __fastcall SetFrac(unsigned __int64 NewFrac);
    void __fastcall SetSign(bool NewSign);
public:
  static const double epsilon; //  = 4.9406564584124654418e-324;
  static const double MaxValue; //  = 1.7976931348623157081e+308;
  static const double MinValue; //  = -1.7976931348623157081e+308;
  static const double PositiveInfinity; //  = 1.0 / 0.0;
  static const double NegativeInfinity; //  = -1.0 / 0.0;
  static const double NaN; //  = 0.0 / 0.0;
  int __fastcall Exponent();
  long double __fastcall Fraction();
  unsigned __int64 __fastcall mantissa();
  __property bool sign = { read = GetSign, write = SetSign };
  __property unsigned __int64 Exp = { read = GetExp, write = SetExp };
  __property unsigned __int64 Frac = { read = GetFrac, write = SetFrac };
  TFloatSpecial __fastcall SpecialType();
  void __fastcall BuildUp(const bool SignFlag, const unsigned __int64 mantissa, const int Exponent);
  String __fastcall toString();
  String __fastcall toString(const TFormatSettings& AFormatSettings);
    String __fastcall toString(const TFloatFormat Format, const int Precision, const int Digits);
  String __fastcall toString(const TFloatFormat Format, const int Precision, const int Digits, const TF
  bool __fastcall IsNan();
  bool __fastcall IsInfinity();
  bool __fastcall IsNegativeInfinity();
  bool __fastcall IsPositiveInfinity();
  __property UInt8 Bytes[unsigned int Index] = { read = GetBytes, write = setbytes };  // 0..7
  __property UInt16 Words[unsigned int Index] = { read = GetWords, write = SetWords }; // 0..3
  static String __fastcall toString(const double Value);
  static String __fastcall toString(const double Value, const TFormatSettings& AFormatSettings);
  static String __fastcall toString(const double Value, const TFloatFormat Format, const int Precision,
  static String __fastcall toString(const double Value, const TFloatFormat Format, const int Precision,
  static double __fastcall parse(const String s);
  static double __fastcall parse(const String s, const TFormatSettings& AFormatSettings);
  static bool __fastcall TryParse(const String s, double& Value);
  static bool __fastcall TryParse(const String s, double& Value, const TFormatSettings& AFormatSettings
  static bool __fastcall IsNan(const double Value);
  static bool __fastcall IsInfinity(const double Value);
  static bool __fastcall IsNegativeInfinity(const double Value);
  static bool __fastcall IsPositiveInfinity(const double Value);
  static int __fastcall Size();
private:
    double& m_Helped;

};

struct TExtendedHelper
{
    TExtendedHelper(long double& Helped) : m_Helped(Helped) {}
    TExtendedHelper(const long double& Helped) : m_Helped(const_cast<long double&>(Helped)) {}
private:
    UInt8 __fastcall InternalGetBytes(unsigned int Index) const;
    UInt16 __fastcall InternalGetWords(unsigned int Index) const;
    void __fastcall InternalSetBytes(unsigned int Index, const UInt8 Value);
    void __fastcall InternalSetWords(unsigned int Index, const UInt16 Value);
    UInt8 __fastcall GetBytes(unsigned int Index) const;
    UInt16 __fastcall GetWords(unsigned int Index) const;
    unsigned __int64 __fastcall GetExp() const;
    unsigned __int64 __fastcall GetFrac() const;
    bool __fastcall GetSign() const;
    void __fastcall setbytes(unsigned int Index, const UInt8 Value);
    void __fastcall SetWords(unsigned int Index, const UInt16 Value);
    void __fastcall SetExp(unsigned __int64 NewExp);
    void __fastcall SetFrac(unsigned __int64 NewFrac);
    void __fastcall SetSign(bool NewSign);
public:
  static const long double epsilon; //  = 4.9406564584124654418e-324L;
  static const long double MaxValue; //  = 1.7976931348623157081e+308L;
  static const long double MinValue; //  = -1.7976931348623157081e+308;
  static const long double PositiveInfinity; //  = 1.0L / 0.0L;
  static const long double NegativeInfinity; //  = -1.0L / 0.0L;
  static const long double NaN; //  = 0.0L / 0.0L;
  int __fastcall Exponent();
  long double __fastcall Fraction();
  unsigned __int64 __fastcall mantissa();
  __property bool sign = { read = GetSign, write = SetSign };
  __property unsigned __int64 Exp = { read = GetExp, write = SetExp };
  __property unsigned __int64 Frac = { read = GetFrac, write = SetFrac };
  TFloatSpecial __fastcall SpecialType();
  void __fastcall BuildUp(const bool SignFlag, const unsigned __int64 mantissa, const int Exponent);
  String __fastcall toString();
  String __fastcall toString(const TFormatSettings& AFormatSettings);
  String __fastcall toString(const TFloatFormat Format, const int Precision, const int Digits);
  String __fastcall toString(const TFloatFormat Format, const int Precision, const int Digits, const TF
  bool __fastcall IsNan();
  bool __fastcall IsInfinity();
    bool __fastcall IsNegativeInfinity();
  bool __fastcall IsPositiveInfinity();
  __property UInt8 Bytes[unsigned int Index] = { read = GetBytes, write = setbytes };  // 0..7 or 0..9
  __property UInt16 Words[unsigned int Index] = { read = GetWords, write = SetWords }; // 0..3 or 0..4
  static String __fastcall toString(const long double Value);
  static String __fastcall toString(const long double Value, const TFormatSettings& AFormatSettings);
  static String __fastcall toString(const long double Value, const TFloatFormat Format, const int Preci
  static String __fastcall toString(const long double Value, const TFloatFormat Format, const int Preci
  static long double __fastcall parse(const String s);
  static long double __fastcall parse(const String s, const TFormatSettings& AFormatSettings);
  static bool __fastcall TryParse(const String s, long double& Value);
  static bool __fastcall TryParse(const String s, long double& Value, const TFormatSettings& AFormatSet
  static bool __fastcall IsNan(const long double Value);
  static bool __fastcall IsInfinity(const long double Value);
  static bool __fastcall IsNegativeInfinity(const long double Value);
  static bool __fastcall IsPositiveInfinity(const long double Value);
  static int __fastcall Size();
private:
  long double& m_Helped;

};

struct TByteHelper
{
    TByteHelper(unsigned char& Helped) : m_Helped(Helped) {}
    TByteHelper(const unsigned char& Helped) : m_Helped(const_cast<unsigned char&>(Helped)) {}
    static const int MaxValue; //  = 255;
    static const int MinValue; //  = 0;
    String __fastcall toString();
    bool __fastcall ToBoolean();
    String __fastcall ToHexString();
    String __fastcall ToHexString(const int MinDigits);
    float __fastcall ToSingle();
    double __fastcall ToDouble();
    long double __fastcall ToExtended();
    static int __fastcall Size();
    static String __fastcall toString(const unsigned char Value);
    static unsigned char __fastcall parse(const String s);
    static bool __fastcall TryParse(const String s, unsigned char& Value);
private:
    unsigned char& m_Helped;

};

struct TShortIntHelper
{
    TShortIntHelper(signed char& Helped) : m_Helped(Helped) {}
    TShortIntHelper(const signed char& Helped) : m_Helped(const_cast<signed char&>(Helped)) {}
    static const int MaxValue; //  = 127;
    static const int MinValue; //  = -128;
    String __fastcall toString();
    bool __fastcall ToBoolean();
    String __fastcall ToHexString();
    String __fastcall ToHexString(const int MinDigits);
    float __fastcall ToSingle();
    double __fastcall ToDouble();
    long double __fastcall ToExtended();
    static int __fastcall Size();
    static String __fastcall toString(const signed char Value);
  static signed char __fastcall parse(const String s);
  static bool __fastcall TryParse(const String s, signed char& Value);
private:
  signed char& m_Helped;

};
```

```
struct TWordHelper
{

    TWordHelper(WORD& Helped) : m_Helped(Helped) {}
    TWordHelper(const WORD& Helped) : m_Helped(const_cast<WORD&>(Helped)) {}
    static const int MaxValue; //  = 65535;
    static const int MinValue; //  = 0;
    String __fastcall toString();
    bool __fastcall ToBoolean();
    String __fastcall ToHexString();
    String __fastcall ToHexString(const int MinDigits);
    float __fastcall ToSingle();
    double __fastcall ToDouble();
    long double __fastcall ToExtended();
    static int __fastcall Size();
    static String __fastcall toString(const WORD Value);
    static WORD __fastcall parse(const String s);
    static bool __fastcall TryParse(const String s, WORD& Value);
private:
  WORD& m_Helped;

};

struct TSmallIntHelper
{

    TSmallIntHelper(short int& Helped) : m_Helped(Helped) {}
    TSmallIntHelper(const short int& Helped) : m_Helped(const_cast<short int&>(Helped)) {}
    static const int MaxValue; //  = 32767;
    static const int MinValue; //  = -32768;
    String __fastcall toString();
    bool __fastcall ToBoolean();
    String __fastcall ToHexString();
    String __fastcall ToHexString(const int MinDigits);
    float __fastcall ToSingle();
    double __fastcall ToDouble();
    long double __fastcall ToExtended();
    static int __fastcall Size();
    static String __fastcall toString(const short int Value);
    static short int __fastcall parse(const String s);
    static bool __fastcall TryParse(const String s, short int& Value);
private:
  short int& m_Helped;

};

struct TCardinalHelper
{
    TCardinalHelper(unsigned int& Helped) : m_Helped(Helped) {}
    TCardinalHelper(const unsigned int& Helped) : m_Helped(const_cast<unsigned int&>(Helped)) {} /* for
    static const int MaxValue; //  = 4294967295;
    static const int MinValue; //  = 0;
    String __fastcall toString();
    bool __fastcall ToBoolean();
    String __fastcall ToHexString();
    String __fastcall ToHexString(const int MinDigits);
    float __fastcall ToSingle();
    double __fastcall ToDouble();
    long double __fastcall ToExtended();
    static int __fastcall Size();
    static String __fastcall toString(const unsigned int Value);
    static unsigned int __fastcall parse(const String s);
    static bool __fastcall TryParse(const String s, unsigned int& Value);
private:
    unsigned int& m_Helped;

};

struct TIntegerHelper
{

    TIntegerHelper(int& Helped) : m_Helped(Helped) {}
    TIntegerHelper(const int& Helped) : m_Helped(const_cast<int&>(Helped)) {} /* for LongInt type too *
    static const int MaxValue; //  = 2147483647;
    static const int MinValue; //  = -2147483648;
    String __fastcall toString();
    bool __fastcall ToBoolean();
    String __fastcall ToHexString();
    String __fastcall ToHexString(const int MinDigits);
    float __fastcall ToSingle();
    double __fastcall ToDouble();
    long double __fastcall ToExtended();
    static int __fastcall Size();
    static String __fastcall toString(const int Value);
    static int __fastcall parse(const String s);
    static bool __fastcall TryParse(const String s, int& Value);
private:
    int& m_Helped;

};

struct TUInt64Helper
{

    TUInt64Helper(unsigned __int64& Helped) : m_Helped(Helped) {}
    TUInt64Helper(const unsigned __int64& Helped) : m_Helped(const_cast<unsigned __int64&>(Helped)) {}
    static const int MaxValue; //  = 18446744073709551615;
    static const int MinValue; //  = 0;
    String __fastcall toString();
    bool __fastcall ToBoolean();
    String __fastcall ToHexString();
    String __fastcall ToHexString(const int MinDigits);
    float __fastcall ToSingle();
    double __fastcall ToDouble();
    long double __fastcall ToExtended();
    static int __fastcall Size();
    static String __fastcall toString(const unsigned __int64 Value);
    static unsigned __int64 __fastcall parse(const String s);
    static bool __fastcall TryParse(const String s, unsigned __int64& Value);
private:
    unsigned __int64& m_Helped;

