Reverse Engineering

3. Analysis of C++ Classes

Ing. Tomáš Zahradnický, EUR ING, Ph.D.
Ing. Josef Kokeš

FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

Czech Technical University in Prague
Faculty of Information Technology
Department of Computer Systems

Version 2019-09-15

Table of Contents

Classes and Structures

The class and struct keywords mean the same thing, the only difference is in their default protection. While class has its content private by default, struct has it public. Thus we can write equivalently:

```
class C {
  public:
     int m_Field;
};
```

```
struct C {
  int m_Field;
};
```

```
class C {
  int m_Field;
};
```

```
struct C {
  private:
    int m_Field;
};
```

Structures

Let's start with a sample `struct` and observe its memory layout:

A Sample Struct

```
struct SampleStruct {
  char  f_Char;
  short f_Short;
  int   f_Int;

  SampleStruct()
    : f_Char(0x11),
      f_Short(0x55AA),
      f_Int(0x12345678)
  {
  }
};

SampleStruct g_Struct[4];
```

The Sample Struct's Memory Layout

```
           f_Char
            | padding
            |  |   f_Short
            |  |   |         f_Int
            |  |   |         |
           -- -- ----- -----------
0x0012FEE0 11 00 AA 55 78 56 34 12   ...UxV4.
0x0012FEE8 11 31 AA 55 78 56 34 12   .1.UxV4.
0x0012FEF0 11 F3 AA 55 78 56 34 12   ...UxV4.
0x0012FEF8 11 9A AA 55 78 56 34 12   ...UxV4.
```

As you can see, all fields in the structure are aligned according to the ABI alignment rules. This is why the `f_Short` field is aligned to a 2-byte boundary. Why is padding `0x00`, then `0x31`, `0xF3`, and `0x9A`?

Structure size determination

The structure can be allocated either on the stack or on the heap. Stack-allocated structures are allocated with the `sub esp, __LOCAL_SIZE` instruction and their size is a `__LOCAL_SIZE` component. If a structure is allocated at runtime on the heap, it is allocated with `new` keyword or a memory allocation function such as `malloc`, `HeapAlloc`, or `LocalAlloc`. These functions take structure size as their argument.

Allocating a structure with `malloc`

```
0x80483c2  <main+18>:  sub   $0xc,%esp
0x80483c5  <main+21>:  push  $0x8                            // Structure size
0x80483c7  <main+23>:  call  0x8048370 <malloc@plt>  // EAX := ptr to the struct
0x80483cc  <main+28>:  movb  $0x11,(%eax)                    // ptr->f_Char=0x11
0x80483cf  <main+31>:  movw  $0x55AA,0x2(%eax)               // ptr->f_Short=0x55AA
0x80483d5  <main+37>:  movl  $0x12345678,0x4(%eax)           // ptr->f_Int=0x12345678
0x80483dc  <main+44>:  add   $0x10,%esp
```

Structure size determination II

Allocation with the new operator

Allocation with the new operator is similar to the previous case, except that a constructor is called for the structure, if one exists. This case is shown below:

Allocating a structure with the new operator

```
0x4006d1   <main(int, const char**)+75>:       mov   $0x8,%edi            // Structure size
0x4006d6   <main(int, const char**)+80>:       callq 0x400580 <_Znwm@plt>// Call new here
0x4006db   <main(int, const char**)+85>:       mov   %rax,%rbx            // Store this pointer into RBX
0x4006de   <main(int, const char**)+88>:       mov   %rbx,%rdi            // Copy this ptr into RDI
0x4006e1   <main(int, const char**)+91>:       callq 0x400752 <SampleStruct::SampleStruct()>

0x400752   <SampleStruct::SampleStruct()>:     push  %rbp
0x400753   <SampleStruct::SampleStruct()+1>:   mov   %rsp,%rbp
0x400756   <SampleStruct::SampleStruct()+4>:   mov   %rdi,-0x8(%rbp)
0x40075a   <SampleStruct::SampleStruct()+8>:   mov   -0x8(%rbp),%rax
0x40075e   <SampleStruct::SampleStruct()+12>:  movb  $0x11,(%rax)         // ptr->f_Char=0x11
0x400761   <SampleStruct::SampleStruct()+15>:  mov   -0x8(%rbp),%rax
0x400765   <SampleStruct::SampleStruct()+19>:  movw  $0x55aa,0x2(%rax)    // ptr->f_Short=0x55AA
0x40076b   <SampleStruct::SampleStruct()+25>:  mov   -0x8(%rbp),%rax
0x40076f   <SampleStruct::SampleStruct()+29>:  movl  $0x12345678,0x4(%rax)// ptr->f_Int=0x12345678
0x400776   <SampleStruct::SampleStruct()+36>:  pop   %rbp
0x400777   <SampleStruct::SampleStruct()+37>:  retq
```

