© The CodeBreakers-Journal, Vol.1, No.2. (2004)
http://www.CodeBreakers-Journal.com

Classes Restoration

Author: Hex

Original version @ XTiN.ORG, translated version @ http://www.apriorit.com

Abstract

Classes restoration is a complicated procedure which requires knowledge of OOP and the way this OOP is organized in specific compiler. Our task is to get class, its methods and members. Class restoration begins with looking for constructor, because here is the memory for object is being allocated and also we can gain some insight into constructor's components. This paper describes how to work with Classes restoration during Reverse Code Engineering processes.

Keywords: *Classes Restoration; Object Descriptors; Reverse Code Engineering*

Contents:

1. Introduction

Classes restoration is a complicated procedure which requires knowledge of OOP and the way this OOP is organized in specific compiler. Our task is to get class, its methods and members. Let’s begin with Delphi, because it’s relatively easy to find a class here. Class restoration begins with looking for constructor, because here is the memory for object is being allocated and also we can gain some insight into constructor’s components. It’s easy to find a constructor in Delphi – we just need to look for a string in which the class name occurs. For example, for TList the next structure can be found:

```
CODE:0040D598 TList           dd offset TList_VTBL
CODE:0040D59C                 dd 7 dup(0)
CODE:0040D5B8                 dd offset aTlist        ; "TList"
CODE:0040D5BC SizeOfObject    dd 10h
CODE:0040D5C0                 dd offset off_4010C8
CODE:0040D5C4                 dd offset TObject::SafeCallException
CODE:0040D5C8                 dd offset nullsub_8
CODE:0040D5CC                 dd offset TObject::NewInstance
CODE:0040D5D0                 dd offset TObject::FreeInstance
CODE:0040D5D4                 dd offset sub_40EA08
CODE:0040D5D8 TList_VTBL           dd offset TList::Grow
CODE:0040D5DC                 dd offset unknown_libname_107
CODE:0040D5E0 aTlist          db 5,'TList'
```

This is, if we can say so, an ‘object descriptor’. Pointer to it is being passed to the constructor. The constructor takes from it the data required for object creation. Using Xref on 40D598 we can find all the places where the constructor is being called. Here is an example of one of such calls:

```
CODE:0040E72E                 mov     eax, ds:TList
CODE:0040E733                 call    CreateClass
CODE:0040E738                 mov     ds:dword_4A45F8, eax
```

The constructor function we named by ourselves. We can determine whether it is really a CreateClass by the contents of the function:

```
CODE:00402F48 CreateClass     proc near               ; CODE XREF:
@BeginGlobalLoading+17p
CODE:00402F48                                         ;
@CollectionsEqual+48p ...

CODE:00402F48                 test    dl, dl
CODE:00402F4A                 jz      short loc_402F54
CODE:00402F4C                 add     esp, 0FFFFFFF0h
CODE:00402F4F                 call    __linkproc__ ClassCreate
CODE:00402F54
CODE:00402F54 loc_402F54:                             ; CODE XREF:
CreateClass+2j
CODE:00402F54                 test    dl, dl
CODE:00402F56                 jz      short locret_402F62
CODE:00402F58                 pop     large dword ptr fs:0

CODE:00402F5F                 add     esp, 0Ch
CODE:00402F62
CODE:00402F62 locret_402F62:                          ; CODE XREF:
CreateClass+Ej
CODE:00402F62                 retn
CODE:00402F62 CreateClass     endp
```

I.e., if there is __linkproc__ ClassCreate inside the function, it’s a constructor. Now we can look at how particularly the class creation happens:

```
CODE:00403200 __linkproc__ ClassCreate proc near       ; CODE XREF:
CreateClass+7p
CODE:00403200                                          ; sub_40AA58+Ap ...
CODE:00403200
CODE:00403200 arg_0           = dword ptr  10h
CODE:00403200
CODE:00403200                 push    edx
CODE:00403201                 push    ecx
CODE:00403202                 push    ebx
CODE:00403203                 call    dword ptr [eax-0Ch]
CODE:00403206                 xor     edx, edx
CODE:00403208                 lea     ecx, [esp+arg_0]
CODE:0040320C                 mov     ebx, fs:[edx]
CODE:0040320F                 mov     [ecx], ebx
CODE:00403211                 mov     [ecx+8], ebp
CODE:00403214                 mov     dword ptr [ecx+4], offset loc_403225
CODE:0040321B                 mov     [ecx+0Ch], eax
CODE:0040321E                 mov     fs:[edx], ecx
CODE:00403221                 pop     ebx
CODE:00403222                 pop     ecx
CODE:00403223                 pop     edx
CODE:00403224                 retn
CODE:00403224 __linkproc__ ClassCreate endp
```

