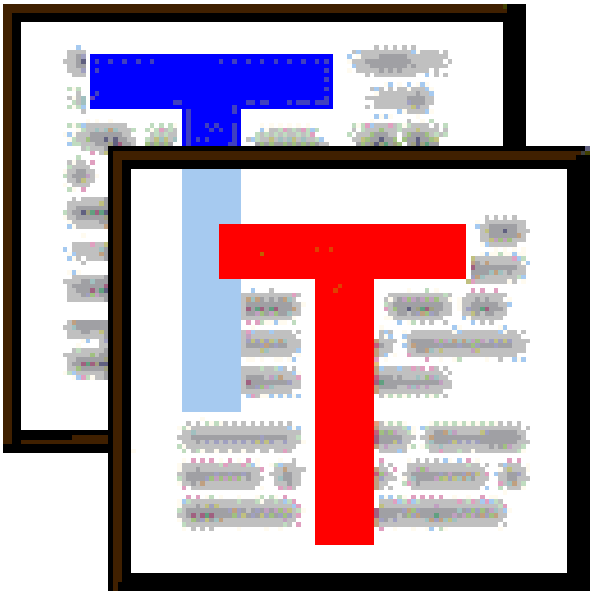
DelphiXE2Cpp11



1 Introduction

Short description

DelphiXE2Cpp11 helps to convert Delphi source code to C++. In contrast to the earlier "Delphi2Cpp", "DelphiXE2Cpp11" processes not only Delphi 7 code but all Delphi language expansions which were added since then. *DelphiXE2Cpp11* also uses the new features of C++11 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.

A comparison of *DelphiXE2Cpp11* and *Delphi2Cpp* is here.

Availability

The actual version of DelphiXE2Cpp11 can be obtained from the TextTransformer websites:

http://www.TextTransformer.com

http://www.TextTransformer.de

2 Installation

The installation is done by the installer DelphiXE2Cpp11Install.exe. All files for projects, examples, source code etc.are copied into the chosen installation directory.

The default path is a sub-folder *DelphiXE2Cpp11* in the user documents folder, like:

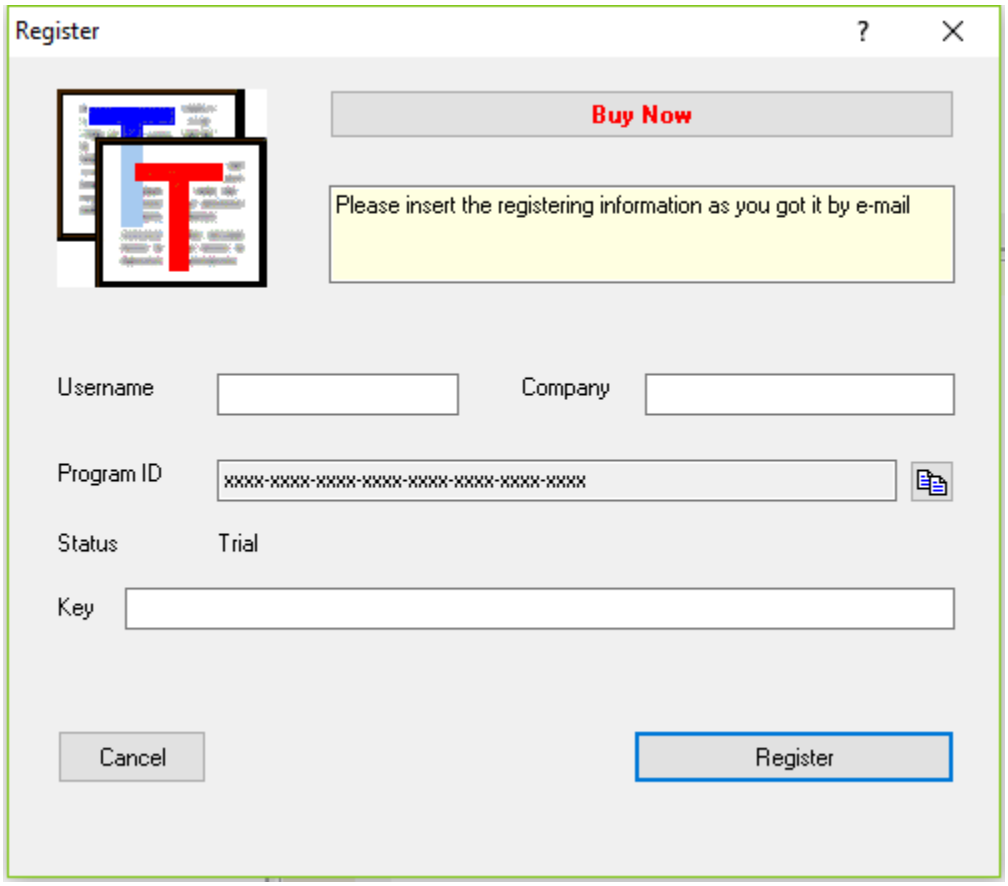
C:\Users\User\Documents\DelphiXE2Cpp11

Regardless of the path, that you chose for the installation, the license file *DelphiXE2Cpp11Lic.dat* will be written at that default path.

3 Registration

If you have bought a license of DelphiXE2Cpp11, **you will get a link to a version of DelphiXE2Cpp11, which you can register.**

The **registration** of DelphiXE2Cpp11, i.e. the removal of trial limitations and the permanent activation of the features, has to be done by the menu: Help->*Registration*. Following dialog is shown:



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

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

After the check of your credit card has been carried out, an **e-mail** which includes the registration data (user name, company and key) is sent to you automatically.
You also will get a link to a version of DelphiXE2Cpp11, which you can register.

It is important to know that if you are downloading DelphiXE2Cpp11 to use on a different computer than the one on which you originally downloaded it, you should transfer it immediately onto removable media, and *not* register it on the first computer.

User name, Company and the **key** then have to be copied unchanged from the e-mail into the corresponding entry fields of the dialog box.Then a click on the **button *Register*** will close the dialog automatically and a message appears, which confirms the success of the registration. A license file *DelphiXE2Cpp11Lic.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 DelphiXE2Cpp11 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:

DelphiXE2Cpp11 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 DelphiXE2Cpp11 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 DelphiXE2Cpp11 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. 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

The main window of the DelphiXE2Cpp11 application consists in a menu, a tool bar and in three windows for the input and for the output.

--

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

--

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

--

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

--

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

--

Options can be saved and reloaded by the buttons

--

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

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

--

The translation is started with F9 or

--

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

--

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

--

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

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

--

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

--

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

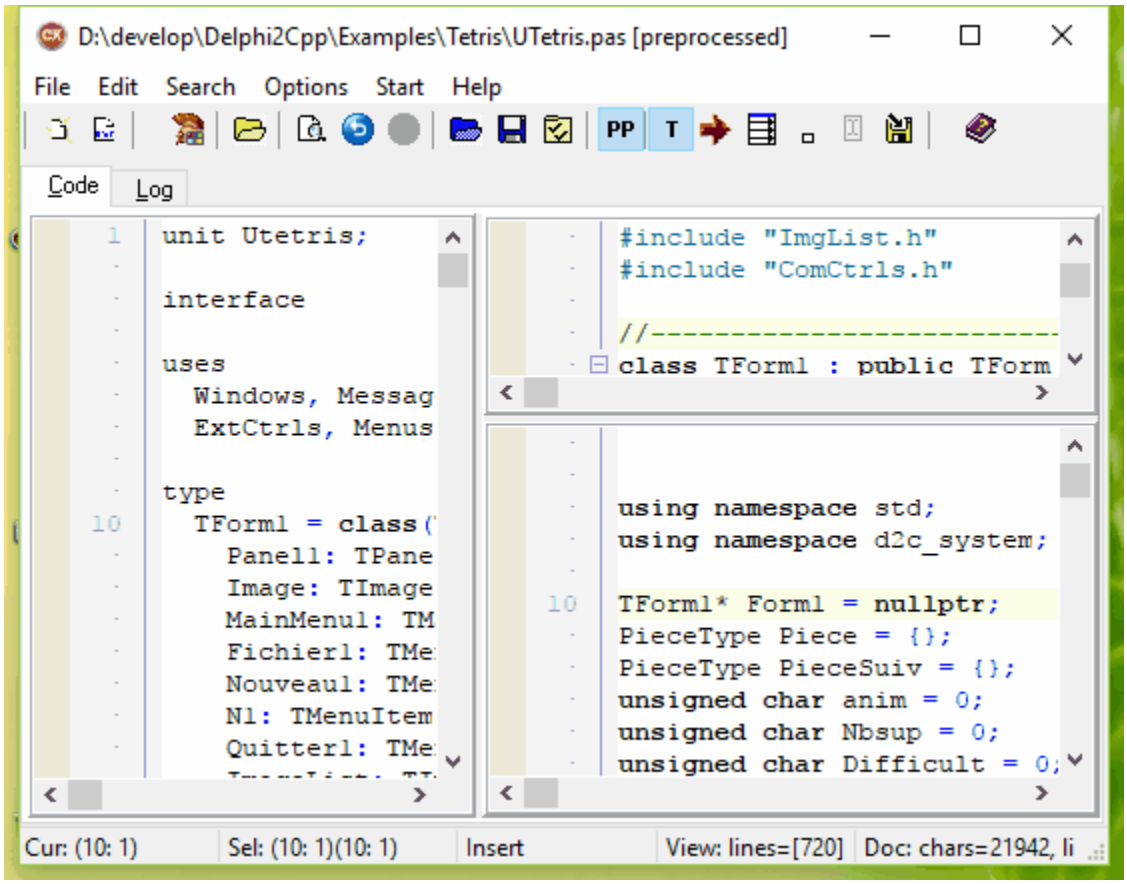
--

This help is shown with F1 or by the button

5.1 Windows

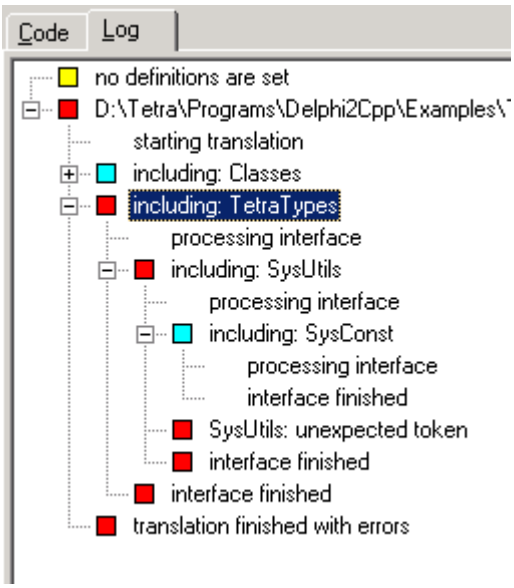
There are three windows in the user interface:

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



5.2 Log panel

The Log panel displays logging messages and errors.



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

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

The picture above is a typical example:
The first line occurs, because no definitions are set in the options.
The red box in front of the filename in the second line means, that there were errors when the file was processed. The cause of the error is marked by the innermost error *SysUtils: unexpected token*. This error is propagated to it's parent nodes.

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

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

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

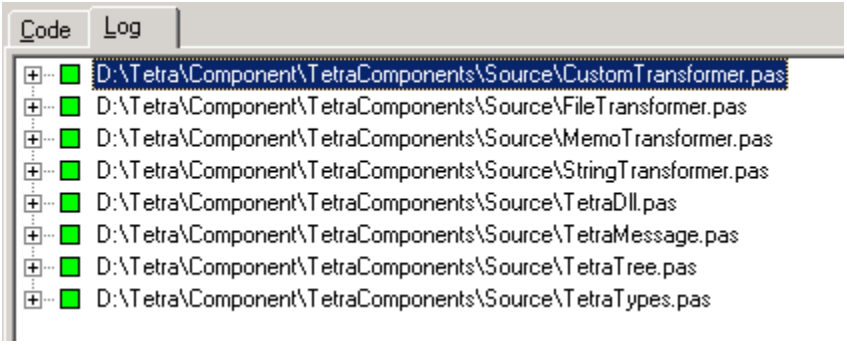
```
function AdjustLineBreaks(const S: string; Style: TTextLineBreakStyle =
       {$IFDEF LINUX} tlbsLF {$ENDIF}
       {$IFDEF MSWINDOWS} tlbsCRLF {$ENDIF}): string;
```

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

--

In the next image you can see an example of the Log panel after use of the file manager, The results

of all files are listed in the tree:

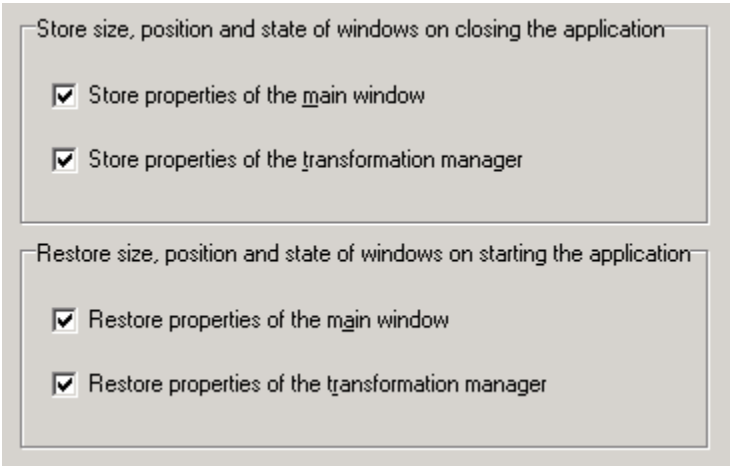


5.3 User options

User options can be accessed in the Options menu at the item "Show user options". These options are saved in the Windows registry and thus persist between different sessions with DelphiXE2Cpp11.

