


Adding Rules to the C4 Conversion Wizard

by Mike Hanson

Published 1998-03-01

Download the code here

As prior versions of Clarion were released, we were often left feeling that our old applications were abandoned. There was never a "suitable" upgrade path: If we wanted our APPs in the new version, we had to rewrite them.

Well, TopSpeed has finally broken this trend with C4's new Conversion Wizard. It miraculously scans through your APP, making changes to accommodate the new platform. I must admit that it's a technological marvel, and I'm often astonished at its capabilities. It handles a wide variety of issues, from changing the template chain to adding a ProgressWindow to Reports.

There's one problem, though. I write my own templates, and I would like to convert them too. Most of my templates are fully described by the PROMPTs, so my biggest need is to change the template chain to match my ABC compatible templates. (e.g.; if my old template chain was "MikeHanson", the new chain is "MikeHansonABC".)

Of course, the first step is to convert the template itself. This involves changing file access statements, #INSERT(%GroupName(Clarion)) to #INSERT(%GroupName(ABC)), and many other modifications. Template conversion is not the focus of this article, though. I'm more concerned about converting the APP once the template has been updated. (See my article in Issue 7 for more information on this complicated topic.)

The converter performs its magic by exporting the old APP to a TXA, modifying that TXA, then importing it into a new APP. If we had to change the templates chains ourselves, we could use a similar technique. Of course, we would do a manual search and replace of the chain name in the TXA file. Creating an automatic conversion facility is definitely the preferable option.

Creating a New Rule DLL

TopSpeed was nice enough to give us the entire source for their converter. You'll find it in C:\CLARION4\CONVSRC. The main engine is contained in CNVENG.PR, CNVENG.CLW, CNVENG.INC and CNVENG.TRN. This source has been used to create C4CNVENG.DLL, which is the foundation for the rest of the conversion modules.

There are two conversion rule sets supplied with C4. The first, CNVRULES, is responsible for converting regular APPs from CW 2.003 to Clarion 4. The second, CNVBETA, is used to convert your applications from interim beta versions of C4. We'll be copying bits and pieces from both of these rule sets to achieve our goals.

When you first look at the conversion source, it seems completely obtuse. It's actually not that hard to create your own rules, though. Just perform the following steps, and you're there. (By the way, my example converter is used for both the BoxSoft SuperTemplates and my public domain offerings, so my converter is called STAB_CNV.)

Step 1 – Make C4CNVENG.LIB

Clarion doesn't supply the LIB file for you to link to C4CNVENG.DLL. However, you can use LIBMAKER.EXE (supplied with Clarion 4) to create C4CNVENG.LIB. Place this file into the C:\CLARION4\LIB directory.

Step 2 – Create Your EXP file

Ultimately, we are trying to create our own DLL that Clarion can use during the conversion process. When creating a DLL, the linker must know what elements to make visible to the outside world. It gets these instructions from an EXP file. When we create APPs with Generator, it automatically creates the EXP for us. In this case, we'll have to do it ourselves.

This is really quite simple. Just copy CNVBETA.EXP to STAB_CNV.EXP, then edit the EXP as necessary. In this case, just edit the first line. You'll end up with something like this:

```
LIBRARY stab_cnv
CODE MOVEABLE DISCARDABLE PRELOAD
DATA MOVEABLE SINGLE PRELOAD
HEAPSIZE 1024
STACKSIZE 1024
EXETYPE WINDOWS
SEGMENTS
ENTERCODE MOVEABLE DISCARDABLE PRELOAD
EXPORTS
InitializeDLL @?
;
;
```

Step 3 – Create the Project File

The TopSpeed development center has used the PR extension for their project files. We must copy one of these files and rename it to use the PRJ extension instead, so that it will be recognized by the C4 IDE. We can use either CNVRULES.PR or CNVBETA.PR as a source. Then you make a few modifications with a text editor. The resulting STAB_CNV.PRJ looks like this:

```
-- TopSpeed Convertor Rules
#noedit
#system win
#model clarion dll
#pragma define(maincode=>off)
#pragma debug(vid=>full)
#compile "stab_cnv.clw"
#pragma link("c4cnveng.lib")
#link "stab_cnv.dll"
#run copyconv.bat
```

The only other change that I've made is to add the #run command. This executes a batch file to copy STAB_CNV.DLL to the BIN directory. (The *#file copy* command from the old TSE project system doesn't work anymore.)