};

struct TInt64Helper
{

    TInt64Helper(__int64& Helped) : m_Helped(Helped) {}
    TInt64Helper(const __int64& Helped) : m_Helped(const_cast<__int64&>(Helped)) {}
    static const int MaxValue; //  = 9223372036854775807;
    static const int MinValue; //  = -9223372036854775808;
    String __fastcall toString();
    bool __fastcall ToBoolean();
    String __fastcall ToHexString();
    String __fastcall ToHexString(const int MinDigits);
    float __fastcall ToSingle();
    double __fastcall ToDouble();
    long double __fastcall ToExtended();
    static int __fastcall Size();
    static String __fastcall toString(const __int64 Value);
    static __int64 __fastcall parse(const String s);
    static bool __fastcall TryParse(const String s, __int64& Value);
private:
    __int64& m_Helped;

};

struct TNativeUIntHelper
{

    TNativeUIntHelper(NativeUInt& Helped) : m_Helped(Helped) {}
    TNativeUIntHelper(const NativeUInt& Helped) : m_Helped(const_cast<NativeUInt&>(Helped)) {}
    static const int MaxValue; //  = 4294967295;
    static const int MinValue; //  = 0;
    String __fastcall toString();
    bool __fastcall ToBoolean();
    String __fastcall ToHexString();
    String __fastcall ToHexString(const int MinDigits);
    float __fastcall ToSingle();
    double __fastcall ToDouble();
    long double __fastcall ToExtended();
    static int __fastcall Size();
    static String __fastcall toString(const NativeUInt Value);
    static NativeUInt __fastcall parse(const String s);
    static bool __fastcall TryParse(const String s, NativeUInt& Value);
private:
    NativeUInt& m_Helped;

};

struct TNativeIntHelper
{

    TNativeIntHelper(NativeInt& Helped) : m_Helped(Helped) {}
    TNativeIntHelper(const NativeInt& Helped) : m_Helped(const_cast<NativeInt&>(Helped)) {}
    static const int MaxValue; //  = 2147483647;
    static const int MinValue; //  = -2147483648;
    String __fastcall toString();
    bool __fastcall ToBoolean();
    String __fastcall ToHexString();
    String __fastcall ToHexString(const int MinDigits);
    float __fastcall ToSingle();
    double __fastcall ToDouble();
    long double __fastcall ToExtended();
    static int __fastcall Size();
    static String __fastcall toString(const NativeInt Value);
    static NativeInt __fastcall parse(const String s);
    static bool __fastcall TryParse(const String s, NativeInt& Value);
private:
    NativeInt& m_Helped;

};

    /*$SCOPEDENUMS ON*/
enum TUseBoolStrs {False,
                                                                 True };

    /*$SCOPEDENUMS OFF*/

struct TBooleanHelper
{

    TBooleanHelper(bool& Helped) : m_Helped(Helped) {}
    TBooleanHelper(const bool& Helped) : m_Helped(const_cast<bool&>(Helped)) {}
    int __fastcall toInteger();
    String __fastcall toString(TUseBoolStrs UseBoolStrs = TUseBoolStrs::False);
    static int __fastcall Size();
    static String __fastcall toString(const bool Value, TUseBoolStrs UseBoolStrs = TUseBoolStrs::False)
    static bool __fastcall parse(const String s);
    static bool __fastcall TryToParse(const String s, bool& Value);
private:
    bool& m_Helped;

};

struct TByteBoolHelper
{

    TByteBoolHelper(unsigned char& Helped) : m_Helped(Helped) {}
    TByteBoolHelper(const unsigned char& Helped) : m_Helped(const_cast<unsigned char&>(Helped)) {}
    int __fastcall toInteger();
    String __fastcall toString();
    static int __fastcall Size();
    static String __fastcall toString(const bool Value);
    static bool __fastcall parse(const String s);
    static bool __fastcall TryToParse(const String s, bool& Value);
private:
    unsigned char& m_Helped;

};

struct TWordBoolHelper
{
    TWordBoolHelper(unsigned short& Helped) : m_Helped(Helped) {}
    TWordBoolHelper(const unsigned short& Helped) : m_Helped(const_cast<unsigned short&>(Helped)) {}
    int __fastcall toInteger();
    String __fastcall toString();
    static int __fastcall Size();
    static String __fastcall toString(const bool Value);
    static bool __fastcall parse(const String s);
    static bool __fastcall TryToParse(const String s, bool& Value);
private:
    unsigned short& m_Helped;

};

struct TLongBoolHelper
{

    TLongBoolHelper(BOOL& Helped) : m_Helped(Helped) {}
    TLongBoolHelper(const BOOL& Helped) : m_Helped(const_cast<BOOL&>(Helped)) {}
    int __fastcall toInteger();
    String __fastcall toString();
    static int __fastcall Size();
    static String __fastcall toString(const bool Value);
    static bool __fastcall parse(const String s);
    static bool __fastcall TryToParse(const String s, bool& Value);
private:
    BOOL& m_Helped;

};
```

14.1.1.6 d2c_sysiter

d2c_sysiter contains Delphi2Cpp helper code to enable range based for-loops.

template<class T> class DynamicArrayIter ...

template<class T> const DynamicArrayIter<T> begin(const DynamicArray<T>& Array) ...
template<class T> const DynamicArrayIter<T> end(const DynamicArray<T>& Array) ...
template <class T, size_t N> T* begin(T (&array_of_const)[N]) ...
template <class T, size_t N> T* end(T (&array_of_const)[N]) ...

template<class T, T minEl, T maxEl> class SetIter ...
template<class T, T minEl, T maxEl> SetIter<T, minEl, maxEl> begin(const System::Set<T, minEl, maxEl>& ASet) ...
template<class T, T minEl, T maxEl> SetIter<T, minEl, maxEl> end(const System::Set<T, minEl, maxEl>& ASet) ...

14.1.1.7 d2c_sysmath

d2c_sysmath contains Delphi2Cpp helper routines for Delphi intrinsic mathematical functions, which aren't provided by C++Builder itself.

int64_t Round(long double d);
int64_t Trunc(long double d);
long double Int(long double d);
int Sqr(int I);
int64_t Sqr(int64_t I);
uint64_t Sqr(uint64_t I);
long double Sqr(long double d);
#if (__BORLANDC__ < 0x0570)
int Random(int I);

```
long double Random( );
long double Sqrt( long double d );
long double Frac( long double d );
long double ArcTan( long double d );
long double Ln( long double d );
long double Sin( long double d );
long double Cos( long double d );
long double Exp( long double d );
#endif
```

14.1.1.8 d2c_sysstring

d2c_sysstring contains some Delphi2Cpp helper functions which are useful at the translation of Delphi code to C++ code for C++Builder.

```
int Pos(char Substr, const AnsiString& S);
int Pos(const WideString& Substr, const WideString& Source );
int Pos(wchar_t C, const WideString& S );

AnsiString Copy(const AnsiString& xs, int Index, int Count);
WideString Copy(const WideString& xs, int Index, int Count);

template <class T> std::vector<T> Copy( const std::vector<T>& V, int Index, int Count) ...
template <size_t N> void d2c_CopyToArray(Char(&CharArray)[N], const String& xs) ...

int d2c_strncmp(const char* xs1, const char* xs2);
int d2c_wcsncmp(const wchar_t* xs1, const wchar_t* xs2);
Char Chr( unsignedchar B );
void Insert( const AnsiString& Source, AnsiString& S, int Index );
void Insert( const WideString& Source, WideString& S, int Index );
void Delete( AnsiString& S, int Index, int Size );
void Delete( WideString& S, int Index, int Size );
#if (__BORLANDC__ <= 0x570)    // not in CBuilder 6
AnsiString StringOfChar( char C, int I );
WideString StringOfChar( wchar_t C, int I );
#endif
void SetString( AnsiString& S, char* Buffer, int Len );
void SetString( WideString& S, wchar_t* Buf, int Len );
void SetLength( AnsiString& S, int Len );
void SetLength( WideString& S, int Len );
int Length(const SmallString<255>& xS);
int Length(const char* xp);
int Length(const wchar_t* xp);
int Length(const AnsiString& S);
int Length(const WideString& S);


#if (__BORLANDC__ >= 0x0610)
UnicodeString Copy(const UnicodeString& S, int Index, int Count);

void SetString( UnicodeString& S, wchar_t* Buf, int Len );
void SetLength( UnicodeString& S, int Len );
int Length(const UnicodeString& S);
```

```
UnicodeString Concat(const UnicodeString& s1);
UnicodeString Concat(const UnicodeString& s1, const UnicodeString& s2);
UnicodeString Concat(const UnicodeString& s1, const UnicodeString& s2, const UnicodeString& s3);
UnicodeString Concat(const UnicodeString& s1, const UnicodeString& s2, const UnicodeString& s3,
                                                                  const UnicodeString& s4);
UnicodeString Concat(const UnicodeString& s1, const UnicodeString& s2, const UnicodeString& s3,
              const UnicodeString& s4, const UnicodeString& s5);
UnicodeString Concat(const UnicodeString& s1, const UnicodeString& s2, const UnicodeString& s3,
              const UnicodeString& s4, const UnicodeString& s5, const UnicodeString& s6);
UnicodeString Concat(const UnicodeString& s1, const UnicodeString& s2, const UnicodeString& s3,
               const UnicodeString& s4, const UnicodeString& s5, const UnicodeString& s6,
               const UnicodeString& s7);
UnicodeString Concat(const UnicodeString& s1, const UnicodeString& s2, const UnicodeString& s3,
               const UnicodeString& s4, const UnicodeString& s5, const UnicodeString& s6,
               const UnicodeString& s7, const UnicodeString& s8);
UnicodeString Concat(const UnicodeString& s1, const UnicodeString& s2, const UnicodeString& s3,
               const UnicodeString& s4, const UnicodeString& s5, const UnicodeString& s6,
               const UnicodeString& s7, const UnicodeString& s8, const UnicodeString& s9);
#endif
```

14.1.1.9 d2c_system

d2c_system contains Delphi2Cpp helper routines, which simulate intrinsic Delphi functions, which are not provided by C++Builder itself.

```
#define ARRAYHIGH(arr) ...
#define ObjectIs(xObj, xIs) ...
```

```
namespace System
{
  const double PI = 3.14159265358979323846; // float 3.14159265358979f;
}

template <class T> void d2c_Move(const wchar_t* Source, DynamicArray<T>& Dest, int Startindex,
unsigned int Count) ...
template <class T> void d2c_Move(const wchar_t* Source, DynamicArray<T>& Dest, unsigned int
Count) ...
void d2c_Move(const wchar_t* Source, Char* Dest, int Startindex, unsigned int Count);

template <class T> T Pred(const T& xT) ...
template <class T> T Succ(const T& xT) ...
template <class T> T Abs(const T xT) ...

void FillChar( void* X, int Count, unsignedchar Value );
void FillChar( char* X, int Count, unsignedchar Value );
void FillChar( wchar_t* X, int Count, unsignedchar Value );
void FillChar( AnsiString& X, int Count, unsignedchar Value );
void FillChar( WideString& X, int Count, unsignedchar Value );

template <typename T> void Val(const AnsiString& S, T& V, int& Code) ...
template <typename T> void Val(const WideString& S, T& V, int& Code) ...
```

```
#if (__BORLANDC__ >= 0x0610)
template <typename T> void Val(const UnicodeString& S, T& V, int& Code) ...
#endif
template <typename T> void Str(T xT, AnsiString& xs) ...
template <typename T> void Str(T xT, WideString& xs) ...
#if (__BORLANDC__ >= 0x0610)
template <typename T> void Str(T xT, UnicodeString& xs) ..
#endif

void Str(double xd, AnsiString& xs);
void Str(long double xd, AnsiString& xs);
void Str(int xd, int xiMinWidth, AnsiString& xs);
void Str(double xd, int xiMinWidth, AnsiString& xs);
void Str(long double xd, int xiMinWidth, AnsiString& xs);
void Str(long double xd, int xiMinWidth, AnsiString& xs);
void Str(double xd, int xiMinWidth, int xiDecPlaces, AnsiString& xs);
void Str(long double xd, int xiMinWidth, int xiDecPlaces, AnsiString& xs);
void Str(const Currency& xcr, int xiMinWidth, int xiDecPlaces, AnsiString& xs);

void Str(double xd, WideString& xs);
void Str(long double xd, WideString& xs);
void Str(int xd, int xiMinWidth, WideString& xs);
void Str(double xd, int xiMinWidth, WideString& xs);
void Str(long double xd, int xiMinWidth, WideString& xs);
void Str(double xd, int xiMinWidth, int xiDecPlaces, WideString& xs);
void Str(long double xd, int xiMinWidth, int xiDecPlaces, WideString& xs);
void Str(const Currency& xcr, int xiMinWidth, int xiDecPlaces, WideString& xs);

#if (__BORLANDC__ >= 0x0610)
void Str(double xd, UnicodeString& xs);
void Str(long double xd, UnicodeString& xs);
void Str(int xd, int xiMinWidth, UnicodeString& xs);
void Str(double xd, int xiMinWidth, UnicodeString& xs);
void Str(long double xd, int xiMinWidth, UnicodeString& xs);
void Str(double xd, int xiMinWidth, int xiDecPlaces, UnicodeString& xs);
void Str(long double xd, int xiMinWidth, int xiDecPlaces, UnicodeString& xs);
void Str(const Currency& xcr, int xiMinWidth, int xiDecPlaces, UnicodeString& xs);
#endif

template <typename T> PChar pchar(const T& xT) ...