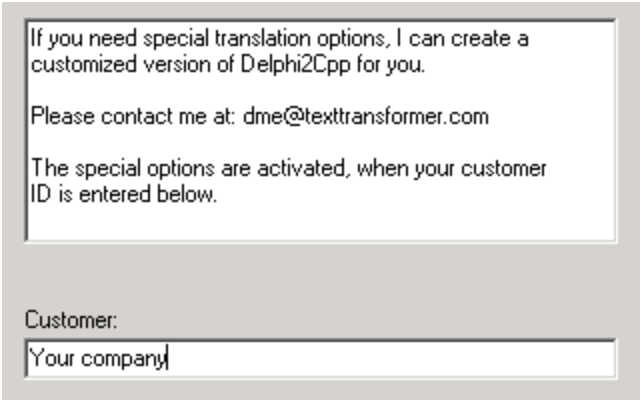
Window positions
Customization

5.3.1 Window positions



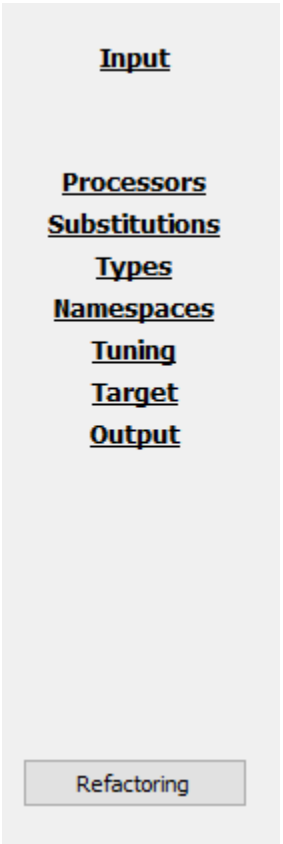
Size positions and state of the main window and the file manager can be stored into the registry and restored from the registry. You can decide to store the values once and than to deactivate a new storage. So the windows will at a new start of DelphiXE2Cpp11 always have the properties that were stored, even if they were change in the previous session.

5.3.2 Customization



5.4 Translation options

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



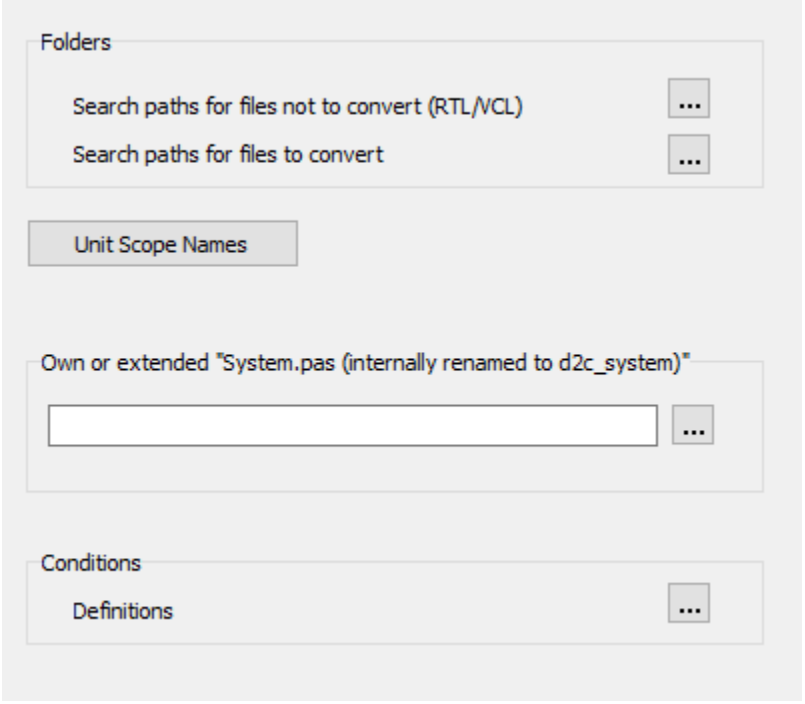
Input options

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

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

5.4.1 Input options

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



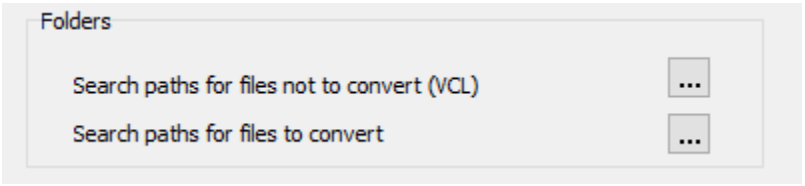
Search paths
Unit Scope Names
System.pas

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

Definitions

5.4.1.1 Search paths

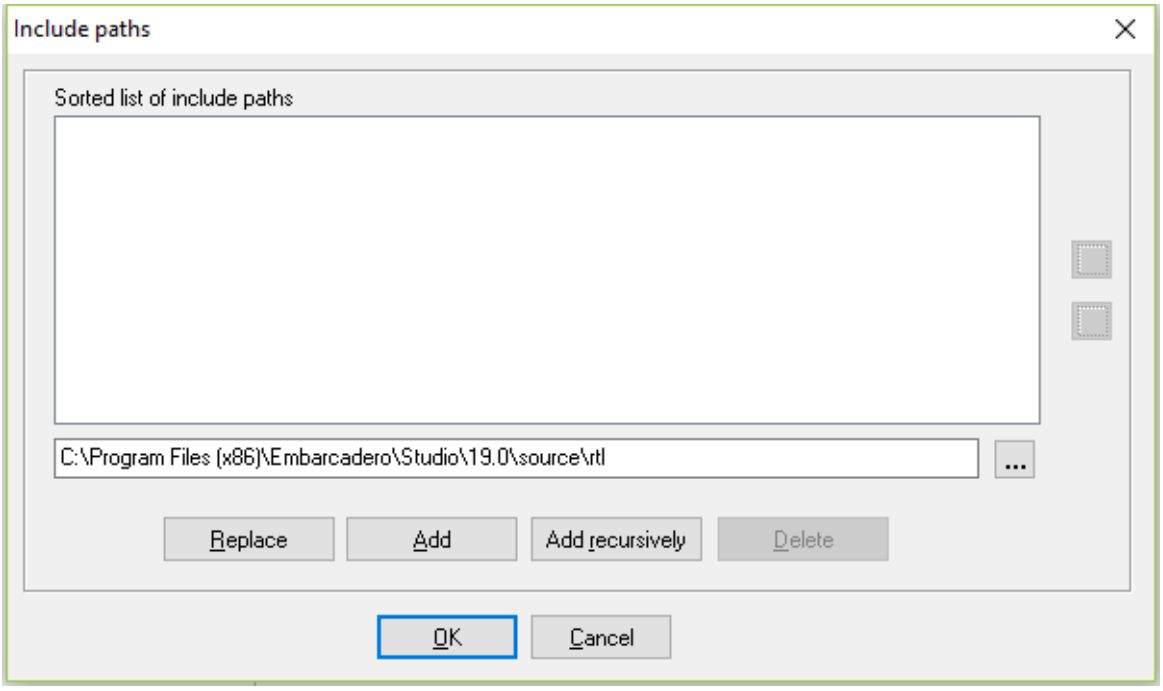
For a correct translation of a Delphi source file the type information of used constants, variables, functions etc.is necessary. If this information is not contained in the actual file, the other used files have to be scanned. As far as these files are in the folder of the source file, they will be found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



These directories are separated into

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

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



As soon as you have clicked at the  "-Button and select a folder, you have the option either to add this folder only or this directory recursively together with all of 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 VCL\RTL

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

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

The paths of the VCL may look like:

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

5.4.1.1.2 Paths to the source files

The paths to the folders of the files, which shall be translated, can be set by a second dialog, analogously the paths to the VCL. DelphiXE2Cpp11 - as Delphi - doesn't make a recursive lookup in the folders. So sub-folders have to be set explicitly too.

5.4.1.1.3 Special VCL headers

DelphiXE2Cpp11 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 DelphiXE2Cpp11 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.

DelphiXE2Cpp11 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.)

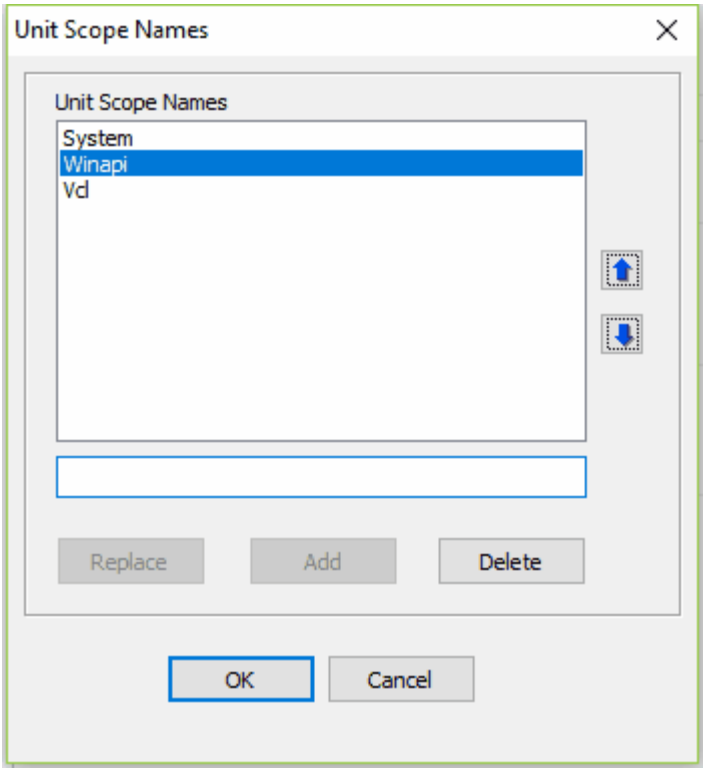
The file ***ShellApi.pas*** is treated in the same C-like manner as *Window.pas*.

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 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

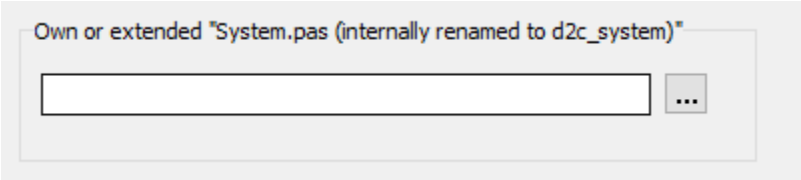
```
uses System.Classes;
```

So, if *System* is in the list of unit scope names, DelphiXE2Cpp11 nevertheless will lookup the file *System.Classes.pas*.

5.4.1.3 Extended "System.pas"

"System.pas" is a source file of special importance in Delphi projects.Fundamental type definitions, procedures and functions are defined in the *System* unit, which is implicitly included in every unit. For example *TObject* is defined there. There are other intrinsic definitions like the *Read*, *Write* or *Str* function, which are accessible in each unit too. These intrinsic function are built into the Delphi compiler. *DelphiXE2Cpp11* 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.



Such an individual *System.pas* called *d2c_system.pas* is in the *Source* folder of the *DelphiXE2Cpp11* 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

DelphiXE2Cpp11 looks up system types and functions etc. in following order::

1. *DelphiXE2Cpp11* will look for declarations at first in your own *System.pas*, if it exists.
2. If the declaration is not found there, *DelphiXE2Cpp11* 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, *DelphiXE2Cpp11* simulates the most important parts of this file.

Mostly DelphiXE2Cpp11 cannot distinguish different elements with the same name. DelphiXE2Cpp11 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.