Step 4 – Create your Source File

A good place to start for this is the CNVBETA.CLW, because it's *much* smaller than CNVRULES.CLW. There are a number of sections that you'll have to deal with.

Step 4a – Header Area

```
MEMBER()
INCLUDE('CNVENG.INC')
MAP
  InitializeDLL,NAME('InitializeDLL')
END
OwnerName EQUATE('BoxSoft SuperTemplates')
```

The MEMBER() statement (instead of PROGRAM) tells the system that we don't have a "Main" program module. This corresponds to the #pragma define(maincode=>off) in the project file.

CNVENG.INC contains the class declarations for the conversion engine.

InitializeDLL is the only routine to be exported as a callable procedure. This is a hook for C4CONV.EXE to call into the DLL. (I'm still not fully certain how the conversion wizard interfaces with my rule classes.)

OwnerName is the name for the conversion rule set. It will be displayed on a new button in the conversion wizard.

Step 4b – Rule Class Declarations

```
ChangeSuperTplClass CLASS(RuleCLass)
Construct             PROCEDURE
TakeSection           FUNCTION(SectionClass SectionMgr |
                              ,InfoTextClass Info |
                              ,STRING SectionHeader) |
                              ,BYTE,VIRTUAL
                    END
ChangeSuperProcCall CLASS(RuleCLass)
Construct             PROCEDURE
TakeSection           FUNCTION(SectionClass SectionMgr |
                              ,InfoTextClass Info |
                              ,STRING SectionHeader) |
                              ,BYTE,VIRTUAL
                    END
```

These are the two rules within our rule set. Each rule is responsible for performing one task. In our case, ChangeSuperTplClass is responsible for changing all of the template chains, and ChangeSuperProcCall is responsible for converting some procedure calls that have changed from the old templates.

All of your Rule classes declarations must contain at least the two methods shown. You can add extra properties and methods of your own, if required.

Step 4c – InitializeDLL Procedure

```
InitializeDLL PROCEDURE
  CODE
```

This is the empty hook procedure defined in our map.

Step 4d – Construct Methods

```
ChangeSuperTplClass.Construct PROCEDURE
  CODE
  SELF.Register(100,OwnerName,'Change SuperTemplate Chains'|
                ,'&Change Tpl Chain:','[COMMON][ADDITION][PROMPTS]')
!-----
ChangeSuperProcCall.Construct PROCEDURE
  CODE
  SELF.Register(230,OwnerName,'Change Procedure Calls' |
                ,'&Procedure Calls:','[SOURCE]')
```

These two Construct methods define the priority, owner, description, prompt, and applicable template sections for each rule.

The *priority* parameter controls the order in which the rules are applied. The best way to determine an appropriate priority is to look at existing conversion rules that do similar things.

The *owner* parameter ensures that the rules are grouped together in the conversion wizard.

The *description* parameter is used for display in various areas of the conversion wizard.

The *prompt* parameter controls the prompt within the group of rules. You should try to keep the highlighted letter unique for each rule in the group.

The *sections* parameter tells the conversion engine which template sections your rule needs to process. There are a number of benefits to this. The conversion process is faster, because each rule applies only to its appropriate sections. In addition, the rules can be simplified to handle the syntax found only within the specified sections. If you're not sure which sections to include, examine other rules that do similar things.