// Pchar call is not created for wchar_t*, PChar instead

template <> inline PChar pchar<wchar_t>(const wchar_t& xT) ...
template <> inline PChar pchar<char>(const char& xT) ..
template <> inline PChar pchar<std::wstring>(const std::wstring& xT) ...


WORD Swap( WORD X );
int Swap( int X );
unsignedint Swap( unsignedint X );
int64_t Swap( int64_t X );

template <class T> unsigned char Hi(const T& xt) ...
template <class T> unsigned char Lo(const T& xt) ...
template <class T> bool Odd(const T xT) ...
template <class T> T Dec(T& xT) ...
template <class T> T Dec(T& xT, int xi) ...
template <class T> T Inc(T& xT) ...
template <class T> T Inc(T& xT, int xi) ...
template <class T> T Sqr(const T& xT) ...
template <class T> T High() ...
template <class T> T High(const T& X) ....
inline int High(const AnsiString& s) ...
inline int High(const WideString& s) ...
inline int High(const UnicodeString& s) ...
template <class T> T Low() ...
template <class T> T Low(const T& X) ...
template <class T, class C> void CastDec(T& xT, C xC) ...
template <class T, class C, class I> void CastDec(T& xT, C xC, I xI) ...
template <class T, class C> void CastDec(T*& xpT, C* xpC) ...
template <class T, class C, class I> void CastDec(T*& xpT, C* xpC, I xI) ...
template <class T, class C, class I> void CastDec(T*& xpT, C xC, I xI) ...
template <class T, class C> void CastInc(T& xT, C xC) ...
template <class T, class C, class I> void CastInc(T& xT, C xC, I xI) ...
template <class T, class C> void CastInc(T*& xpT, C xpC) ...
template <class T, class C, class I> void CastInc(T*& xpT, C* xpC, I xI) ...
template <class T, class C, class I> void CastInc(T*& xpT, C xC, I xI) ...
template <class TargetType, class SouceType, class Value> void CastAssign(SouceType* target,
Value v) ...
void Assert( bool expr );
void Assert( bool expr, const AnsiString& Msg );
void Halt( int Exitcode = - 1 );
bool Assigned(void* P);
template<class R, class... Args >  bool Assigned(R __fastcall ( __closure * Func) (Args ...)) ...
void* Ptr(int Address);
template <class T> void* Addr(const T& X) ...
template <typename CH> int charLen(const CH* src) ...
String d2c_LoadResourceString(int Ident);

template<class T, T minEl, T maxEl> System::Set<T, minEl, maxEl> CreateSetFromRange(T First, T
Last) ...
template<class T, unsigned char minEl = 0, unsigned char maxEl = 255> System::Set<T, minEl,
maxEl> CreateSetFromRange(unsigned char First, unsigned char Last) ...
template <unsigned char sz> bool operator == (SmallString<sz> SmallS, AnsiString AnsiS) ...
template <unsigned char sz> bool operator == (AnsiString AnsiS, SmallString<sz> SmallS) ...
template <unsigned char sz> bool operator != (SmallString<sz> SmallS, AnsiString AnsiS) ...
template <unsigned char sz> bool operator != (AnsiString AnsiS, SmallString<sz> SmallS) ...

class ENoDefaultConstructorError : public std::exception ...


void ThrowAbstractError(const String& xsClassName); // d2c
void ThrowNoDefaultConstructorError(const String& xsClassName); // d2c
```

14.1.1.10 d2c_systypes

d2c_systypes contains Delphi2Cpp helper types.

```
// definitions in one word are needed at the C++Builder for properties
typedef short int shortint;
typedef unsigned char unsignedchar;
typedef signed char signedchar;
typedef unsigned int unsignedint;
typedef unsigned short unsignedshort;
typedef SmallString<255> ShortString;

namespace System {

typedef std::wstring::size_type d2c_size_t;
const d2c_size_t d2c_npos = std::wstring::npos;

template <int v> struct Int2Type ...
typedef Int2Type<0> uniquetype;

template <size_t Count, typename Elem> void ArrAssign(Elem* Dest, const Elem* Src) ...
template <int Count1, int Count2, typename Elem> void ArrAssign(Elem (*Dest)[Count2], const Elem
(*Src)[Count2]) ...
template <class T> void Initialize(DynamicArray<T>& DynArr, const T* pArr, int Count) ...
template <class T> DynamicArray<T> CreateDynArray(const T* pArr, int Count) ...
template <class T, class InputIterator> DynamicArray<T> CreateDynArray(InputIterator first,
InputIterator last) ...
template <class T> DynamicArray<T> CreateDynArray(const std::initializer_list<T>& List, int Count) ...
template<typename T> using TArray = DynamicArray<T>;
template <class T> void SetLength(DynamicArray<DynamicArray<T>>& MultiDimArr, int Dim1, int
Dim2) ...
```

14.1.2 C++ code for other compilers

The code for Visual C++ in "..\Delphi2Cpp\Source\VisualC" uses the specific *property* extension of this compiler, while properties are eliminated completely in the code for other compilers in "..\DelphiXE2Cpp11\Source\Other". The code consists in .cpp and .h files with the following names. The names denote the subject of the files. The most important of these files is d2c_system, which provides basic functions like "High" and "Succ" etc. The other files also are supplementing missing parts of System.pas and other helper functions. The include directives of needed files are inserted into the generated code automatically by DelphiXE2Cpp11.

d2c_config
d2c_convert
d2c_openarray
d2c_smallstring
d2c_smallstringconvert
d2c_sysconst
d2c_syscurr
d2c_sysdate
d2c_sysfile
d2c_sysiter
d2c_sysmac
d2c_sysmarshal
d2c_sysmath
d2c_sysmem
d2c_sysmeta
d2c_sysmonitor
d2c_sysobject
d2c_sysstring
d2c_system
d2c_systypes

d2c_sysvariant
DelphiSets
OnLeavingScope

The complete code also contains:

and an rtl-folder with the pre-translated files of the Delphi RTL

14.1.2.1 d2c_config

d2c_config can be used to set definitions for different targets. At the moment only Windows 64 bit is completely supported.

For Visual C++ a constant "`_CPP_VER`" for the used version of C++ is calculated in dependence from the value of "`__cplusplus`".

```
#if __cplusplus >= 201103L || (defined(_MSC_VER) && _MSC_VER >= 1900)
  #if __cplusplus == 201704L
    #define _CPP_VER 100
  #else
    #if __cplusplus == 201703L
     #define _CPP_VER 17
    #else
      #define _CPP_VER 14
    #endif
  #endif
#else
 #define _CPP_VER 98
#endif
```

The file also contains a constant, which defines the base index for strings:

```
const int StringBaseIndex = 0;
```

14.1.2.2 d2c_convert

d2c_convert contains *Delphi2Cpp* helper functions to convert different string and array types into each other.

```
System::AnsiString wstr2str(const System::WideString& xs);
System::WideString str2wstr(const System::AnsiString& xs);
System::AnsiChar wchar2char(System::WideChar xc);
System::WideChar char2wchar(System::AnsiChar xc);

System::AnsiString wstr2astr(const System::WideString& xs);
System::AnsiString ustr2astr(const System::UnicodeString& xs);
System::AnsiString sstr2astr(const System::SmallString<255>& xs);
```

```
System::WideString astr2wstr(const System::AnsiString& xs);
System::WideString ustr2wstr(const System::UnicodeString& xs);  // see
WStrFromUStr
System::WideString sstr2wstr(const System::SmallString<255>& xs);

System::UnicodeString astr2ustr(const System::AnsiString& xs);
System::UnicodeString wstr2ustr(const System::WideString& xs);  // see
UStrFromWStr
System::UnicodeString sstr2ustr(const System::SmallString<255>& xs);

void* astr2address(const System::AnsiString& xs, int index = 0);
void* wstr2address(const System::WideString& xs, int index = 0);
void* ustr2address(const System::UnicodeString& xs, int index = 0);
void* sstr2address(const System::SmallString<255>& xs, int index = 0);

System::PWideChar address2pwchar(void* p);

System::PAnsiChar astr2pchar(const System::AnsiString& xs, int index = 0);
System::PAnsiChar wstr2pchar(const System::WideString& xs, int index = 0);
System::PAnsiChar ustr2pchar(const System::UnicodeString& xs, int index = 0);
System::PAnsiChar sstr2pchar(const System::SmallString<255>& xs, int index = 0);

System::PWideChar astr2pwchar(const System::AnsiString& xs, int index = 0);
System::PWideChar wstr2pwchar(const System::WideString& xs, int index = 0);
System::PWideChar ustr2pwchar(const System::UnicodeString& xs, int index = 0);
System::PWideChar sstr2pwchar(const System::SmallString<255>& xs, int index = 0);
System::PWideChar ustr2punichar(const System::UnicodeString& xs, int index = 0);

inline System::ShortString ustr2sstr(const System::UnicodeString& xs) ..
inline System::ShortString astr2sstr(const System::AnsiString& xs) ..
template <class T> T* array2ptr(const std::vector<T>& s, int offset = 0) ..
inline unsigned char* bytearray2pbyte(const std::vector<unsigned char>& s, int
offset = 0) ..
inline System::PAnsiChar bytearray2pchar(const std::vector<unsigned char>& s, int
 offset = 0) ...
inline void* bytearray2pvoid(const std::vector<unsigned char>& s, int offset = 0)
...
inline System::PAnsiChar wchararray2pchar(const std::vector<System::WideChar>& s,
int offset = 0) ...
inline System::PWideChar wchararray2pwchar(const std::vector<System::WideChar>& s
, int offset = 0) ...
template <typename Type, Type Low, Type High> TSet<Type, Low, High> IntToSet( int
 xi ) ...
template <typename Type, Type Low, Type High> int SetToInt( const TSet<Type, Low,
High>& xsi) ...
template <typename Type, Type Low, Type High> unsigned char ToByte(const TSet<
Type, Low, High>& xset) ...

// to smallstringconvert.h
//template <unsigned char sz = 255> System::SmallString<sz> astr2sstr(const
std::string& xs)
//template <unsigned char sz> System::SmallString<sz> wstr2sstr(const
std::wstring& xs)
//template <unsigned char sz> System::SmallString<255> sstr2sstr(const
System::SmallString<sz>& xs)
```

14.1.2.3 d2c_openarray

d2c_openarray contains *Delphi2Cpp* helper code to simulate Delphi open arrays. For constant open arrays simply std::vectors are used in the C++ translation. But there case, where open arrays are passed as var-parameters. Strings, SmallStrings, and fixed arrays con be passed to such parameters as well as dynamic arrays. For that case Delphi2Cpp uses a special template type *OpenArrayRef*, which is defined in *d2c_openarray.*

```
template <class T>
class OpenArrayRef
{
public:
      OpenArrayRef(std::vector<T>& v);
      OpenArrayRef(std::basic_string<T>& s);

      ...