So, the command

```
CODE:0040E72E mov eax, ds:TList
```

loads contents into EAX to the address of TList, i.e. it’s TList_VTBL. Since we use Delphi, here is the Borland’s convention of __fastcall is being used (parameters are being passed in the next order: EAX, EDX, ECX, stack...). It means that the pointer to the virtual methods table is being passed to the function CreateClass as a first parameter. Further EAX is not changing and gets into __linkproc__ClassCreate, and here we see:

```
CODE:00403203                 call    dword ptr [eax-0Ch]
```

Where is it going? The pointer to TList_VTBL=0x40D5D8 is still lying in EAX. 0x40D5D8-0xC=40D5CC, and this is

```
CODE:0040D5CC                 dd offset TObject::NewInstance
```

This is the ancestor’s constructor. So, TList is inherited by TObject. Let’s look what is in the depth:

```
CODE:00402F0C TObject::NewInstance proc near          ; DATA XREF:
CODE:004010FCo
CODE:00402F0C                                          ; CODE:004011DCo ...
CODE:00402F0C                 push    eax
CODE:00402F0D                 mov     eax, [eax-1Ch]
CODE:00402F10                 call    __linkproc__ GetMem
CODE:00402F15                 mov     edx, eax
CODE:00402F17                 pop     eax
CODE:00402F18                 jmp     TObject::InitInstance
CODE:00402F18 TObject::NewInstance endp
```

The value of EAX is the same, so 0x40D5D8-0x1C=0x40D5BC. Thus, the object size which is stored in 0x40D5BC, is being passed into GetMem

```
CODE:0040D5BC SizeOfObject    dd 10h
```

So, the total size of object members =0x10. The function TObject::InitInstance doesn't do anything special, it's just stuffs object members with zero and sets the value of pointer to VTBL in the just created instance of the object. Then the exit from CreateClass will happen and the pointer to the instance of the object will be returned into EAX.

That's why the call of constructors looks like:

```
CODE:0040E72E                 mov     eax, ds:TList
CODE:0040E733                 call    CreateClass
CODE:0040E738                 mov     ds:dword_4A45F8, eax
```

2. Restoration of the object structure

We have known the object size already. It's 0x10, where 0x4 bytes were taken by the pointer to VTBL. But there are 0xC bytes left and they contain object members, so we need to find them. Here an intuition is required. First of all, objects can't be created for no particular reason and members can be filled either in constructor (fully or partly) or after creating by Set-methods. Our TList in the constructor is being stuffed with zero through **rep stosd** (in TObject::InitInstance). So there is no info about class members in the constructor. Thus let's trace life cycle after the creation.

In our example the pointer to the instance of the class is being driven into global variable dword_4A45F8. So we can just set breakpoint on reading from dword_4A45F8 and look at how the object methods will be called. First event:

```
CODE:0041319D mov     eax, [ebp+var_4]
CODE:004131A0 mov     edx, ds:pTList
CODE:004131A6 mov     [eax+30h], edx  ; copied a pointer to the instance of
an object
CODE:004131A9 jmp     short loc_4131BD
.............
CODE:004131BD
CODE:004131BD loc_4131BD:                             ; CODE XREF:
sub_4130BC+EDj
CODE:004131BD xor     eax, eax
CODE:004131BF push    ebp
CODE:004131C0 push    offset loc_413276
CODE:004131C5 push    dword ptr fs:[eax]
CODE:004131C8 mov     fs:[eax], esp
CODE:004131CB mov     eax, [ebp+var_4]
CODE:004131CE mov     edx, [eax+18h]
CODE:004131D1 mov     eax, [ebp+var_4]
CODE:004131D4 mov     eax, [eax+30h] ;'implicit passing of a pointer to the
object itself'
CODE:004131D7 call    Classes::TList::Add(void *)
```

