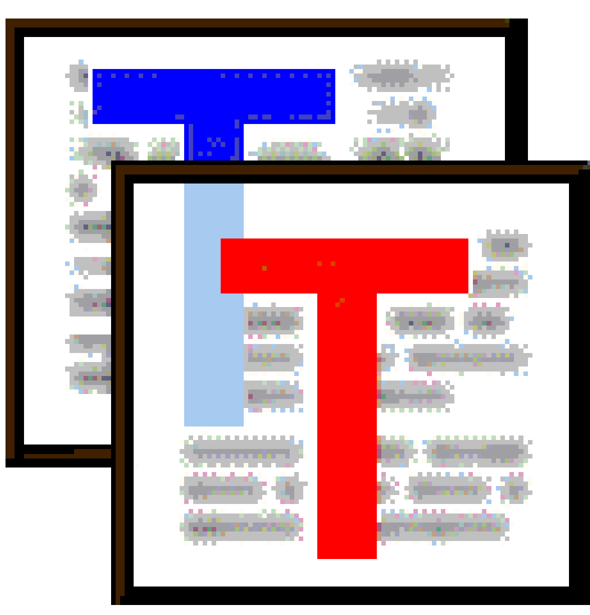
Delphi2Cpp

© 2012 Dr. Detlef Meyer-Eltz



1 Introduction

Short description

Delphi2Cpp helps to convert Delphi 4/5 source code to C++. In many cases a manual post-processing of the produced code will be required. However, it is aim of the program to keep the amount of the post-processing as small as possible.

Availability

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

http://www.TextTransformer.com

http://www.TextTransformer.de

2 Editions

There is a student - formerly called standard - version and a professional version of Delphi2Cpp. A summarizing survey of the differences is the following table.

Delphi2Cpp	**student**	**professional**
Preprocessor	+	+
Translator	+	+
Use as command line tool	-	+
Separated preprocessing and translation	-	+
File manager	-	+
Substitution of identifiers afterwards	-	+
Possibility to prepare the Delphi code	-	+
Runtime class information	-	+
Creation of C-Code	-	+
Definition of prefixes for properties	-	+
Creation of namespaces	-	+
Overwriting "System.pas"	-	+
Processing inline assembler	-	+
Meta capabilities with own TObject	-	+

+ : feature exists
- : feature is not available

An overview of the provided source code is here.

© 2012 Dr. Detlef Meyer-Eltz

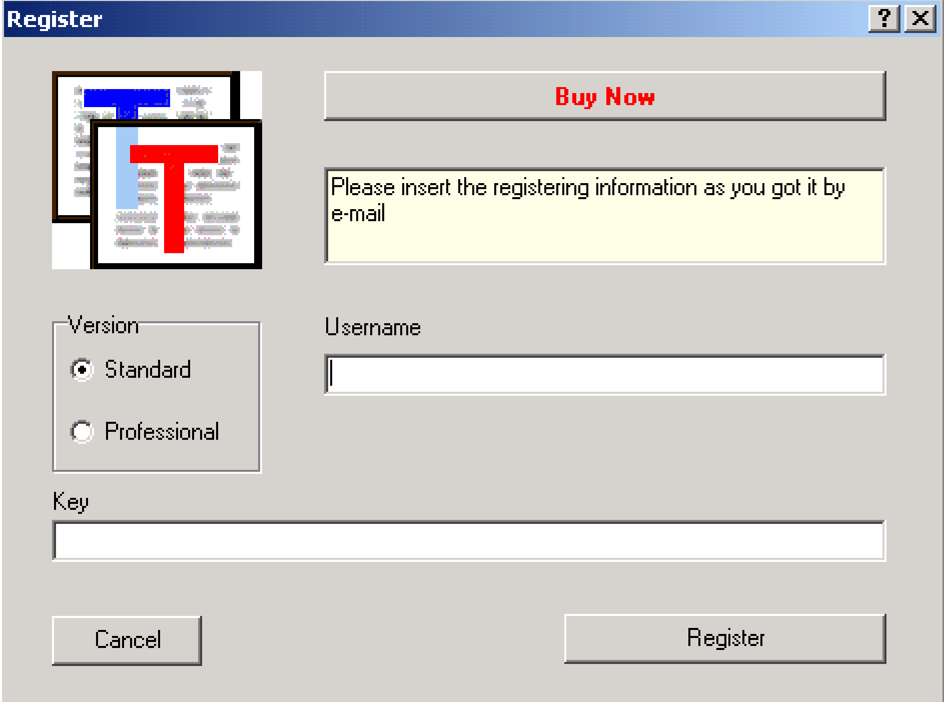
3 Installation

The installation is done by the installer Delphi2CppInstall.exe. All files are copied into the chosen installation directory.
Delphi2Cpp doesn't need any additional files like dll's etc.

4 Registration

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

The **registration** of Delphi2Cpp, i.e. the permanent activation of the features, can be done either in the little dialog, which is shown, when the trial version is started or inside of the running program by the menu: Help->*Registration*. The last case is explained here in more detail.



For the registration of the **Standard version** you must transmit a **user name** (at least eight characters) and your address details and the details on the method of payment. For the registration of the **Professional Version** in addition a **program ID** (see below) is required.

Forms for the corresponding inputs are displayed in your Internet browser, if you are on line and click the **Buy Now** button.

© 2012 Dr. Detlef Meyer-Eltz

After the check of your credit card has been carried out, an **e-mail** which includes the registration data (user name and key) is sent to you automatically. These must be assigned to the appropriate fields of the dialog box shown above. But first select, whether a registration of the **standard or the professional version** shall be carried out. **User name** and the **key** then have to be copied unchanged from the e-mail into the corresponding entry fields of the dialog box.
Then the **button *Register*** will close the dialog automatically and a message appears, which confirms the success of the registration.

Program ID for the registration of the Professional version

The program ID that is required for the registration of the Professional version is shown as soon as you select the button Professional in the dialog box. An additional field appears with a combination of numbers and letters.

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. The registered professional version can be executed only on the computer on which it originally was installed.

It is important to know that if you are downloading Delphi2Cpp to use the Professional version 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.
(For the standard version there is no such restriction. You can arbitrarily transfer it.)

Upgrade to Professional Version

If you have registered the Standard Version of the TextTransformers, in the dialog appears a button, by which you can upgrade your license to the Professional Version, instead of the **Buy Now** button

Upgrade to Professional Version.

5 How to start

You will get good C++ translations of your Delphi code only, if you make the correct settings in dialog for the translation options, which can be shown by the button . There are two main decisions to make.

1. C++ Builder or other compiler

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

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

Delphi2Cpp has to know the types and signatures of procedures and functions in your Delphi source code to make a correct translation. That's no problem as far as these information stems from your own source code. You simply have to set the paths to your source code at the according place in the options dialog.
But all Delphi code implicitly also includes the *System* unit and most Delphi code uses at least the *Sysutils* unit too. Already translated C++ code for these both units is part of the Delphi2Cpp installation. In the same folder there are pas-files with the Delphi interface parts of these units. If no other units from the Delphi RTL/VCL are used in your code, you will get the best translation results, if you select the path to these pas-files as search path for the files not to convert. Mostly your code will depend on more units of the Delphi RTL/VCL. In this case you should select the paths to the original Delphi RTL/VCL. If there is more than one edition of the Delphi RTL/VCL installed on your computer, than please choose the oldest version (beginning from Delphi 4), because Delphi2Cpp might have difficulties to parse more recent Delphi language extensions.
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. 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 .

© 2012 Dr. Detlef Meyer-Eltz

6 User interface

The main window of the Delphi2Cpp application consists in a menu, a tool bar and in three windows for the input and for the output.

--

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

--

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

--

By this button the conversion of a Delphi dpr project file to a translated unit and a C++Builder project file is started. In addition the project options can be supplemented with the paths to the found source files and a management for the conversion of the whole project can be created.

--

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 include paths to the directories with the preprocessed VCL and you also select your preprocessed Delphi sources, the enabling of the translator suffices for the fastest possible translation. 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 can be saved with the options.
The *overwritten System.pas* gets always preprocessed, even if the button is disabled.

--

The translation is started with F9 or

--

In the professional version of Delphi2Cpp 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.

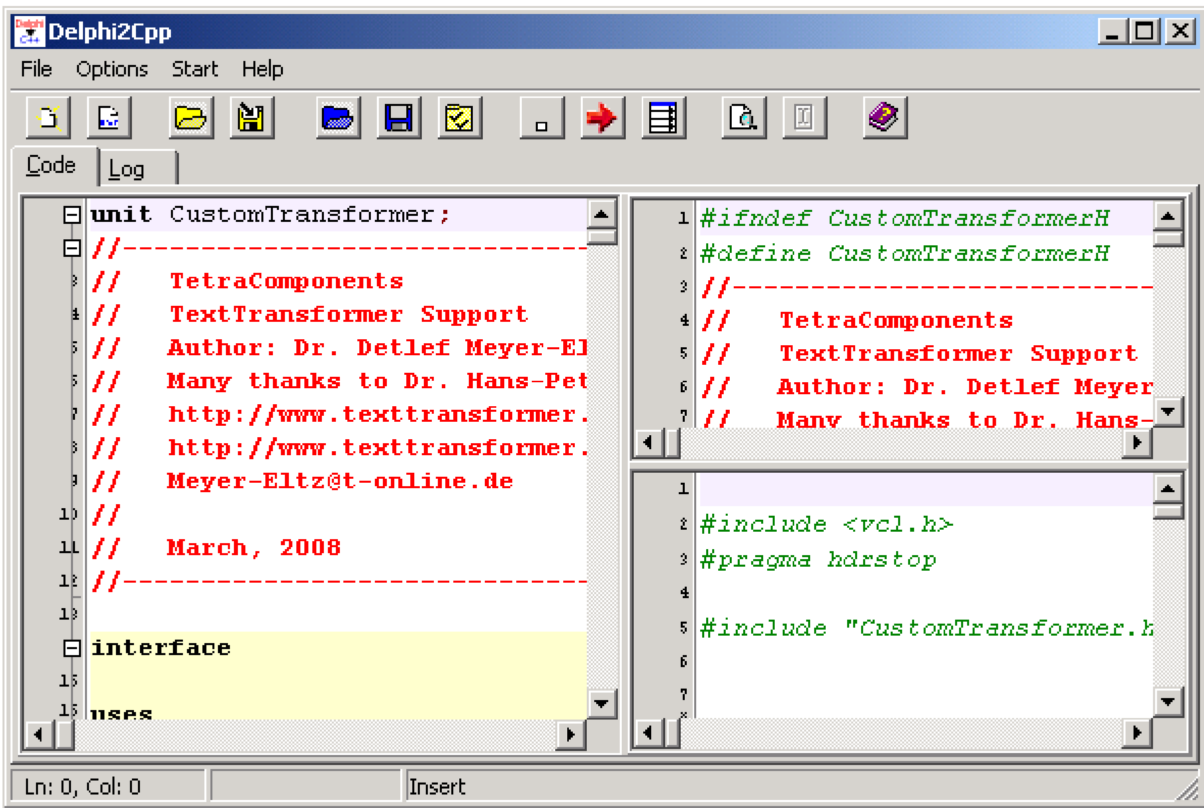
--

This help is shown with F1 or by the button

6.1 Windows

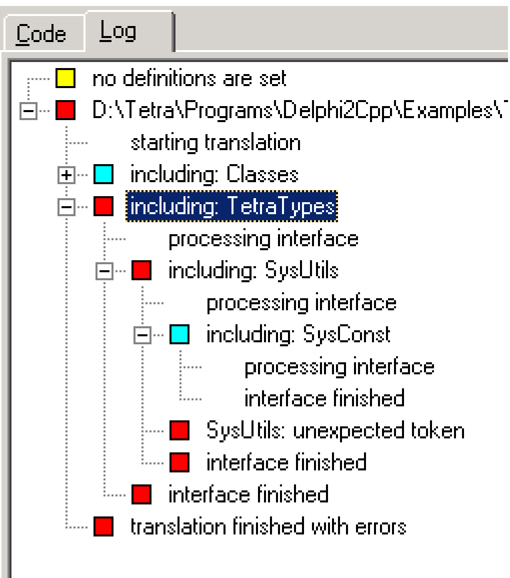
There are three windows in the user interface:

1. the left window shows the Delphi source code
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



6.2 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.

In the **professional version** of Delphi2Cpp *SysUtils* can be opened by a double click on the node with the error message. In the standard version you can open the file as normal. If the translation of *SysUtils.pas* is started, it stops at:

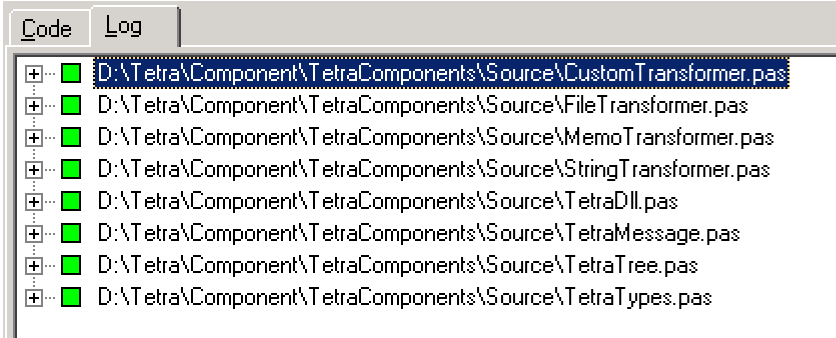
```
function AdjustLineBreaks(const S: string; Style: TTextLineBreakStyle =
            ): string;
```

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* is defined, there is no value assigned to *TTextLineBreakStyle* in the result of the preprocessor.