Structure size determination III

Other methods

Structure size can also be determined from the instructions manipulating with the structure. If the structure is copied or an array of structures of the same type is used, instructions such as `mul` often contain the size of our structure. E.g.:

```
for( i=0; i<N; ++i )
  pStructureArray[i].f_Field=0;
```

Suboptimal with `mul`

```
  xor ecx, ecx              // i=0
  mov esi, dword ptr [ebp-8]// ESI := array base
loop:
  mov eax, ecx
  mul STRUCTURE_SIZE
  mov edi, esi
  add edi, eax              // EDI = ESI + size*i
  mov byte ptr [edi], 0ffh
  inc ecx
  cmp ecx, 100
  jb loop
```

Better with `add`

```
  mov $0x600ec8,%rax // Array base
loop:
  movb $0x11,(%rax)
  add $0xc,%rax      // Move to the next item
  cmp $0x603da8,%rax // Array end
  jne loop
```

Inheritance I

A structured data type can contain another structured data type as its member. There is nothing special about this. Structured data types can inherit from other structured data types. Let's see what happens when a multiple inheritance is used. Suppose we have a `class` inheriting from `CUnknown` (see the Polymorphism section) and from another `class Rect`. The base class `CUnknown` implements reference counting, while `Rect` contains 4 `int`s. Together they form the `CRefcountedRect` class.

```
typedef struct Rect {
  int f_X, f_Y, f_Width, f_Height;
} Rect, *RectPtr, **RectHandle;

class CRefcountedRect :  public CUnknown, public Rect {
  public:
    virtual HRESULT STDMETHODCALLTYPE QueryInterface( REFIID riid, LPVOID* ppvObject );
                                      CRefcountedRect(int x, int y, int w, int h)
                                       : m_Area(w*h) {
                                          f_X = x; f_Y = y; f_Width = w; f_Height = h;
                                      };
    virtual                           ~CRefcountedRect();
    virtual int                       get_Area() const { return m_Area; }
    const Rect*                       get_Rect() const { return static_cast<const Rect*>(this); }

  protected:
    int m_Area;
};
```

Inheritance II

Memory layout of the CRefcountedRect

```
                 CRefcountedRect*-|   pVMT (see later)
7ffffffe0c0: CUnknown*---------+-->d0 0c 40 00 00 00 00 00
                                   m_RefCount
7ffffffe0c8:                       01 00 00 00 00 00 00 00

                                   f_X         f_Y
7ffffffe0d0: Rect*---------------->10 00 00 00 20 00 00 00
                                   f_Width     f_Height
7ffffffe0d8:                       40 00 00 00 80 00 00 00
                                   f_Area
7ffffffe0e0:                       00 20 00 00
```

Polymorphism I

Once a structured data type contains virtual methods, it must contain a virtual method table (VMT). If the object inherits from multiple objects with virtual methods, it inherits two or more VMTs. Each VMT is a table of function pointers to virtual methods; all virtual methods are called via this table. VMT is stored as the first item in the class, followed by first base class data. Then the second VMT follows, second data, etc.

```
#include <Unknwn.h>
class CUnknown : public IUnknown {
  public:
    virtual HRESULT STDMETHODCALLTYPE QueryInterface( REFIID riid, LPVOID* ppvObject );
    virtual ULONG   STDMETHODCALLTYPE AddRef(void);
    virtual ULONG   STDMETHODCALLTYPE Release(void);

    CUnknown() : m_RefCount(1) {}
    virtual ~CUnknown() { m_RefCount = -1; }

  protected:
    volatile ULONG __declspec(align(8)) m_RefCount;
};

ULONG STDMETHODCALLTYPE CUnknown::AddRef(void) {
  return InterlockedIncrement(&m_RefCount);
}

ULONG STDMETHODCALLTYPE CUnknown::Release(void) {
  ULONG ulNewValue = InterlockedDecrement(&m_RefCount);
  if (ulNewValue == 0)
    delete this;
  return ulNewValue;
}
```