Now look into Classes::TList::Add:

```
CODE:0040EA28 __fastcall Classes::TList::Add(void *) proc near
CODE:0040EA28                                         ; CODE XREF:
@RegisterClass+9Bp
CODE:0040EA28                                         ;
@RegisterIntegerConsts+20p ...
CODE:0040EA28 push    ebx
CODE:0040EA29 push    esi
CODE:0040EA2A push    edi
CODE:0040EA2B mov     edi, edx
CODE:0040EA2D mov     ebx, eax ; a kind of This
CODE:0040EA2F mov     esi, [ebx+8] ; addressing to the object member №1
CODE:0040EA32 cmp     esi, [ebx+0Ch] ; addressing to the object member №3
CODE:0040EA35 jnz     short loc_40EA3D
CODE:0040EA37 mov     eax, ebx
CODE:0040EA39 mov     edx, [eax] ;addressing to TList->pVTBL
CODE:0040EA3B call    dword ptr [edx]
CODE:0040EA3D
CODE:0040EA3D loc_40EA3D:                             ; CODE XREF:
Classes::TList::Add(void *)+Dj
CODE:0040EA3D mov     eax, [ebx+4] ; addressing to the object member №2
CODE:0040EA40 mov     [eax+esi*4], edi
CODE:0040EA43 inc     dword ptr [ebx+8]
CODE:0040EA46 mov     eax, esi
CODE:0040EA48 pop     edi
CODE:0040EA49 pop     esi
CODE:0040EA4A pop     ebx
CODE:0040EA4B retn
CODE:0040EA4B __fastcall Classes::TList::Add(void *) endp
```

That is… 3 last members have been found. All of them are of 4 bytes size. To simplify the work with classes in IDA Pro we use structures. Classes are the same structures, anyway: After using the next structure:

```
00000000 TList_obj struc ; (sizeof=0X10)
00000000 pVTBL dd ?
00000004 Property1 dd ?
00000008 Property2 dd ?
0000000C Property3 dd ?
00000010 TList_obj ends
```

things become more clear:

```
CODE:0040EA28 __fastcall Classes::TList::Add(void *) proc near
CODE:0040EA28                                         ; CODE XREF:
@RegisterClass+9Bp
CODE:0040EA28                                         ;
@RegisterIntegerConsts+20p ...
CODE:0040EA28 push    ebx
CODE:0040EA29 push    esi
CODE:0040EA2A push    edi
CODE:0040EA2B mov     edi, edx
CODE:0040EA2D mov     ebx, eax
CODE:0040EA2F mov     esi, [ebx+TList_obj.Property2]
CODE:0040EA32 cmp     esi, [ebx+TList_obj.Property3]
CODE:0040EA35 jnz     short loc_40EA3D
CODE:0040EA37 mov     eax, ebx
CODE:0040EA39 mov     edx, [eax+TList_obj.pVTBL]
CODE:0040EA3B call    dword ptr [edx] ;TList::Grow
CODE:0040EA3D
CODE:0040EA3D loc_40EA3D:                             ; CODE XREF:
Classes::TList::Add(void *)+Dj
CODE:0040EA3D mov     eax, [ebx+TList_obj.Property1]
CODE:0040EA40 mov     [eax+esi*4], edi
CODE:0040EA43 inc     [ebx+TList_obj.Property2]
CODE:0040EA46 mov     eax, esi
CODE:0040EA48 pop     edi
CODE:0040EA49 pop     esi
CODE:0040EA4A pop     ebx
CODE:0040EA4B retn
CODE:0040EA4B __fastcall Classes::TList::Add(void *) endp
```

Think of VBTL look and it will be easy to guess that:

```
CODE:0040EA3B call    dword ptr [edx]
```

is TList::Grow, because

```
CODE:0040D5D8 pVTBL dd offset TList::Grow
```

Now we can make a deeper analyze of the class members. For example, if we have a look at the next code:

```
CODE:0040EA3D mov     eax, [ebx+TList_obj.Property1]
CODE:0040EA40 mov     [eax+esi*4], edi
CODE:0040EA43 inc     [ebx+TList_obj.Property2]
```

we can say that Property2 is a counter for the list elements, because it increases when an element is added. And Property1 is the pointer to the array of list elements. Property 2 in this array is an index. Property 3 is the maximum number of the elements in a list, as method TList::Grow is being called just when Property2==Property3. We found out this by using logic. Now, when all is clear, we may look in Help and give names to the members:

```
CODE:0040EA28 __fastcall Classes::TList::Add(void *) proc near
CODE:0040EA28                                         ; CODE XREF:
@RegisterClass+9Bp
CODE:0040EA28                                         ;
@RegisterIntegerConsts+20p ...
CODE:0040EA28                 push    ebx
CODE:0040EA29                 push    esi
CODE:0040EA2A                 push    edi
CODE:0040EA2B                 mov     edi, edx
CODE:0040EA2D                 mov     ebx, eax
CODE:0040EA2F                 mov     esi, [ebx+TList_obj.Count]
CODE:0040EA32                 cmp     esi, [ebx+TList_obj.Capacity]
CODE:0040EA35                 jnz     short loc_40EA3D
CODE:0040EA37                 mov     eax, ebx
CODE:0040EA39                 mov     edx, [eax+TList_obj.pVTBL]
CODE:0040EA3B                 call    dword ptr [edx]
CODE:0040EA3D
CODE:0040EA3D loc_40EA3D:                             ; CODE XREF:
Classes::TList::Add(void *)+Dj
CODE:0040EA3D                 mov     eax, [ebx+TList_obj.Items]
CODE:0040EA40                 mov     [eax+esi*4], edi
CODE:0040EA43                 inc     [ebx+TList_obj.Count]
CODE:0040EA46                 mov     eax, esi

CODE:0040EA48                 pop     edi
CODE:0040EA49                 pop     esi
CODE:0040EA4A                 pop     ebx
CODE:0040EA4B                 retn
CODE:0040EA4B __fastcall Classes::TList::Add(void *) endp
```