In the professional version the results of all files are listed in the tree:

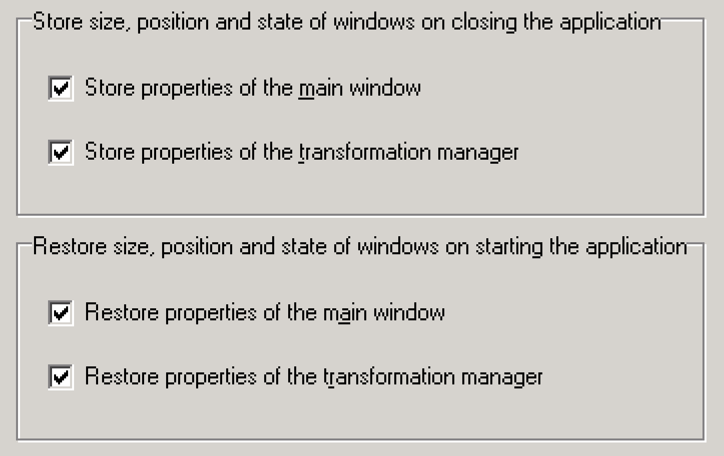


6.3 User options

User options can be accessed in the Options menu at the item "Show user options". These options are saved in the Windows registry and thus persist between different sessions with Delphi2Cpp.

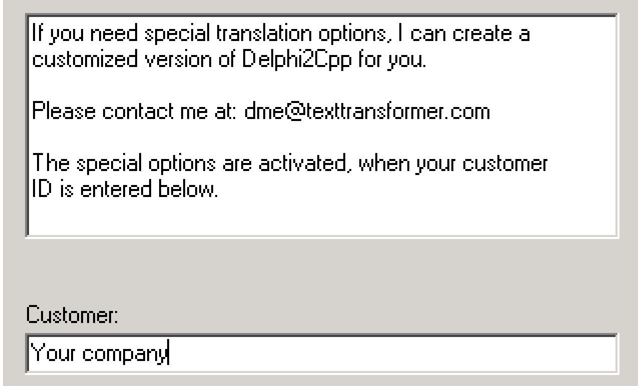
Window positions
Customization

6.3.1 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 Delphi2Cpp always have the properties that were stored, even if they were change in the previous session.

6.3.2 Customization



6.4 Translation options

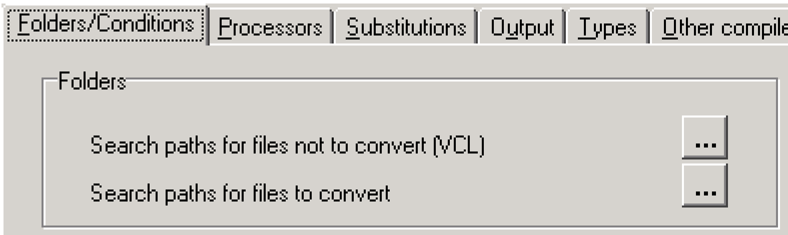
In the options dialog you can set

- Include paths
- System.pas
- Definitions
- Processors
- Substitutions of identifiers
- Types
- Output
- Other compiler
- Default options

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

6.4.1 Include paths

Per default all files are scanned for type information, constants and for global variables which are included in the source file and which are placed in the same folder as the actual Delphi source file. You can select more include directories in the options dialog.

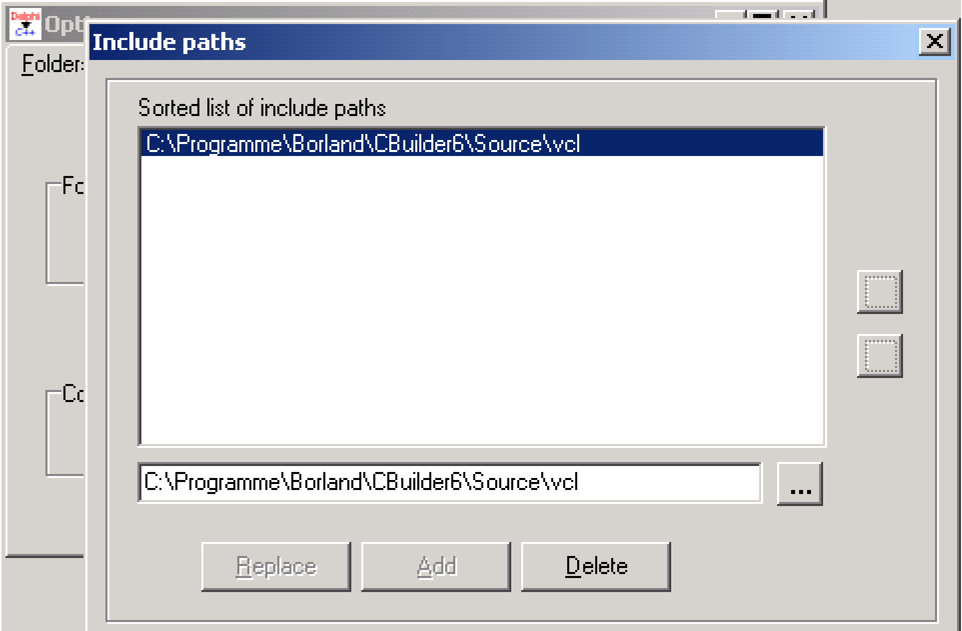


These directories are separated into

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

This difference is really relevant only, though, when the the latter ones shall be found out automatically or if a management shall be created automatically. The files, which shall be translated can be selected also manually and the paths to those files too. That's even recommended, because the automatic process is fault-prone.

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



In addition to the both kinds of files just mentioned, a special path to an own *System.pas* can be set.

Please also notice, that there are some special treatments of other special files.

6.4.1.1 Paths to the VCL

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 as search paths in the project options.

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

The paths of the 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
```

© 2012 Dr. Detlef Meyer-Eltz

6.4.1.2 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. Delphi2Cpp - as Delphi - doesn't make a recursive lookup in the folders. So sub-folders have to be set explicitly too. However Delphi2Cpp can try to lookup all these sub-folders once automatically, if you select the according options at the conversion of a Delphi dpr-file .

6.4.1.3 Special VCL headers

Delphi2Cpp tries to parse ***System.pas*** always in addition to the other included files. *System.pas* contains the declaration of *TObject* and many other frequently used functions, procedures, records and classes.There are some workarounds in Delphi2Cpp to parse more recent versions of *System. pas,* which uses features, which don't exist in Delphi 4/5, but not all versions of *System.pas* were available for tests.
If *System.pas* cannot be found in the specified include paths, a part of the content of this file is simulated.
In the professional version of Delphi2Cpp 1.4.0 and later you can include your *own extended System. pas*.

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

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

The unit ***BDE*** is used in database units, but there is no *BDE.pas*. The Delphi compiler doesn't need this file because there is a *BDE.dcu* . The interface is declared in the file *BDE.int* instead.*Delphi2Cpp* also will look for *BDE.int* in the include paths. The folder for this file has to be set there, e.g. CBuilder6/ 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 CBuilder 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*.

Like it was mentioned for *System.pas* above Delphi2Cpp also cannot really parse the file ***SyncObjs. pas*** in more recent versions of the VCL. There are workarounds in Delphi2Cpp to parse the version of the BDS 2006.

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

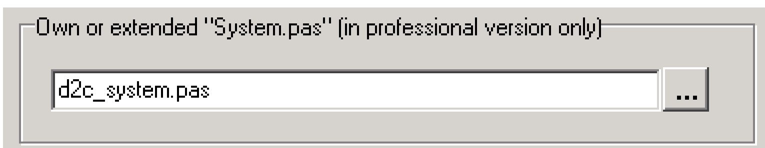
6.4.2 Extended "System.pas"

"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*

© 2012 Dr. Detlef Meyer-Eltz

function, which are accessible in each unit too. These intrinsic function are built into the Delphi compiler. *Delphi2Cpp* must know the signatures of such intrinsic functions and tries to find them in the *System.pas*. So the original incomplete System.pas either has to be replaced by an extended copy or a the original *System.pas* has to be supplemented by an additional source file.

In the options dialog you can set the name of such an additional *System.pas* extension file. However, if you use a complete replacement of the *System.pas*, as in the case, that you work with the pre-translated code, leave this field empty.



Such an individual *System.pas* called *d2c_system.pas* is in the *Source* folder of the *Delphi2Cpp* installation.

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 do so is disabled for all other files.

Lookup algorithm

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

1. *Delphi2Cpp* will look for declarations at first in your own *System.pas*, if it exists.
2. If the declaration is not found there, *Delphi2Cpp* will look in the System.pas of your Delphi installation, if the path to this file is set in the options.
3. If neither an own System.pas exists nor the path to the original *System.pas* is set, *Delphi2Cpp* simulates the most important parts of this file.

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

d2c_system.pas makes a lot of use of the (*_ ... _*) comments to insert pre-translated C++ code. The use of the predefined identifier Cpp normally isn't suitable for the overwritten *System.pas*, since the "{$else}"-branch with the Delphi declarations is removed from the preprocessor before the translator

© 2012 Dr. Detlef Meyer-Eltz

can read them.

6.4.2.1 SetString

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

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

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

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

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

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

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

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

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

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

The translation with Delphi2Cpp results in:

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

6.4.2.2 Memory management

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

© 2012 Dr. Detlef Meyer-Eltz

d2c_system.pas.

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

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

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

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

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

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

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

6.4.2.3 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.

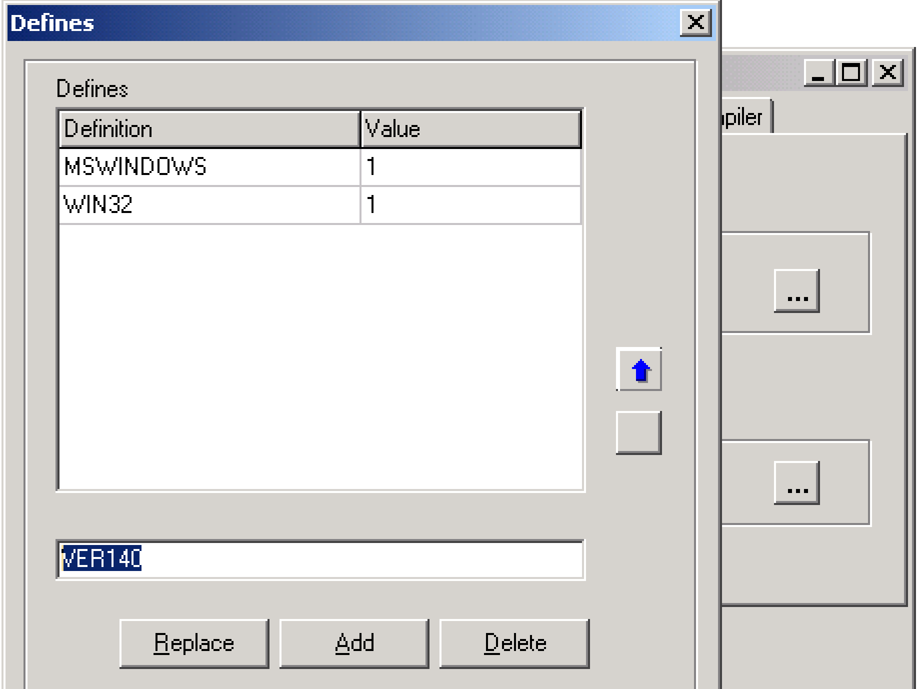
```
Inc( i )  -> i++
```

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.

© 2012 Dr. Detlef Meyer-Eltz

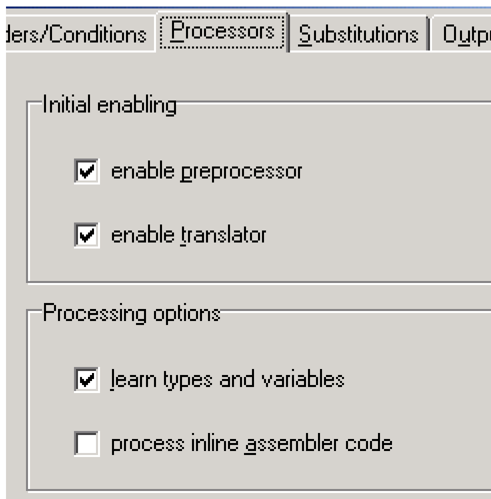
6.4.3 Definitions

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



There are limitations for the evaluation of such expressions.

6.4.4 Processors



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 to **process inline assembler** is available in the professional version only.

6.4.5 Substitutions

Substitution tables for identifiers can be shown and changed on the second register page of the option dialog.

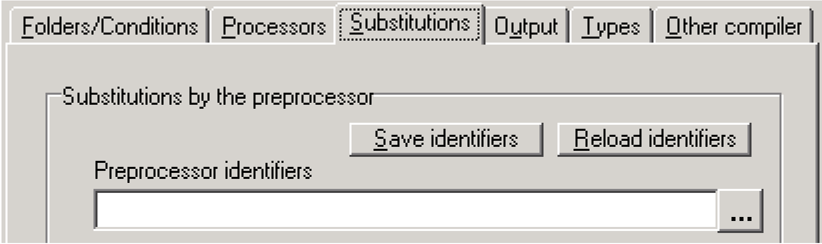
Identifiers can be changed both,

- by the preprocessor
- by the translator

6.4.5.1 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.

© 2012 Dr. Detlef Meyer-Eltz

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.

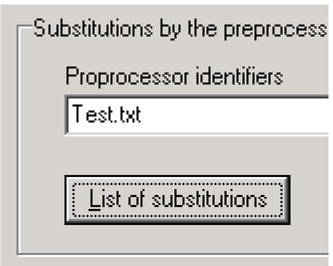
```
...
SetLength
Setscrollinfo
SetSelection
...
```

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.

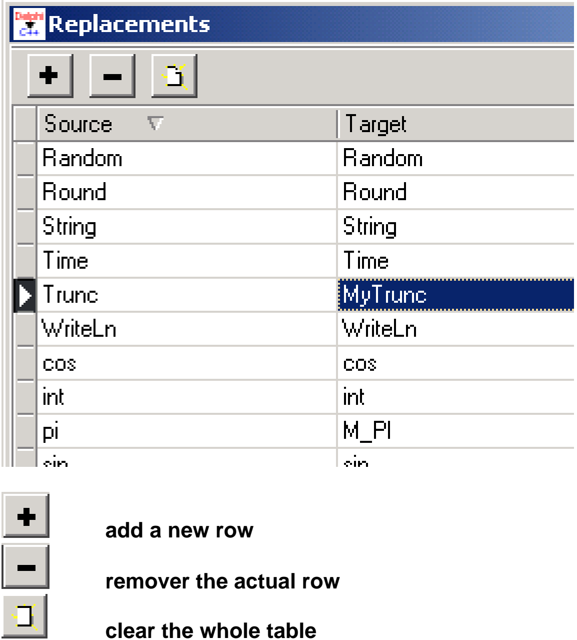
If you edit the list in an external editor, you have to reload the list by the button ***Reload identifiers***, otherwise the changes will not have an effect.

6.4.5.2 Substitutions in the preprocessor

A substitution table for the preprocessor can be shown and changed on the second register page of the option dialog.



If you click on the button "List of substitutions", the table is shown.
In the first column of the table 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.



add a new row

remover the actual row

clear the whole table

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.

at the unification of the notations, the notation found by chance first usually is used as standard for all further equivalent identifiers in the Delphi code. By means of the substitution table a certain notation can be forced.

E.g. there is a member function of TStream called "Write". This function is declared in "classes.hpp" of the CBuilder. If this function is called in the translated code, the name has to be written exactly as "Write", because C++ is case sensitive.