Polymorphism II

Once a structured data type contains virtual methods, it must contain a virtual method table (VMT). If the object inherits from multiple objects with virtual methods, it inherits two or more VMTs. Each VMT is a table of function pointers to virtual methods; all virtual methods are called via this table. VMT is stored as the first item in the class, followed by first base class data. Then the second VMT follows, second data, etc.

```
#include <Unknwn.h>
class CUnknown : public IUnknown {
  public:
    virtual HRESULT STDMETHODCALLTYPE QueryInterface( REFIID riid, LPVOID* ppvObject );
    virtual ULONG   STDMETHODCALLTYPE AddRef(void);
    virtual ULONG   STDMETHODCALLTYPE Release(void);

    CUnknown() : m_RefCount(1) {}
    virtual ~CUnknown() { m_RefCount = -1; }

  protected:
    volatile ULONG __declspec(align(8)) m_RefCount;
};

ULONG STDMETHODCALLTYPE CUnknown::AddRef(void) {
  return InterlockedIncrement(&m_RefCount);
}

ULONG STDMETHODCALLTYPE CUnknown::Release(void) {
  ULONG ulNewValue = InterlockedDecrement(&m_RefCount);
  if (ulNewValue == 0)
    delete this;
  return ulNewValue;
}
```

VMT

```
&CUnknown::QueryInterface
&CUnknown::AddRef
&CUnknown::Release
&CUnknown::~CUnknown
```

Polymorphism III

The layout of the `CUnknown` class would then be:

Object layout

```
this --> +0 pVMT -------> +0 &CUnknown::QueryInterface
         +4 padding       +4 &CUnknown::AddRef
         +8 m_RefCount    +8 &CUnknown::Release
                          +c &CUnknown::~CUnknown
```

Calling a method through a VMT

```
40245C mov  eax,dword ptr [ebp+8]// Load this into EAX
40245F mov  ecx,dword ptr [eax]  // Load this->pVMT into ECX
402461 mov  edx,dword ptr [ebp+8]// Load this into EDX
402464 push edx                  // Push this as the first arg.
402465 mov  eax,dword ptr [ecx+4]// Retrieve the method from VMT [AddRef]
402468 call eax                  // Call AddRef
```

Note 1: You might have noticed that the `AddRef` method does not receive its argument in `ECX`, contrary to what we might have expected. This is because the `STDMETHODCALLTYPE` changes the calling convention to `__stdcall` and this means all parameters are pushed onto the stack. Even for methods!

Note 2: The class contains a padding in front of the `m_RefCount` field because of the `__declspec(align(8))`. This is required because the InterlockedXXX functions require aligned data.

Setting object's VMT (Windows)

```
CUnknown::CUnknown():
  00401110 push ebp
  00401111 mov  ebp,esp
  00401113 sub  esp,44h
  00401116 push ebx
  00401117 push esi
  00401118 push edi
  00401119 mov  dword ptr [ebp-4],ecx             // this was passed in ECX
  0040111C mov  ecx,dword ptr [ebp-4]
  0040111F call IUnknown::IUnknown (4011D0h)      // An implicit constructor
  00401124 mov  eax,dword ptr [ebp-4]
  // Assign the VMT pointer to this->pVMT offset
  00401127 mov  dword ptr [eax],offset CUnknown::'vftable' (4293BCh)
  0040112D mov  eax,dword ptr [ebp-4]
  00401130 mov  dword ptr [eax+8],1               // Set m_RefCount=1
  00401137 mov  eax,dword ptr [ebp-4]
  0040113A pop  edi
  0040113B pop  esi
  0040113C pop  ebx
  0040113D mov  esp,ebp
  0040113F pop  ebp
  00401140 ret
```

CUnknown's VMT

```
004293BC  80 22 40 00 c0 14 40 00  ."@.A.@.
004293C4  20 23 40 00 10 14 40 00   #@...@.
```

Polymorphism IV

If a class is abstract and has some methods defined, it still has a VMT. Its constructor (even an implicit one) assigns it to the `this->pVMT` field. If some of the virtual methods are not implemented, what should their slot in the VMT contain?

```
#include <Unknwn.h>
class CUnknown : public IUnknown {
  public:
    virtual HRESULT STDMETHODCALLTYPE QueryInterface( REFIID riid, LPVOID* ppvObject );
    virtual ULONG   STDMETHODCALLTYPE AddRef(void);
    virtual ULONG   STDMETHODCALLTYPE Release(void);

    CUnknown() { m_RefCount = 1; }
    virtual ~CUnknown() { m_RefCount = -1; }

    protected:
      volatile ULONG __declspec(align(8)) m_RefCount;
};

// Implementations of AddRef and Release remain the same
```