5.4.1.3.1 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);
```

DelphiXE2Cpp11 uses such declarations - by removing the underscore - if nothing else is found. Indeed, just for the *SetString* function. *DelphiXE2Cpp11* 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 DelphiXE2Cpp11 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 DelphiXE2Cpp11 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 DelphiXE2Cpp11 results in:

```
void __fastcall SetString( AnsiString& S, char* Buffer, int Len )
{
  S[0] = Len;
```

```
  if ( Buffer != NULL )
      memmove( &S[1], Buffer, Len );
}
```

5.4.1.3.2 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*. (DelphiXE2Cpp11 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;
```

5.4.1.3.3 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;
```

```
}
```

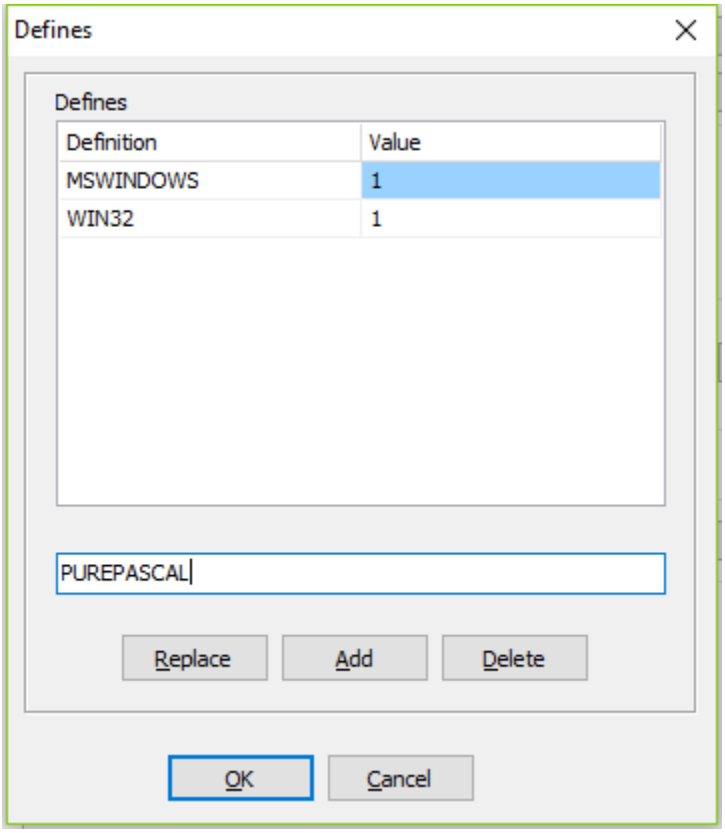
For integer types *Inc and Dec* are converted automatically to the C++ incrementing and decrementing operators. E.g.

```
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.

5.4.1.4 Definitions

Delphi code often contains directives for conditional compilation of parts of the source text. DelphiXE2Cpp11 evaluates such directives too. You can set the definitions in the option dialog



There are limitations for the evaluation of such expressions.

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

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

5.4.1.4.1 Windows.pas

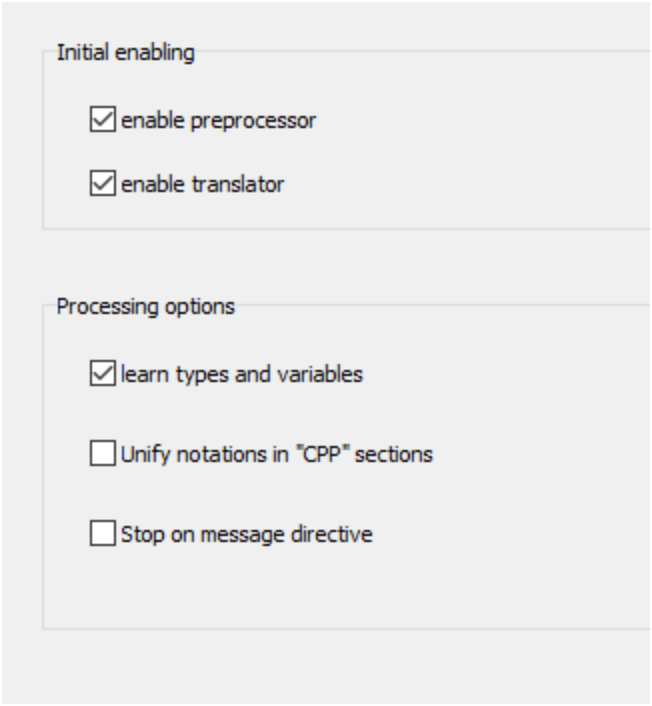
If there is no Definition set either of CPUX86 or of Win64 the Windows.pas cannot be parsed. That's because of the following code:

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

5.4.2 Processor options

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



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

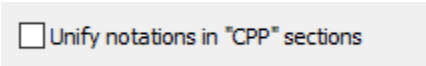
The *overwritten System.pas* gets always preprocessed, even if the option to do so is disabled.

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

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

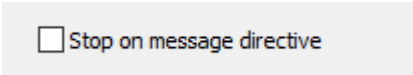
The option Stop on message directive determine what happens, if a message directive would remain in the pre-processed code.

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*

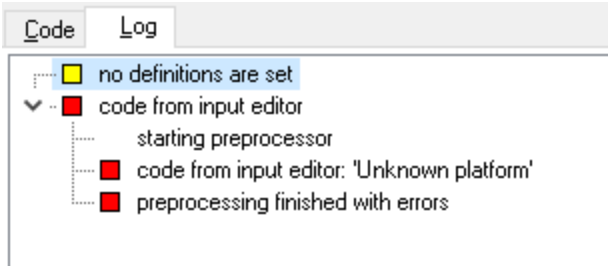
alternative. Therefore the function definition would be reduced to a function declaration.

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



On the other side, there are messages which you might want to ignore. In the following case DelphiXE2Cpp11 isn't able to calculate the correct result of the condition:

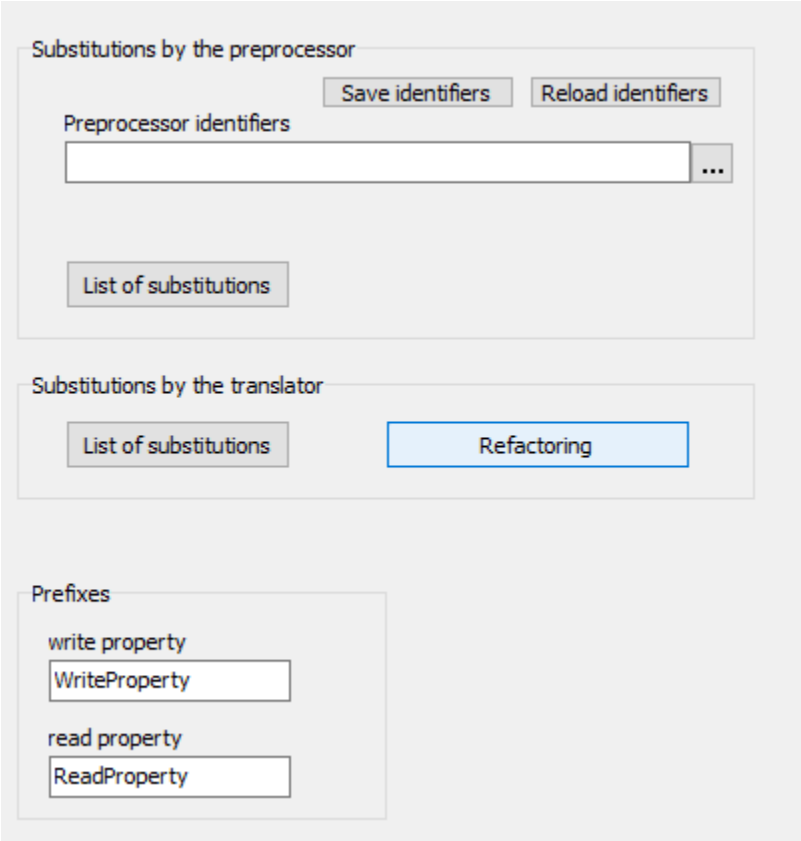
```
{$IF SizeOf(Extended) <> SizeOf(TExtended80Rec)}
  {$MESSAGE ERROR 'TExtended80Rec has incorrect size'}
{$ENDIF }
```

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

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

5.4.3 Substitution options

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



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

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

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

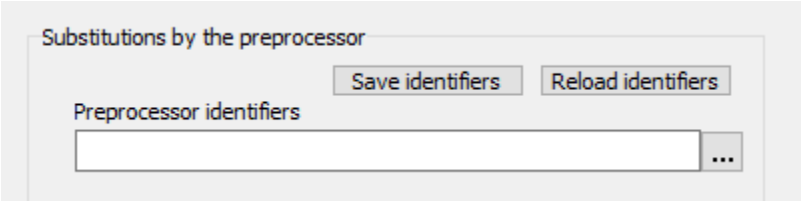
3. the substitutions by the translator are executed after the code already has been parsed.
4. also some kinds of refactoring can be done now.

5. 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: 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

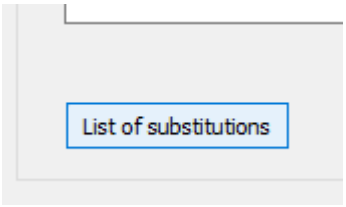
MinComp
MaxComp
NaN
Infinity
NegInfinity

Sum
SLICE

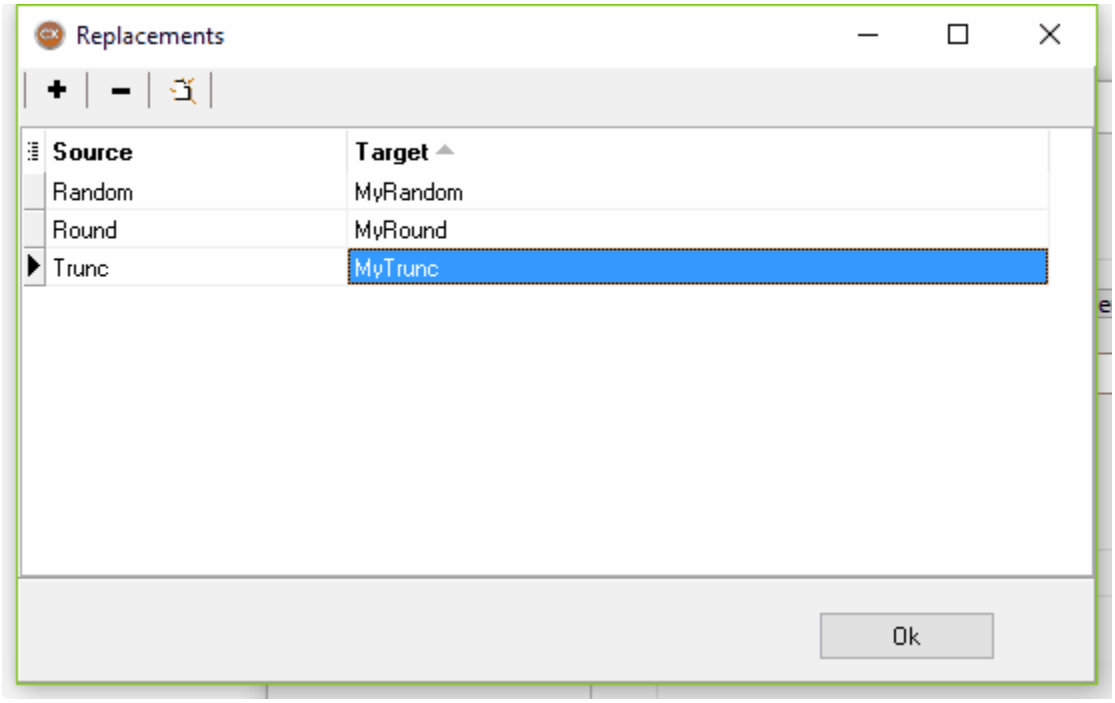
Sorry, the list may not be complete

5.4.3.2 Substitutions in the preprocessor

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



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



add a new row

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; .
```

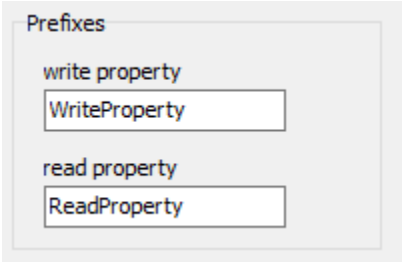
2. The identifier is already recognized by the translator before the substitution takes place. Therefore it can be substituted by something completely different, without affecting the translation process. E.g.

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

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

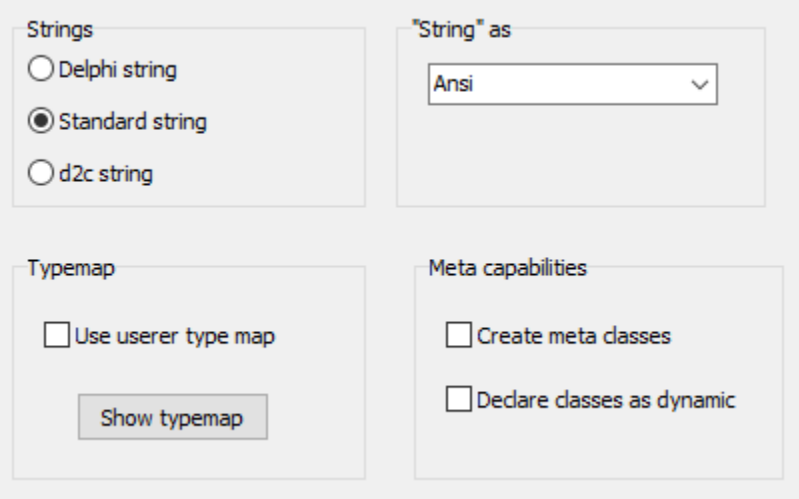
5.4.3.4 Prefixes for properties

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.



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.

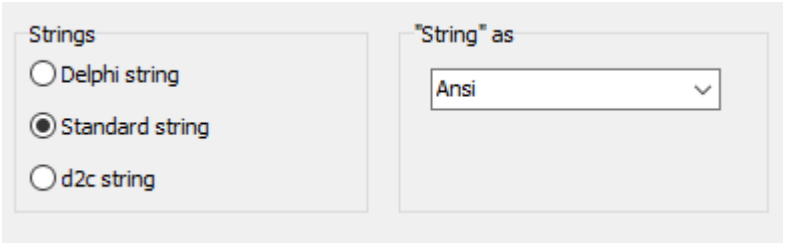
the insertion of macros to access runtime class information

d2c_config.h !!!

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

5.4.4.1 String types

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



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';
```