```
Stream.write(PChar(s)^, Length(s));
```

->

```
Stream->Write( S.c_str( ), S.Length( ) );
```

The same purpose is accomplished by use of the list of identifiers. The items of this list are overwritten by the items of the substitution table.

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.

© 2012 Dr. Detlef Meyer-Eltz

6.4.5.3 Substitutions of the translator

Similar to the substitution table for the preprocessor there is a second substitution table for the translator. This option is accessible in the professional version of Delphi2Cpp only.

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.

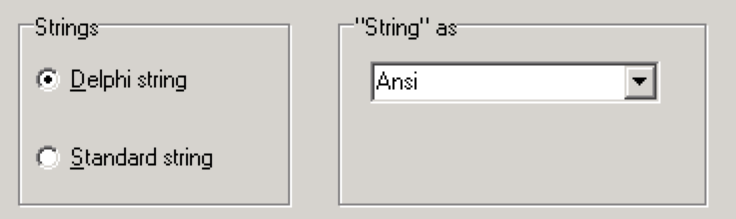
6.4.6 Types

On the *Types* page of the options dialog you can chose how the string types *AnsiString*, *WideString* and *String* are translated.

In the professional version of Delphi2Cpp there is an option to create namespaces for each unit.

6.4.6.1 String types

On the *Types* page of the options dialog you can chose how the string types *AnsiString*, *WideString* and *String* are translated.



If ***Delphi string*** is selected, the three types of strings keep their names, but if you select ***Standard string*** following replacements are carried out:

© 2012 Dr. Detlef Meyer-Eltz

```
AnsiString -> std::string
WideString -> std::wstring
String -> String  // the user has to define: typedef std::string String; or  typedef std::wstring Strin

AnsiChar -> char
WideChar -> wchar_t
Char -> Char // the user has to define: typedef char Char; or  typedef wchar_t Char;
```

Also the treatment of string parameters will be changed. E.g.

```
procedure foo1 (const s : string);
procedure foo2 (s : string);
procedure foo3 (var s : string);
```

will be translated for ***Delphi strings*** to::

```
void __fastcall foo1( const String s );
void __fastcall foo2( String s );
void __fastcall foo3( String& s );
```

For ***standard strings*** the translation is:

```
void __fastcall foo1( const String& s );
void __fastcall foo2( String s );
void __fastcall foo3( String& s );
```

If the **C-like** option is enabled, Strings are passed as pointers. For standard strings:

```
extern "C" void __fastcall foo1( const String* s );
extern "C" void __fastcall foo2( const String* s );
extern "C" void __fastcall foo3( String* s );
```

The ***""String" as*** option determines the type, which is associated with the word *String*. E.g.

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

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

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

and for the ***Wide*** association to

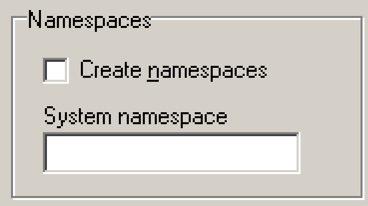
String S;
S = L"hallo";

The ***Unicode*** association is treated like the *Wide* association till now.

6.4.6.2 Creating namespaces

In the professional version there is a option, to create a namespace for each unit.

© 2012 Dr. Detlef Meyer-Eltz



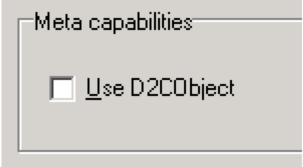
If the option is enabled in C++ header files the namespaces are put in front of types from other units and in the C++ implementation files according uses clauses are inserted.

If an extended System.pas is set, the namespace for it is "d2c_system" per default, notwithstanding of the actual file name. This expression can be changed, if another name is set in the field for the system namespace. If no extended System.pas is set, the namespace for it is "System" per default and it can be changed in the same way.

You also can use the table of the substitutions of the translator, to change the namespace for a file. So it is possible, to create the same namespace for the contents of different files.

6.4.6.3 Meta capabilities

The use of the classes *TObject/TD2CObject* and *TMetaClass/TD2CMetaClass* to reproduce Delphi's Meta capabilities can be enabled on the *Types* page of the options in the professional version of Delphi2Cpp.



These classes are defined in the files *d2c_systobj.h* and *d2c_systobj.cpp* and are part of Delphi2Cpp professional installation.
Alternatively you can use MFC-like macros

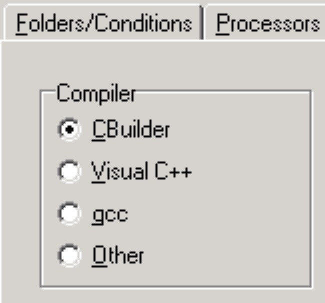
6.4.7 Output

By the *Output* options on the third register page of the options dialog you can change the kind and style of the generated C++ code.

Compiler
Precompiled header
Verbose
Special treatment of some VCL functions
C like output

6.4.7.1 Compiler

In the *Output* options you can chose the kind of c++-compiler, for which the output shall be produced.



CBuilder

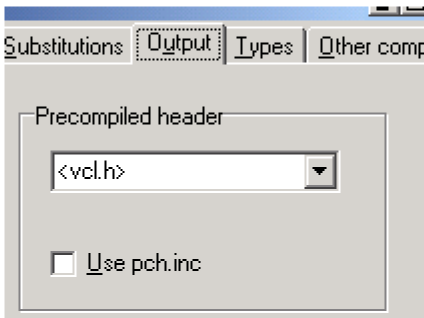
CBuilder is made on top of a Delphi-Compiler and has some C++ extensions to cope with language features of Delphi, which cannot be reproduced adequately with the standard C++. This extended C++ is the primary target of the translation made by Delphi2Cpp.

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 will be more compiler specific conversions.
If the generated C++ code shall be used with other compilers than the CBuilder, 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. It is recommended in this case to switch off the special treatment of VCL functions too and to use standard strings instead of the Delphi strings.

6.4.7.2 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.
Delphi2Cpp can insert the according marker into the generated code.



There are three options:

1. <vcl.h>

normally used with CBuilder. Delphi2Cpp also appends the line:

```
#pragma hdrstop
```

if this option is chosen.

2. "stdafx.h"

normally used with Visual C++.

3. No marker for a precompiled header at all

for other compilers like gcc.

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

6.4.7.2.1 pch.inc

If the file manager of the professional version 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 CBuilder.

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

6.4.7.3 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.
It is nor recommended to switch off this option.

6.4.7.4 Special treatment of some VCL functions

Some Delphi VCL functions are made to member functions in the CBuilder VCL.. Delphi2Cpp converts the generated C++ code accordingly for some of the frequently used function. You can switch off this special treatment and write your own C++ functions instead. This is recommended for example in the case, that you generate C++ code for another Compiler than CBuilder.

© 2012 Dr. Detlef Meyer-Eltz

6.4.7.5 C like output

This option is accessible in the professional version only.

By this option Delphi2Cpp can generate C like code: Then

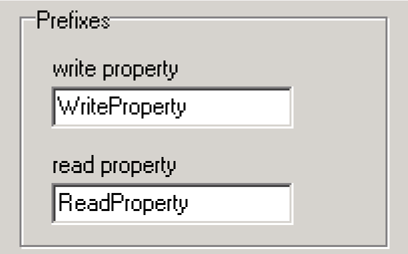
1. an *extern "C"* declaration is put in front of functions and global variables
2. instead of references of structures, their addresses are passed to functions. This also concerns String parameter: *Strings* are passed as pointers.
3. the extension of the generated source file is ".c" instead of ".cpp"

6.4.8 Other compiler

If you create C++ code for another compiler than CBuilder all properties are replaced by pairs of functions. You can change the prefixes for the function names and in the professional version the insertion of macros to access runtime class information can be activated.

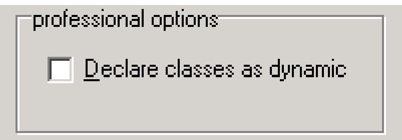
6.4.8.1 Prefixes for properties

If you create C++ code for another compiler than CBuilder all properties are replaced by pairs of functions. You can change the prefixes for the function names on the last register page of the options dialog.



6.4.8.2 runtime class information

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 or by TD2CObject. If you need runtime information and don't like to use TD2CObject, you can activate the insertion of these macros in the options dialog at the page Other compiler.



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:

MFC	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

6.4.9 Default options

Delphi2Cpp saves the current setting of project options, if the box *Default* is checked, when the dialog is closed.

The setting is saved in the project folder as

```
\Delphi2Cpp\Projects\Default.prj
```

When the project options dialog is opened for the first time after a restart of Delphi2Cpp or if the button *Reset to default* is pressed, *Default.prj* is reloaded again (and the *Default* box is unchecked). If the file *Default.prj* doesn't exist internally pre-configured options are used as default instead.

6.5 Translation

If you have a Delphi project file for the files, which shall be translated, it is recommended to start with a conversion of this dpr-file. At this conversion the same steps are executed as for other files too (see below), but in addition search paths can be found, which are needed for the translation of the other files too.

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.

© 2012 Dr. Detlef Meyer-Eltz

6.5.1 Preprocessing

A preprocessor fulfils two tasks:

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

6.5.1.1 Conditional compilation

Delphi2Cpp 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++ are evaluated.

From version 1.0.1 on 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.

6.5.1.2 Unification of notations

While Delphi code is case insensitive, C++ code is case sensitive. So different notations of identifiers have to be unified. Delphi2Cpp uses a simple approach to do that. As soon a a new identifier is recognized it is put into a table and all further notations of this identifier are replaced by the first one. After one or several files have been processed the table can be saved.

This unification is done by the preprocessor, which also is responsible for the conditional compilation.

6.5.2 Scanning dependencies

Most Delphi units depend on other units, which are included in the uses clause. Delphi2Cpp scans the included files in so far, as they are placed either in the same directory as the actual file or in a directory, which is set in the dialog for the include paths.
The translation will produce the best results if the **Delphi 4/5 VCL** is included. In this case, however,

© 2012 Dr. Detlef Meyer-Eltz

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.
It should be possible to use **newer versions of the VCL** too. But in this case the some parts of code will not be analysed correctly and the parser gets even slower..

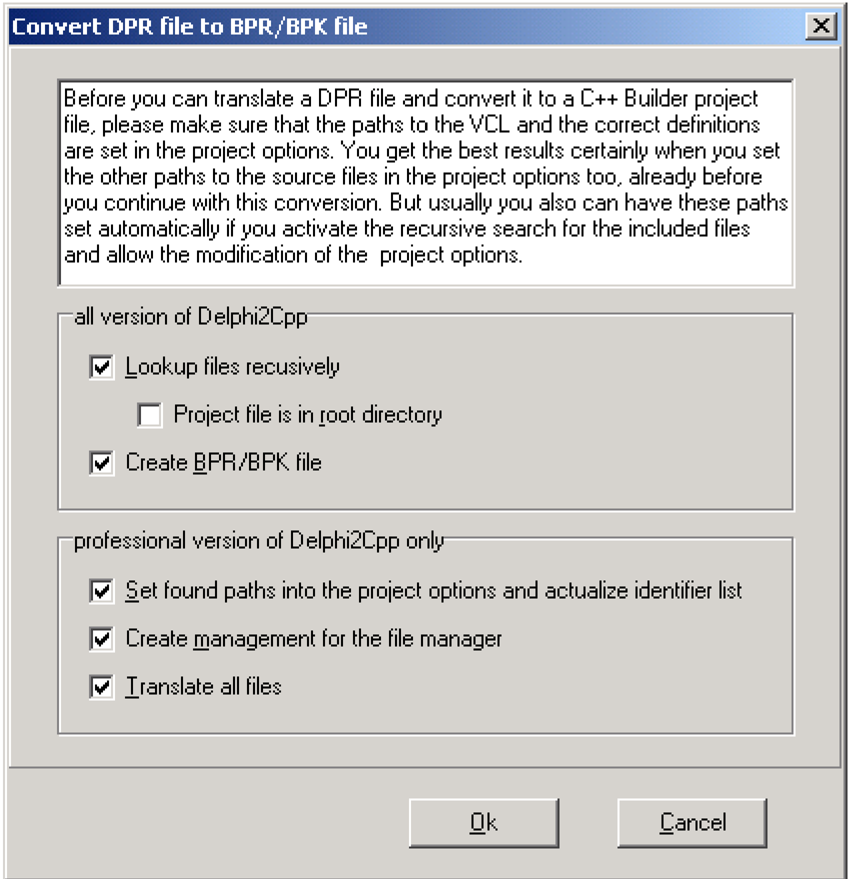
6.5.3 Writing the C++ code

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.

6.5.4 Converting dpr-files

If you have a Delphi project file (*dpr*) for the files, which shall be translated, it is recommended to start with a conversion of this dpr-file. At this conversion the same steps are executed as for other files too, but in addition the project options can be supplemented with the paths to the found source files and a management for the conversion of the whole project can be created.The conversion of the Delphi project file is started by the according item in the *Start* menu or by the button

At first you can select some options now:



© 2012 Dr. Detlef Meyer-Eltz

Lookup files recursively
Project file is in root directory
Create BPR/BPK file
Set found paths into the project options
Create management for the file manager
Translate all files

6.5.4.1 Lookup files recursively

Delphi2Cpp can parse and convert Delphi *dpr*-files, but doesn't process other kinds of Delphi's newer project files like *dproj*-files, which contain the search paths for all used units. Nevertheless Delphi2Cpp can often find these paths by searching recursively for the files included in the units. Though, units of the same name then may not exist in the directory tree repeatedly.
If the project file is not in the topmost directory of this tree, you will have to select this directory.

The recursive lookup is possible only at the conversion of a dpr-file and must be enabled. The found paths can be inserted into the project options automatically then.

6.5.4.2 Project file is in root directory

If the project file selected for the conversion of Delphi dpr-files is not in the topmost directory of all directories, which contain the units of the project, you will have to select this directory.

6.5.4.3 Creating BPR/BPK files

At the conversion of Delphi dpr-files at first such a file is translated like any other Delphi source file. But, if the according option was enabled, a C++Builder *bpr* project file or a *bpk*-file is created afterwards too. For this reason there are four frame files in the source folder:

```
\Source\Application_default.bpr
\Source\Console_default.bpr
\Source\Lib_default.bpr
\Source\Package_default.bpk
```

for GUI applications, console applications, libraries and packages respectively.these frame files all contain the placeholders

%PROJECT%
%MAINSOURCE%
%INCLUDEPATH%

for the name of the project, the main source and the search paths.

Delphi2Cpp **ignores technical settings like compiler switches** at this translation. The generated C++Builder project are just for a quick start with the translated code. You will have to set your special options manually.You can adapt the frames for your own preferred C++Builder project settings. But the number and order of the placeholders must be maintained.

If you load the options into the C++Builder, it will at first complain about the **missing resource file** and then create a new one. The translated application might use **components**, which are not installed in your IDE. This will be denounced too. If the code of these components is in a sub-directory of the project, Delphi2Cpp will have added it to the generated C++Builder project, You will have to remove it and install the components instead.

6.5.4.4 Actualize project options

At the conversion of Delphi dpr-files paths to the used units are found. If wanted, these are inserted into the project options automatically. The list of identifiers is updated or created then too.

6.5.4.5 Automatic creation of managements

At the end of a conversion of a Delphi dpr file a management of all used files is created automatically, if the according option was enabled.

6.5.4.6 Translate all files

With the professional version of Delphi2Cpp all files of a selected Delphi project can be translated at once, if they are in a common directory tree. You only have to enable the according option, when you start to create a C++Builder project file. Of course you also can use this possibility, if you are working with another compiler.

6.6 File manager

The File-Manager is accessible in the professional version of Delphi2Cpp only!

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:

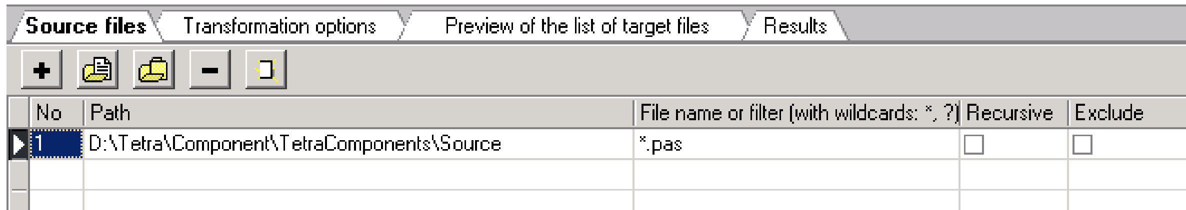
© 2012 Dr. Detlef Meyer-Eltz

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

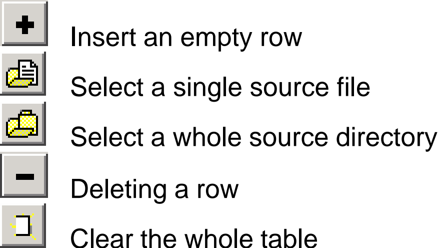
The settings, inclusive of the select folders and files, can be stored as a management and loaded when required newly.

6.6.1 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

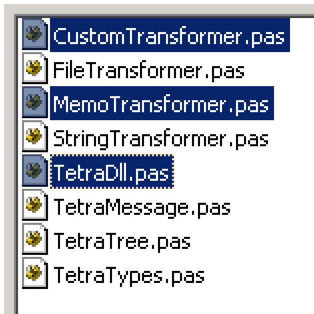
Select a single source file

Select a whole source directory

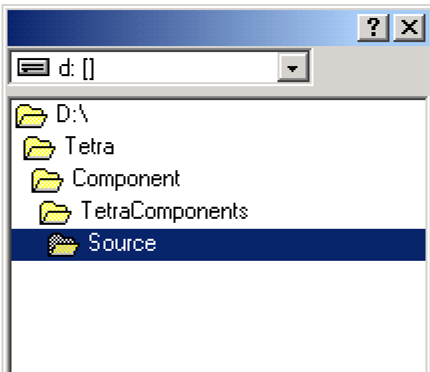
Deleting a row

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.

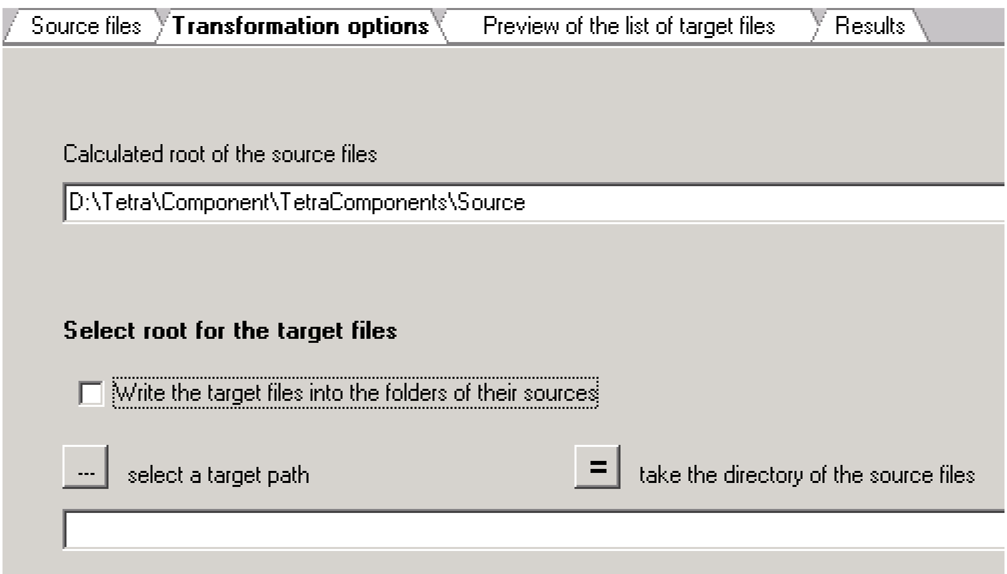
Exclude

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.

6.6.2 Translation options

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

© 2012 Dr. Detlef Meyer-Eltz



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

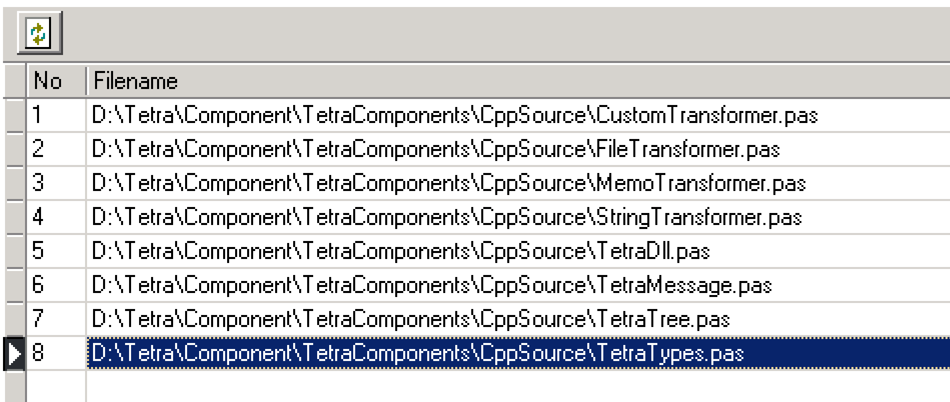
a dialog for the selection of a different target directory is opened.

The button:

can help to navigate faster to the new target directory

6.6.3 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 .

6.6.4 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

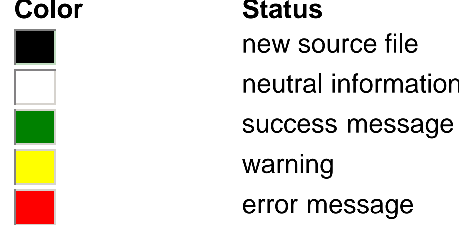
When the translations are started, the page is changed to the Results-page automatically.

6.6.5 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

6.6.6 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.

7 Use in command line mode

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

7.1 Parameter

The professional version of Delphi2Cpp.exe can be controlled either by a management, which was produced with the file manager or by parameters for the source and target files.
In the first case a call has the form:

Delphi2Cpp -p PROJECT -m MANAGEMENT

and in the second case:

Delphi2Cpp -p PROJECT -s SOURCE [-t TARGET] [-r]

Expressions in brackets are optional.
If a path contains spaces, it has to be quoted.

Parameter	Meaning	Examples
-p PROJECT	Delphi2Cpp project	cbuilder_vcl_ge.prj

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

-p PROJECT

The parameter -p must be followed by the path of the Delphi2Cpp project, with the options by which the files of the source directory shall be translated.

-m MANAGEMENT

The parameter -m is followed by the path to a Delphi2Cpp management, which specifies the source and target files.
If an -m paramerter is provided, -s, -t and -r are ignored.

-s SOURCE

The parameter -s must be followed by a specification of the files, which shall be translated.
In the simplest case this a specification is the path of a single file, like "C:\dir\source.pas". To transform all "pas" files of a directory, you can use a mask like: "C:\dir*.pas;*.dpr".
If there is no directory specified in the mask, all according files of the actually directory will be translated. If there is no special extension specified in the mask, all files of the directory will be translated. E.g.: "ab?.*" will chose all files of the directory beginning with "ab" followed by a single character, e.g. "ab1.pas", "ab2.pas" and "ab_.pas". **Attention**: in this case Delphi2Cpp will try to translate also files with other extensions than "*.pas". This will lead to errors for "*.txt" files or "*.inc"-files etc.

-t TARGET

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

-r RECURSIVE

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

-pause

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

© 2012 Dr. Detlef Meyer-Eltz

8 What is translated

The most important translation services of Delphi2Cpp are listed briefly in the following. However, the translation results aren't guaranteed for all points always, since Delphi2Cpp works with a simplified registration of type information. This system doesn't always suffice for a correct translation in individual cases.

8.1 File organization

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. The header file is enclosed into a sentinel:

```
#ifndef testH
#define testH

...

#endif
```

and the source file uses the precompiled VCL:

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

8.2 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
```

8.3 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"
```

8.4 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.

8.5 Comments

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

```
(*...*)
```

to

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

8.6 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
property	__property
:=	=
=	==
<>	!=
and	&&
boolean	bool

8.7 Simple type identifiers

Simple type names for the special C++ expansions of the CBuilders are required now and then. While e.g. the type "cardinal " usually is translated as " unsigned int", this isn't permitted in the following context:

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

Delphi2Cpp therefore produces a type definition for a simple identifier:

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

8.8 Rearrangements of expressions

Variable declarations are a typical example of a simple rearrangements:

```
Name : Typ;        ->             Typ Name;
```

A little bit more complicated rearrangements are required in instructions. E.g.

© 2012 Dr. Detlef Meyer-Eltz

```
for factor := expr1 to expr2 do
->
for ( factor = expr1; factor != expr2; factor ++ )
```

8.9 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;
```

© 2012 Dr. Detlef Meyer-Eltz

```
  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;
```

8.10 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"
```

8.11 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 "||".

© 2012 Dr. Detlef Meyer-Eltz

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 " & ".

8.12 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.	& * + - ! ~
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 )
```

8.13 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. Delphi2Cpp 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 ) );
```

8.14 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 ) );
}
```

8.15 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;
```

8.16 Calls of procedures and functions

In contrast to Delphi the calls of procedures and functions in C++ have to end with parenthesis even then, if no parameters are passed.

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

8.17 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*.

8.18 Temporary variables

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

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

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

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

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

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

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

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

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

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

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

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

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

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

This becomes to:

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

{
```

© 2012 Dr. Detlef Meyer-Eltz

```
    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.

8.19 Calls of inherited procedures and functions

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

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

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

```
inherited.foo  ->  inherited::foo()
```

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

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

->

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

8.20 Ancestors

If no ancestor type is specified when declaring a new object class, Delphi automatically uses *TObject* as the ancestor. In C++ this has to be made explicit.

