Reverzní inženýrství

3. Analýza tříd v C++

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

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra informační bezpečnosti

Verze 2020-09-04

Obsah

Třídy a struktury

Klíčová slova `class` a `struct` znamenají přesně to samé, jediný rozdíl spočívá v odlišné výchozí ochraně prvků: Zatímco `class` má standardně svůj obsah `private`, `struct` ho má `public`. Tím pádem jsou následující dvojice zápisů ekvivalentní:

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

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

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

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

Struktury

Podívejme se na vzorovou `struct` a způsob, jak je uložena v paměti:

Vzorová struktura

```
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];
```

Rozložení SampleStruct v paměti

```
                f_Char
                 |  vycpávka (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.
```

Jak vidíme, všechna pole ve struktuře jsou zarovnána podle zarovnávacích pravidel ABI. To je důvod, proč je pole `f_Short` zarovnáno na 2-bajtovou hranici. Proč je ale vycpávka jednou `0x00`, pak `0x31`, `0xF3`, a `0x9A`?

Určení velikosti struktury I

Struktura může být alokována buď na zásobníku, nebo na haldě. Struktury na zásobníku jsou alokovány instrukcí `sub esp, __LOCAL_SIZE` a jejich velikost je součástí výrazu `__LOCAL_SIZE`. Pro alokování struktury za běhu na haldě používáme klíčové slovo `new` nebo funkci pro alokaci paměti jako `malloc`, `HeapAlloc`, nebo `LocalAlloc`. Tyto funkce přijímají velikost struktury jako svůj argument.

Alokace struktury pomocí `malloc`

```
0x80483c2  <main+18>:  sub   $0xc,%esp
0x80483c5  <main+21>:  push  $0x8                        // Velikost struktury
0x80483c7  <main+23>:  call  0x8048370 <malloc@plt>      // EAX := ukazatel na str.
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
```

Určení velikosti struktury II

Alokace pomocí operátoru `new`

Alokace pomocí operátoru `new` je podobná jako v předchozím případě, až na to, že je pro strukturu zavolán její konstruktor, pokud existuje. Tento případ vypadá následovně:

Alokace struktury pomocí operátoru `new`

```
0x4006d1   <main(int, const char**)+75>:      mov   $0x8,%edi            // Velikost struktury
0x4006d6   <main(int, const char**)+80>:      callq 0x400580 <_Znwm@plt>// Volání operátoru new
0x4006db   <main(int, const char**)+85>:      mov   %rax,%rbx            // Uložit ukazatel this do RBX
0x4006de   <main(int, const char**)+88>:      mov   %rbx,%rdi            // Zkopírovat ukazatel this do 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
```

Určení velikosti struktury III

Další metody

Velikost struktury se dá dále odvodit z instrukcí, které s ní pracují. Pokud má být obsah struktury zkopírován nebo je použito pole struktur stejného typu, můžeme často využít instrukce jako `mul` k zjištění velikosti struktury. Například:

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

Neoptimálně pomocí `mul`

```
  xor ecx, ecx                // i=0
  mov esi, dword ptr [ebp-8]// ESI := zač. pole
loop:
  mov eax, ecx
  mul VELIKOST_STRUKTURY
  mov edi, esi
  add edi, eax                // EDI = ESI + vel*i
  mov byte ptr [edi], 0ffh
  inc ecx
  cmp ecx, 100
  jb loop
```

Lépe pomocí add

```
  mov $0x600ec8,%rax // začátek pole
loop:
  movb $0x11,(%rax)
  add $0xc,%rax      // posun na další prvek
  cmp $0x603da8,%rax // konec pole
  jne loop
```

Dědičnost I

Strukturovaný datový typ může jako svoji položku využít další strukturovaný datový typ. Na tom není nic zvláštního. Strukturovaný datový typ může také dědit z dalších strukturovaných datových typů. Podívejme se, jak vypadá případ vícenásobné dědičnosti. Předpokládejme, že máme `class` dědící z `CUnknown` (viz dále v sekci Polymorfismus) a z další třídy `Rect`. Základní třída `CUnknown` implementuje počítání odkazů, zatímco `Rect` obsahuje 4 `int`y. Společně tvoří třídu `CRefcountedRect`.

```
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;
};
```