Polymorphism V

If a class is abstract and has some methods defined, it still has a VMT. Its constructor (even an implicit one) assigns it to the `this->pVMT` field. If some of the virtual methods are not implemented, what should their slot in the VMT contain?

```
#include <Unknwn.h>
class CUnknown : public IUnknown {
  public:
    virtual HRESULT STDMETHODCALLTYPE QueryInterface(
    virtual ULONG   STDMETHODCALLTYPE AddRef(void);
    virtual ULONG   STDMETHODCALLTYPE Release(void);

    CUnknown() { m_RefCount = 1; }
    virtual ~CUnknown() { m_RefCount = -1; }

    protected:
      volatile ULONG __declspec(align(8)) m_RefCount;
};

// Implementations of AddRef and Release remain the same
```

Original VMT

```
&CUnknown::QueryInterface
&CUnknown::AddRef
&CUnknown::Release
&CUnknown::~CUnknown
```

Polymorphism VI

If a class is abstract and has some methods defined, it still has a VMT. Its constructor (even an implicit one) assigns it to the `this->pVMT` field. If some of the virtual methods are not implemented, what should their slot in the VMT contain?

```
#include <Unknwn.h>
class CUnknown : public IUnknown {
  public:
    // QueryInterface is left undefined
    virtual ULONG   STDMETHODCALLTYPE AddRef(void);
    virtual ULONG   STDMETHODCALLTYPE Release(void);

    // Constructor is left undefined
    virtual ~CUnknown() { m_RefCount = -1; }

    protected:
      volatile ULONG __declspec(align(8)) m_RefCount;
};

// Implementations of AddRef and Release remain the same
```

VMT with pure methods

```
&_purecall
&CUnknown::AddRef
&CUnknown::Release
&CUnknown::~CUnknown
```

Setting object's VMT (Linux 64-bit)

```
IUnknown::IUnknown():                                // An implicit constructor
  00400fdc push   %rbp
  00400fdd mov    %rsp,%rbp
  00400fe0 mov    %rdi,-0x8(%rbp)                   // this pointer stored in a local var
  00400fe4 mov    -0x8(%rbp),%rax                   // this pointer to RAX
  00400fe8 movq   $0x4014d0,(%rax)                  // VMT assigned to this->pVMT
  00400fef pop    %rbp
  00400ff0 retq
```

```
CUnknown::CUnknown():                                // An implicit constructor
  00400fee push   %rbp
  00400fef mov    %rsp,%rbp
  00400ff2 sub    $0x10,%rsp
  00400ff6 mov    %rdi,-0x8(%rbp)
  00400ffa mov    -0x8(%rbp),%rax
  00400ffe mov    %rax,%rdi                         // this passed in RDI
  00401001 callq  0x400fd8 <IUnknown::IUnknown()>   // Implicit constructor called here
  00401006 mov    -0x8(%rbp),%rax
  0040100a movq   $0x401490,(%rax)                  // VMT assigned to this->pVMT
  00401011 leaveq
  00401012 retq
```

IUnknown's VMT

```
00000000004014d0 <_ZTV8IUnknown+16>:
 +0    0000000000400840 <__cxa_pure_virtual@plt>
 +8    0000000000400840 <__cxa_pure_virtual@plt>
 +10   0000000000400840 <__cxa_pure_virtual@plt>
 +18   0000000000000000 empty
```

CUnknown's VMT

```
0000000000401490 <_ZTV8CUnknown+16>:
 +0    0000000000400840 <__cxa_pure_virtual@plt>
 +8    0000000000400976 <CUnknown::AddRef()>
 +10   0000000000400994 <CUnknown::Release()>
 +18   0000000000000000 empty
```

Polymorphism VII

Pure virtual methods' addresses in the VMT are replaced by the address of the `_purecall` function. This function calls a purecall handler if one's present and then aborts the program if not already aborted by the handler.