```
TNewClass = class ...

->

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

8.21 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
Addition of missing constructors
Virtual constructors
Problems with constructors

8.21.1 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 (without parameters) the base class constructor has to be called in the initialization list with the according parameters.The constructors of the ancestor classes are executed in Delphi only, if they are called explicitly from in the written code. In such cases *Delphi2Cpp* tries to find this call and puts it into the initialization list:

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

->

__fastcall foo::foo ( TComponent * Owner )
: inherited   ( Owner )
{ }
```

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

Base class constructors without parameters are called automatically in C++. *Delphi2Cpp* preserves the original calls of such constructors as line comments.

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

->

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

8.21.2 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:

© 2012 Dr. Detlef Meyer-Eltz

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

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

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

```
constructor Derived.Create;
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() )
{
}
```

8.21.3 Addition of missing constructors

Unlike in Delphi, constructors of base classes cannot be called directly in C++. Additional constructor have to be defined in the derived class. Delphi2Cpp inserts missing constructors in C++ automatically. So, resuming the previous example, an additional standard constructor is created, which can be used

with all classes, which are derived from *TObject*:

```
__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
};
```

8.21.4 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 methods. 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 );
```

In side 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 *TD2CObject* class for C++Builder or the *TObject* class for other compilers is used. The files d2c_systobj.h and d2c_systobj.cpp, where these classes are defined, are part of the Delphi2Cpp professional installation.

8.21.5 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 to be initialized in at the beginning of the constructor code in the initialization list. But sometimes the value can depend on other calculations and *Delphi2Cpp* cannot recognize this.

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 )
}
```

© 2012 Dr. Detlef Meyer-Eltz

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 Delphi2Cpp 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 );
```

8.22 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 ( )
```

Delphi2Cpp tempts to find calls of destructors of the base class and to comment them out in C++. Thereby is assumed that the destructor of the base class is virtual. 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;
```

8.23 class methods

A Delphi class method can be called through a class reference or an object reference, like C++ static methods.

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

The first intuition would be a translation in the following way:

© 2012 Dr. Detlef Meyer-Eltz

```
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 = new TBase;
pBase->ClassMethod( 0 );  // calling through a object reference
```

But there are two reasons why this translation doesn't suffice.

1. 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).
2. Delphi class methods can be virtual, C++ static methods cannot.Therefore *Delphi2Cpp* has to use a tricky construction to reproduce this ability of Delphi.:

Both problems can be solved, if an additional pointer to an instance of the class is passed to the function.

non virtual class methods
virtual class methods

So the translation looks like:

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

...

TBase* pBase = NULL;
int i = 0;
TBase::ClassMethod( TBase::__tp.SObjectType(), 0 );
pBase = new TBase;
pBase->ClassMethod( pBase, 0 );
```

8.23.1 non virtual class methods

The translation of a definition of a non virtual class method looks like:

```
int __fastcall /*static*/ TBase::ClassMethod( const TBase* xpTHIS, int xi ) {
  int result = 0;
  const TBase* pTHIS = (TBase*) xpTHIS;
  ...
  return result;
}
```

pTHIS represents Delphi's class *Self*. In the C++ translation a pointer to an instance of the class is passed as parameter from outside of the method and assigned to *pTHIS*. The cast from *xpTHIS* to *pTHIS* doesn't do anything here. It's just a formal line of code, which is repeated in all class functions and which is necessary in cases, where the class function is virtual.
The code which is indicated with "..." is interpreted as an implicit with statement:

```
with pTHIS do
  ...
```

A complete example demonstrates this:

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

The explicit with statement "with Create do" is part of the implicit with statement, so that *Create* is called with *pTHIS*.

```
int __fastcall /*static*/ TBase::ClassMethod( const TBase* xpTHIS, int xi ) {
  int result = 0;
  const TBase* pTHIS = (TBase*) xpTHIS;
  /*# with Create do */
  {
    TBase* with0 = (TBase*) pTHIS->Create();  <-  implicit with
    {
      with0->Init();
      with0->Done();
      delete with0;
    }
  }
  result = xi;
  return result;
}
```

8.23.2 virtual class methods

The trick, which makes the static method virtual is, that an additional virtual method is called for the passed class pointer.

```
TDerived = class(TBase)
public
  class function ClassVirtual(xi: Integer): Integer; override;
```

becomes:

```
class TDerived: public TBase {
public:
  static int __fastcall vs_ClassVirtual( const TBase* xpTHIS, int xi ) const
  { return xpTHIS->vs_ClassVirtual(xpTHIS, xi); }
  virtual int __fastcall vs_ClassVirtual( const TBase* xpTHIS, int xi ) const;
```

The definition of the virtual class method analogous to the example of the non virtual class method demonstrates the difference:

```
class function TDerived.ClassVirtual(xi: Integer): Integer;
begin
  with Create do  <-- new object from a virtual constructor
  begin
    Init;
    Done;
    Free;
  end;
```

© 2012 Dr. Detlef Meyer-Eltz

```
  result := xi;
end;
```

Here the cast of the parameter *xpTHIS* to *pTHIS* is necessary to get the actual type. Because all overwritten virtual functions must have the same signatures, it is not possible to declare the class parameter with the type of the actual class. But as the method is called through redirection of the class pointer the typecast is guaranteed to be safe.

```
int __fastcall /*static*/ TDerived::vs_ClassVirtual( const TBase* xpTHIS, int xi ) const {
  int result = 0;
  const TDerived* pTHIS = (TDerived*) xpTHIS;
  /*# with Create do */
  {
    TDerived* with1 = (TDerived*) pTHIS->Create();
    {
      with1->Init();
      with1->Done();
      delete with1;
    }
  }
  result = xi;
  return result;
}
```

8.24 Meta information

Delphi has special capabilities to use runtime type information (RTTI) There are class methods that operate on class references instead of objects, which uses this information.

C++ Builder has special language extensions and uses the special class *TMetaClass* to copy these capabilities partly.

The most important of these capabilities can be reproduced not only with C++ Builder but with other compilers too by means of the *TObject* class and the *TMetaClass*, which are defined in the files *d2c_systobj.h and d2c_systobj.cpp* and are part of Delphi2Cpp professional installation.

Even for C++ Builder the use of the according classes *TD2CObject* and *TD2CMetaClass* is necessary, if virtual construction of classes with *Create* is needed,

The use of these classes has to be enabled in the options dialog.

TObject/TD2CObject
static type objects

8.24.1 TObject/TD2CObject

The class *TObject*, which is defined in the files *d2c_systobj.h* and *d2c_systobj.cpp* can be used with other compilers than C++ Builder as basis for a substitute of *TObject* of the Borland VCL. The class *TD2CObject* can be used with C++ Builder to enrich the capabilities of *TObject*, which already exists there. Analogously *TMetaClass* and *TD2CMetaClass* are used with other compilers or with C++ Builder.

TObject/TD2CObject and *TMetaClass/TD2CMetaClass* mainly consist of the same functions:

The methods of *TMetaClass/TD2CMetaClass* correspond to the static methods of *TObject/TD2CObject*. They have the same functionality but differ in parameters. Whereas the *TObject/TD2CObject* methods take a *TClass/TD2CClass* (*TD2CMetaClass*/TMetaClass**) as the first parameter, this parameter is implicit in the *TMetaClass/TD2CMetaClass* methods.

These methods are known from Delphi and their description can be found in the Delphi help files. There is a difference, though: name and type information result from *std::type_info* in C++ and their values are different from the values in Delphi.

```
virtual const char* __fastcall ClassName() const;
virtual bool __fastcall ClassNameIs(const char* S) const;
virtual TClass __fastcall ClassParent() const;
virtual void * __fastcall ClassInfo() const;
virtual bool __fastcall InheritsFrom(TClass aClass) const;
virtual TObject* __fastcall NewInstance() const;
virtual TObject* __fastcall Create() const;
```

The implementation was made by template meta programming.

8.24.2 static type objects

The reconstruction of the methods of *TObject* and *TClass* of the Delphi VCL in C++ was made by means of template meta-programming. The basic idea stems from Marcin Wudraczyk:

http://marcin.wudarczyk.pl/education/tipstricks/metacls.htm

though his implementation had to be changed in some important points to fit to the design of the VCL.

The basic idea is, to store the meta information of classes in a static members of the classes. These members and the functions to retrieve the information are inserted into the classes by means of two macros (*metaobject* and *metaobject_impl*). That's done, when the according option is enabled.

Functions that are implemented by the macros are:

const TClass VClassType() const

delivers the dynamical TClass

const TObject* VObjectType() const

delivers a const object of the class

const TClass SClassType()

delivers the static type TClass

8.25 abstract methods

Like Delphi also C++ knows abstract methods. So till version 1.4.3 Delphi2Cpp did the most natural way of translation, as in the example:

```
procedure foo; virtual; abstract;
->
```

© 2012 Dr. Detlef Meyer-Eltz

```
void __fastcall foo() = 0;
```

But opposed to Delphi. in C++ no objects can be created from classes with abstract methods. To allow this creation in C++ too, the method is now defined such, that it throws an exception, if it is called:

```
void __fastcall foo ()
{ throw std::runtime_error("abstract function called"); }
```

If the Delphi code was correct, this method never will be called.

```
#include <stdexcept>
```

where *std::runtime_error* is defined is included automatically.

8.26 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() )
```

8.27 Creation of instances of classes

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

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

8.28 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.
A simplified approach is made by Delphi2Cpp, which will lookup resource strings in the current module only and which uses default strings in the case, that no resource files are added to the project.

If the Delphi unit *Test* contains the section:

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

then the translated code will contain the definition:

```
#define Test_SIndexError d2c_LoadResourceString( (int)  "Index out of bounds: %d" )
```

At positions, where resource strings are used, they are now substituted be the according macro.

```
Exception ( SIndexError )
->
Exception ( Test_SIndexError )
```

The function *d2c_LoadResourceString* creates a temporary *ResourceString* structure, which is passed to the Delphi function *LoadResourceString.* If a real resource file is created and used, *d2c_LoadResourceString* can be called with the identifier which is assigned to the strind in the resource.

8.29 Special RTL/VCL-functions

Some functions of the *Delphi RTL/VCL* either don't exist in the *C++Builder* counterpart or have become to member functions of the *String* classes. The conversion of calls of the latter kind of functions into calls of the according member functions is done automatically by *Delphi2Cpp.* For Delphi I/O routines there is a ready translated C++ file. In addition the calls of some compile time functions and some other special functions is done automatically. See the following examples:

```
var
 i, j : Integer;
 p1 : Pointer;
 s1, s2 : String;
 iset : set Of int;
 obj : TObject;
 e :TEnum;
                                                     / std::string
begin
 Assigned( obj );         ->  ( obj != NULL );
 Copy(s1, i, j);          ->  s1.SubString( i, j );  / s1.substr( i - 1, j );
 Dec(i);                  ->  i--;
 Dec(i, j);               ->  i -= j;
 Dec(e1);                 ->  e1--;
 Delete(s1, i, j);        ->  s1.Delete( i, j );     / s1.erase( i - 1, j );
 Dispose(p1);             ->  delete p1;
 Exclude(iset, i);        ->  iset >> i;
 FreeAndNil(p1);          ->  delete p1; p1 = NULL;
 High(TEnum);             ->  /*# High(TEnum) */ 2;
 High(strarray);          ->  strarray.High;
 High(type);              ->  High<type>(); // defined in d2c_system.pas
 Inc(i);                  ->  i++;
 Inc(i, j);               ->  i += j;
 Inc(e1);                 ->  e1++;
 Include(iset, i);        ->  iset << i;
 Insert(s1, s2, i);       ->  s2.Insert( s1, i );    / s2.insert( i - 1, s1 );
 Length(s1);              ->  s1.Length( );          / s1.length( );
 Length(strarray);        ->  strarray.Length;
 Low(TEnum);              ->  /*# Low(TEnum) */ 0;
 Low(strarray);           ->  strarray.Low;
 Low(type);               ->  Low<type>();  // defined in d2c_system.pas
 New(obj);                ->  obj = new obj;
```

© 2012 Dr. Detlef Meyer-Eltz

```
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

8.29.1 I/O routines

Delphi has text and file I/O library routines, which are quite different from C++ I/O routines. So they cannot be substituted automatically by according routines of the C++ standard library. A direct counterpart of the Delphi in C++ was made instead by translation and adaptation of the according parts of the f*ree pascal FCL*. It is contained in the files *d2c_sysfile.h and d2c_sysfile.cpp* in the source folder of the Delphi2Cpp installation. The *GNU Lesser General Public License* which apply to the FCL also applies to these files. The translation was made for *Windows* with the 0x86 processor. The best matching declarations are contained in *d2c_system.pas*.

With *d2c_file.h* and *d2c_sysfile.cpp* the behavior of the Delphi I/O routines is reproduced in C++ quite exactly. For example:

```
var
  t : TextFile;

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

becomes:

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

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

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

8.29.2 Read(Ln)/Write(Ln) routines

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

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

becomes:

© 2012 Dr. Detlef Meyer-Eltz

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

8.29.3 Formatting parameters

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

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

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

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

This is possible also for the *Write(Ln)* procedure, which accepts further output parameters too, because such calls are divided into a series calls by *Delphi2Cpp*.