};
```

14.1.2.4 d2c_smallstring

d2c_smallstring contains *Delphi2Cpp* helper code to simulate Delphi Short*String*'s. d2c_smallstringconvert contains some conversion routines between *SmallString*'s and other types.

14.1.2.5 d2c_sysconst

d2c_sysconst contains *Delphi2Cpp* helper constants for formatting or to create error messages.

14.1.2.6 d2c_syscurr

d2c_syscurr contains *Delphi2Cpp* helper code to simulate the Delphi currency type.

14.1.2.7 d2c_sysdate

d2c_sysdate contains *Delphi2Cpp* helper code to simulate the Delphi *DateTime* type.

14.1.2.8 d2c_sysfile

d2c_sysfile contains *Delphi2Cpp* helper code for basic file reading and writing routines.

```
struct TFileRec {
  THandle Handle;
  int Mode;
  WORD Flags;
  unsigned int RecSize;
  unsigned char _private [ 28 ];
  unsigned char UserData [ 32 ];
  Char Name [ 260 ];
};

template <typename T>
struct TTypedFile
{
  THandle Handle;
  int Mode;
  WORD Flags;
  unsigned int RecSize;
  unsigned char _private [ 28 ];
  unsigned char UserData [ 32 ];
  Char Name [ 260 ];
};

typedef char TextBufA [ 260 ];

struct TTextRec {
  THandle Handle;
  int Mode;
  WORD Flags;
  unsigned int BufSize;
  unsigned int BufPos;
  unsigned int BufEnd;
  char* BufPtr;
  void* OpenFunc;
  void* InOutFunc;
  void* FlushFunc;
  void* CloseFunc;
  unsigned char UserData [ 32 ];
  Char Name [ 260 ];
  WORD CodePage;
  char LineEnd[3];
  TextBufA Buffer;
};


typedef TTypedFile<unsigned char> file;
typedef TTextRec Text;
typedef TTextRec *ptext;

// System.h enum TTextLineBreakStyle {tlbsLF, tlbsCRLF, tlbsCR };

extern bool  FileNameCaseSensitive;
extern bool CtrlZMarksEOF;
extern TTextLineBreakStyle DefaultTextLineBreakStyle;

const Char DirectorySeparator = _T('\\');
const Char DriveSeparator = _T(':');
const Char PathSeparator = _T(';');
const int MaxPathLen = 260;

extern THandle UnusedHandle;
extern THandle StdInputHandle;
extern THandle StdOutputHandle;
extern THandle StdErrorHandle;

const bool LFNSupport = true;
const char ExtensionSeparator = '.';
extern TSet < UChar, 0, 255 > AllowDirectorySeparators;
extern TSet < UChar, 0, 255 > AllowDriveSeparators;
extern bool FileNameCaseSensitive;
extern bool CtrlZMarksEOF;

const int fsFromBeginning = 0;
const int fsFromCurrent = 1;
const int fsFromEnd = 2;

const THandle feInvalidHandle = ((THandle) - 1 );  //return value on FileOpen
error


/* file input modes */
const int fmClosed = 0xD7B0;
const int fmInput = 0xD7B1;
const int fmOutput = 0xD7B2;
const int fmInOut = 0xD7B3;
//const int fmAppend = 0xD7B4;
extern TTextRec ErrOutput, Output, Input, Stdout, Stderr;
//extern unsigned char FileMode;
// System.h WORD IOResult( );

const int fmAppend = 0xD7B4;   // unknown in C++Builder 6?


typedef void ( * FileFunc )( TTextRec&  );


/****************************************************************************
                          Untyped File Management
****************************************************************************/

void AssignFile( TFileRec& f, const String& Name );
void AssignFile( TFileRec& f, Char c );
void Assign( TFileRec& f, const String& Name );

void Rewrite( TFileRec& f, int l = 128 );
void Reset( TFileRec& f, int l = 128 );
void CloseFile( TFileRec& f );
void Close( TFileRec& f );

void BlockWrite( TTypedFile<unsigned char>& f, void*& buf, int64_t Count, int&
Result );
void BlockWrite( TTypedFile<unsigned char>& f, void*& buf, int Count, int& Result
);
void BlockWrite( TTypedFile<unsigned char>& f, void*& buf, WORD Count, WORD&
Result );
void BlockWrite( TTypedFile<unsigned char>& f, void*& buf, unsigned int Count,
unsigned int& Result );
void BlockWrite( TTypedFile<unsigned char>& f, void*& buf, WORD Count, int&
Result );
void BlockWrite( TTypedFile<unsigned char>& f, void*& buf, int Count );
void BlockRead( TTypedFile<unsigned char>& f, void*& buf, int Count, int& Result
);
void BlockRead( TTypedFile<unsigned char>& f, void*& buf, int64_t Count );
```

```
int64_t FileSize( TFileRec& f ); // FileSize can't be used on a text
TTypedFile<unsigned char>.
int64_t FilePos( TFileRec& f, int l = 128 );
void Seek( TTypedFile<unsigned char>& f, int64_t Pos );
void Rename( TTypedFile<unsigned char>& f, const char* P );
void Rename( TTypedFile<unsigned char>& f, const wchar_t* P );
void Rename( TTypedFile<unsigned char>& f, const AnsiString& s );
void Rename( TTypedFile<unsigned char>& f, const WideString& s );
void Rename( TTypedFile<unsigned char>& f, char c );
void Rename( TTypedFile<unsigned char>& f, wchar_t c );
bool Eof( TFileRec& f, int l = 128 );
void Truncate( TFileRec& f, int RecSize );

template <typename T> void AssignFile( TTypedFile<T>& f, const String& Name ) ...
template <typename T> void AssignFile( TTypedFile<T>& f, const Char* P ) ...
template <typename T> void AssignFile( TTypedFile<T>& f, Char c ) ...
template <typename T> int64_t FileSize( TTypedFile<T>& f ) ...
template <typename T> bool Eof( TTypedFile<T>& f ) ...
template <typename T> void CloseFile( TTypedFile<T>& f ) ...
template <typename T> void Close( TTypedFile<T>& f ) ...
template <typename T> void Reset( TTypedFile<T>& f, int l = -1 ) ...
template <typename T> void Rewrite( TTypedFile<T>& f, int l = -1 ) ...
template <typename T> void Write( TTypedFile<T>& f, void* buf ) ...
template <typename T> void Read( TTypedFile<T>& f, void* buf ) ...
template <typename T> void Truncate( TTypedFile<T>& f ) ...


/*****************************************************************************
                            Text File Management
*****************************************************************************/


void Assign( TTextRec& f, const String& Name );
void Assign( TTextRec& t, Char c );
void AssignFile( TTextRec& f, const String& Name );
void AssignFile( TTextRec& t, Char c );
void CloseFile( TTextRec& t );
void Close( TTextRec& t );
void Rewrite( TTextRec& t );
void Reset( TTextRec& t );
void Append( TTextRec& t );
// System.h void Flush( TTextRec& t );
void Erase( TTextRec& t );
void Rename( TTextRec& t, const char* P );
void Rename( TTextRec& t, const wchar_t* P );
void Rename( TTextRec& t, const AnsiString& s );
void Rename( TTextRec& t, const WideString& s );
void Rename( TTextRec& t, char c );
void Rename( TTextRec& t, wchar_t c );
bool Eof( TTextRec& t );
bool Eof( );
bool EoLn( TTextRec& t );
bool EoLn( );
bool SeekEof( TTextRec& t );
bool SeekEof( );
bool SeekEoLn( TTextRec& t );
bool SeekEoLn( );
void SetTextBuf( TTextRec& t, void* buf, int size );
void SetTextLineEnding(TTextRec& t, String& Ending);


void Write( TTextRec& t, const SmallString<255> s, int Len = 0 );
void Write( TTextRec& t, const char* P, int Len = 0 );
void Write( TTextRec& t, const AnsiString& s, int Len = 0 );
void Write( TTextRec& t, const wchar_t* P, int Len = 0 );
void Write( TTextRec& t, const WideString& s, int Len = 0 );
void Write( TTextRec& t, long l, int Len = 0 );
void Write( TTextRec& t, int l, int Len = 0 );
void Write( TTextRec& t, unsigned int l, int Len = 0 );
void Write( TTextRec& t, unsigned short l, int Len = 0 );
void Write( TTextRec& t, uint64_t q, int Len = 0 );
void Write( TTextRec& t, int64_t i, int Len = 0 );
void Write( TTextRec& t, long double r, int rt = -1, int fixkomma = -1 );
void Write( TTextRec& t, double r, int rt = -1, int fixkomma = -1 );
void Write( TTextRec& t, Currency c, int fixkomma = -1, int Len = 0 );
void Write( TTextRec& t, bool b, int Len = 0 );
void Write( TTextRec& t, char c, int Len = 0 );
void Write( TTextRec& t, unsigned char c, int Len = 0 );
void Write( TTextRec& t, wchar_t c, int Len = 0 );
void WriteLn( TTextRec& t );
void WriteLn( TTextRec& t, const SmallString<255> s, int Len = 0 );
void WriteLn( TTextRec& t, const char* P, int Len = 0 );
void WriteLn( TTextRec& t, const AnsiString& s, int Len = 0 );
void WriteLn( TTextRec& t, const wchar_t* P, int Len = 0 );
void WriteLn( TTextRec& t, const WideString& s, int Len = 0 );
void WriteLn( TTextRec& t, long l, int Len = 0 );
void WriteLn( TTextRec& t, int l, int Len = 0 );
void WriteLn( TTextRec& t, unsigned int l, int Len = 0 );
void WriteLn( TTextRec& t, unsigned short l, int Len = 0 );
void WriteLn( TTextRec& t, uint64_t q, int Len = 0 );
void WriteLn( TTextRec& t, int64_t i, int Len = 0 );
void WriteLn( TTextRec& t, long double r, int rt = -1, int fixkomma = -1 );
void WriteLn( TTextRec& t, double r, int rt = -1, int fixkomma = -1 );
void WriteLn( TTextRec& t, Currency c, int fixkomma, int Len = -1 );
void WriteLn( TTextRec& t, bool b, int Len = 0 );
void WriteLn( TTextRec& t, char c, int Len = 0 );
void WriteLn( TTextRec& t, unsigned char c, int Len = 0 );
void WriteLn( TTextRec& t, wchar_t c, int Len = 0 );
void Write( const SmallString<255> s, int Len = 0 );
void Write( const char* P, int Len = 0 );
void Write( const AnsiString& s, int Len = 0 );
void Write( long l, int Len = 0 );
void Write( int l, int Len = 0 );
void Write( unsigned int l, int Len = 0 );
void Write( uint64_t q, int Len = 0 );
void Write( int64_t i, int Len = 0 );
void Write( long double r, int rt = -1, int fixkomma = -1 );
void Write( double r, int rt = -1, int fixkomma = -1 );
void Write( Currency c, int fixkomma = -1, int Len = 0 );
void Write( bool b, int Len = 0 );
void Write( char c, int Len = 0 );
void WriteLn( );
void WriteLn( const SmallString<255> s, int Len = 0 );
void WriteLn( const char* P, int Len = 0 );
void WriteLn( const AnsiString& s, int Len = 0 );
void WriteLn( long l, int Len = 0 );
void WriteLn( int l, int Len = 0 );
void WriteLn( unsigned int l, int Len = 0 );
void WriteLn( uint64_t q, int Len = 0 );
void WriteLn( int64_t i, int Len = 0 );
void WriteLn( long double r, int rt = -1, int fixkomma = -1 );
void WriteLn( double r, int rt = -1, int fixkomma = -1 );
void WriteLn( Currency c, int fixkomma = -1, int Len = 0 );
void WriteLn( bool b, int Len = 0 );
void WriteLn( char c, int Len = 0 );
void Write( wchar_t c, int Len  = 0);
void WriteLn( wchar_t c, int Len = 0 );
void Write( const wchar_t* P, int Len = 0 );
void Write( const WideString& s, int Len = 0 );
void WriteLn( const wchar_t* P, int Len = 0 );
void WriteLn( const WideString& s, int Len = 0 );