We have restored the structure, let's look into the class methods.

3. Looking for the class methods

Methods can be: public/private (protected), virtual/non-virtual and static.
Static methods can't be found because after the compilation was made they look like common procedures. Affiliation of such function with a specific class is also impossible to determine. But is there a sense in such search? If the function is called somewhere in the class methods, it, anyway, will be viewed while the code is being extracted. Otherwise, it is wasting of time. Virtual functions are easy to find to– they all are in VTBL. But how we should look for non-virtual ones? Let's think of OOP: when the object methods are called, the pointer to the object itself is implicitly passed to them. In fact, it means that each method accepts the pointer to the object as its first parameter. I.e., if the method was declared as __fastcall, the pointer to the object will be pushed into EAX. But for __cdecl or __stdcall methods it's the first parameter in the stack. Let's look on where is the pointer to the object is stored…absolutely right! In dword_4A45F8. On XREF to 4A45F8 we can find lots of non-virtual methods. Further we can set a breakpoint on 4A45F8 and trace the copying of a pointer to the instance to find where else the call of methods can take place. All is easy in our example, because global variable is used. But what we should do, if the local variable is used or if the code can't be executed (for example, we research driver's code or the code is not allowed for execution)? Here we need a specific method.

Step-by-step:

1) We have to find all the points of constructor's calls.

For each call:

2) Trace where the pointer to the instance of an object is being written (local variable)
3) Looking through the function which has called the constructor for all the calls of the object methods
4) If there are no such calls, look at the next call of the constructor, otherwise look for all xref to the method that had been found. In such way we can find calls that are not beside the constructor. And, as we know that the first parameter is the pointer to an object, we can go to each **xref** and look where else the pointer to an object was used. And in such way we are going up the levels of the code, till we reach a deadlock or the method that had been found.
5) Reviewing the next method that had been found

For example, we have found Classes::TList::Add method. On one of the Xref we find Classes::TList::Add method here:

```
CODE:0040F020 TThreadList::Add proc near              ; CODE XREF:
TCanvas::`...'+9Ep
CODE:0040F020                                         ;
Graphics::_16725+C4p
CODE:0040F020
CODE:0040F020 var_4           = dword ptr -4
CODE:0040F020
CODE:0040F020                 push    ebp
CODE:0040F021                 mov     ebp, esp
CODE:0040F023                 push    ecx
CODE:0040F024                 push    ebx
CODE:0040F025                 mov     ebx, edx
CODE:0040F027                 mov     [ebp+var_4], eax
CODE:0040F02A                 mov     eax, [ebp+var_4]
CODE:0040F02D                 call    TThreadList::LockList
CODE:0040F032                 xor     eax, eax
CODE:0040F034                 push    ebp
CODE:0040F035                 push    offset loc_40F073
CODE:0040F03A                 push    dword ptr fs:[eax]
CODE:0040F03D                 mov     fs:[eax], esp
CODE:0040F040                 mov     eax, [ebp+var_4]
CODE:0040F043                 mov     eax, [eax+4]
CODE:0040F046                 mov     edx, ebx
CODE:0040F048                 call    TList::IndexOf
CODE:0040F04D                 inc     eax
CODE:0040F04E                 jnz     short loc_40F05D
CODE:0040F050                 mov     eax, [ebp+var_4]
CODE:0040F053                 mov     eax, [eax+4]
CODE:0040F056                 mov     edx, ebx
CODE:0040F058                 call    Classes::TList::Add(void *)
```

I.e. we have found TList::IndexOf method.