Source code of the `_purecall` function from `purevirt.c`

```
void __cdecl _purecall( void ) {
  _purecall_handler purecall = (_purecall_handler) DecodePointer(__pPurecall);
  if( purecall != NULL )
  {
    purecall();
    /* shouldn't return, but if it does, we drop back to default behaviour */
  }
  #if defined (_DEBUG)
    _NMSG_WRITE(_RT_PUREVIRT);
  #endif /* defined (_DEBUG) */
  /* do not write the abort message */
  _set_abort_behavior(0, _WRITE_ABORT_MSG);
  abort();
}
```

Virtual Method Tables in Reverse Engineering

As you have noticed, constructors set the object's VMT if one's present. This happens even if there's no constructor. In that case an implicit constructor is used (e.g. see the `IUnknown` ctor). Since each VMT is a global table shared by all instances of the same object, we can:

- use a VMT pointer to tell whether an unknown object is of a certain type by comparing its VMT pointer to a list of VMT pointers;
- examine pointers in the object's memory; if they point to a VMT, we've discovered multiple inheritance;
- examine pointers in each VMT and identify code belonging to the object.

The above information can be further extended by studying the object's type information, which is used in RTTI.

Motivation I

Slide no. 9 uses `static_cast<const Rect*>(this)` to cast the `this` pointer into a pointer to a `Rect`. The `Rect*` pointer is also a "`this`" pointer of a derived class and is different from the `CRefcountedRect*` `this` pointer. What code was behind the `static_cast`?

```
CRefcountedRect::get_Rect() const:
  004027C0  push  ebp
  004027C1  mov   ebp,esp
  004027C3  sub   esp,48h
  004027C6  push  ebx
  004027C7  push  esi
  004027C8  push  edi
  004027C9  mov   dword ptr [ebp-4],ecx                  // Store this into ebp-4
  004027CC  cmp   dword ptr [ebp-4],0                    // Static_cast does nothing for a NULL pointer
  004027D0  je    CRefcountedRect::get_Rect+1Dh (4027DDh)
  004027D2  mov   eax,dword ptr [ebp-4]                  // Load this into EAX
  004027D5  add   eax,10h                                // Move the this pointer by 16 bytes
  004027D8  mov   dword ptr [ebp-48h],eax                // Store the casted result
  004027DB  jmp   CRefcountedRect::get_Rect+24h (4027E4h)
  004027DD  mov   dword ptr [ebp-48h],0                  // Static_cast failed, store NULL as the result
  004027E4  mov   eax,dword ptr [ebp-48h]
  004027E7  pop   edi
  004027E8  pop   esi
  004027E9  pop   ebx
  004027EA  mov   esp,ebp
  004027EC  pop   ebp
  004027ED  ret
```

Motivation II

How do we go back from `Rect*` to `CRefcountedRect*`? When we want to upcast an object pointer, we can try `dynamic_cast`:

```
const Rect* pRect = pRefcountedRect->get_Rect();
const CRefcountedRect* pRefCountedRect2 = dynamic_cast<const CRefcountedRect*>(pRect);
printf("pRefcountedRect=%p\npRect=%p\npRefcountedRect2=%p\n", pRefcountedRect, pRect, pRefCountedRect2);
```

```
1>c:\users\...\tokens.cpp(892): error C2683: 'dynamic_cast' : 'Rect' is not a polymorphic type
```

Upcasting was not possible, since `Rect` was not polymorphic, i.e. did not have a VMT. We can add one by adding a `virtual` destructor:

```
typedef struct Rect {
  int f_X, f_Y, f_Width, f_Height;
  virtual ~Rect();
} Rect, *RectPtr, **RectHandle;
```

The result

```
pRefcountedRect  = 0012FE98
pRect            = 0012FEA8
pRefCountedRect2 = 0012FE98
```

How did `dynamic_cast` know that it was possible to cast this pointer to a `Rect` into a pointer to `CRefcountedRect`?

The typeid operator I

Run Time Type Identification (RTTI) is used when typeid or dynamic_cast operators are used. Let's see what typeid provides to us:

```
const std::type_info& rtiCRefcountedRect = typeid(CRefcountedRect);

// Copy type_info reference into the rti_CRefcountedRect local
00402CA7  mov        dword ptr [ebp-0B4h],offset CRefcountedRect 'RTTI Type Descriptor' (432110h)

rtiCRefcountedRect:
  __vfptr   0x004295c4                          const type_info::'vftable'* // A pointer to the VMT
  _M_data   0x00000000                          void *
  _M_d_name 0x00432118 ".?AVCRefcountedRect@@" char [1]                     // A var-sized mangled class name
```

The type_info class is defined in typeinfo.h as follows:

```
class type_info {
  public:
  ...
  private:
    void *_M_data;     // What is this?
    char _M_d_name[1]; // A variable sized mangled name
};
```

There's a class name, a valuable piece of information!