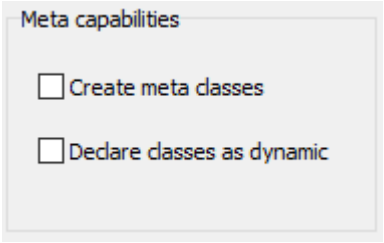
is translated for an ***Ansi*** association to:

```
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, DelphiXE2Cpp11 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 Delphi2Cpp hasn't been re-implemented in DelphiXE2Cpp11 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".

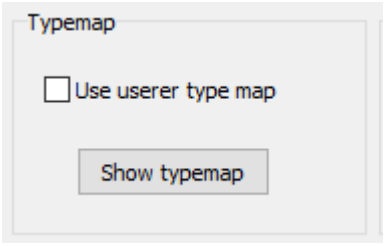
The following table compares the class names and functions of the MFC and Delphi:

class CObject	class TObject
struct CRuntimeClass	class TMetaClass
CObject::GetRuntimeClass	TObject::ClassType

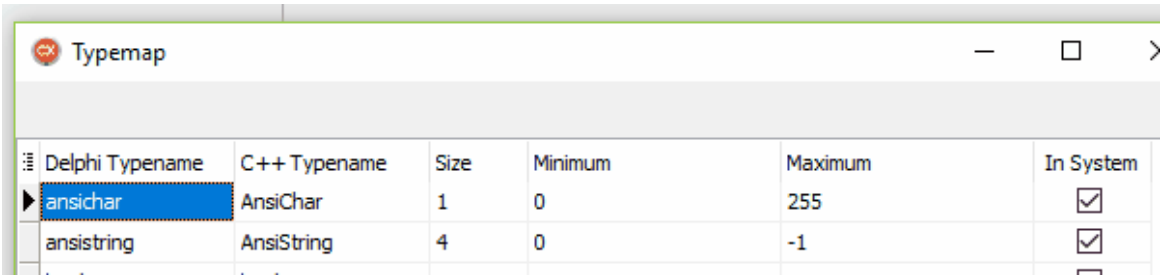
CRuntimeClass::IsDerivedFrom	TMetaClass::InheritsFrom
CObject::IsKindOf	TObject::InheritsFrom
CRuntimeClass::CreateObject	TMetaClass::Create
CObject::CreateObject	TObject::Create

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:



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
```

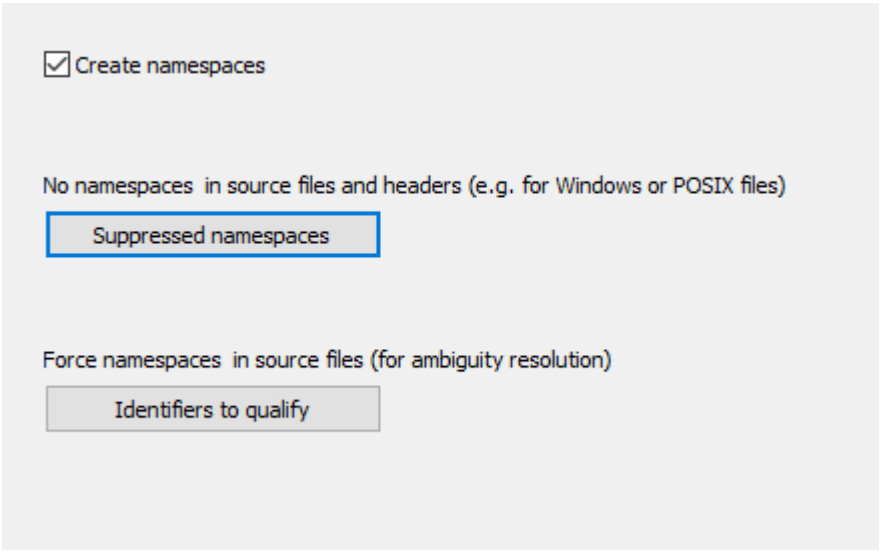
size_t

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.

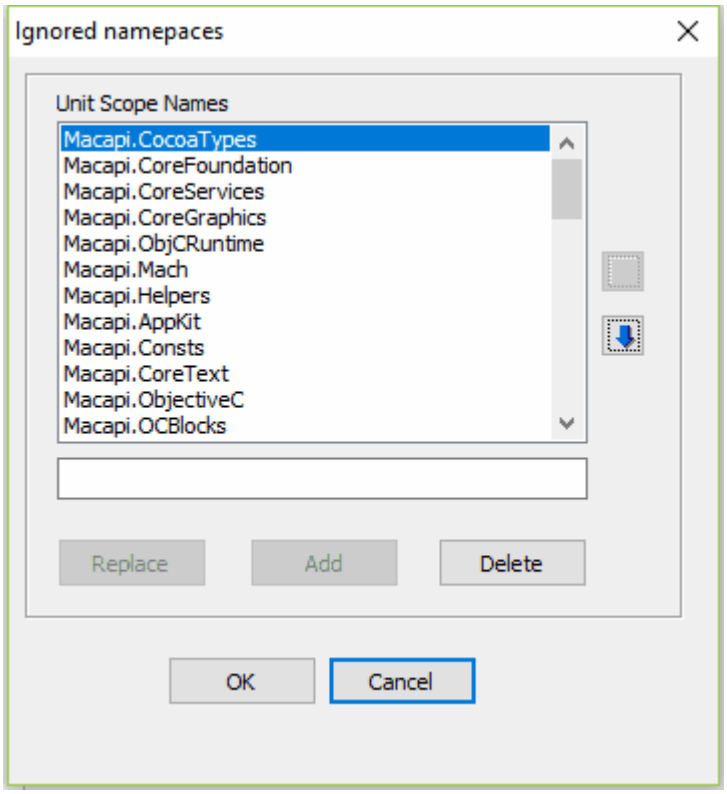
5.4.5 Namespace options

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



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

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



At the example the namespaces for files of the OSX-API are suppressed. For Windows it is recommended to Suppress "WINAPI::Windows".

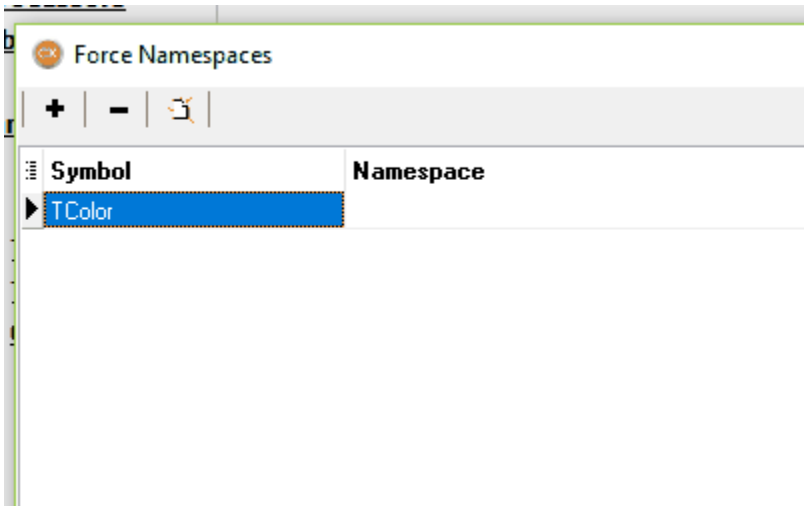
// todo namespaces for all external types are suppressed automatically

In WINAPI.Windos BOOL is defined as

BOOL = LongBool;

Therefore this type will be qualified in the translated code as: WINAPI::Windows::BOOL.

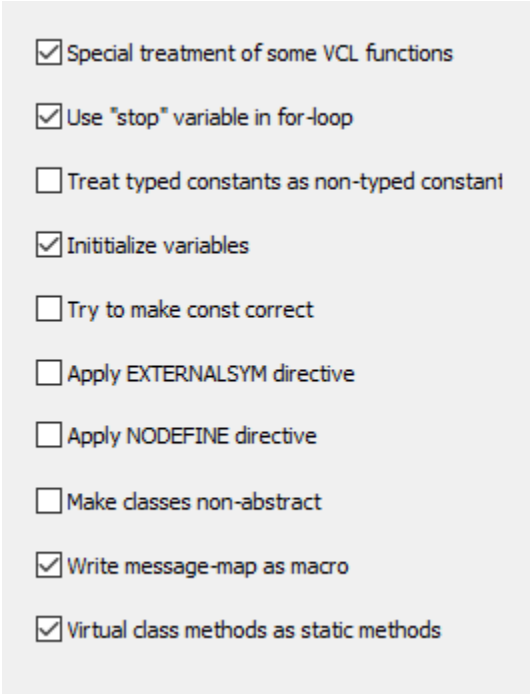
The other way round identifiers for types can be listed, for which a qualification shall be forced:



If the cell for the namespace is empty, DelphiXE2Cpp11 will lookup the namespace, if it has a value, this is set as namespace.

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 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.. *DelphiXE2Cpp11* 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. *DelphiXE2Cpp11* 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 DelphiXE2Cpp11 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, *DelphiXE2Cpp11* 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++. *DelphiXE2Cpp11* 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.