Dědičnost II

Rozložení `CRefcountedRect` v paměti

```
                CRefcountedRect*-|    pVMT (viz dále)
7fffffffe0c0: CUnknown*--------+-->d0 0c 40 00 00 00 00 00
                                      m_RefCount
7fffffffe0c8:                         01 00 00 00 00 00 00 00
                                      f_X          f_Y
7fffffffe0d0: Rect*--------------->10 00 00 00 20 00 00 00
                                      f_Width      f_Height
7fffffffe0d8:                         40 00 00 00 80 00 00 00
                                      f_Area
7fffffffe0e0:                         00 20 00 00
```

Polymorfismus I

Pokud strukturovaný datový typ obsahuje virtuální metody, musí obsahovat také tabulku virtuálních metod (VMT). Pokud objekt dědí z více objektů s virtuálními metodami, dědí dvě nebo více VMT. Každá VMT je tabulkou ukazatelů na virtuální metody; všechny virtuální metody jsou volány prostřednictvím této tabulky. VMT je uložena jako první položka třídy, následovaná daty této třídy. Poté přijde druhá VMT, druhá data, atd.

```
#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;
}
```

Polymorfismus II

Pokud strukturovaný datový typ obsahuje virtuální metody, musí obsahovat také tabulku virtuálních metod (VMT). Pokud objekt dědí z více objektů s virtuálními metodami, dědí dvě nebo více VMT. Každá VMT je tabulkou ukazatelů na virtuální metody; všechny virtuální metody jsou volány prostřednictvím této tabulky. VMT je uložena jako první položka třídy, následovaná daty této třídy. Poté přijde druhá VMT, druhá data, atd.

```
#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
```

Polymorfismus III

Rozložení třídy `CUnknown` pak je:

Rozložení objektu

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

Volání metody skrze VMT

```
40245C mov  eax,dword ptr [ebp+8]// Načíst this do EAX
40245F mov  ecx,dword ptr [eax]  // Načíst this->pVMT do ECX
402461 mov  edx,dword ptr [ebp+8]// Načíst this do EDX
402464 push edx                  // Uložit this jako první arg.
402465 mov  eax,dword ptr [ecx+4]// Načíst adresu metody z VMT [AddRef]
402468 call eax                  // Zavolat metodu
```

Pozn. 1: Možná jste si všimli, že navzdory očekávání metoda `AddRef` nedostala svůj argument v `ECX`. Důvodem je to, že `STDMETHODCALLTYPE` mění volací konvenci na `__stdcall`, a v té se všechny argumenty předávají přes zásobník. Včetně argumentů metod!

Pozn. 2: Ve třídě nacházíme před polem `m_RefCount` zarovnání, které jsme si vyžádali pomocí `__declspec(align(8))`. To bylo nutné, protože funkce InterlockedXXX vyžadují zarovnaná data.

Nastavení VMT objektu (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 bylo předáno v ECX
  0040111C mov  ecx,dword ptr [ebp-4]
  0040111F call IUnknown::IUnknown (4011D0h)     // Implicitní konstruktor
  00401124 mov  eax,dword ptr [ebp-4]
  // Zápis ukazatele na VMT do this->pVMT
  00401127 mov  dword ptr [eax],offset CUnknown::'vftable' (4293BCh)
  0040112D mov  eax,dword ptr [ebp-4]
  00401130 mov  dword ptr [eax+8],1              // Nastavení 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
```

VMT v CUnknown

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

Polymorfismus IV

I pokud je třída abstraknı́, ale má deklarované nějaké metody, má svoji VMT. Konstruktor (i implicitní) ji přiřadí do pole `this->pVMT`. Otázka zní: Pokud některé virtuální metody nejsou implementovány, jak by měl vypadat jejich ukazatel ve VMT?