8.30 RegisterComponents

Since components are an important feature of Delphi, a special translation routine was made for their registration.

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

8.31 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);
```

© 2012 Dr. Detlef Meyer-Eltz

```
var
  foo : Array[TEnum] Of String;

->

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

8.32 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.

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

The use of set's is translated as follows:

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

->

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

or

type TIntSet = set of 1..250;

->

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

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

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

a helping macro and a helping type is created:

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

 MySet = test__1;
```

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

8.34 in-operator

The in-operator of Delphi is replaced by the "Contains" function of the Set class in C++.

8.35 Static arrays

Static arrays in C++ are declared siimilar 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, i.e. the undermost index is 0 and the topmost index is the size of the array minus 1.
Up to the version 1.2.4. of Delphi2Cpp it was the strategy to size the C++ arrays as the upper bound of the original Delphi array plus one. This strategy was changed with version 1.2.5. From now on the size of the C++ array is determined by the number of elements, i.e. that genuine zero based arrays are used.

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

8.36 Dynamic arrays

Dynamic arrays are simulated in the CBuilder 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;
```

© 2012 Dr. Detlef Meyer-Eltz

```
->
DynamicArray < double > MyFlexibleArray;  // CBuilder
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.

8.37 Array indices

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

If the lower bound of an array isn't null, Delphi2Cpp corrects an index by which the array is accessed automatically by subtraction of the lower bound.
Example:

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

is translated to:

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

8.38 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.

8.38.1 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.

8.38.2 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 ) );
}
```

8.38.3 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 Delphi2Cpp 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.

8.38.4 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.

8.38.4.1 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.

8.38.4.2 array of const for other compilers

array of const is reproduced for other compilers Delphi2Cpp 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.

8.38.4.3 array of const vs. set's

Delphi2Cpp decides by the expected parameter type how the set argument is translated:

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

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

 ->

#define test__0 (System::Set< AnsiString, 0, 255 >() << AnsiString ( "hello" ) << AnsiString ( "world"

void __fastcall foo (  TVarRec* arr, const int arr_size );
void __fastcall bar ( TStringSet set );

foo ( ARRAYOFCONST(( "hello", "world" )) );
bar ( test__0 );
```

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

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

->

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

© 2012 Dr. Detlef Meyer-Eltz

```
}
```

8.39 Returning arrays

In Delphi arrays can be returned from functions. This is not possible in C++. Instead a pointer to the first element of the array is returned and used as argument for "memcpy".

```
TObjectArray = array[1..5] of TObject;
function CreateArray: TObjectArray;

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

->

```
typedef TObject* TObjectArray [ 5/*# range 1..5*/ ];
TObject* const * __fastcall CreateArray( );

void __fastcall test( )
{
  TObjectArray arr2;
  memcpy( arr2, CreateArray(), sizeof( TObjectArray ) );
}
```

In cases, where the static array is locally constructed inside of a function, a helping array in the file scope is used for an intermediate copy of the array.

```
function CreateArray: TObjectArray;
begin
  result[1] := TObject.Create;
  result[2] := TObject.Create;
  result[3] := TObject.Create;
  result[4] := TObject.Create;
  result[5] := TObject.Create;
end;
```

->

```
TObjectArray ArrayUnit__0;

TObject* const * __fastcall CreateArray( )
{
  ArrayUnit__0[1 - 1] = new TObject;
  ArrayUnit__0[2 - 1] = new TObject;
  ArrayUnit__0[3 - 1] = new TObject;
  ArrayUnit__0[4 - 1] = new TObject;
  ArrayUnit__0[5 - 1] = new TObject;
  return ArrayUnit__0;
}
```

8.40 Properties

There is a counterpart to the Delphi properties only in the extended C++ of the CBuilder. You can set the option to produce C++ code for other compilers than the CBuilder. In this case properties are eliminated

Instead of the properties their read and write specifications are replaced by two functions whose

© 2012 Dr. Detlef Meyer-Eltz

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.

```
property List : TList read AList write SetList;
->
/* __property TList * List */
TList * ReadPropertyList( ) const;
void WritePropertyList( TList * Value );
```

These functions regulate the access to the fields or methods, which originally were set in the property. The visibility of these fields or methods usually is private or protected. In the "ReadProperty" function the originally set 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 the parameters are passed to the originally method.

```
TList * COptions :: ReadPropertyList( ) const {
  return AList;
}
void COptions :: WritePropertyList( TList * Value ) {
  SetList ( Value );
}
```

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.

```
List := Value;
Value := List;
->
WritePropertyList( Value );
Value = ReadPropertyList( );
```

8.41 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++ the abbreviated form is translated to longer notation: object->property[index]

8.42 for loop's

Delphi 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;
```

© 2012 Dr. Detlef Meyer-Eltz

A straightforward translation of this code - as done with Delphi2Cpp up to version 1.5.0 - 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, which is produced by Delphi2Cpp from version 1.5.1 on:

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

8.43 with-statements

In C++ there are no with-statements. Therefore all elements must be completely qualified.

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

procedure test(OrderDate: TDate);      void __fastcall test ( TDate * OrderDate )
begin                                  {
  with OrderDate do                     /*with OrderDate do*/
    if Month = 12 then                  if ( OrderDate->Month == 12 )
    begin                                 {
      Month := 1;                           OrderDate->Month = 1;
      Year := Year + 1;                     OrderDate->Year = OrderDate->Year + 1;
    end                                   }
    else                                else
      Month := Month + 1;                 OrderDate->Month = OrderDate->Month + 1;
end;                                   }
```

At the translation of more complex with-statements a temporary variable is created:

```
                                 ->     {
with TMyObject.Create do                  /*# with TMyObject.Create do */
  try                                     TMyObject* tmp2 = new TMyObject;
    // do something with the object       try
  finally                                 {
    Free;                                   // do something with the object
  end;                                    }
                                          __finally
                                          {
                                            tmp2->Free();
                                          }
                                        }
```

© 2012 Dr. Detlef Meyer-Eltz

8.44 Nested functions

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

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

implementation

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

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

begin
 iClassVar := 1;
 iFunctionVar := 2;
 NestedTest1(3, 4);
 result := result + iTwiceParam;
end;
```

->

```
class TNested: public System::TObject {
  public:
  int iClassVar;
  int __fastcall Test( int iOuterParam, int iTwiceParam, String s );
  public: inline __fastcall TNested () {}
  private: // originally nested
  void __fastcall NestedTest( int iInnerParam, int iTwiceParam, int& iFunctionVar, int& iOuterParam, in
};

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


int __fastcall TNested::Test( int iOuterParam, int iTwiceParam, String s )
{
  int result = 0;
  const char cSeparate = ':';
  int iFunctionVar = 0;
  iClassVar = 1;
  iFunctionVar = 2;
  NestedTest1( 3, 4, iFunctionVar, iOuterParam, result );
  result = result + iTwiceParam;
  return result;
}


int __fastcall TNested::Test ( int iOuterParam, int iTwiceParam )
{
  int result;
  int iFunctionVar;
```

© 2012 Dr. Detlef Meyer-Eltz

```
  iClassVar = 1;
  iFunctionVar = 2;
  NestedTest( 3, 4, iOuterParam, result, iFunctionVar );
  result = result + iTwiceParam;
  return result;
}
```

From Delphi2Cpp 1.5.2 on only the parameters, variables and constants are passed, which are actually required. It is taken into account that at multiple nesting possible parameters which aren't needed in a function itself but in a function subordinated to it are passed through. The "result" variable for the return value of the topmost function is submitted to subordinate functions under the name "xResult" and therefore is distinguished from the "result" variables there.

Other possibilities to translate nested functions are discussed here:

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

8.45 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;
```

8.46 Initializing arrays

Array's are initialized very simply in Delphi. The initialization of an array of *TStyleRecord*'s:

```
TStyleRecord = record
  Name    : string;
  Color   : TColor;
  Style   : TFontStyles;
end;
```

could look like:

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

This was translated in the earlier versions of *Delphi2Cpp* as:

```
TStylesArray DefaultStyles = {
 { "tnone" /*Name*/, clBlack /*Color*/, arrays__0 /*Style*/ },
 { "tstring" /*Name*/, clMaroon /*Color*/, arrays__1 /*Style*/ },
 { "tcomment" /*Name*/, clNavy /*Color*/, arrays__2 /*Style*/ }
};
```

It is a problem however, that such initializations are only possible, if the elements are built in types of C. Such initializations of elements of the type *TFontStyle* aren't permitted. Beginning with version 1.2.4., *Delphi2Cpp* therefore generates initialization functions which are called within the initialization routine of the corresponding unit.

```
void DefaultStylesInit( )
{
  DefaultStyles[0].Name = "tnone";
  DefaultStyles[0].Color = clBlack;
  DefaultStyles[0].Style = arrays__0;
  DefaultStyles[1].Name = "tstring";
  DefaultStyles[1].Color = clMaroon;
  DefaultStyles[1].Style = arrays__1;
  DefaultStyles[2].Name = "tcomment";
  DefaultStyles[2].Color = clNavy;
  DefaultStyles[2].Style = arrays__2;
}

void DefaultStylesInit( )
{
  DefaultStylesInit0( );
  DefaultStylesInit1( );
  DefaultStylesInit2( );
}
```

8.47 finally

The Delphi keyword *finally* is substituted by the keyword *__finally* for CBuilder. For other compilers it is translated by:

```
catch(...)
{
  ... // statements
  throw;
}
... // statements
```

But this translation will not handle "exit" or "goto" or other non-exception ways of leaving the current code block. If you 'return' from within a 'try' block, the 'finally' code will be skipped.

© 2012 Dr. Detlef Meyer-Eltz

8.48 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.

In CBuilder 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 in 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 CBuilder 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 )
```

If the CBuilder option is not set, only the comments are written, but not the macros.

© 2012 Dr. Detlef Meyer-Eltz

8.49 threadvars

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

```
threadvar
 x: Integer;
```

CBuilder has an according keyword *__thread.* which could be used:

```
int __thread x;
```

But Delphi2Cpp chooses the equivalent possibility for the same declaration, which also is used by Visual C++,

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

For gcc the declaration is translated as:

```
__thread int x;
```

8.50 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++ now 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;
```

8.51 Namespaces

In the professional version there is a option, to create a namespace for each unit.
In C++ header files the namespaces 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;
```

© 2012 Dr. Detlef Meyer-Eltz

```
...

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 CBuilder have a using clause at their end. That's why Delphi2Cpp 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.

The identifier of the namespace of an overwritten System.pas is always put in front of the according types, even if the option to use namespaces is deactivated.

8.52 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, which 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);
```

Delphi2Cpp supports a *lightweight and fast* solution for other compilers, that was found by *Tamas Demjen* :

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

His detailed explanation of his approach is in the file "Events.rtf": According to his solution the closure above has to be defined as:

```
typedef event1< void, TObject* > TNotifyEvent;
```

The return type and the parameters of the original member pointer become template arguments of an *event* class. Internally this class holds a pointer to a standard C++ member function and to an instance of a class with this function. This makes the assignment of an event handler to an event looking complicated. A special *Connect* method has to be called instead of a simple assignment. However *Delphi2Cpp* creates the according code automatically for you. For example:

```
Button1.OnClick.Connect<TAppManager,
                        TAppManager::Button1Click>(AppManager);
```

Here *Button1.OnClick* is an event of *TButton*, *TAppManager* is a class with the event handler *Button1Click* and *AppManager* is an instance of that class. 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);

// assigning the event handler to another button
Button2.OnClick = Button1.OnClick;
```

8.53 Libraries

Delphi2Cpp can translate library files for Dll's like the following example from the Delphi help. It shows a DLL with two exported functions, Min and Max.

```
library MinMax;
```

© 2012 Dr. Detlef Meyer-Eltz

```
function min(X, Y: integer): integer; stdcall;
begin
  if X < Y then min := X else min := Y;
end;

function max(X, Y: integer): integer; stdcall;
begin
  if X > Y then max := X else max := Y;
end;

exports
  min,
  max;

begin
end.
```

->

```
extern "C" __declspec(dllexport) int __stdcall max( int X, int Y );
extern "C" __declspec(dllexport) int __stdcall min( int X, int Y );

int __stdcall min( int X, int Y )
{
  int result = 0;
  if ( X < Y )
    result = X;
  else
    result = Y;
  return result;
}

int __stdcall max( int X, int Y )
{
  int result = 0;
  if ( X > Y )
    result = X;
  else
    result = Y;
  return result;
}
```

The Delphi help recommends: "If you want your DLL to be available to applications written in other languages, it's safest to specify **stdcall** in the declarations of exported functions." However, the names of such exported functions get a special "decorated" signature in order to facilitate language features like overloading. To avoid such name mangling a module definition (.def-) file can be used in the Dll project. Delphi2Cpp creates module definition files automatically.

9 What is partially translated

Some features of Delphi can ba translated partly only.