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();
```

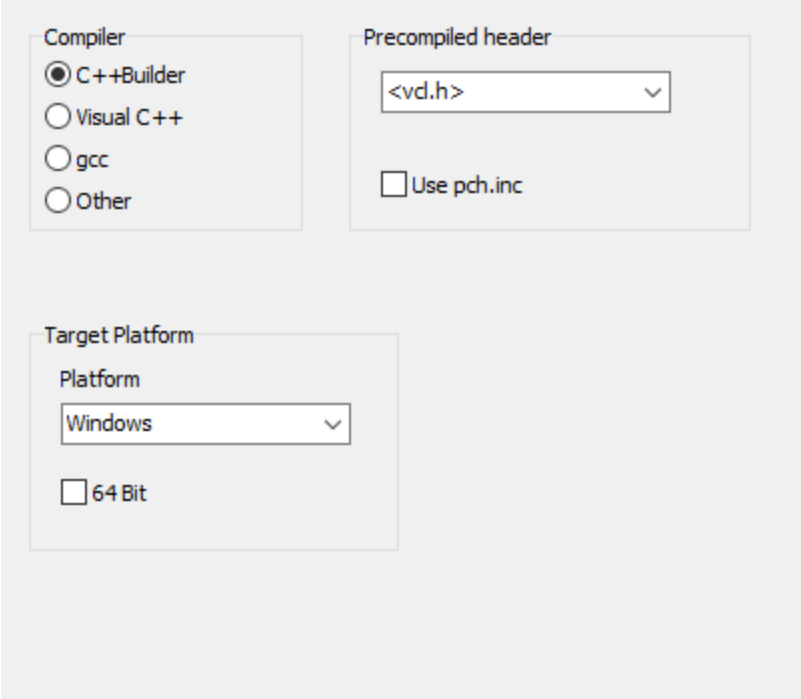
If there are overridden Delphi virtual class methods, the option *Virtual class methods as static methods* has to be disabled. The method then becomes

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

DelphiXE2Cpp11 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..



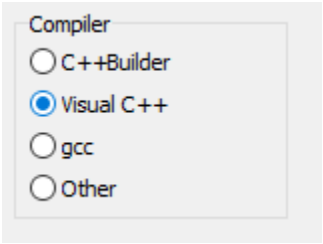
Compiler

Precompiled header

Target platform

5.4.7.1 Compiler

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



C++Builder

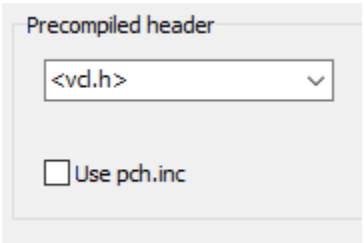
C++Builder is made on top of a Delphi-Compiler and has some C++ extensions to cope with language features of Delphi, which cannot be reproduced adequately with the standard C++.

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.

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 DelphiXE2Cpp11 then will insert into the generated code.



There are three options:

1. <vcl.h>

normally used with C++Builder. DelphiXE2Cpp11 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.

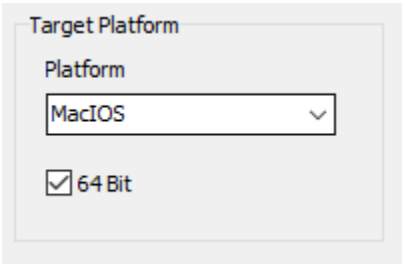
5.4.7.2.1 pch.inc

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.

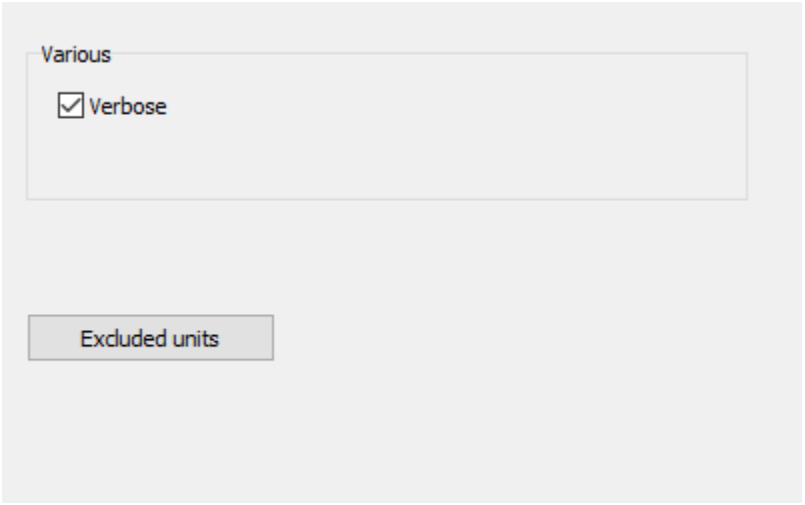


The alternative platforms are: Window, Linux or MacIOS. The selected platform makes no big difference in the generated C++ code, because *DelphiXE2Cpp11* aims to generate portable C++ code. But nevertheless some types are functions might be named differently for different platform. The source code for 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.

5.4.8 Output options

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



Excluded units

The output of using statements for units used in a Delphi source file can be suppressed with this option. A reason why one might want to do this is, that the types from the used file are removed by refactoring or that the used file cannot be translated at all.

5.4.8.1 Verbose

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

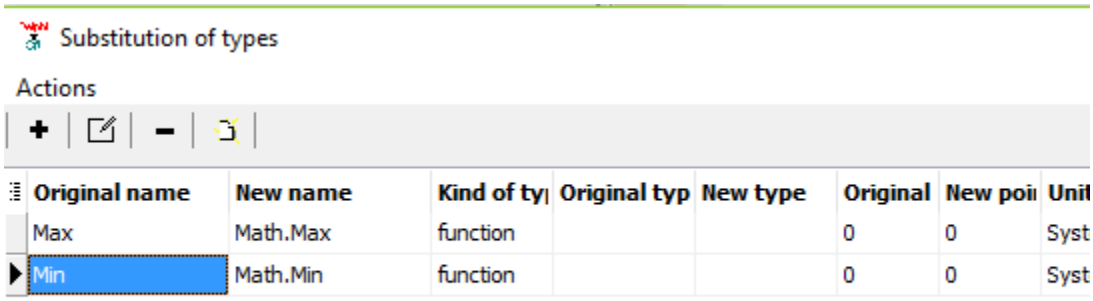
To distinguish these comments from converted comments, which stem from the Delphi source code, they are marked with a hash character (octothorpe) '#'.

E.g.:

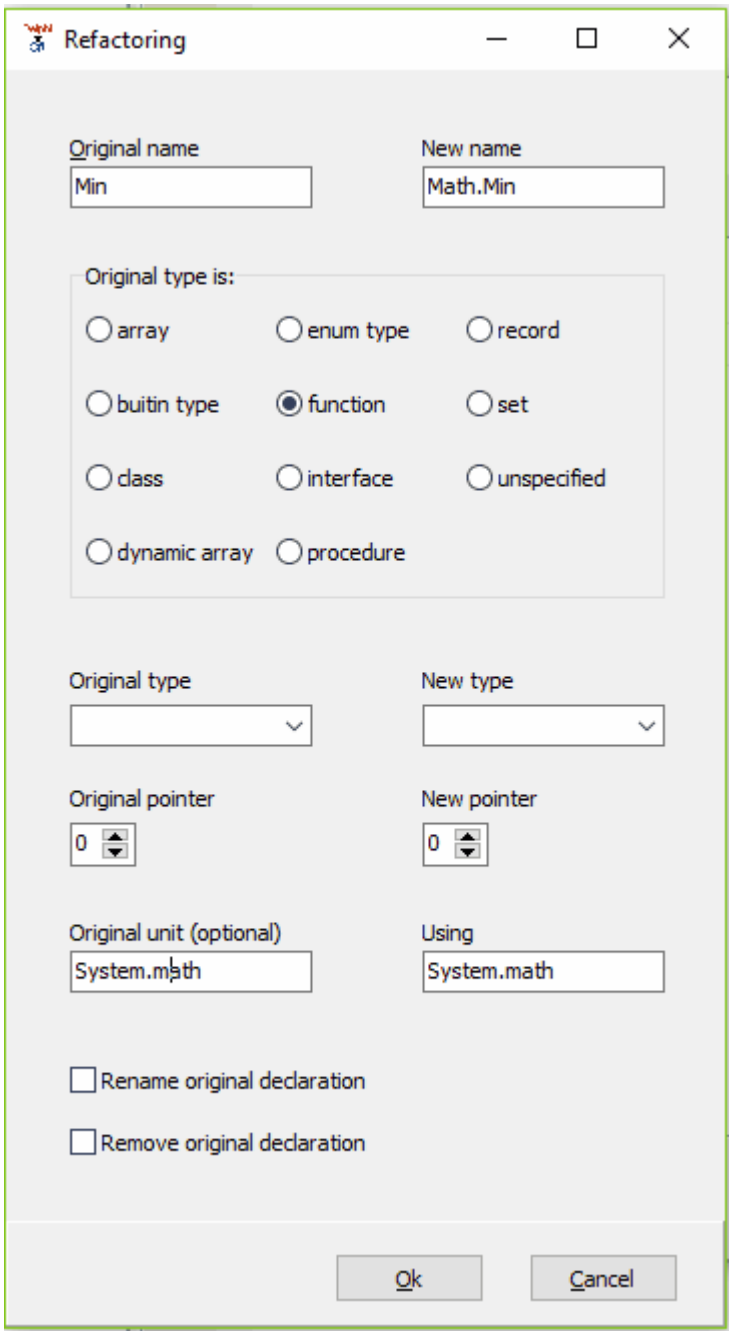
```
WORD Words[4/*# range 0..3*/];
```

5.4.9 Refactoring

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



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



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

"Original name" and "New name"

The original name of a variable, function or constant in the Delphi source code will be changed to the new name in the C++ output. The input in the field is treated case insensitive in the same way as the

source code by the pre-processor. For example with input in the image above all occurrences of "Min" in the source code will be changed to to "Math.Min", regardless whether "Min"; "min", "mIN" or any other case occurs in the code.
If the identifier for the original name isn't contained in the list of notations, 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"

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

Original unit

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

Using

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

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

Rename original declaration
Remove original declaration

not implemented yet

5.5 Translation

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

Three steps are executed for a translation:

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

5.5.1 Preprocessing

A preprocessor fulfils two tasks:

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

5.5.1.1 Conditional compilation

DelphiXE2Cpp11 uses a preprocessor (pretranslator), which prepares the source text so that directives for the conditional compilation are evaluated and removed.
Conditional expressions like

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

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

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