The `typeid` operator II

When we look at assembly code of our code on the last slide, there's a type mismatch. While the result is a `type_info` reference, the value assigned into `rtiCRefcountedRect` is a `_RTTITypeDescriptor` pointer. The internals of this structure are hidden in a private `rtti.h` file. A similar structure, `TypeDescriptor`, with the same layout, is found in `ehdata.h` and describes the first field in the structure to be a VMT pointer:

```
typedef struct TypeDescriptor {
  #if defined(_WIN64) || defined(_RTTI) /*IFSTRIP=IGN*/
    const void * _EH_PTR64 pVFTable; // Field overloaded by RTTI
  #else
    DWORD hash; // Hash value computed from type's decorated name
  #endif
  void * _EH_PTR64 spare; // reserved, possible for RTTI
  char name[]; // The decorated name of the type; 0 terminated.
} TypeDescriptor;
```

Note: Normally, when using the `typeid` operator, the compiler changes it into a `mov` instruction assigning the `_RTTITypeDescriptor*` into the result (wherever possible), or calls the `__RTtypeid` function which returns an `_RTTITypeDescriptor*`.

The typeid operator III

```
extern "C" PVOID __CLRCALL_OR_CDECL __RTtypeid (
  PVOID inptr // Pointer to polymorphic object
) throw(...)
{
  if (!inptr) {
    throw bad_typeid ("Attempted a typeid of NULL pointer!");
    return NULL;
  }

  __try {
    // Ptr to CompleteObjectLocator should be stored at vfptr[-1]
    _RTTICompleteObjectLocator *pCompleteLocator = (_RTTICompleteObjectLocator *)  ((*((void***)inptr))[-1]) ;
    if (((const void *)pCompleteLocator->pTypeDescriptor) != NULL) {
      return (PVOID) COL_PTD(*pCompleteLocator);
    }
    else
    {
      throw __non_rtti_object("Bad read pointer - no RTTI data!");
      return NULL;
    }
  }
  __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION
                                  ?  EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
  {
    throw __non_rtti_object ("Access violation - no RTTI data!");
    return NULL;
  }
}
```

Dynamic_cast to a void* I

The dynamic_cast operator is typically used for upcasting (from a base class in direction from the base class to derived classes). When casting to a void* the __RTCastToVoid function from rtti.cpp is called:

```
void* pv = dynamic_cast<void*>(pRect);
  0040264B  mov   eax,dword ptr [pRect]
  0040264E  push  eax
  0040264F  call  __RTCastToVoid (4032C8h)
  00402654  add   esp,4
  00402657  mov   dword ptr [pv],eax
extern "C" PVOID __CLRCALL_OR_CDECL __RTCastToVoid (
  PVOID inptr // Pointer to polymorphic object
) throw(...)
{
  if (inptr == NULL)
    return NULL;

  __try {
    return FindCompleteObject((PVOID *)inptr);
  }
  __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION
                              ?  EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
  {
    throw __non_rtti_object ("Access violation - no RTTI data!");
    return NULL;
  }
}
```

Dynamic_cast to a void* II

```
static PVOID __CLRCALL_OR_CDECL FindCompleteObject (PVOID *inptr) // Pointer to polymorphic object
{
  // Ptr to CompleteObjectLocator should be stored at vfptr[-1]
  _RTTICompleteObjectLocator *pCompleteLocator = (_RTTICompleteObjectLocator *) ((*((void***)inptr))[-1]);
  char *pCompleteObject = (char *)inptr - COL_OFFSET(*pCompleteLocator);

  // Adjust by construction displacement, if any
  if (COL_CDOFFSET(*pCompleteLocator))
    pCompleteObject -= *(int *)((char *)inptr - COL_CDOFFSET(*pCompleteLocator));

  return (PVOID) pCompleteObject;
}
```

As you can see, the function extracts an `_RTTICompleteObjectLocator` pointer from within a field in front of the VMT. The structure is private and undocumented.