Further we see that we are in the method of TthreadList object and TList is its member. Here we have nothing to look at. Let's assume that there are no more xref to Classes::TList::Add. Go in TList::IndexOf method and look at its xref. One of them directs us here:

```
CODE:0040EE38 TList::Remove   proc near               ; CODE XREF:
TThreadList::Remove+28p

CODE:0040EE38                                         ;
TCollection::RemoveItem+Bp ...
CODE:0040EE38                 push    ebx
CODE:0040EE39                 push    esi
CODE:0040EE3A                 mov     ebx, eax
CODE:0040EE3C                 mov     eax, ebx
CODE:0040EE3E                 call    TList::IndexOf
CODE:0040EE43                 mov     esi, eax
CODE:0040EE45                 cmp     esi, 0FFFFFFFFh
CODE:0040EE48                 jz      short loc_40EE53
CODE:0040EE4A                 mov     edx, esi
CODE:0040EE4C                 mov     eax, ebx
CODE:0040EE4E                 call    TList::Delete
CODE:0040EE53
CODE:0040EE53 loc_40EE53:                             ; CODE XREF:
TList::Remove+10j
CODE:0040EE53                 mov     eax, esi
CODE:0040EE55                 pop     esi
CODE:0040EE56                 pop     ebx
CODE:0040EE57                 retn
CODE:0040EE57 TList::Remove   endp
```

So, TList::Delete and TList::Remove are found. And so forth for all xref and variables that contain a pointer to the instance of a class. Here is an example of looking through the variable:

```
CODE:0041319D mov     eax, [ebp+var_4]
CODE:004131A0 mov     edx, ds:pTList
CODE:004131A6 mov     [eax+30h], edx  ;a pointer to the instance of an
object is being copied
CODE:004131A9 jmp     short loc_4131BD
```

We see below:

```
CODE:00413236 mov     eax, [eax+30h]
CODE:00413239 mov     edx, [ebp+var_10]
CODE:0041323C call    TList::Get
```

How we can identify public or private methods? We can try to do that only when all the set of methods is found. Private methods are called only inside the other object methods. I.e. we should look at xref. While looking for methods we advise to number them first. It means as you find the method, you name it Object1::Method1, Object1::Method2 and so on, and when all the methods are found you may begin restoration of type and number of elements.

4. Determination of the number of method arguments

For __cdecl и __stdcall there are few things to tell about, you just need to look on how much of them have IDA found and subtract the 1 (i.e. the 1 is a pointer to the instance of an object, and others are method arguments). There are more complications for __fastcall. First we need to remember the sequence order of arguments: EAX,EDX,ECX,stack. The analyze begins with how much arguments that had been transmitted via stack does IDA have counted. If there are at least one, we add to it 3 (3 register's plus the ones for stack). As first argument is allocated for This, we need to subtract the 1 from the number. The summary value is the net number of arguments. If there are no stack arguments, we look at the beginning of the function. Delphi tries not to spoil arguments values, so each __fastcall function begins with copying from registers EAX, EDX and ECX in such way:

```
mov esi, edx ; first parameter
mov ebx, eax ; pThis
mov edi, ecx ; second parameter
```

Depending on the number of registers that are being copied, one can conclude what is the number of arguments. For example:

```
CODE:0040EBE0 TList::Get      proc near              ; CODE XREF:
@GetClass+1Dp
CODE:0040EBE0                                        ;
@UnRegisterModuleClasses+24p ...
CODE:0040EBE0
CODE:0040EBE0 var_4           = dword ptr -4
CODE:0040EBE0
CODE:0040EBE0                 push    ebp
CODE:0040EBE1                 mov     ebp, esp
CODE:0040EBE3                 push    0
CODE:0040EBE5                 push    ebx
CODE:0040EBE6                 push    esi
CODE:0040EBE7                 mov     esi, edx
CODE:0040EBE9                 mov     ebx, eax
CODE:0040EBEB                 xor     eax, eax
```

There are 2 arguments, 1 of them is pThis, thus TList::Get has 1 argument.

```
CODE:004198CC                 push    ebp
CODE:004198CD                 mov     ebp, esp
CODE:004198CF                 add     esp, 0FFFFFF8Ch
CODE:004198D2                 push    ebx
CODE:004198D3                 push    esi
CODE:004198D4                 push    edi
CODE:004198D5                 mov     [ebp+var_C], ecx
CODE:004198D8                 mov     [ebp+var_8], edx
CODE:004198DB                 mov     [ebp+var_4], eax
```

There are 3 arguments, 1 of them is for pThis, so total is 2 arguments. We should remind you that we restore the number of arguments in initial method which is described in Delphi, and in IDA, naturally, while declaring the function type we should write all the arguments in consideration with This. Types of arguments try to determine on your own.