```
  {$DEFINE EXTENDEDHAS10BYTES}
{$ENDIF}
```

The size is taken from the type-map.

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

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

The source code has to be corrected by hand then.

Include directives are executed too.

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

The file *filename* is included into the source.

The definitions can be set in the options dialog.

5.5.1.2 Unification of notations

While Delphi code is case insensitive, C++ code is case sensitive. So different notations of identifiers have to be unified. DelphiXE2Cpp11 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. DelphiXE2Cpp11 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:

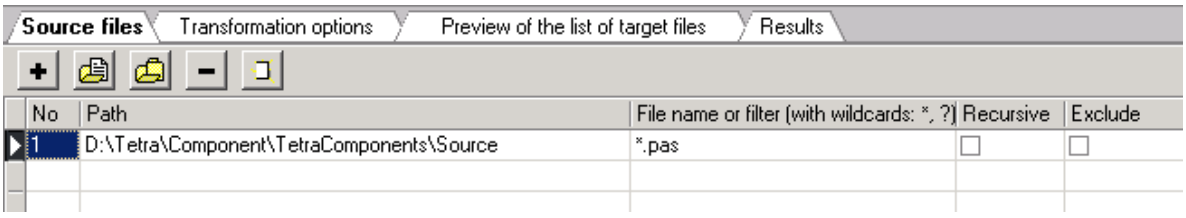
At first the button in the tool bar of the manager for executing the translations is deactivated since no source files are selected yet. Only if this has happened and options are set as requested, the translation can be started. Before starting the translations, you can check the list of the files which will be produced. There is a page of his own for each of these steps in the file manager:

1. Source files
2. Translation options
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 Selecting source files

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



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


Insert an empty row

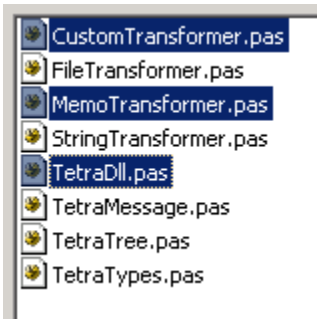

Select a single source file


Select a whole source directory

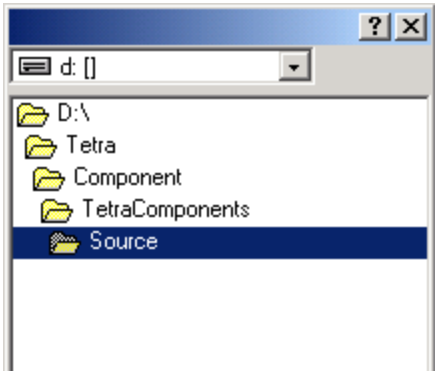

Deleting a row


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

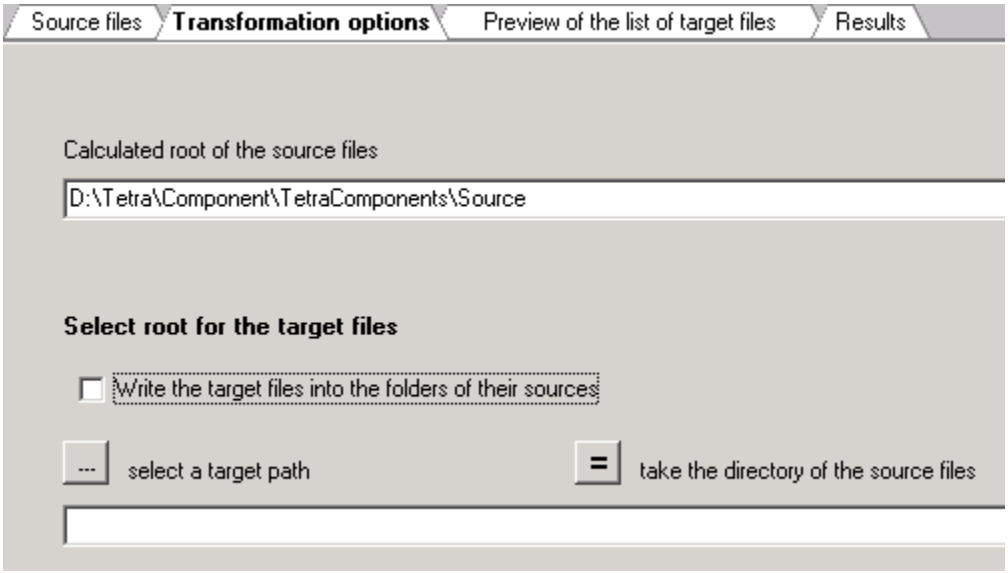
The check box in this field can be activated only for folders. If it is activated, then all files in the sub-folders of the shown directory are transformed too.

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.

5.6.2 Translation options

The path for the target files have to be selected on the second page of the file manager.



Writing text into a specified folder

If the check box "Write the target files into the folders of their sources" is deactivated, the input fields for the target folders are enabled.

By the button

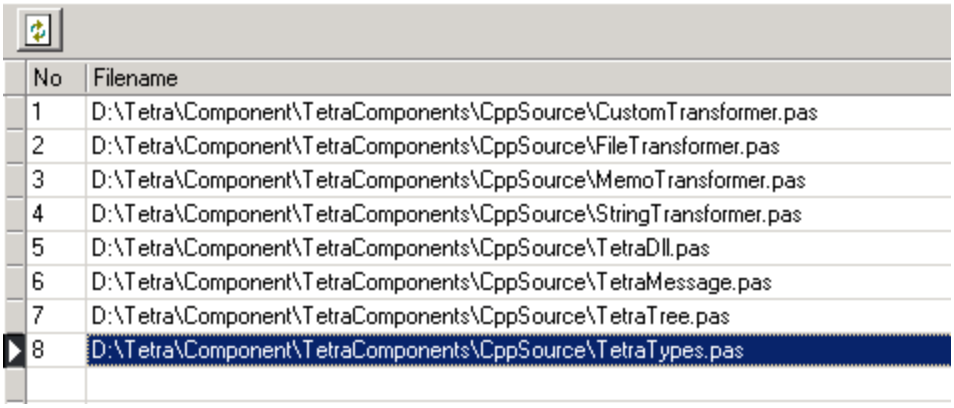
a dialog for the selection of a different target directory is opened.

The button:

can help to navigate faster to the new target directory

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  .

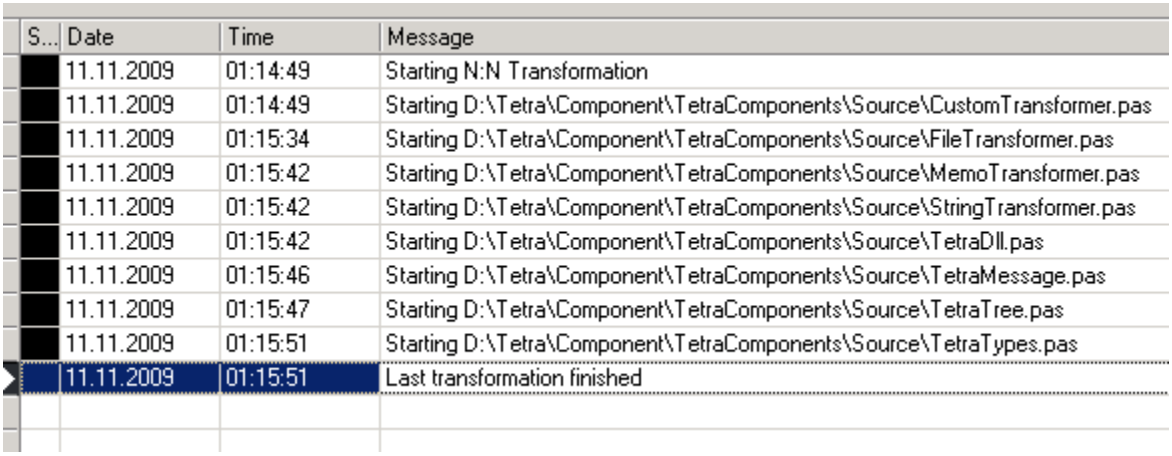
5.6.4 Starting the translation

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

DelphiXE2Cpp11.exe can be called from the command line too. You then have to pass some parameters.

6.1 Parameter

DelphiXE2Cpp11.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:

DelphiXE2Cpp11 -p PROJECT -m MANAGEMENT

and in the second case:

DelphiXE2Cpp11 -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	DelphiXE2Cpp11 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 DelphiXE2Cpp11 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 DelphiXE2Cpp11 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 DelphiXE2Cpp11 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;
```

In such cases it is recommended either to let the pre-processor substitute such expressions or to modify the source code. Otherwise DelphiXE2Cpp11 simply ignores such ampersands, that means "&type" becomes "type". In that case the parser will stop at that position, because of the unexpected *type* keyword. If "&type" is substituted for example by "amps_type" everything works well. You even can let the translator make a second substitution from "amps_type" to "type", if you like. This substitution is made after "amps_type" has been recognized as an identifier.

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

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

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

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

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

```
&End
```

7.1.3 Simple substitutions

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

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

7.1.4 String constants and single characters

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

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

7.1.5 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;
```

DelphiXE2Cpp11 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 *DelphiXE2Cpp11*. There also is an extended System.pas, which has to be set in the translation options, to let DelphiXE2Cpp11 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"      // partially in Delphi2Cpp trial version
#include "d2c_systypes.h"
#include "d2c_sysconst.h"
#include "d2c_syscurr.h"
#include "d2c_sysdate.h"
#include "d2c_sysobj.h"    // not in Delphi2Cpp trial version
#include "d2c_openarray.h"  // not in Delphi2Cpp trial version
#include "d2c_sysexcept.h"
#include "d2c_sysmath.h"    // partially in Delphi2Cpp trial version
#include "d2c_sysstring.h"  // partially in Delphi2Cpp trial version
#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 DelphiXE2Cpp11 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 DelphiXE2Cpp11 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 interfacesdepends 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

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
Virtual 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 *DelphiXE2Cpp11* 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++. *DelphiXE2Cpp11* 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 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**

```
Base = class
public
  constructor Create(arg : Integer);
  destructor Destroy;
private
  FList : TList;
  FI : Integer;
  FTimeOut: Longint;
end;
```

C++ translation

```
class Base: public System::TObject {
  friend class Derived;
  public:
  __fastcall Base( int arg );
  __fastcall ~Base( );
  private:
  TList* FList;
  int FI;
  int FTimeOut;
  public: inline __fastcall Base () {} <- dangerous
};
```

```
  constructor Base.Create(arg : Integer);
  begin
  end;
```

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

If the members are initialized explicitly in Delphi, *DelphiXE2Cpp11* tries to find the according statements and puts them into the initialization list of the class constructor:

```
constructor Base.Create(arg : Integer);
begin
  FList := TList.Create;
  FI := arg;
  if arg <> $00 then
    FTimeOut := arg
  else
   FTimeOut := DefaultTimeout;
end;
```

```
__fastcall Base::Base( int arg )
 : FI(arg),
   FList(new TList),
   FTimeOut(0)
{
  if ( arg != 0x00 )
    FTimeOut = arg;
  else
    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 *DelphiXE2Cpp11* tries to take care about such cases. But *DelphiXE2Cpp11* cannot find all such hidden dependencies, as in the following example:

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

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

In such cases constructors have to be corrected manually like:

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

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

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

implementation

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

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

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

Obviously, this is not correct. Therefore DelphiXE2Cpp11 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;
}
```

7.4.1.2.2.3 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 DelphiXE2C++11 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
};
```

7.4.1.2.2.4 Virtual constructors

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

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

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

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

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

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

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

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 *DelphiXE2Cpp11*cannot recognize this.

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

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

```
  x,y : Double;
end;


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

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

After translation the two constructors become ambiguous:

```
__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 ))
{
}
```

They not only have the same signature now, but also the same name. In such cases the conflict has to be avoided manually. For example you can remove the second constructor and define a static function instead:

```
TCoordinate* __fastcall TCoordinate::CreatePolar( double Radius, double Angle )
{
  return new TCoordinate(Radius * cos( Angle ), Radius * sin( Angle )
}
```

At all positions, where the second constructor shall be used, the new function has to be used instead. This positions can be found easily, because DelphiXE2Cpp11 inserts the original name of the constructor as a comment into the translated code, if the *Verbose* option is enabled and if the name is not written exactly as "Create".

```
P = /*CreatePolar*/ new TCoordinate( AX, AY );

->

P = TCoordinate::CreatePolar( AX, AY );
```

7.4.1.2.3 Destructors

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

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

DelphiXE2Cpp11 tempts to find calls of destructors of the base class and to comment them out in C++. Thereby is assumed that the destructor of the base class is virtual. This has to be checked by the user.

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

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.Therefore *DelphiXE2Cpp11* has to use a tricky construction to reproduce this ability of Delphi.
2. In the defining declaration of a class method, the identifier *Self* represents the class where the method is called. In C++ however inside of a static function there is no counterpart to Delphi's *Self* (*this* isn't defined her).

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

non virtual class methods
virtual class methods
Self instance

7.4.1.2.4.1 non virtual class methods

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

```
type
 TBase = class(TObject)
 public
    class function ClassMethod(xi: Integer): Integer;
 end;

...

var
  pBase: TBase;
  i : Integer;
begin
  i := TBase.ClassMethod(0);  // calling through a class reference
  // ...
  i := pBase.ClassMethod(0);  // calling through an object reference
```

This is translated in the following way:

```
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
```

7.4.1.2.4.2 virtual class methods

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 DelphiXE2Cpp11 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.

7.4.1.2.4.3 Self instance

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

example:

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

DelphiXE2C++11 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 DelphiXE2Cpp11 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 a private or protected member is visible anywhere in the module where its class is declared. In C++ a private or protected member is visible only inside of the class. So the translation by *DelphiXE2C++11* makes all classes and records in the same module to friends of each other.

```
class TStringBuilder : public System::TObject
{
  typedef System::TObject inherited;

  friend struct WordRec;
  friend struct LongRec;
...
```

A complete solution would list all free functions of the unit too.

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 explicit.
(DelphiXE2Cpp11 ignores the {$M+} directive, which would make them public.)

7.4.1.2.7 Creation of instances of classes

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

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

7.4.1.3 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 keyword "__interface" is used to construct interfaces:

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

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

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.

7.4.2 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 *Delphi2C#*.

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, DelphiXE2Cpp11 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.

```
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;
}
```

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.

7.4.2.5.2 Static array parameter

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

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


procedure foo(Arr: Array of Char);

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

is translated to:

```
void foo( const char* Arr, int Arr_maxidx );

void bar( )
{
  char chararray [ 10 ];
  foo( chararray, MAXIDX( chararray ) );
}
```

7.4.2.5.3 Dynamic array parameter

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

```
type
strarray = Array of String;
procedure Check(aSources : strarray);
->
typedef DynamicArray < String > strarray;
void __fastcall Check( const strarray& aSources);
```

In this case DelphiXE2Cpp11 translates such a parameter as a reference to a dynamic array. The parameter is translated like an open array however, if the type isn't defined explicitly. In this case the C ++ compiler will fail, if it is tried to pass a dynamic array to this function.

7.4.2.5.4 array of const

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

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

However, while all elements of an open array have the same type, elements of different types can be passed as an *array of const*. Indeed the *array of const* is an open array of *TVarRec* elements and *TVarRec* is a variant type which 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.

7.4.2.5.4.1 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.

7.4.2.5.4.2 array of const for other compilers

array of const is reproduced for other compilers DelphiXE2Cpp11 by an *OpenArray* class defined in d2c_openarray.h, which is based on a std::vector.

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

->

void foo ( const OpenArray<TVarRec>& Args );
```

For the call of such functions a type definition ***ARRAYOFCONST*** is used:

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

The type definition is:

```
typedef OpenArray<TVarRec>  ARRAYOFCONST;
```

Since this class has the *size* method an additional parameter isn't necessary.

7.4.2.5.4.3 array of const vs. set's

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

```
type
TCharSet = set of Char;

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

foo(['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 DelphiXE2Cpp11 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 DelphiXE2Cpp11*) 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 DelphiXE2Cpp11 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 DelphiXE2Cpp11 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 DelphiXE2Cpp11 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 *DelphiXE2Cpp11* 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.

7.5 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

// todo

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 - and array parameters: The different cases of single

parameters and how they are translated are listed below.