void Read( TTextRec& t, SmallString<255>& s );
void Read( TTextRec& t, char* s, int maxlen = 0x7FFFFFFF );
void Read( TTextRec& t, AnsiString& s );
void Read( TTextRec& t, char& c );
void Read( TTextRec& t, WideString& s );
void Read( TTextRec& t, wchar_t& c );
void Read( TTextRec& t, unsigned int& u );
void Read( TTextRec& t, unsigned short& u );
void Read( TTextRec& t, long double& v );
void Read( TTextRec& f, Currency& v );
void Read( TTextRec& t, double& v );
void Read( TTextRec& t, int64_t& i );
void Read( TTextRec& t, uint64_t& q );
void ReadLn( TTextRec& t);
void ReadLn( TTextRec& t, SmallString<255>& s );
void ReadLn( TTextRec& t, char* s, int maxlen = 0x7FFFFFFF );
void ReadLn( TTextRec& t, AnsiString& s );
void ReadLn( TTextRec& t, char& c );
void ReadLn( TTextRec& t, WideString& s );
void ReadLn( TTextRec& t, wchar_t& c );
void ReadLn( TTextRec& t, int& l );
void ReadLn( TTextRec& t, unsigned int& u );
void ReadLn( TTextRec& t, unsigned short& u );
void ReadLn( TTextRec& t, long double& v );
void ReadLn( TTextRec& f, Currency& v );
void ReadLn( TTextRec& t, double& v );
void ReadLn( TTextRec& t, int64_t& i );
void ReadLn( TTextRec& t, uint64_t& q );
void Read( SmallString<255>& s );
void Read( char* s, int maxlen = 0x7FFFFFFF );
void Read( AnsiString& s );
void Read( char& c );
void Read( WideString& s );
void Read( wchar_t& c );
void Read( int& l );
void Read( unsigned int& u );
void Read( long double& v );
void Read( double& v );
void Read( Currency& v );
void Read( int64_t& i );
void Read( uint64_t& q );
void ReadLn( );
void ReadLn( SmallString<255>& s );
void ReadLn( char* s, int maxlen = 0x7FFFFFFF );
void ReadLn( AnsiString& s );
void ReadLn( char& c );
void ReadLn( WideString& s );
void ReadLn( wchar_t& c );
void ReadLn( int& l );
void ReadLn( unsigned int& u );
void ReadLn( unsigned short& u );
void ReadLn( long double& v );
void ReadLn( double& v );
void ReadLn( Currency& v );
void ReadLn( int64_t& i );
void ReadLn( uint64_t& q );

template <unsigned char sz> void Write( TTextRec& t, SmallString<sz> s, int Len =
0 ) ...
template <unsigned char sz> void WriteLn( TTextRec& t, SmallString<sz> s, int Len
= 0 ) ...
template <unsigned char sz> void Write( SmallString<sz> s, int Len = 0 ) ...
template <unsigned char sz> void WriteLn( SmallString<sz> s, int Len = 0 ) ...
template <unsigned char sz> void Read( TTextRec& t, SmallString<sz>& s ) ...
template <unsigned char sz> void ReadLn( TTextRec& t, SmallString<sz>& s ) ...
template <unsigned char sz> void Read( SmallString<sz>& s ) ...
template <unsigned char sz> void ReadLn( SmallString<sz>& s ) ...


void SetLineBreakStyle(System::Text& T, TTextLineBreakStyle Style);
WORD GetTextCodePage(const System::Text T);
void SetTextCodePage(System::Text& T, WORD CodePage);
void Flush(System::Text& T);

void InOutError();
void SetInOutRes(int NewValue);
```

14.1.2.9 d2c_sysiter

d2c_sysiter contains *Delphi2Cpp* helper code to enable range based for-loops.

14.1.2.10 d2c_sysmac

d2c_sysmac contains *Delphi2Cpp* helper macros for message maps.

```
#define BEGIN_MESSAGE_MAP ...
#define VCL_MESSAGE_HANDLER(msg,type,meth) ...
#define END_MESSAGE_MAP(base) ...
```

14.1.2.11 d2c_sysmarshal

d2c_sysmarshal is a part of the translated *System.pas*.

14.1.2.12 d2c_sysmath

d2c_sysmath contains *Delphi2Cpp* helper routines for Delphi intrinsic mathematical functions.

```
int64_t Round( long double d );
long double Frac( long double d );
int64_t Trunc( long double d );
extern int RandSeed;
long double Sqr( long double d );
long double Sqrt( long double d );
long double ArcTan( long double d );
long double Ln( long double d );
long double Sin( long double d );
long double Cos( long double d );
long double Exp( long double d );
long double Int( long double d );
int64_t Trunc( long double d );
```

14.1.2.13 d2c_sysmem

d2c_sysmem contains *Delphi2Cpp* helper routines for memory management.

```
void* AllocMem(d2c_size_t Size);
void GetMem(void*& P, d2c_size_t Size);
void* GetMemory(d2c_size_t Size);
void* ReallocMemory(void* P, d2c_size_t Size);
void ReallocMem(void*& P, d2c_size_t Size);
void FreeMem(void* P);
void FreeMem(void*& P, d2c_size_t Size);
void FreeMemory(void* P);
void FreeMemory(void* P, d2c_size_t Size);
```

14.1.2.14 d2c_sysmonitor

d2c_sysmonitor contains *Delphi2Cpp* helper code from the original System.pas with the definition of *TMonitor*,

14.1.2.15 d2c_sysmeta

d2c_meta contains *Delphi2Cpp* helper code to partially simulate Delphi class references. The details are protected and can be seen by customers of Delphi2Cpp only.

```
class TMetaClass ...
template <typename Class> class ClassRef ...
template <class Class> ClassRef+ class_id()
```

14.1.2.16 d2c_sysobject

d2c_sysobject contains *Delphi2Cpp* helper code from the original System.pas with the definition of *TObject*.

14.1.2.17 d2c_sysstring

d2c_sysstring contains *Delphi2Cpp* helper routines for string operations.

```
int PCharLen(const AnsiChar* P);
int PWCharLen(const WideChar* P);
String Copy(const String& xs, int Index, int Count);

template <class T> std::vector<T> Copy(const std::vector<T>& V, int Index, int
Count) ...
template <size_t N> void d2c_CopyToArray(Char(&CharArray)[N], const String& xs)
...
int d2c_wcsncmp(const wchar_t* xs1, const wchar_t* xs2);
int d2c_strncmp(const char* xs1, const char* xs2);
Char LowerCase(Char C );
void Insert(const String& Source, String& S, d2c_size_t Index );
void Delete(String& S, int Index, d2c_size_t Size );
String StringOfChar(Char C, int l );
std::string StringOfChar(char C, int l);
void SetString(String& S, Char* Buffer, d2c_size_t Len );
void SetLength(String& s, d2c_size_t newLength);
void SetString(String& s, Char* Buffer, d2c_size_t Length);
void SetString(AnsiString& s, const PAnsiChar Buffer, d2c_size_t Length);
void SetString(PShortString s, PAnsiChar Buffer, d2c_size_t Len);
void SetLength(String& S, d2c_size_t Len );
void SetLength(UTF8String& S, d2c_size_t Len);


String Concat(const String s1);
String Concat(const String s1, const String s2);
String Concat(const String s1, const String s2, const String s3);
String Concat(const String s1, const String s2, const String s3,
              const String s4);
String Concat(const String s1, const String s2, const String s3,
              const String s4, const String s5);
String Concat(const String s1, const String s2, const String s3,
              const String s4, const String s5, const String s6);
String Concat(const String& s1, const String& s2, const String& s3,
              const String& s4, const String& s5, const String& s6,
              const String& s7);
String Concat(const String& s1, const String& s2, const String& s3,
              const String& s4, const String& s5, const String& s6,
              const String& s7, const String& s8);
String Concat(const String& s1, const String& s2, const String& s3,
              const String& s4, const String& s5, const String& s6,
              const String& s7, const String& s8, const String& s9);
```

14.1.2.18 d2c_system

d2c_system is the most important of the Delphi2Cpp helper files, because it contains routines, which simulate intrinsic Delphi functions.

```
#define MAXIDX(x) ...
#define ARRAYHIGH(arr) ...
#define SLICE(a, n) ...
#define ObjectIs(xObj, xIs) ...
#define INTFOBJECT_IMPL_IUNKNOWN(BASE) ...


extern int Argc;
extern PPChar Argv;
extern HINSTANCE HInstance;

const double PI = 3.14159265358979323846; // float 3.14159265358979f;


void FillChar(void* Dest, NativeInt Count, AnsiChar Value);
void FillChar(String& X, int Count, wchar_t Value );
void FillChar(std::string& X, int Count, char Value);


template <class T> std::vector<T> Slice(const T* arr, int n) ...

void Move(const wchar_t* Source, String& Dest, unsigned int Count);

WORD swap(WORD X);
int swap(int X);
unsigned int swap(unsigned int X);
int64_t swap(int64_t X);

template <class T> T Dec(T& xT) ...
template <class T> T Dec(T& xT, int xi) ...
bool Inc(bool& b);
template <class T> T Inc(T& xT) ...
template <class T> T Inc(T& xT, int xi) ...
template <class T> unsigned char Hi(const T& xt) ...
template <class T> unsigned char Lo(const T& xt) ...
template <class T> bool Odd(const T xT) ...

template <class T> T AtomicDecrement(std::atomic<T>& Target) ...
template <class T> T AtomicDecrement(std::atomic<T>& Target, T Value) ...
template <class T> T AtomicIncrement(std::atomic<T>& Target) ...
template <class T> T AtomicIncrement(std::atomic<T>& Target, T Value) ...
template <class T> T AtomicExchange(std::atomic<T>& Target, T Value) ...
template <class T> T AtomicCmpExchange(std::atomic<T>& Target, T NewValue, T
Comparand) ...
template <class T> T AtomicCmpExchange(std::atomic<T>& Target, T NewValue, T
Comparand, bool& Succeeded) ...

template <class T, class C> void CastDec(T& xT, C xC) ...
template <class T, class C, class I> void CastDec(T& xT, C xC, I xI) ...
template <class T, class C> void CastDec(T*& xpT, C* xpC) ...
template <class T, class C, class I> void CastDec(T*& xpT, C* xpC, I xI) ...
template <class T, class C, class I> void CastDec(T*& xpT, C xC, I xI) ...
template <class T, class C> void CastInc(T& xT, C xC) ...
template <class T, class C, class I> void CastInc(T& xT, C xC, I xI) ...
template <class T, class C> void CastInc(T*& xpT, C xpC) ...
template <class T, class C, class I> void CastInc(T*& xpT, C* xpC, I xI) ...
template <class T, class C, class I> void CastInc(T*& xpT, C xC, I xI) ...
template <class TargetType, class SouceType, class Value> void
CastAssign(SouceType* target, Value v) ...


template <class T> T Sqr(const T& xT) ...
template <class T> constexpr T High() ...
template <class T> constexpr T High(const T& X) ...
template <class T> d2c_size_t High(const std::vector<T>& X) ...
template <unsigned char sz> d2c_size_t High(System::SmallString<sz>& X) ...
template <class T> T Succ(T xT) ...
template <class T> T Pred(T xT) ...
System::UnicodeString::size_type High(const System::UnicodeString& X);
System::AnsiString::size_type High(const System::AnsiString& X);
template <class T> T Low() ...
template <class T> constexpr d2c_size_t Low(const std::vector<T>& X) ...
template <class T> constexpr T Low(const T& X) ...
template<typename T, unsigned N> d2c_size_t Low(const T(&v)[N]) ...
template<typename T, unsigned N> d2c_size_t High(const T(&v)[N]) ...
System::UnicodeString::size_type Low(const System::UnicodeString& X);
System::AnsiString::size_type Low(const System::AnsiString& X);

void Assert( bool expr );
void Assert(bool expr, const std::wstring& Msg);

template <typename T> bool Assigned(const std::vector<T>* P) ...
template <typename T> bool Assigned(const std::vector<T>& P) ...
template <typename T> bool Assigned(std::function<T> P) ...