Dynamic_cast to a void* III

```
static PVOID __CLRCALL_OR_CDECL FindCompleteObject (PVOID *inptr) // Pointer to polymorphic object
{
  // Ptr to CompleteObjectLocator should be stored at vfptr[-1]
  _RTTICompleteObjectLocator *pCompleteLocator = (_RTTICompleteObjectLocator *) ((*((void***)inptr))[-1]);
  char *pCompleteObject = (char *)inptr - COL_OFFSET(*pCompleteLocator);

  // Adjust by construction displacement, if any
  if (COL_CDOFFSET(*pCompleteLocator))
    pCompleteObject -= *(int *)((char *)inptr - COL_CDOFFSET(*pCompleteLocator));

  return (PVOID) pCompleteObject;
}
```

As you can see, the function extracts an `_RTTICompleteObjectLocator` pointer from within a field in front of the VMT. The structure is private and undocumented.

WHO CARES?

Dynamic_cast to a void* IV

```
static PVOID __CLRCALL_OR_CDECL FindCompleteObject (PVOID *inptr) // Pointer to polymorphic object
{
  // Ptr to CompleteObjectLocator should be stored at vfptr[-1]
  _RTTICompleteObjectLocator *pCompleteLocator = (_RTTICompleteObjectLocator *)  ((*((void***)inptr))[-1]) ;
  char *pCompleteObject = (char *)inptr - pCompleteLocator->offset;

  // Adjust by construction displacement, if any
  if (pCompleteLocator->cdOffset)
    pCompleteObject -= *(int *)((char *)inptr - pCompleteLocator->cdOffset);

  return (PVOID) pCompleteObject;
}
```

As you can see, the function extracts an `_RTTICompleteObjectLocator` pointer from within a field in front of the VMT. The structure is:

```
typedef struct _RTTITypeDescriptor {
  void* __vftbl;   // VMT pointer
  void* data;      // ??
  char  d_name[1]; // Mangled data type name
} _RTTITypeDescriptor, TypeDescriptor;

typedef struct _RTTICompleteObjectLocator {
  DWORD                           signature;                // version of the structure, COL_SIG_REV0==0
  LONG                            offset;                   // offset of this VMT in the complete class
  LONG                            cdOffset;                 // construction displacement offset
  TypeDescriptor*                 pTypeDescriptor;
  _RTTIClassHierarchyDescriptor*  pClassHierarchyDescriptor;
} _RTTICompleteObjectLocator;
```

Dynamic_cast to a non-void* I

When we use the dynamic_cast operator with a non-void* data type, MSVC calls the __RTDynamicCast internal function instead.

```
const CRefcountedRect* pRefCountedRect2 = dynamic_cast<const CRefcountedRect*>(pRect);
  00402C58  push  0                                                   // 0 for pointers, 1 for refs
  00402C5A  push  offset CRefcountedRect 'RTTI Type Descriptor' (432110h) // To coerce to
  00402C5F  push  offset Rect 'RTTI Type Descriptor' (4320FCh)        // Coerce from
  00402C64  push  0                                                   // A VMT offset in the object
  00402C66  mov   eax,dword ptr [ebp-0A8h]                            // An object to coerce
  00402C6C  push  eax
  00402C6D  call  __RTDynamicCast (40331Eh)
  00402C72  add   esp,14h
  00402C75  mov   dword ptr [ebp-0ACh],eax                            // Store the coerced result
```

As we can see, passing an _RTTITypeDescriptor* is enough to verify whether CRefcountedRect derives from Rect. How is this information verified?

Dynamic_cast to a non-void* II

```
extern "C" PVOID __CLRCALL_OR_CDECL __RTDynamicCast (
  PVOID inptr,      // Pointer to polymorphic object
  LONG VfDelta,     // Offset of vfptr in object
  PVOID SrcType,    // Static type of object pointed to by inptr
  PVOID TargetType, // Desired result of cast
  BOOL isReference) // TRUE if input is reference, FALSE if input is ptr
  throw(...)
  {
    PVOID pResult=NULL;
    _RTTIBaseClassDescriptor *pBaseClass;

    // dynamic_cast returns nothing for a NULL ptr
    if (inptr == NULL)
      return NULL;

    __try {
      PVOID pCompleteObject = FindCompleteObject((PVOID *)inptr);
      _RTTICompleteObjectLocator *pCompleteLocator=(_RTTICompleteObjectLocator*) ((*((void***)inptr))[-1]) ;

      // Adjust by vfptr displacement, if any
      inptr = (PVOID *) ((char *)inptr - VfDelta);

      // Calculate offset of source object in complete object
      ptrdiff_t inptr_delta = (char *)inptr - (char *)pCompleteObject;

      if (!(CHD_ATTRIBUTES(*COL_PCHD(*pCompleteLocator)) & CHD_MULTINH)) {// if not multiple inheritance
        pBaseClass = FindSITargetTypeInstance( pCompleteLocator, (_RTTITypeDescriptor *) SrcType,
                                               (_RTTITypeDescriptor *) TargetType );
      } else if ...
      // Branches for multiple non-virtual and virtual inheritance.
```