```
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

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

Other compilers

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

```
void __fastcall Foo(PInteger& param);
void __fastcall Foo(MyRecord param);
void __fastcall Foo(const MyRecord& param);
void __fastcall Foo(MyRecord& param);
void __fastcall Foo(MyRecord& param);

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

```
void Foo(PInteger& param);
void Foo(MyRecord param);
void Foo(const MyRecord& param);
void Foo(MyRecord& param);
void Foo(MyRecord& param);

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

There is a problem with variable 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 *DelphiXE2Cpp11* 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 *DelphiXE2Cpp11*. In cases where such routines are used, *DelphiXE2Cpp11* 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 DelphiXE2Cpp11 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++, 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;
    }
```

Delphi2Cpp replaced the inner functions by new member functions and tried to pass all necessary variables and constants to that function.There were difficulties with result variables and with undeclared types at the declaration of the inner functions however. At the current solution no variables etc. have to be passed explicitly, because *[&]* captures the environment by reference.The drawback of the current solution is, that it doesn't work, if sn inner function is called reflexively. In such cases a manually post-processing in necessary.

7.8.6.1 Reflexive nested routines

Sometimes a nested function calls itself. An example is the nested routine *AppendFormat* inside of *DateTimeToString* in *System.SysUtils*. A translation produces the lambda function:

```
  auto AppendFormat = [&](PChar Format) -> void
```

```
{
  ...
  AppendFormat(ustr2pwchar(AFormatSettings.ShortDateFormat));
```

Here 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". This can be fixed manually very easily:

```
std::function<void(PChar Format)> AppendFormat;

AppendFormat = [&](PChar Format) -> void
{
  ...
  AppendFormat(ustr2pwchar(AFormatSettings.ShortDateFormat));
```

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 *DelphiXE2Cpp11*. 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 DelphiXE2Cpp11 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. DelphiXE2Cpp11 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 *DelphiXE2Cpp11*.

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, DelphiXE2C++11 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 DelphiXE2Cpp11 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 was read, the "ReadProperty" function is used and the "WriteProperty" function is called in all places, where originally a value was assigned to a property.

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

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;
}
```

DelphiXE2Cpp11 can produce both code variants, depending on the option to *Use "stop" variable in for-loop* or not..

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.

7.11.2 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. *DelphiXE2Cpp11* 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.`

7.11.3 with-statements

In C++ there are no with-statements. Therefore DelphiXE2Cpp11 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
    if Month = 12 then
    begin
      Month := 1;
      Year := Year + 1;
    end
    else
      Month := Month + 1;
end;
```

```
{
  /*# with OrderDate do */
  {
    auto& with0 = OrderDate;
    if(with0.Month == 12)
    {
      with0.Month = 1;
      with0.Year = with0.Year + 1;
    }
    else
     with0.Month = with0.Month + 1;
  }
}
```

7.11.4 Initialization/Finalization

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

```
initialization

pTest := CTest.Create;

finalization

pTest.Free();
```

->

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

void Tests_finalization()
{
  delete pTest;
}

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

7.12 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 DelphiXE2Cpp11 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::TClass 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 DelphiXE2Cpp11 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 DelphiXE2Cpp11. 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. The exact definition of the ClassRef class is tricky and works only, because DelphiXE2Cpp11 also inserts some additional helper code into every class declaration. The following code demonstrates how a small class factory using class references is converted from Delphi to C++:

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

 TBaseClass = class of TBase;

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

 TDerivedClass = class of TDerived;


implementation

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

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


->


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<TDerived1> ClassRefType;
  ClassRefType* ClassType() const {return  System::class_id<TDerived1>();}
  TDerived1* Create() {return new TDerived1();}
  static TDerived1* SCreate() {return new TDerived1();}
  System::String ClassName() {return L"TDerived1";}
  static System::String SClassName() {return L"TDerived1";}
  TDerived1();
};

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.

7.13 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).

DelphiXE2Cpp11 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 DelphiXE2Cpp11 the old Delphi2Cpp used a similar solution from Tamas Demjen :

http://tweakbits.com/articles/events/index.html

7.17 Libraries

DelphiXE2Cpp11 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. DelphiXE2Cpp11 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

.

8.1 Unicode

DelphiXE2Cpp11 is able to process Delphi files which uses non ANSI characters for identifiers or in comments. For example:

```
unit Unicode;

interface


(* DelphiXE2Cpp11 □  □ Unicode *)

type

  □□ = record
    □□: string;
    □□ : string;
  end;


implementation

   //□ □□ □ □□ (xìngzhì)
   procedure□ (A□ □ : □□ );
  begin
     WriteLn(A□ .□□ );
     WriteLn(A□ .□□ );
  end;



end.
```

becomes to:

```
/* DelphiXE2Cpp11 □  □ Unicode */

struct □□
{
   System::String □□ ;
   System::String □□ ;
};

---------


   //□ □□ □ □□ (xìngzhì)

void □ (□□ A□□ )
{
   WriteLn(A□ □ .□□ );
   WriteLn(A□ □ .□□ );
}
```

8.2 Unit scope names

DelphiXE2Cpp11 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*. DelphiXE2Cpp11 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 () cor`
`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;
IntDivide(a: type; b:
type): resultType;
Modulus(a: type; b:
type): resultType;
LeftShift(a: type; b:
type): resultType;
RightShift(a: type; b:
type): resultType;
LogicalAnd(a: type; b:
type): resultType;
LogicalOr(a: type; b:
type): resultType;
LogicalXor(a: type; b:
type): resultType;
BitwiseAnd(a: type; b:
type): resultType;
BitwiseOr(a: type; b:
type): resultType;
BitwiseXor(a: type; b:
type): resultType;
```

```
/          friend resultType operator / (const type A, const type B);
div        friend resultType operator / (const type A, const type B);
mod        friend resultType operator % (const type A, const type B);
shl        friend resultType operator << (const type A, const type B);
shr        friend resultType operator >> (const type A, const type B);
and        friend resultType operator && (const type A, bool B);
or         friend resultType operator || (const type A, bool B);
xor        friend resultType operator XOR (const type A, bool B); // c
and        friend resultType operator & (const type A, bool B);
or         friend resultType operator | (const type A, bool B);
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++. DelphiXE2Cpp11 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;
```

DelphiXE2Cpp11 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, DelphiXE2Cpp11 creates explicit calls to the member function. For example:

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

becomes to:

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

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

8.4 Class helpers

There is no counterpart to class/record helpers in C++. However it is possible to translate Delphi code using class helpers to C++. This 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 DelphiXE2Cpp11 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;
```

DelphiXE2Cpp11 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 DelphiXE2Cpp11 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

DelphiXE2Cpp11 cannot distinguish a generic type and a normal type with the same name in the same unit. There are such cases in System-pas. E.g.

```
IEnumerator = interface(IInterface)
IEnumerator<T> = interface(IEnumerator)
```

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
{
  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);}
public:
  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.

9 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
Inline assembler code isn't converted
const-correctness is an important concept in C++, but hardly supported in Delphi
API functions often are specified too vaguely in Delphi

9.1 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 **professional version** of *Delphi2Cpp*, there is a minimalistic option to convert Delphi comments and Delphi expressions and to substitute identifiers. The option wasn't taken over here to *DelphiXE2Cpp11*, because it is of little use and because in the actual Delphi *RTL* the definition of *PUREPASCAL* can be set, to avoid the use of assembler code at all,

9.2 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 DelphiXE2Cpp11, 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 DelphiXE2Cpp11 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.

9.3 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 thirs 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;
```

DelphiXE2Cpp11 converts a Cardinal to unsigned int. But it's not possible to assign an unsigned int* to PDWORD or to LPDWORD in C++,

10 What is not translated

There are some principle problems - listed below - at the conversion of Delphi code to C++ which cannot be resolved by an automatic translator. But even things which DelphiXE2C++11, normally can handle may fail in complex nested cases. Sometimes Delphi2C# generates explicit "todo"-comments where something has to be completed manually.

Some Delphi constructs, which aren't, automatically translated yet are:

- Inline assembler code in Delphi and C++ almost are identically. DelphiXE2Cpp11 doesn't translate these parts.but only copies them.
- *Delphi2C#* always assumes unique names.But e.g. there might be symbols from the operation system, which differ in notation..
- Some problems with constructors remain. E.g. *DelphiXE2Cpp11* cannot distinguish constructors with equal signatures.
- Manual post-processing to achieve const-correctness is necessary.
- 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.

- Little effort has been done to test the COM technologies of the Delphi ActiveX framework..

Special problems:

Reflexive nested routines
lifetime extension of bound variables
Constructors and destructors may not call virtual methods of derived classes
https://isocpp.org/wiki/faq/strange-inheritance#calling-virtuals-from-ctors

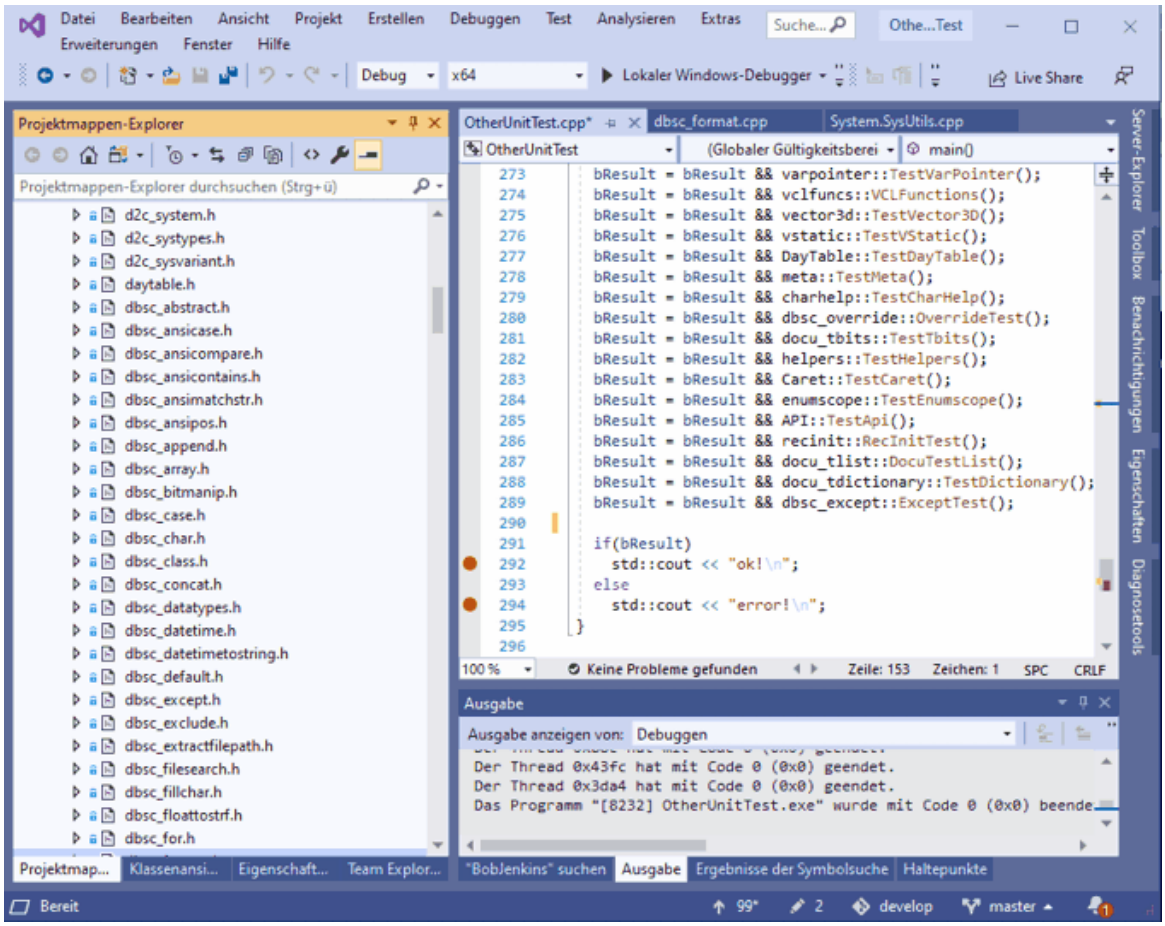
11 Unit tests

The quality of the translation results of Delphi code to C++ with DelphiXE2Cpp11 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 DelphiXE2Cpp11 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++.



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

11.1 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 DelphiXE2Cpp11 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 DelphiXE2Cpp11 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;
}
```

11.2 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 DelphiXE2Cpp11 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.

11.3 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
```

12 Pretranslated C++ code

DelphiXE2Cpp11 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.

12.1 Delphi RTL/VCL

User Delphi code is based on the Delphi RTL and the VCL How about the C++ version of these files?

C++ Builder

The C++ Builder already has its own version of the Delphi RTL/VCL with C++ interface files. *DelphiXE2Cpp11* 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 *DelphiXE2Cpp11* 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 API functions and types of the operating system such that they are conform to Delphi. In C++ this conversion isn't necessary. Also some parts of System.pas aren't needed in C++.

Therefore some parts of the Delphi RTL are pre-translated and prepared to use with the code translated by *DelphiXE2Cpp11*. Because Embarcadero has the copyright of the Delphi RTL/VCL the translated parts cannot be shipped with the *DelphiXE2Cpp11* installer. However as customer of *DelphiXE2Cpp11* you certainly will have a license of Delphi too and as owner you also have the right to use the translated code. So you can get the C++ version of the Delphi code, if you send a prove of the Delphi ownership to me or simply send the Delphi RTL code to me.