template <typename CH> int charLen(const CH* src) ...
template <class T> void* Addr(const T& X) ...
template <typename T> void Val(const std::wstring& S, T& V, int& Code) ...
template <typename T> void Val(const std::string& S, T& V, int& Code) ...
template <typename T> void Str(T xT, String& xs) ...
template <typename T> void Str(T xT, std::string& xs) ...

void Str(double xd, std::string& xs);
void Str(long double xd, std::string& xs);
void Str(int xd, int xiMinWidth, std::string& xs);
void Str(double xd, int xiMinWidth, std::string& xs);
void Str(long double xd, int xiMinWidth, std::string& xs);
void Str(long double xd, int xiMinWidth, std::string& xs);
void Str(double xd, int xiMinWidth, int xiDecPlaces, std::string& xs);
void Str(long double xd, int xiMinWidth, int xiDecPlaces, std::string& xs);
void Str(const System::Currency& xcr, int xiMinWidth, int xiDecPlaces,
std::string& xs);

void Str(double xd, String& xs);
void Str(long double xd, String& xs);
void Str(int xd, int xiMinWidth, String& xs);
void Str(double xd, int xiMinWidth, String& xs);
void Str(long double xd, int xiMinWidth, String& xs);
void Str(long double xd, int xiMinWidth, String& xs);
void Str(double xd, int xiMinWidth, int xiDecPlaces, String& xs);
void Str(long double xd, int xiMinWidth, int xiDecPlaces, String& xs);

template <typename T> PChar pchar(const T& xT) ...
template <> inline PChar pchar<wchar_t>(const wchar_t& xT) ...
template <> inline PChar pchar<char>(const char& xT) ...
template <> inline PChar pchar<std::wstring>(const std::wstring& xT) ...


//http://stackoverflow.com/questions/4770968/storing-function-pointer-in-stdfunct
ion
template <typename Signature> std::function<Signature> fptr_cast(void* f) ...

//http://stackoverflow.com/questions/20833453/comparing-stdfunctions-for-equality
template<typename T, typename... U> d2c_size_t getAddress(std::function<T(U...)>
f) ...


HMODULE FindResourceHInstance(unsigned int Module);  // dummy function
HMODULE FindClassHInstance( const TMetaClass* ClassType );
HMODULE FindHInstance( void* Address );

template <class T> T Default() ...
```

14.1.2.19 d2c_systypes

d2c_systypes contains *Delphi2Cpp* helper types and routines.

```
typedef std::wstring::size_type d2c_size_t;
const d2c_size_t d2c_npos = std::wstring::npos;

// some types which originally were defined in System.Types.pas as EXTERNALSYM
```

```
...


template <int v> struct Int2Type ..
typedef Int2Type<0> uniquetype;

template <class T> void SetLength(std::vector<std::vector<T>>& MultiDimArr, int
Dim1, int Dim2) ...
template <class T> void Delete(std::vector<T>& arr, int Index, int Size) ...
template <d2c_size_t Count, typename Array> void ArrAssign(Array& Dest, const
Array& Src) ...
template <int Count1, int Count2, typename Array> void ArrAssign(Array& Dest,
const Array& Src) ...
template <typename T> void CopyArray(std::vector<T>& Dest, const std::vector<T>&
Source, uint64_t Count) ...
```

14.1.2.20 d2c_sysvariant

d2c_sysvariant contains the *Delphi2Cpp* helper types *TVarRec*, *TVarData* and *ArrayOfConst*.

14.1.2.21 DelphiSets

DelphiSets contains the *Delphi2Cpp* helper type definition from Daniel Flower to simulate Delphi Sets.

```
template <typename Type, Type Low, Type High> class TSet ...


/* DelphiSets.h
================================================================================
===

      Dan's Substitute Sets ** (c) Copyright 2011, Daniel Flower
      A high-performance template class that emulates Delphi's sets.

      Authorized for unlimited use in any Delphi2Cpp project.
```

14.1.2.22 OnLeavingScope

OnLeavingScope is a file from Craig Scott, which is used as *Delphi2Cpp* helper to simulate finally-statements.

```
/**
 * The contents of this file are based on the article posted at the
 * following location:
 *
 *    http://crascit.com/2015/06/03/on-leaving-scope-part-2/
 *
 * The material in that article has some commonality with the code made
 * available as part of Facebook's folly library at:
 *
 *    https://github.com/facebook/folly/blob/master/folly/ScopeGuard.h
 *
 * Furthermore, similar material is currently part of a draft proposal
 * to the C++ standards committee, referencing the same work by Andrei
 * Alexandresu that led to the folly implementation. The draft proposal
 * can be found at:
 *
 *   http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4189.pdf
 *
 * With the above in mind, the content below is made available under
 * the same license terms as folly to minimize any legal concerns.
 * Should there be ambiguity over copyright ownership between Facebook
 * and myself for any material included in this file, it should be
 * interpreted that Facebook is the copyright owner for the ambiguous
 * section of code concerned.
 *
 *   Craig Scott
 *   3rd June 2015
 *
  * ---------------------------------------------------------------------
 *
 * Copyright 2015 Craig Scott
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *    http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```

14.1.2.23 RTL core

There is a ready translation of core files of the Delphi RTL, that are needed nearly in every Delphi project. However, there are copyright restrictions that must be met when these files are shipped.

The RTL core comprises following code:

Parts of the System.pas (including replacements for intrinsic routines, see above).

System.Actions
System.Analytics
System.AnsiStrings
System.Character
System.Classes
System.ConvUtils
System.DateUtils
System.Diagnostics
System.Generics.Collections
System.Generics.Defaults
System.Hash
System.IniFiles

System.IOUtils
System.Masks
System.Math
System.RTLConsts
//System.Rtti.cpp
System.StdConvs
System.StrUtils
//System.SyncObjs
System.TimeSpan
//System.TypInfo
System.UITypes
//System.Zip
System.ZLib
System.ZLibConst
System.Win.Registry

System.Internal.ExcUtils
System.SysConst
System.SysUtils
System.Types
//System.Variants

Winapi.Messages.h
Winapi.Windows

14.1.2.23.1 System.h

System.h is a file of the RTL core. It is a manually created header file which is included at the top position in every header files that *Delphi2Cpp* produces. *System.h* contains fundamental type definitions analogously to the interface part of the Delphi *System.pas*. It also contains some often used macros.

14.1.3 Visual C++ configuration

For Visual C++ projects following options are recommended:

1. set the x64 option in the menu.

2. Additional dependencies may be:

   version.lib
   mpr.lib
   netapi32.lib
   Rpcrt4.lib
   Dbghelp.lib

3. set C++ version to C++17

Normally in Visual C++ the __cplusplus preprocessor macro has a fixed value, but d2c_config uses this value to calculate the used version of C++. There fore in Visual C++ projects the option:

/Zc:__cplusplus

has to be set in the **Additional options** pane of the **Command Line** property page in the properties for **C/C++** .

4. If newly translated code is compiled for the first time, it might be useful to disable the warning to "possible loss of data" at type conversions:

C++ -> extended -> deactivate warnings

4244
4267

5. If C++98 code is compiled with Visual C++ you should disable secure warnings

C++ Preprocessor -> preprocessor definitions

 _CRT_SECURE_NO_WARNINGS
 _SCL_SECURE_NO_WARNINGS

This is not necessary fro C++11, because here the secure commands

```
strcpy_s
strncpy_s
wcscpy_s
```

etc. are defined.

14.1.4 Special Delphi units

It already has been explained that for other compilers than C++Builder the System.pas has to be treated in a special way. But it is recommended also to prepare some other files of the Delphi RTL. That are the API files, System.Types.pas. Users of Delphi2Cpp with valid Delphi license get the ready prepared pas-files together with the pre-translated RTL files.

System.pas

Because Delphi2Cpp provides ready prepared C++ substitutes for the System.pas and also an own System.pas is used to control the output generation, the original System.pas still is needed for the translation of the Delphi sources. Parts which are missing in the own System.pas are taken from here.

The Delphi2Cpp pre-processor cannot evaluate SizeOf expressions. The following condition:

```
{$IFDEF EXTENDEDIS10BYTES}
  {$IF SizeOf(Extended) <> SizeOf(TExtended80Rec)}
    {$MESSAGE ERROR 'TExtended80Rec has incorrect size'}
  {$ENDIF }
```

```
{$ENDIF EXTENDEDIS10BYTES}
```

is therefore replaced by

```
{$IFDEF D2C}
// d2c cannot check size
{$ELSE}
  {$IF SizeOf(Extended) <> SizeOf(TExtended80Rec)}
    {$MESSAGE ERROR 'TExtended80Rec has incorrect size'}
  {$ENDIF }
{$ENDIF}
```

and the identifiers *D2C* is defined in the project file for the Windows 64 bit result.

Several classes, most important *TObject*, aren't defined if the definition for SYSTEM_HPP_DEFINES_OBJECTS isn't set. But this definition doesn't suffice. If for example the NODEFINE directive for the string type is disabled, this will force Delphi2Cpp to insert the *System* namespace in header files before *String*: *System::String*. This is desired and applies to a lot of other NODEFINE directives in *System.pas* too.

If one tries to translate System.pas despite of the set definitions there remain some messages like:

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

These parts have to be prepared too. These parts are in the implementation part however and do not harm, if *System.pas* only is used for the translation of other files.

API files

Though the API files, e.g. the Winapi--files for Windows, are needed for the translation of the other Delphi files, they mostly don't have to be translated themselves. Their purpose is just to provide the C++ API types and constants for Delphi. The C++ code, which is generated from the Delphi sources just should use the original types and constants. There are some special directives written into the Delphi code that let make the C++ Builder access the original API. Delphi2Cpp also uses these directives.

System.Types.pas

The NODEFINE directives here should be disabled. C++Builder defines these type in an extra C++ header. But for Delphi2Cpp1 translated code these definitions remain in *System.Types.h*.

System.Variants.pas/System.VarUtils.pas

Under Windows Delphi Variant is a reduplication of the VARIANT structure in OAldl.h. A C++ application should use the original Windows types..Until now Delphi2Cpp offers no special support for the conversion of Delphi code using Variants etc. However Delphi2Cpp supports TVarRec. Advice from users is welcome.

14.2 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

14.2.1 Bugs in the Delphi RTL/VCL

In some cases DelphiXE22Cpp11 cannot process a unit though the Delphi compiler can That's because the automatically generated parser of Delphi2Cpp is more strict than the Delphi parser, which 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 Delphi2Cpp 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

```
(*$HPPEMIT 'DEFINE_GUID(IID_IOTAStreamModifyTime,0x49F2F63F,0x60CB,0x4FD4,0xB1,0x2F,0x81,0x67,0xFC,0x79
...
(*$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

->
(*$HPPEMIT 'DEFINE_GUID(IID_IOTAStreamModifyTime,0x49F2F63F,0x60CB,0x4FD4,0xB1,0x2F,0x81,0x67,0xFC,0x79
...
(*$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
```

\rtl\osx\Macapi.ObjectiveC.pas

there are several occurrences of:

```
{$ELSE Defined(...
```

This syntax isn't documented and seems not to be used anywhere else (with one exception in SysUtils). The Code can be processed, when {$ELSE is changed to {$ELSEIF

Flaws in the RTL of RAD Studio 10.4.1 Alexandria

```
{$IF SizeOf(Extended) >= 10)}
->
{$IF SizeOf(Extended) >= 10}
```

```
{$IF SizeOf(Extended) = 10)}
->
{$IF SizeOf(Extended) = 10}
```

Posix.SysSocket.pas

function CMSG_NXTHDR

System.pas

```
{$ELSEIF defined(CPUX64) and defined(Linux)) }
->
{$ELSEIF defined(CPUX64) and defined(Linux) }
```

```
{$IF Defined(OSX64) or defined(IOSSIMULATOR) and defined(CPUX64))}
->
{$IF Defined(OSX64) or defined(IOSSIMULATOR) and defined(CPUX64)}
```

```
{$ELSEIF defined(CPUX64) and defined(Linux)) }
->
{$ELSEIF defined(CPUX64) and defined(Linux) }
```

```
procedure Set8087CW(NewCW: Word);
function Get8087CW: Word;
procedure SetMXCSR(NewMXCSR: UInt32);
procedure SetMXCSRExceptionFlag(NewExceptionFlag: UInt32);
procedure ClearMXCSRStatus(ExceptionFlag: UInt32);
{$IF defined(CPUX86) and defined(ASSEMBLER)}


{$IF not (defined(PC_MAPPED_EXCEPTIONS) or defined(SJLJ_BASED_EXCEPTIONS)) or defined(ZCX_BASED_EXCEPTI
->
{$IF not (defined(PC_MAPPED_EXCEPTIONS) or defined(SJLJ_BASED_EXCEPTIONS)) or defined(ZCX_BASED_EXCEPTI
```

System.Rtti.pas: unknown SizeOf(TValue)

Winapi.DXFile.pas cannot parse:

```
(*$HPPEMIT '#include "dxfile.h"'{*)
(*$HPPEMIT '#include "rmxfguid.h"'{*)
(*$HPPEMIT '#include "rmxftmpl.h"'{*)
```

14.2.2 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 *Delphi2Cpp* offers the possibility to prepare the Delphi source code such, that *Delphi2Cpp* 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 CPP. In fact the second method is based on the first, because the *Delphi2Cpp* pre-processor converts the CPP part into the special comments and the Delphi2C# translator than simply removes the special brackets (*#_ ... _#*).