```
#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;
};

// Implementace AddRef a Release zůstávají beze změny
```

Polymorfismus V

I pokud je třída abstraknı́, ale má deklarované nějaké metody, má svoji VMT. Konstruktor (i implicitní) ji přiřadí do pole `this->pVMT`. Otázka zní: Pokud některé virtuální metody nejsou implementovány, jak by měl vypadat jejich ukazatel ve VMT?

```
#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;
};

// Implementace AddRef a Release zůstávají beze změny
```

Původní VMT

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

Polymorfismus VI

I pokud je třída abstraknı́, ale má deklarované nějaké metody, má svoji VMT. Konstruktor (i implicitní) ji přiřadí do pole `this->pVMT`. Otázka zní: Pokud některé virtuální metody nejsou implementovány, jak by měl vypadat jejich ukazatel ve VMT?

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

    // Konstruktor není definován
    virtual ~CUnknown() { m_RefCount = -1; }

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

// Implementace AddRef a Release zůstávají beze změny
```

VMT s abstraktními metodami

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

Nastavení VMT objektu (Linux 64-bit)

```
IUnknown::IUnknown():                          // Implicitní konstruktor
  00400fd0 push   %rbp
  00400fd1 mov    %rsp,%rbp
  00400fd4 mov    %rdi,-0x8(%rbp)              // Ukazatel this uložen do lok. prom.
  00400fd8 mov    -0x8(%rbp),%rax              // Ukazatel this do RAX
  00400fdc movq   $0x4014d0,(%rax)             // VMT zapsána do this->pVMT
  00400fe3 pop    %rbp
  00400fe4 retq
```

```
CUnknown::CUnknown():                          // Implicitní konstruktor
  00400ff0 push   %rbp
  00400ff1 mov    %rsp,%rbp
  00400ff4 sub    $0x10,%rsp
  00400ff8 mov    %rdi,-0x8(%rbp)
  00400ffc mov    -0x8(%rbp),%rax
  00401000 mov    %rax,%rdi                    // this se předává v RDI
  00401003 callq  0x400fd0 <IUnknown::IUnknown()>  // Zde je volán implicitní konstruktor
  00401008 mov    -0x8(%rbp),%rax
  0040100c movq   $0x401490,(%rax)             // VMT zapsána do this->pVMT
  00401013 leaveq
  00401014 retq
```

VMT rozhraní IUnknown

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

VMT třídy CUnknown

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

Polymorfismus VII

Adresy abstraktních metod ve VMT jsou nahrazeny adresou funkce `_purecall`. Tato funkce zavolá handler purecall, pokud existuje, a potom ukončí program, pokud už ho neukončil sám handler.

Zdrojový kód funkce `_purecall` z `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();
}
```

Tabulky virtuálních metod v reverzním inženýrství

Jak jste si všimli, pokud má objekt VMT, tak konstruktor nastaví ukazatel na ni. K tomu dochází i tehdy, když konstruktor neexistuje — v takovém případě to udělá implicitní konstruktor (např. konstruktor `IUnknown`). Protože VMT je globální tabulkou sdílenou všemi instancemi jednoho objektu, můžeme:

- použít ukazatel na VMT k určení, zda je neznámý objekt určitého typu porovnáním jeho ukazatele na VMT se seznamem známých VMT;
- prozkoumat ukazatele v paměti objektu; pokud ukazují na VMT, našli jsme vícenásobnou dědičnost;
- prozkoumat ukazatele v každé VMT a identifikovat kód, který patří danému objektu.

Výše uvedené informace můžeme dále rozšířit studiem informace o typu objektu, která je použita v RTTI.

Motivace I

Na slajdu 9 používáme `static_cast<const Rect*>(this)` k přetypování ukazatele `this` na ukazatel na `Rect`. Ukazatel `Rect*` je také ukazatelem "`this`" odvozené třídy a je odlišný od ukazatele `CRefcountedRect* this`. Jaký kód se skrývá za `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                  // Ulož this do ebp-4
  004027CC  cmp   dword ptr [ebp-4],0                    // Static_cast NULL neudělá nic
  004027D0  je    CRefcountedRect::get_Rect+1Dh (4027DDh)
  004027D2  mov   eax,dword ptr [ebp-4]                  // Načti this do EAX
  004027D5  add   eax,10h                                // Posuň ukazatel this o 16 bajtů
  004027D8  mov   dword ptr [ebp-48h],eax                // Ulož přetypovaný výsledek
  004027DB  jmp   CRefcountedRect::get_Rect+24h (4027E4h)
  004027DD  mov   dword ptr [ebp-48h],0                  // Static_cast selhal, výsledkem bude NULL
  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
```