Step 4e – TakeSection Method

```
ChangeSuperTplClass.TakeSection FUNCTION(SectionClass SectionMgr|
                                        ,InfoTextClass Info|
                                        ,STRING SectionHeader)
cLine CSTRING(MaxLineLen),AUTO
i LONG(1)
StrQ     QUEUE
OldS       CSTRING(50)
NewS       CSTRING(50)
         END!QUEUE
  CODE
  DO BuildStrQ
  SELF.Buttons=Action:Apply
  Info.AddTitle('Template Name Changed: '&SectionHeader&' section')
  LOOP
    SectionMgr.GetLine(i,cLine)
    SELF.Lexer.TakeLine(cLine)
    IF SectionHeader='[COMMON]'
      IF SELF.Lexer.GetToken(1)='FROM'
        DO TryReplace
      END
    ELSIF SectionHeader='[ADDITION]'
      IF SELF.Lexer.GetToken(1)='NAME'
```

```
        DO TryReplace
      END
    ELSE
      DO TryReplace
    END
    IF SectionMgr.LineChanged(i,cLine)
      SectionMgr.SetLine(i,cLine)
    END
    i+=1
  WHILE i<=SectionMgr.GetLineCount()
  RETURN Level:Benign
!-------------------------------------
BuildStrQ ROUTINE
  StrQ.OldS = 'SuperSecurity'
  StrQ.NewS = 'SuperSecurityABC'
  ADD(StrQ); ASSERT(~ERRORCODE())
  !
  StrQ.OldS = 'MikeHanson'
  StrQ.NewS = 'MikeHansonABC'
  ADD(StrQ); ASSERT(~ERRORCODE())
  !
  StrQ.OldS = 'MHResize'
  StrQ.NewS = 'MikeHansonABC'
  ADD(StrQ); ASSERT(~ERRORCODE())
  !
  StrQ.OldS = 'SuperOddsAndEnds'
  StrQ.NewS = 'MikeHansonABC'
  ADD(StrQ); ASSERT(~ERRORCODE())
!-------------------------------------
TryReplace ROUTINE
  DATA
j BYTE,AUTO
k BYTE,AUTO
TokenStart USHORT,AUTO
TokenEnd USHORT,AUTO
  CODE
  LOOP j = 1 TO RECORDS(StrQ)
    GET(StrQ, j); ASSERT(~ERRORCODE())
    k = SELF.Lexer.FindToken(StrQ.OldS)
    IF k
      TokenStart = SELF.Lexer.GetStartChrPos(k)
      TokenEnd = SELF.Lexer.GetEndChrPos(k)
      cLine = SUB(cLine, 1, TokenStart-1) & StrQ.NewS |
                  & cLine[TokenEnd+1 : LEN(cLine)]
      SELF.Lexer.TakeLine(cLine)
      Info.AddLine(StrQ.OldS & ' template changed to ' |
                   & StrQ.NewS & ' template', i)
      BREAK
    END
  END
```

This method is responsible for changing the template chain. It's similar to Clarion's chain converter, except that it understands multiple template chains. First it loads a local queue with old chains and their new counterparts. Notice that I'm amalgamating several of my old public domain template chains into a single chain.

SELF.Buttons is used to specify which buttons are available when a potential change is being previewed in manual mode. Because this rule applies to internal template sections and not source code, some of the buttons (like UnComp) are not applicable.

Info.AddTitle defines the title describing the type of change being made. It is displayed in the information box in the bottom left corner of the modification preview window.

SectionMgr.GetLine requests the next line from the section. Then SELF.Lexer.TakeLine parses the line into tokens.

At this point, each of the section types is checked for the expected tokens. If it's the COMMON section, then the template chain name will be preceded by the word "FROM". If it's the ADDITION section, then the template chain name will be preceded by the word "NAME". In the PROMPTS section, it could be almost anywhere. If applicable, we DO TryReplace to change the template chain name.

If the routine is successful, SectionMgr.SetLine is called to remember the changes.

The TryReplace routine simply checks each queue entry, looking for old template chain names. If any are found, the token is replaced with the new chain name (via string splicing), and the resulting line is passed back into the Lexer class for re-parsing. To perform the slicing, our code uses methods from the Lexer class, including FindToken, GetStartChrPos, and GetEndCharPos. In addition, a descriptive line is added to explain what's happening.

I'm not going to list the TakeSection method for the other rule. It is quite similar, and you can always look at the source to see how it works.

Step 5 – Make the DLL

Now that you've got your PRJ, EXP and CLW, it's time to make your DLL. Load the PRJ and hit the lightning bolt. With any luck, you should have a new conversion DLL. Don't forget that the DLL needs to be in C:\CLARION4\BIN for it to work.

Step 6 – Update C4CONV.INI

Finally, you must update C:\CLARION4\BIN\C4CONV.INI so the conversion wizard knows about your new DLL. Just add another entry to the RuleDLLs section, as follows:

```
[RuleDLLs]
1=CNVRULES.DLL
2=CNVBETA.DLL
3=STAB_CNV.DLL
```

Conclusion

Well, that's all there is to it. Of course, these were very simple conversion rules. We didn't have to worry about parsing entire commands and substituting other commands with different parameters. I would think, however, that these rules would accommodate 99% of people with non-Clarion templates. For those of you with more complex needs, just peruse CNVRULES.CLW. It contains a plethora of rules for you to copy and modify.

The fantastic thing about all this is that the conversion wizard provides all of the support for the user interface, importing and exporting the TXAs, aborting changes, etc. We just add the specific intelligence for our rules, and it handles the rest. All in all, it's an impressive blend of power, simplicity and flexibility. Congratulations, TopSpeed, on a job well done!

Reader Comments

Add a comment