In the section about the *overwritten System.pas* there are examples and explanations how to use this feature.

14.2.2.1 Comments (*#_ ... _#*)

Delphi2Cpp 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:

```
(*#_#include "math.h"_#*)
```

->

```
#include "math.h"
```

Remark: in the first version of Delphi2Cpp program these parenthesis were defined as (*_ ... _*). This led to errors in code like in WinAPI.DXGI1_2.pas:

```
function GetDisplayModeList1(
  (* [in] *) EnumFormat: DXGI_FORMAT;
  (* [in] *) Flags: UINT;
  (*_Inout_*) var pNumModes: UINT;
  (*_Out_writes_to_opt_(*pNumModes,*pNumModes)*) out pDes: DXGI_MODE_DESC1): HRESULT; stdcall;
```

14.2.2.2 Predefined identifier Cpp

In addition the the definitions which the user can set in the translation options the identifier *CPP* always is defined in *Delphi2Cpp*. The pre-processor treats this identifier in a special manner. The pre-processor 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:

```
{$ifdef CPP}
  out << s << endl;
{$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;

Delphi2Cpp II ignores the part of code in the `{$else}`-section completely, but it is visible to the Delphi compiler. So, this special way 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 CPP unification option.

14.2.3 Delphi directives to support C++Builder

There are four directives defined in Delphi to support the generation of C++ header files for C++Builder. All the Delphi translations of Windows interfaces don't have to be translated back, but simply are left out by means of these directives. In Delphi2Cpp II they work for other compilers too and you also can use them for your own purposes.
All these directives only have an effect in the global parts of units.

$HPPEMIT
$EXTERNALSYM

$NODEFINE
$NOINCLUDE

These directives can have an impact on the notations of the according types.

14.2.3.1 $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'}
```

14.2.3.2 $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.

Delphi2Cpp II 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}
```

14.2.3.3 $NODEFINE

The *NODEFINE* directive prevents the specified symbol from being included in the C++ header file, while allowing some information to be output to the OBJ file.
Such symbols are expected in special files for C++Builder. For example for C++Builder there is a file " System.Types.h" where the types TSize, TPoint and, TRect are defined in C++ manner. In System.Types.pas these types are marked with *NODEFINE*.

For other target compilers it is recommended to disable the NODEFINE option. Types like the just mentioned TSize, TPoint and, TRect remain then in the translated files.

Syntax:

```
{$NODEFINE identifier}
```

Example:

```
type
       Temperature = type single;
       {$NODEFINE Temperature}
```

14.2.3.4 $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.
```

14.2.3.5 Impact on notations

Types, which are marked as "EXTERNALSYM" or "NODEFINE" are not written into the generated C++ output, if the according option is enabled.. External symbols are provided by the operating system. Therefore the notation which is used in the API of the operation system has to be set in the list of notations.

For example in System.pas there is:

```
  PByte         = ^Byte;        {$NODEFINE PByte}       { defined in sysmac.h }
```

In this case "PBYTE" from Windows.h could be used. (However most symbols which are marked with NODEFINE don't exist in the API and would have to be defined in your own utility files if the NODEFINE option isn't disabled.)

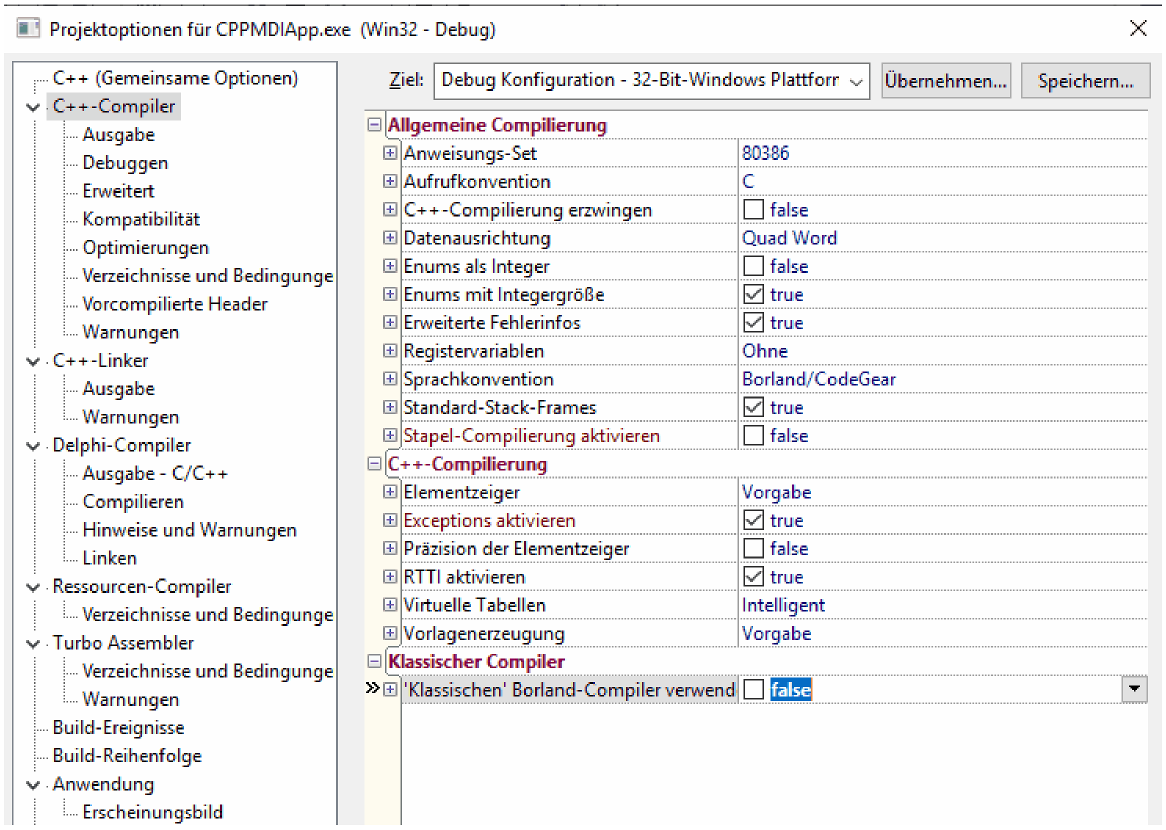
15 Delphi projects

Delphi2Cpp II doesn't convert project files (*.dproj), but if you use C++Builder, Delphi form files (*.dfm) can be reused. However it is recommended to create and maintain Delphi project files (*.dpr) with C++Builder.

For other compilers all these files are not converted.

15.1 Clang

Enter topic text here.


Projektoptionen für CPPMDIApp.exe (Win32 - Debug)

C++ (Gemeinsame Optionen)
C++-Compiler
- Ausgabe
- Debuggen
- Erweitert
- Kompatibilität
- Optimierungen
- Verzeichnisse und Bedingunge
- Vorcompilierte Header
- Warnungen

C++-Linker
- Ausgabe
- Warnungen

Delphi-Compiler
- Ausgabe - C/C++
- Compilieren
- Hinweise und Warnungen
- Linken

Ressourcen-Compiler
- Verzeichnisse und Bedingunge

Turbo Assembler
- Verzeichnisse und Bedingunge
- Warnungen

Build-Ereignisse
Build-Reihenfolge
Anwendung
- Erscheinungsbild

Ziel: Debug Konfiguration - 32-Bit-Windows Plattforr

Allgemeine Compilierung	
Anweisungs-Set	80386
Aufrufkonvention	C
C++-Compilierung erzwingen	☐ false
Datenausrichtung	Quad Word
Enums als Integer	☐ false
Enums mit Integergröße	✓ true
Erweiterte Fehlerinfos	✓ true
Registervariablen	Ohne
Sprachkonvention	Borland/CodeGear
Standard-Stack-Frames	✓ true
Stapel-Compilierung aktivieren	☐ false
C++-Compilierung	
Elementzeiger	Vorgabe
Exceptions aktivieren	✓ true
Präzision der Elementzeiger	☐ false
RTTI aktivieren	✓ true
Virtuelle Tabellen	Intelligent
Vorlagenerzeugung	Vorgabe
Klassischer Compiler	
'Klassischen' Borland-Compiler verwend	☐ false

15.2 dpr Files

Delphi project files with the extension "dpr" are listing all files that are used in a project and contain the code, which starts the application. Normally such files only contains code, which is generated by the Delphi IDE:Though Delphi2Cpp II converts such dpr files to C++ files, it is recommended not to use the converted file, but to let C++Builder create and maintain this file. What C++Builder exactly does isn't documented anywhere and it changes with different versions of C++Builder.

The default dpr file for a VCL forms application looks like:

```
program Project1;

uses
  Vcl.Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

The according file created by C++Builder XE10 Tokyo 2 looks like:

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop
#include <tchar.h>
//---------------------------------------------------------------------------
USEFORM("Unit1.cpp", Form1);
//---------------------------------------------------------------------------
int WINAPI _tWinMain(HINSTANCE, HINSTANCE, LPTSTR, int)
{
  try
  {
     Application->Initialize();
     Application->MainFormOnTaskBar = true;
     Application->CreateForm(__classid(TForm1), &Form1);
     Application->Run();
  }
  catch (Exception &exception)
  {
     Application->ShowException(&exception);
  }
  catch (...)
  {
     try
     {
       throw Exception("");
     }
     catch (Exception &exception)
     {
       Application->ShowException(&exception);
     }
  }
  return 0;
}
//---------------------------------------------------------------------------
```

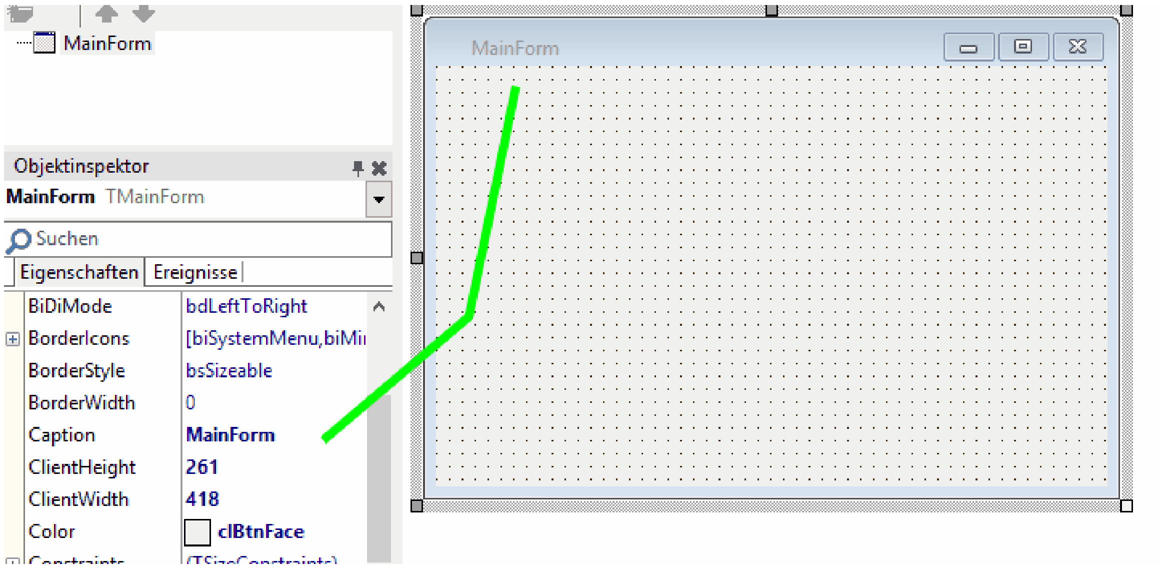
15.3 dfm Files

To reuse Delphi form files, that are files with the extension "dfm", you have to create a C++Builder project manually like your Delphi project, but with empty forms and without user code. It is important however, that the forms in this dummy project get the exactly the same names as the according Delphi forms and also the the corresponding units have to have identical names. The tool *MainFormExchange* assists the following steps.