Variant parts in records
Visibility of class members
Manipulations with class-reference types

9.1 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)
```

9.2 Visibility of class members

In Delphi a private or protected member is visible anywhere in the module where its class is declared. In C++ a private or protected member is visible only in the class. So the tranlation makes all classes in the same module to friends of each other. But as problem remains, that an access to these elements from code in the same unit, but outside of the class is forbidden in C++.
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.
(Delphi2Cpp ignores the {$M+} directive, which would make them public.)

9.3 class-reference type

In Delphi operations can be performed on a class itself, rather than on instances of a class. With *TClass* references to classes can be declared and used. CBuilder has as counterpart to Delphi's *TClass*:

```
typedef TMetaClass* TClass
```

Beginning with version 1.4.5 the installation of the professional Version contains *substitutes for TObject and TMetaClass*, which was developed especially for *Delphi2Cpp*. By these classes much of the Delphi Meta capabilities can be simulated. But still some of the possibilities of Delphi to operate with such class references are not translated correctly by *Delphi2Cpp* or even cannot be translated at all.
In the professional version of Delphi2Cpp the macros *DECLARE_DYNAMIC* and *IMPLEMENT_DYNAMIC* can help.

© 2012 Dr. Detlef Meyer-Eltz

9.4 inline assembler

The **student version** of *Delphi2Cpp* doesn't process inline assembler code. 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*, you can enable a processing of the assembler code:

- Delphi comments are converted to C++ comments
- Delphi expressions used inside of the assembler code are analysed and converted
- identifiers can be substituted by other identifiers

These actions often suffice to get operating code. However, assembler expert knowledge is needed in individual cases. The result depends on the target compiler:

C++Builder:

Much inline assembler code of Delphi also works with C++Builder. But there are a few constructs of Delphi inline assembler which are not permitted in C++Builder and there are differences in the manner how parameters are passed to functions and how values are returned.

Other compilers:

Local labels are converted to global labels for other compilers than C++Builder. Local labels are labels that start with an at-sign '@'. They don’t have to be declared and their scope is restricted from the *asm* reserved word to the end of the *asm* statement that contains it. Other compilers don't know them. Therefore they are converted to normal labels.Example:

```
asm
@SomeLoop:
  jmp @SomeLoop
end;
asm
  push  esi
  { Some remarks }
@SomeLoop:
  jmp @SomeLoop
  pop esi
end;
```

->

```
_asm
{

 SomeLoop__0:
       jmp SomeLoop__0
}
_asm
{

   push  esi
     //  Some remarks
 SomeLoop__0:
     jmp SomeLoop__0
   pop esi
}
```

9.5 const-correctness

Compared with the concept the *const*-correctness in C++ the use of *const* in Delphi is very limited. In

the Delphi *const*-section true constants are declared whose values cannot change and the keyword *const* also can be used to declare constant parameters. No values can be assigned to constant parameters and they cannot be passed to routines, where *var* parameters are expected. But unlike C++, Delphi does not permit methods to be marked as *const*. The VCL pendant of the C++Builder is not designed for C++ *const*-correctness.

If the translated Delphi code simply should compile, it would be the best to ignore the *const*-qualifier totally. But it is the aim of Delphi2Cpp, that the created C++ code should be C++-like code and the translation also is orientated at the way the C++Builder produces C++-header files from Delphi sources. C++Builder leaves the *const* qualifiers for parameters. For example:

```
TMyClass = class
private
  FObject : TObject;
public
  constructor Create(const Obj: TObject);
```

The declaration of a constructor is translated by C++Builder and accordingly by Delphi2Cpp to

```
__fastcall TMyClass( const TObject* Obj );
```

But this leads to a problem in the body of the constructor, where the parameter is assigned to a member of the class:

```
__fastcall TMyClass::TMyClass( const TObject* Obj )
 : FObject(Obj)
{
}
```

Compiling this code produces the error: E2034 conversion of 'const TObject *' to 'TObject *' not possible. So a cast is necessary, which strips the const qualifier away:

```
__fastcall TMyClass::TMyClass( const TObject* Obj )
 : FObject((TObject*)Obj)
{
}
```

or more precisely:

```
__fastcall TMyClass::TMyClass( const TObject* Obj )
 : FObject(const_cast<TObject*>(Obj))
{
}
```

This example suggests to leave out the *const*-qualifier at the translation anyway as mentioned above. You can correct the code in this way, but there are other cases where the const-qualifier should be preserved.

For other compilers than C++Builder the methods, which are created for the read-specifiers of properties are made const-methods.

© 2012 Dr. Detlef Meyer-Eltz

10 What is not translated

At first Delphi2Cpp is concentrated on code constructs occurring the most frequently in Delphi code. In more complex cases Delphi2Cpp even can fail at constructs described as translatable in this manual. In the result, there may by some quotes of the original Delphi code within the C++ code. They have to be eliminated by hand. Sometimes Delphi2Cpp generates an explicit comment which points out with the word "todo" that here still is something to do.

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

- Little effort has been done to test the COM technologies of the Delphi ActiveX framework.
- Inline assembler code in Delphi and C++ almost are identically. Delphi2Cpp doesn't translate these parts.but only copies them.
- *Delphi2Cpp* always assumes unique names.So, e.g functions with the same name, which differ in the signatures only cannot be distinguished.
- Some problems with constructors remain. E.g. *Delphi2Cpp* cannot distinguish constructors with equal signatures.
- Manual post-processing to achieve const-correctness is necessary.

Special problems:

char type problem
finally problem

10.1 char type problem

The char type in C++ is a signed type, whereas it is an unsigned type in Object Pascal. It is rare that a situation would occur in which this difference would be a problem. A customer sent me the following example:

```
var c : Char;
c := someUpperAsciiValue;
if c < #32 then
  ; // c is between 0 and 31.
```

gets converted to:

```
char c;
c = someUpperAsciiValue;
if (c < '\x20')
  ; // c is between -127 and 31.
```

He has fixed the code by casting c to Byte there.

10.2 finally problem

For other compilers than CBuilder there are cases, where the 'finally' code will be skipped..

11 Pretranslated C++ code

Parts of the Delphi RTL/VCL are pre-translated to C++ and optionally you can prepare parts of your own Delphi code with pre-translated C++ code, if you have to translate your modified Delphi sources again and again.

11.1 Delphi RTL/VCL

There is some ready to use C++ code translated from the free pascal sources:

http://www.freepascal.org/

This code covers

- missing parts of Delphi's System.pas for C++Builder and
- most parts of the System unit and the Sysutils.unit for Visual C++
- most parts of the System unit and the Sysutils.unit for Linux

Other units of the VCl can be translated with Delphi2Cpp, but an automatic translation just of these two files is insufficient, firstly because the *System.pas* is incomplete and secondly because they depend on assembler code and on calls of API functions of the operating System. It would be nonsense to retranslate the Delphi adaptations to the C++ Windows or Linux API back to C++.

<u>Restrictions</u>

The code was made originally by automatic translation with a special version of *Delphi2Cpp: FreePascal2Cpp*. Some parts of the generated C++ code were improved manually. Large parts were tested and are working well, but there isn't any guarantee that the code is completely bug-free. There is no translation for the Variant class and only a minimal implementation of a Currency class. The formatting of real types in free pascal depend on a special binary layout of that types. The formatting couldn't be reproduced exactly. Free pascal uses *Mersenne twister* for the generation of random numbers.This part was not translated back to C++ because there are a lot of C++ implementations of *Mersenne twister* in C++

http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/VERSIONS/C-LANG/c-lang.html

Currently a primitive random generator based on C++ standard functions is used. Such standard function were preferred to direct translations of Delphi functions at other places too. So there might be slight differences to the original Delphi behavior. For example the error position returned in the Code-parameter of the Val procedure, is a dummy only in C++.

Despite of these restrictions for many users this provided C++ code will make the migration of their Delphi code much easier.

<u>Other operating systems and compilers</u>

Though free pascal is made fur use on many different operating systems, the translation here is for Visual C++ under Windows and gcc under Linux only. Of course it would be desirable if the translation would cover all other operating systems and compilers too. But that's much effort and it has to be doubted that somebody wants to produce C++ code e.g. for OS2 or Amiga with a Delphi2Cpp. In addition it is not desirable to reproduce the complicated folder structure in which the free pascal code

© 2012 Dr. Detlef Meyer-Eltz

is organized. It was a lot of work to find just the needed parts of code for Windows and Linux in the in the scattered files. Ohter parts of the free pascal sources can be translated on demand.

11.1.1 C++ code for C++Builder

The C++ Builder already has its own version of the Delphi RTL/VCL with C++ interface files. If the option to produce C++ for C++ Builder is enabled Delphi2Cpp tries to optimize the translated code to work together with these libraries. For the parts which are missing in System.pas, there is pre-translated code in the folder:

..\Delphi2Cpp\Source\CBuilder

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

..\Delphi2Cpp\Source\d2c_system.pas

This file includes the headers of the files for the missing parts of *System.pas*:

```
#include "d2c_systypes.h"
#include "d2c_systobj.h"
#include "d2c_sysmath.h"
#include "d2c_sysstring.h"
#include "d2c_sysfile.h"
```

11.1.2 C++ code for Visual C++

For Visual C++ most parts of the most basic Delphi units *System.pas* and *Sysutils.pas* are translated from the free pascal sources:(http://www.freepascal.org/). You can find the code in the folder

..\Delphi2Cpp\Source\Other

The C++ counterpart of *Sysutils.pas* is written in a single source file. System.pas however is split into some smaller files. But all headers of these smaller files are put together into "System.h", which is:

```
#ifndef SystemH
#define SystemH

#if defined( WIN32 ) || defined( WIN64 )
#define windows 1
#endif

#include "d2c_system.h"
#include "d2c_systypes.h"
#include "d2c_sysconst.h"
#include "d2c_syscurr.h"
#include "d2c_sysdate.h"
#include "d2c_systobj.h"
#include "d2c_openarray.h"
#include "d2c_sysexcept.h"
#include "d2c_sysmath.h"
#include "d2c_sysstring.h"
#include "d2c_sysfile.h"
```

© 2012 Dr. Detlef Meyer-Eltz

```
using namespace System;

#endif // SystemH
```

There are three pas-file, which contains the according interface declarations. These pas-files have to be put into the search paths for files not to convert (VCL) in the option dialog of Delphi2Cpp.

11.1.3 C++ code for Linux

Under Linux with gcc the same code for the System unit and for the SysUtils unit can be used as for Visual C++. The code is in the folder:

..\Delphi2Cpp\Source\Other

For use under Linux or with other compilers than VisualC++ either the line

```
#include "stdafx.h"
```

has to be removed from the sources or a dummy header with this name has to be made. Some definitions, which are created by Visual Studio automatically are presupposed:

```
_UNICODE for unicode/widestring applications
_CONSOLE for console applications
_USRDLL  for dll's
```

Under Linux the command line arguments aren't accessible before entering the main function of a program. This function therefore should start with something like:

```
  System::Argv = argv;
  System::Argc = argc;
```

11.1.4 Copyright and editions

Copyright of the basic file version

Explanation: in the following license "library" means
the following files:

System.pas / d2c_system.pas
System.h
d2c_sysconst.h
d2c_syscurr.h d2c_syscurr.cpp
d2c_sysdate.h d2c_sysdate.cpp
d2c_sysexcept.h d2c_sysexcept.cpp
d2c_sysfile.h, d2c_sysfile.cpp
d2c_sysstring.h d2c_sysstring.cpp
d2c_systypes.h

d2c_varrec.h
d2c_smallstring.h
Windows.pas
Sysutils.pas
Sysutils.h Sysutils.cpp

AS THEY ARE CONTAINED IN THE FREE TRIAL VERSION OF Delphi2Cpp.

This library is derived from the FreePascal library:

http://www.freepascal.org/

FreePascal is published under the terms of GNU Lesser General
Public License and the same terms apply to this library.

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with d2c_sysfile.h/cpp; if not, write to the Free Software
Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Copyright of the extended file version

The terms of the license above don't apply to extended versions
of these files, which are distributed with commercial versions of
Delphi2Cpp. Individual licenses are applied to them.
The library doesn't depend on the commercial extensions and the
the commercial extensions only originates from the author
Dr. Detlef Meyer-Eltz or might use code which has no copyright restrictions

Copyright (C) <2011> <Dr. Detlef Meyer-Eltz>

The extended version of this file is authorized for unlimited use in any
Delphi2Cpp project.

Files	Delphi2Cpp trial	Delphi2Cpp student	Delphi2Cpp professional
System.pas	++	++	++
Windows.pas	++	++	++
Sysutils.pas	++	++	++
d2c_system.pas	++	++	++
d2c_sysconst.h	++	++	++
d2c_syscurr.h/.cpp	++	++	++

d2c_sysdate.h/.cpp	++	++	++
d2c_sysexcept.h/.cpp	++	++	++
d2c_sysfile.h/.cpp	++	++	++
d2c_sysmath.h/.cpp	+	++	++
d2c_openarray.h/.cpp	-	++	++
d2c_sysstring.h/.cpp	+	++	++
d2c_system.h/.cpp	+	++	++
d2c_systobj.h/.cpp	-	+ (dummy class)	++
d2c_systypes.h	++	++	++
d2c_varrec.h	++	++	++
d2c_smallstring.h	++	++	++
d2c_sysutils.h/.cpp	++	++	++

++ : completely contained
+ : partly contained
- : not contained

11.2 Preparing Delphi code

For users who would like to continue to develop their Delphi code and in parallel also need the C++ code updated, the professional version Delphi2Cpp offers a possibility to avoid to have to postprocess the output again and again. In the Delphi code C++ code fragments can already be inserted, which then become an integral part of the translated code. This can be done in two ways:

insertions by the comments (*_ ... _*)
substitutions by the predefined identifier Cpp

In the section about the *overwritten System.pas* there are examples and explanations how to use this feature.

11.2.1 Comments (*_ ... _*)

In the professional version of Delphi2Cpp the translator interprets the extended Delphi brackets (*_ ... _*) in a special way. A text in such brackets is taken unchanged into the C++ code.

E.g. you can force in a simple way that an additional header is included into the C++ code:

```
(*_ #include "math.h" _*)
```

->

```
#include "math.h"
```

11.2.2 Predefined identifier Cpp

In the professional version of Delphi2Cpp the identifier *Cpp* is pre-defined in the translator. The preprocessor interprets code areas, in which a test for the definition of Cpp succeeds, a special way: they are enclosed in the special brackets (*_ ... _*) so that the translator then writes these text areas unchanged into the C++ code.

For example:

© 2012 Dr. Detlef Meyer-Eltz

```
{$ifdef Cpp}
  out << s << endl;
{$else}
  WriteLn(s);
{$endif}
```

After pre-processing the translator gets:

(*_ out << s << endl; _*)

and because of the special treatment of the brackets *(*_..._*)*, the final C++ output is:

out << s << endl;

Delphi2Cpp 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.

11.2.3 Delphi directives to support CBuilder

There are four directives defined in Delphi to support the generation of C++ header files for C++Builder. All the Delphi translations of Windows interfaces don't have to be translated back, but simply are left out by means of these directives. In Delphi2Cpp 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

11.2.3.1 $HPPEMIT

The *HPPEMIT* directive 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'}
```

11.2.3.2 $EXTERNALSYM

The *EXTERNALSYM* directive prevents the specified Pascal symbol from appearing in C++ header files. This might be useful if a type has to be defined in Delphi and the type is defined in C++ elsewhere.
In Delphi2Cpp there is no difference between this directive and the *NODEFINE* direktive.