Dynamic_cast to a non-void* III

```
      if (pBaseClass != NULL)
      {
        // Calculate ptr to result base class from pBaseClass->where
        pResult = ((char *) pCompleteObject) + PMDtoOffset(pCompleteObject, BCD_WHERE(*pBaseClass));
      }
      else
      {
        pResult = NULL;

        if (isReference)
          throw bad_cast("Bad dynamic_cast!");
      }
    }
    __except (GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION
                                  ?  EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
      pResult = NULL;
      throw __non_rtti_object("Access violation - no RTTI data!");
    }

    return pResult;
  }
```

Dynamic_cast to a non-void* IV

Each VMT[−1] contains the `_RTTICompleteObjectLocator* pLoc` pointer and the `pLoc->pTypeDescriptor->d_name` field tells us the type name! Having the name is the first step. We can go further by examining the below structures [2, 3] and discover the entire hierarchy!

```
typedef struct PMD {
  ptrdiff_t mdisp; //vftable offset
  ptrdiff_t pdisp; //vftable offset
  ptrdiff_t vdisp; //vftable offset (for virtual base class)
};

typedef const struct _s_RTTIBaseClassDescriptor {
  TypeDescriptor                *pTypeDescriptor;
  DWORD                         numContainedBases;
  PMD                           where;
  DWORD                         attributes;
  _RTTIClassHierarchyDescriptor *pClassHierarchyDescriptor;
} _RTTIBaseClassDescriptor;

typedef const struct _s_RTTIBaseClassArray {
  _RTTIBaseClassDescriptor *pArrayOfBaseClassDescriptors[1]; // A variable sized array
} _RTTIBaseClassArray;

typedef const struct _s_RTTIClassHierarchyDescriptor {
  DWORD               signature;
  DWORD               attributes;
  DWORD               numBaseClasses;
  _RTTIBaseClassArray *pBaseClassArray;
} _RTTIClassHierarchyDescriptor;
```

Dynamic_cast in g++

A brief glance

Let's check where the VMT and object metadata is stored in g++. The following code comes from libstdc++, namely gcc-4.9-4.9.2/gcc-4.9.2/libstdc++-v3/libsupc++/dyncast.cc:

```
extern "C" void * __dynamic_cast (
  const void *src_ptr,                        // object started from
  const __class_type_info *src_type,          // type of the starting object
  const __class_type_info *dst_type,          // desired target type
  ptrdiff_t src2dst                           // how src and dst are related
)
{
  const void *vtable = *static_cast <const void *const *> (src_ptr);
  const vtable_prefix *prefix = adjust_pointer <vtable_prefix> (vtable,
                                      -offsetof (vtable_prefix, origin));
  const void *whole_ptr = adjust_pointer <void> (src_ptr, prefix->whole_object);
  const __class_type_info *whole_type = prefix->whole_type;
  ...
```

We can see that the pointer to the VMT is also the first member of the structured type, and the metadata also precedes the VMT. There's a difference between MSVC and g++ — if a `NULL` pointer is passed to `dynamic_cast` in MSVC a `NULL` is returned, while g++ crashes dereferencing a `NULL` reference.

Type Information in Reverse Engineering

Type information presents another useful source of information about the reverse engineered target. We can extract the following:

- class name;
- class hierarchy.

There are IDA Pro scripts that do this work for us [1] and dump the hierarchy.

If we wanted to find this information ourselves, we would have to parse the code section and look for a typical constructor code — assigning the VMT to the first data member of the object. From the VMT[-1] pointer we could get to the type information and extract it.

Bibliography

- Igorsk: *Reversing Microsoft Visual C++ Part II: Classes, Methods and RTTI*. Available online at `http://www.openrce.org/articles/full_view/23`, 2006.
- Microsoft Corp.: *rttidata.h*: Available online at `http://read.pudn.com/downloads10/sourcecode/os/41823/WINCEOS/COREOS/CORE/CORELIBC/CRTW32/RTTI/rttidata.h__.htm`.
- Passion_wu128: *rtti.h*: Available online at `http://m.blog.csdn.net/blog/passion_wu128/38511957`, 2014.