Some helping code is already delivered with the *DelphiXE2Cpp11* installer:
It is recommended to prepare some files of the Delphi RTL

12.1.1 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 DelphiXE2Cpp11 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:

..\DelphiXE2Cpp11\Source\C++Builder

You also should use the extended *System.pas* extension in

d2c_config
d2c_convert
d2c_smallstringconvert
d2c_sysexcept

d2c_sysfile
d2c_sysinit
d2c_sysiter
d2c_sysmath
d2c_sysmem
d2c_sysobj
d2c_sysopenarr
d2c_sysstring
d2c_system
d2c_systypes

12.1.2 C++ code for other compilers

The code for Visual C++ in "..\DelphiXE2Cpp11\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_meta
d2c_openarray
d2c_smallstring
d2c_smallstringconvert
d2c_sysconst
d2c_syscurr
d2c_sysdate
d2c_sysdynarr
d2c_sysfile
d2c_sysiter
d2c_sysmac
d2c_sysmath
d2c_sysmem
d2c_sysstring
d2c_system
d2c_systypes
d2c_sysvariant
DelphiSets
OnLeavingScope

The complete code also contains:

d2c_sysmarshal
d2c_sysmonitor

and an rtl-folder with the pre-translated files of the Delphi RTL

12.1.3 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 DelphiXE2Cpp11 with valid Delphi license get the ready prepared pas-files together with the pre-translated RTL files.

System.pas

Because DelphiXE2Cpp11 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 DelphiXE2Cpp11 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 DelphiXE2Cpp11 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. DelphiXE2Cpp11 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 DelphiXE2Cpp111 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 OAIdl.h. A C++ application should use the original Windows types..Until now DelphiXE2Cpp11 offers no special support for the conversion of Delphi code using Variants etc. However DelphiXE2Cpp11 supports TVarRec. Advice from users is welcome.

12.2 Case study: SysUtils

Beneath System.pas System.SysUtils.pas is the most important unit of the Delphi RTL. It consists in about 34739 lines of code (RAD Studio 10.2.2 Tokyo). After applying the following definitions:

```
[DEFINITIONS]
CPUX64=1
D2C=1
MSWINDOWS=1
PUREPASCAL=1
WIN64=1
```

all special code parts for 32 bit, non-windows systems and assembler are removed and only 21670 lines of code remain.

The generated C++ header file consists in 14000 lines of code with 66 differences to the manually post-processed file, the generated C++ source file consists in 21224 lines of code with 482 differences. Differences are shown in the text comparing program *WinMerge* and often cover more than one line of code.

More exactly

Header:

16 additional friend declaration(-groups)
19 changes for const-correctness
4 std::atomic variables
7 concerning the API
3 concerning constants

12.2.1 Example: CP_ACP

CP_ACP is defined in *WinNls.h*

```
#define CP_ACP                    0           // default to ANSI code page
```

For Delphi this constant has to be created too, but in a manner that is compatible also with C++Builder. Therefore are the following lines in *System.pas*:

```
(*$HPPEMIT '#if !defined(CP_ACP)' *)
(*$HPPEMIT 'static const System::Word CP_ACP = System::Word(0);' *)
(*$HPPEMIT '#endif' *)
const
  CP_ACP  = 0;
...
{$NODEFINE CP_ACP}
...
```

These lines consist in three parts:

1. With *HPPEMIT CP_ACP* is defined for C++Builder for operating systems, where *CP_ACP* isn't defined.
2. The constant is defined for Delphi
3. With *NODEFINE* the Delphi constant becomes out of scope for C++Builder.

There is a second definition of *CP_ACP* in *Winapi.Windows.pas*

```
{$EXTERNALSYM CP_ACP}
CP_ACP                   = 0;             { default to ANSI code page }
```

For Delphi, this is identical to the declaration in *System.pas*. For C++Builder the constant is made invisible again by the *EXTERNALSYM* macro.

DelphiXE2Cpp11 has to lookup both Delphi constants, to be able to translate the code where they are used correctly.Because of the *NODEFINE* and *EXTERNALSYM* macros the constants aren't written into the generated C++ code. (The unit *Winapi.Windows* isn't used in the C++ code at all.) Only the *HPPEMIT* part is written into the generated System.h:

```
#if !defined(CP_ACP)
static const System::Word CP_ACP = System::Word(0);
#endif
```

Under Windows however, simply the definition in *WinNls.h* is used. (Like *Winapi.Windows* also the generated translations of the *System* unit aren't used in translated Delphi projects. There ready to use prepared files instead.)

12.3 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

12.3.1 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 DelphiXE2Cpp11 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 DelphiXE2Cpp11 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
```

12.3.2 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 *DelphiXE2Cpp11* offers the possibility to prepare the Delphi source code such, that *DelphiXE2Cpp11* 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 *DelphiXE2Cpp11* 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.

12.3.2.1 Comments (*#_ ... _#*)

DelphiXE2Cpp11 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 old 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;
```

12.3.2.2 Predefined identifier Cpp

In addition the the definitions which the user can set in the translation options the identifier *CPP* always is defined in *DelphiXE2Cpp11*. 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;

DelphiXE2Cpp11 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.

12.3.3 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 DelphiXE2Cpp11 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.

12.3.3.1 $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'}
```

12.3.3.2 $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.

DelphiXE2Cpp11 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}
```

12.3.3.3 $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}
```

12.3.3.4 $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.
```

12.3.3.5 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.)

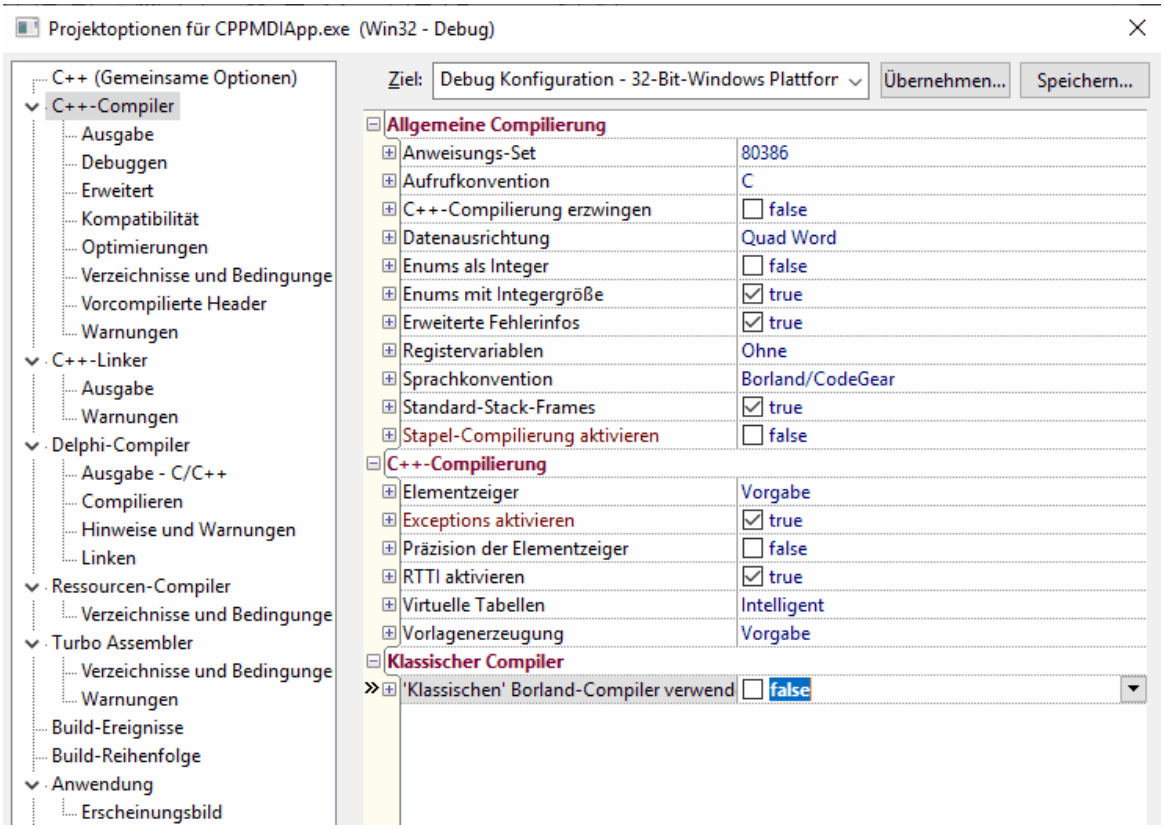
13 Delphi projects

DelphiXE2Cpp11 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.

13.1 Clang

Enter topic text here.



13.2 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 DelphiXE2Cpp11 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;
}
//---------------------------------------------------------------------------
```

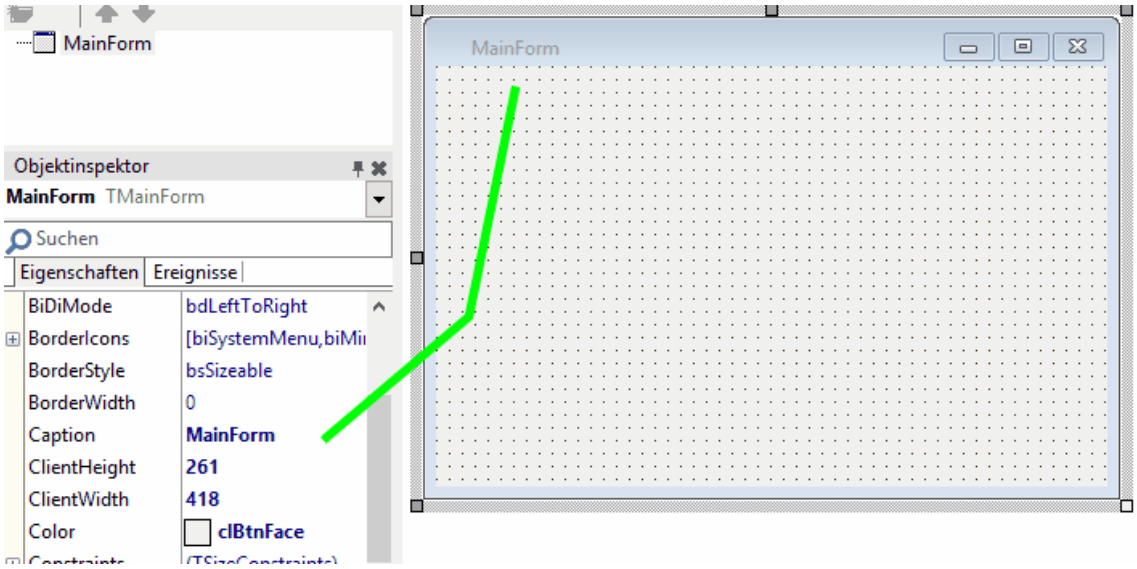
13.3 dfm Files

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.

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.



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":

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 DelphiXE2Cpp11 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.

14 Formatting

The generated C++ code should be readable, but little effort was made to make it beautiful. There are free pretty-printers available, which have a lot of options to format the code just as you like it. I recommend:

http://universalindent.sourceforge.net/

With UniversalIndentGUI "you change the value of a parameter and directly see how your reformatted code will look like. Save your beauty looking code or create an anywhere usable batch/shell script to reformat whole directories or just one file even out of the editor of your choice that supports external tool calls."

15 DelphiXE2Cpp11 versus Delphi2Cpp

DelphiXE2Cpp11 is based on the experiences with the previous program *Delphi2Cpp*, which translates Delphi 7 code only.

Delphi2Cpp supports old style programs based on AnsiChars and AnsiStrings as well as programs based on WideChar and UnicodeStrings.
DelphiXE2Cpp11 produces Unicode based code only and presumes 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:

DelphiXE2Cpp11 processes the Delphi language expansions which were added since Delphi 7

DelphiXE2Cpp11 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 DelphiXE2Cpp11 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 Delphi2Cpp.
- there is an option to create class reference types by which classes can be created.

Classes.pas TList

16 TextTransformer

DelphiXE2Cpp11 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

17 Service

There is also a service to make translations of Delphi source code for you. So you don't have to buy the program:

http://www.texttransformer.com/D2C_TranslationService_en.html

or in German at:

http://www.texttransformer.de/D2C_TranslationService_ge.html

I also like make extensions of DelphiXE2Cpp11 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

- B -

- C -

- D -

- E -

- F -

- G -

- H -

- I -

- K -

- L -

- M -

- N -

- O -

- P -

- R -

- S -

- T -

- U -

- V -

- W -

- X -

- Z -