There are several ways to accomplish this task. Below is a description how to proceed, if the C++Builder project shall be placed into the same folder as the existing Delphi project.

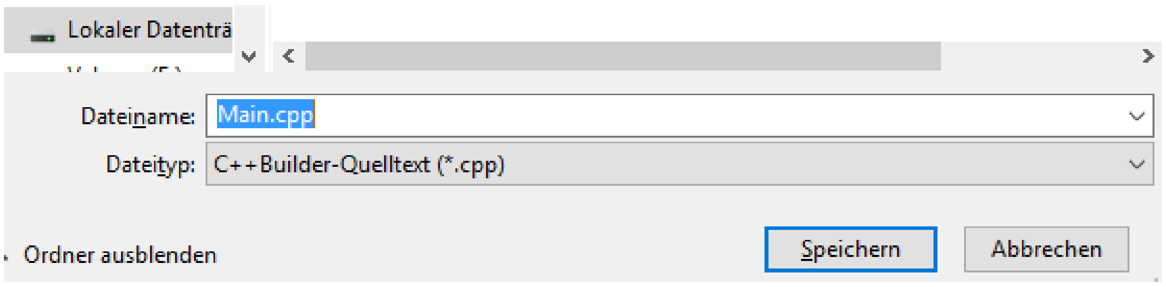
Create a new C++Builder VCL application. You have to choose a configuration that compiles C++11 code.

Let's assume that the name of your Delphi main form is "MainForm", then rename the automatically created main form to this name.



BiDiMode	bdLeftToRight
BorderIcons	[biSystemMenu,biMi
BorderStyle	bsSizeable
BorderWidth	0
Caption	MainForm
ClientHeight	261
ClientWidth	418
Color	clBtnFace

Now you can save the project into the same folder where your Delphi project is saved. At first you will be prompted to enter the name of the main unit. Here you have to choose the same name as the Delphi main form has. Of course it will have the "cpp" extension instead of the original "pas" extension. Let's assume the original file calls "Main.pas":



Dateiname: Main.cpp

Dateityp: C++Builder-Quelltext (*.cpp)

Now it is important, that you rename your original form file "Main.dfm" to another temporary name. If you don't do that, you will be prompted to overwrite the original file. But we still need it.

Next you will be prompted to choose a name for the precompiled header file. It is recommended to take the name of the Delphi project file plus PCH1.h

```
<Delphi projectname>PCH<n>.h
```

Finally you have to choose a name for the C++Builder project. Again it is recommended to take the name of your Delphi project.

`<Delphi projectname>`.cbproj

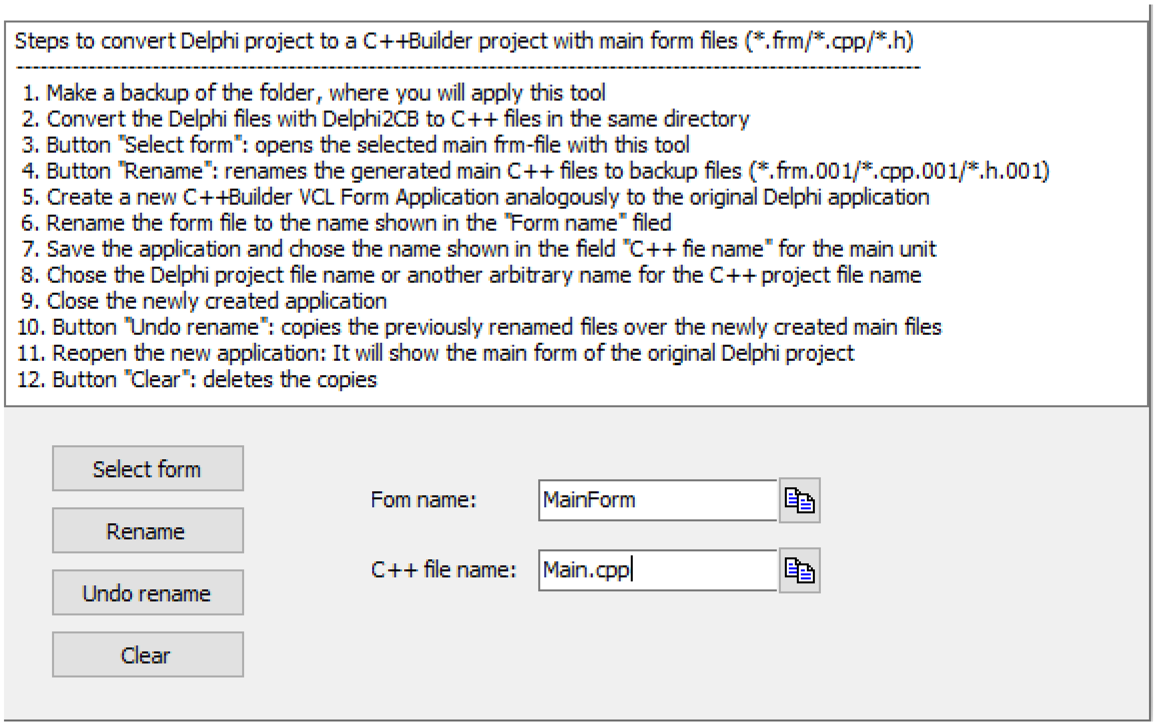
Now you have to close the C++Builder project, delete the automatically created main form file "Main.dfm" and rename the original Delphi main form back to "Main.dfm". This is also a good moment to overwrite the C++ files that were created by the C++Builder IDE with the files of the Delphi2Cpp II translation.

Now you can reopen the C++Builder project. If all components, that you used in the Delphi main form are installed in C++Builder too, the form that you know from your Delphi project is shown identically in C++Builder now.

If you used Components, which are not installed in C++Builder, you will get according error messages. You either can ignore them with the risk that something gets lost on your form or you can cancel and install the needed components first.

15.4 Tool: MainFormExchange

There is a tool called *MainFormExchange* that assists the creation of C++Builder projects.



```
Steps to convert Delphi project to a C++Builder project with main form files (*.frm/*.cpp/*.h)
----------------------------------------------------------------------------------------------
 1. Make a backup of the folder, where you will apply this tool
 2. Convert the Delphi files with Delphi2CB to C++ files in the same directory
 3. Button "Select form": opens the selected main frm-file with this tool
 4. Button "Rename": renames the generated main C++ files to backup files (*.frm.001/*.cpp.001/*.h.001)
 5. Create a new C++Builder VCL Form Application analogously to the original Delphi application
 6. Rename the form file to the name shown in the "Form name" filed
 7. Save the application and chose the name shown in the field "C++ fie name" for the main unit
 8. Chose the Delphi project file name or another arbitrary name for the C++ project file name
 9. Close the newly created application
10. Button "Undo rename": copies the previously renamed files over the newly created main files
11. Reopen the new application: It will show the main form of the original Delphi project
12. Button "Clear": deletes the copies
```

Fom name: MainForm

C++ file name: Main.cpp

After you have selected the main form of the Delphi application, the name of the form and the name of the according C++ unit are shown in the fields of *MainFormExchange*. As in the previous example the main form is called *MainForm* and the main unit is called *main.cpp* in the picture.

When the button *Rename* is pressed, the following renaming is carried out:

Original name	New name	
Main.dfm	Main,dfm.001	original Delphi form file
Main.h	Main,h.001	generated from Main.pas
Main.cpp	Main,cpp.001	generated from Main.pas

Now the steps 5 to 9 shown at the top of image above have to be executed. As result there now is a new C++Builder VCL Form Application analogously to the original Delphi application. Next the original

Delphi form file and the generated C++ files have to be restored, by pressing the button *Undo rename*. At this step also a backup of the manually created clean C++Builder files is made:

Original name	New name	
Main.dfm	Main.dfm.000	newly created clean C++Builder form file
Main.h	Main.h.000	newly created clean C++Builder header file
Main.cpp	Main.cpp.000	newly created clean C++Builder source file
Main,dfm.001	Main,dfm	original Delphi form file
Main,h.001	Main,h	generated from Main.pas
Main,cpp.001	Main,cpp	generated from Main.pas

The 000-backup files are created in order to be able to make comparisons with the generated files in the case of an error. By pressing the *Clear button*, they are deleted.

If all went well, the basic framework for the C ++ version of the original Delphi application now exists. You have to add the d2c helper files now and the rest of the C++ files, that were generated from the Delphi source files.

16 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."

17 Delphi2Cpp 2 compared to Delphi2Cpp 1

Delphi2Cpp2 is based on the experiences with the previous program *Delphi2Cpp 1*, which translates Delphi 7 code only.

Delphi2Cpp 1 supports old style programs based on AnsiChars and AnsiStrings as well as programs based on WideChar and UnicodeStrings.
Delphi2Cpp I2 produces Unicode based code only and preferes the use of a C++11 compiler.
The old Delphi2Cpp also had the option to let "String" be defined as a standard string. This option is removed for the C++Builder target too: for C++Builder Strings are UnicodeStrings, for other compilers there are more options.

Some more differences are:

Delphi2Cpp 2 processes the Delphi language expansions which were added since Delphi 7

Delphi2Cpp 2 also uses the new features of *C++11* to improve the translation results.

- initializing arrays by means of a std::initializer_list

- nested functions are simulated by means of lambda-functions
- with-statements can be rewritten by use of a with-variable of the auto-type
- the behavior of finally is simulated by use of a lambda expression
- for-in loops are converted to range-based for loops.

In addition there are some more changes in Delphi2Cpp 2 to improve or to simplify the translation:

- desired type information can be set in the type-map of the options
- identifiers with an ampersand prefix can be treated correctly
- C-style array return values are converted to reference parameters
- array properties become to Getter/Setter-methods with the array as reference parameter
- the calculation of operator precedence is much more accurate then in the first version of Delphi2Cpp.
- there is an option to create class reference types by which classes can be created.

Classes.pas TList

18 DelphiXE2Cpp11 versus Delphi2CB

Delphi2CB is an low-priced extract of the actual Delphi2Cpp II for users of *C++Builder*.

In contrast to Delphi2Cpp II Delphi2CB

- only uses the additional C++Builder keywords like __property, __finally etc.
- only supports C++Builder as compiler
- only supports Windows as target platform
- only supports Delphi-strings and has no type options at all

19 TextTransformer

Delphi2Cpp II 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/DelphiPrettyPrint_en.html

20 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

or in German at:

http://www.texttransformer.de/D2C_TranslationService_ge.html

I also like make extensions of Delphi2Cpp II 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

or in German at:

http://www.texttransformer.de/Contact_ge.html

Index