Syntax:

```
{$EXTERNALSYM identifier}
```

Example:

```
type
  size_t : LongWord;
  {$EXTERNALSYM size_t}
```

11.2.3.3 $NODEFINE

The *NODEFINE* directive prevents the specified symbol from being included in the C++ header file, while allowing some information to be output to the OBJ file. However, in Delphi2Cpp there is no difference between this directive and the *EXTERNALSYM* direktive. When you use *NODEFINE*, it is your responsibility to define any necessary types with *HPPEMIT*. For example:

Syntax:

```
{$NODEFINE identifier}
```

Example:

```
type
       Temperature = type double;
       {$NODEFINE Temperature}
       {$HPPEMIT 'typedef double Temperature'}
```

11.2.3.4 $NOINCLUDE

The *NOINCLUDE* directive prevents the specified file from being included in header files generated for C++.

Syntax:

```
{$NOINCLUDE filename}
```

Example:

```
 {$NOINCLUDE Unit1} // removes #include Unit1.
```

© 2012 Dr. Detlef Meyer-Eltz

12 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."

13 TextTransformer

Delphi2Cpp was 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

14 Service

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

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

or in German at:

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

I also like make extensions of Delphi2Cpp 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

© 2012 Dr. Detlef Meyer-Eltz

Index

- - -

-- 16, 57

- " -

"String" as 21

- % -

%INCLUDEPATH% 30
%MAINSOURCE% 30
%PROJECT% 30

- (-

(*_ ... _*) 85
(*_..._*) 13

- @ -

@ 78

- _ -

__closure 74
__fastcall 24
__finally 71
__thread 73
_CONSOLE 82
_UNICODE 82
_USRDLL 82

- + -

++ 16, 57

- < -

<< 57

- > -

>> 57

- A -

Abs 13
Absolute address 73
abstract methods 55
ActiveX 80
Adaption of parameters 44
AddError 35
AddMessage 35
Address 13
AddWarning 35
Ancestor 46
and 41, 42
AnsiString 21
Application_default.bpr 30
Array 61
array of const 64, 65
Array parameter 45, 62
array result 66
Array size 59, 60
ARRAYOFCONST 64, 65
arrays 70
arrays assignement 43
Assembler 17, 80
Assigned 57
assignment 43
at-sign 78
Automatic creation of managements 31

- B -

Backup 35
Base class 47, 51
BDE 13
BDE.dcu 13
BDE.int 13
BDE.pas 13
Beautifier 88
BEGIN_MESSAGE_MAP 72
bitwise operator 41
BlockRead 58
BlockWrite 58

© 2012 Dr. Detlef Meyer-Eltz

boolean operator 41
BPK files 30
BPR files 30
BPR-file 29

- C -

C++ Builder 4
C++ header 29
C++ source file 29
Call 44
Case sensibility 38
Case sensitivity 28
Case-sensivity 19
Cast 42
CBuilder 2, 24, 39, 71, 73
Char 21
char type 80
character array 43
Class creation 56
class method 51, 52, 53
ClassInfo 54
ClassName 54
ClassNameIs 54
ClassParent 54
class-reference type 77
ClassType 26
Clear types and variables 6, 28
Clear windows 6
C-like 21
CObject 26
COM technologies 80
command line mode 36
Command line parameter 36
Comments 39
Compile time functions 13
Compiler 4, 24, 26
Conditional compilation 17, 28
Conflicting names 80
Connect 74
Console_default.bpr 30
const 78
const parameter 78
const_cast 78
constant 78
const-correctness 78
Constructor 46, 47, 48
Constructors of the base class 80
Contact 88
Converting dpr-files 29
Copy 57
Copyright 83
Cpp 13, 85
Create 26, 54
CreateObject 26
Creating BPR/BPK files 30
Creation of managements 31
CRuntimeClass 26
Currency 81
Customization 88

- D -

d2c_LoadResourceString 56
d2c_openarray 82
d2c_sysconst 82
d2c_syscurr 82
d2c_sysdate 82
d2c_sysexcept 82
d2c_sysfile 82
d2c_sysfile.cpp 58
d2c_sysfile.h 58
d2c_sysmath 82
d2c_sysstring 82
d2c_system 82
d2c_system.pas 13, 15
d2c_systobj 23, 49, 54, 82
d2c_systypes 82
Daniel Flower 60
Dec 16, 57
Decimals 59
DECLARE_DYNAMIC 26
DECLARE_DYNCREATE 26
declspec(thread) int x; 73
Default array-property 67
Default options 27
Default.prj 27
def-file 75
Definition 4, 17
Delete 15, 57
Delphi ActiveX framework 80
Delphi I/O routines 13
Delphi project 29
Delphi RTL/VCL 4, 81, 82
Delphi string 21
Delphi2Cpp 2

© 2012 Dr. Detlef Meyer-Eltz

DelphiSets.h 60
Demjen 74
Dependencies 11, 28
designintf.pas 13
Destructor 51
Directive 17, 86
Directives 28
Dispose 57
Dll 75
DPR-file 29, 30
dproj-file 30
dsgnintf.pas 13
Dynamic array 61
dynamic_cast 56
DynamicArray 61

- E -

E2034 78
END_MESSAGE_MAP 72
Enumerated types 16, 59
equality operators 42
event 74
Event handling 74
Events.rtf 74
Exclude 57
Excluding individual files 32, 34
Extended "System.pas" 13
extended System.pas 4
extern 38
EXTERNALSYM 87

- F -

FCL 58
File manager 31
File organization 38
finalization 70
Finalization part 80
finally 71
finally problem 80
FmtLoadStr 56
for loop 67
Formatting 88
Formatting of real types 81
Formatting parameters 59
Free pascal 81, 82
FreeMem 13, 15
FreePascal FCL 58
FreePascal2Cpp 81
friend 77
function 46
function call 44
Function name 26, 43
Functions 43

- G -

gcc 24, 73, 81, 82, 83
GetMem 13, 15
GetRuntimeClass 26
GNU Lesser General Public License 58

- H -

High 13, 57, 61, 62
HPPEMIT 86

- I -

I/O routines 58
Identifier notation 28
IMPLEMENT_DYNAMIC 26
IMPLEMENT_DYNCREATE 26
Inc 16, 57
Include 57
Include directive 28
Include paths 11
Included files 28
Inheritance 46
inherited 46
InheritsFrom 26, 54
initialization 70
Initialization lists 47
Initialization part 80
Initializing arrays 70
inline assembler 17, 78
Inline assembler code 80
in-operator 61
Insert 57
IsDerivedFrom 26
is-operator 56

© 2012 Dr. Detlef Meyer-Eltz

- K -

Keyword 21

- L -

Last error position 6
Learning types and variables 6, 17
Length 57, 61
Lib_default.bpr 30
Library 6, 75
Linux 81, 83
LoadResourceString 56
LoadStr 56
Local label 78
Log panel 8
Log-file 33
Lookup algorithm 13
Lookup files recursively 30
Low 13, 57, 61

- M -

-m 36
Management 31, 36
Mangement 36
Marcin Wudraczyk 55
MAXIDX 63
MAXIDX(x) 61
memcpy 43, 66
Memory management 13, 15
Mersenne twister 81
Message handlers 72
Meta cpapbilities 54
Meta cpapbilities, enabling 23
meta information 55
metaobject 55
metaobject_impl 55
Method pointers 74
MFC 26
Microsoft Foundation Classes 26
Missing constructor 48
module definition file 6, 75
Move 13
MSWINDOWS 4

- N -

N:1 33
N:N 33
name space 13
Names of helping variables 21
namespace 13, 22, 73
Nested functions 69, 80
New 15, 57
NewInstance 54
NODEFINE 87
NOINCLUDE 87
Notation 19

- O -

ObjectIs 56
Odd 13
Open array 45, 62
OpenArray 64, 65
Operator 41
operator precedence 42
Options 11
or 41, 42
order of type definitions 40
Other compiler 4, 26, 61, 66
Other compilers 71
Output options 23
Overwriting "System.pas" 13
Overwritten System.pas 56

- P -

-p 36
Package_default.bpk 30
PAnsiChar 57
Parameter 44
Paths to the source file 31
-pause 36
pch.inc 25
plain old data types 58
POD types 58
Pos 57
PP-button 6
precedence of operators 42
Precompiled header 24

© 2012 Dr. Detlef Meyer-Eltz

Pred 13
Prefix 26
Preprocessed code 4
Preprocessor 6, 17, 28
Pretranslated C++ code 81
Pretranslator 28
Pretty-printer 88
Preview of the target files 34
procedure 46
procedure call 44
Procedures 43
processing inline assembler 17
Processors 17
professional edition 2
professional version 6, 8, 22, 23, 36, 54, 73
Project file 11
propertiy 24
property 26, 66

- R -

-r 36
Range 60
Reaarangements 39
read 66
Read procedure 58
ReadLn procedure 58
ReadProperty 66
Real types formatting 81
ReallocMem 13, 15
Record 76
Reference 13
Refresh 34
RegisterComponents 57, 59
Registration 3
Replacement 18
Reset to default 27
resource string 56
result 43, 69
Results 35
Returning arrays 66
return-statement 43
RTTI 26
Runtime class information 26
runtime_error 55

- S -

-s 36
SClassType 55
Search path to the source files 11, 13
Search path to the VCL 11, 12
Search paths to source files 31
Selecting source files 32
Self 52
Service 88
set 60, 65
Set class 60, 61
SetLength 57
SetString 15
ShellApi.pas 13
shortint 39
Simple substitutions 39
Single characters 41
Size of an array 61
standard edition 2
Standard string 21
Start a translation 35
Static array 61
Static array parameter 63
static method 52
static type objects 55
std::runtime_error 55
std::type_info 54
std::vector 61
stdafx.h 24, 25
stdcall 75
stdexcept 55
Str procedure 59
Strarting the translation 6
strcpy 43
String 21
String constant 41
string parameter 45
String type 4
student edition 2
Substitution 18
Substitution in the translator 21
Substitution of the preprocessor 19
Substitution table 19
SubString 57
Succ 13
Synchronizing Delphi and C++ code 81

© 2012 Dr. Detlef Meyer-Eltz

SyncObjs.pas 13
System 81, 82
System unit 4
System. 13
System.pas 4, 11, 13, 15, 56, 73
System::Set 21, 60
Sysutils 81, 82
Sysutils unit 4

- T -

-t 36
Tamas Demjen 74
T-button 6
TClass 77
TD2CMetaClass 23, 54
TD2CObject 23, 49, 54
template meta programming 55
temporary file 35
Temporary variables 45
TextTransformer 88
threadvar 73
TMetaClass 23, 26, 54, 77
TObject 13, 23, 26, 46, 49, 54
Tool bar 6
Translating all files of a Delphi project 31
Translation 27
Translation options 4, 11, 33
Translation service 88
Translator 6, 17
TSet 21, 60
ttm 36
TVarRec 64, 65
type cast 15
type checking 56
type identifier 39
type name 39
typedef 40
Types option 21

- U -

Unification of notations 28
Union 76
Unit frame 6
UniversalIndentGUI 88
unsignedchar 39
unsignedint 39
Upgrade 3
Use pch.inc 24
User options 10
Uses clauses 38
using 73

- V -

Val procedure 81
variable 38
Variant 76
Variant class 81
Variant types 80
VCL 11, 28
VCL_MESSAGE_HANDLER 72
VClassType 55
VCL-functions 57
vector 61
virtual class method 53
Virtual constructors 49
Visibility 77
Visual C++ 24, 73, 81, 82
VisualC 24
VObjectType 55
void pointer casts 42
void* 42

- W -

waiting for definiens 40
wchar_t 21
WideString 21
Width 59
Window position 10
Window size 10
Windows 58, 81
Windows API 13
Windows interfaces 86
Windows messages 72
Windows.pas 13
WinProcs.pas 13
WinTypes.pas 13
with-statement 68
write 66
Write procedure 58, 59
WriteLn procedure 58, 59

© 2012 Dr. Detlef Meyer-Eltz

WritePropery 66

xResult 69

© 2012 Dr. Detlef Meyer-Eltz