Motivace II

Jak se vrátíme z `Rect*` k `CRefcountedRect*`? Pro zpětné přetypování ukazatele na objekt můžeme zkusit `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
```

Zpětné přetypování není možné, protože `Rect` není polymorfický, tzn. nemá VMT. Tu můžeme přidat doplněním `virtual` destruktoru:

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

Výsledek

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

Jak mohl `dynamic_cast` vědět, že je možné ukazatel na `Rect` přetypovat na ukazatel na `CRefcountedRect`?

Operátor typeid I

Pro realizaci operátorů typeid a dynamic_cast se používá Run Time Type Information (RTTI). Podívejme se, co nám typeid přináší za informace:

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

// Zkopíruj odkaz na type_info do lok. prom. rti_CRefcountedRect
00402CA7  mov         dword ptr [ebp-0B4h],offset CRefcountedRect `RTTI Type Descriptor' (432110h)

rtiCRefcountedRect:
  __vfptr   0x004295c4                          const type_info::`vftable'* // Ukazatel na VMT
  _M_data   0x00000000                          void *
  _M_d_name 0x00432118 ".?AVCRefcountedRect@@" char [1]                     // Dekorované jméno třídy
```

Třída type_info je definována v typeinfo.h jako:

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

Našli jsme jméno třídy. To je velmi cenná informace!

Operátor typeid II

Když se pečlivěji podíváme na assembler z minulého slajdu, zjistíme nesoulad typů. Proměnná `rtiCRefcountedRect` má být ukazatel na `type_info`, ale ukládáme do ní ukazatel na `_RTTITypeDescriptor`. Vnitřnosti této struktury jsou ukryty v neveřejném souboru `rtti.h`. V `ehdata.h` však nalezneme strukturu `TypeDescriptor` se stejným rozložením; ta deklaruje, že prvním prvkem struktury je ukazatel na VMT:

```
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;
```

Pozn.: Za normálních okolností kompilátor nahradí operátor `typeid` buď instrukcí `mov` zapisující výsledek na výstup, nebo (pokud to nebylo možné) zavolá funkci `__RTtypeid`, která vrátí `_RTTITypeDescriptor*`.

Operátor typeid 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 na void* I

Operátor `dynamic_cast` se typicky používá ke zpětnému přetypování (ze základní třídy směrem k odvozené třídě). Při přetypování na `void*` se volá funkce `__RTCastToVoid` z `rtti.cpp`:

```
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 na 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;
}
```

Jak vidíte, funkce si vyzvedne ukazatel na `_RTTICompleteObjectLocator` z pole ležícího před VMT. Tato struktura je privátní a nedokumentovaná.

Dynamic_cast na 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;
}
```

Jak vidíte, funkce si vyzvedne ukazatel na `_RTTICompleteObjectLocator` z pole ležícího před VMT. Tato struktura je privátní a nedokumentovaná.

A KOHO TO ZAJÍMÁ?

Dynamic_cast na 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;
}
```

Jak vidíte, funkce si vyzvedne ukazatel na _RTTICompleteObjectLocator z pole ležícího před VMT. Tato struktura je:

```
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 na ne-void* I

Pokud použijeme operátor `dynamic_cast` s datovým typem odlišným od `void*`, zavolá MSVC interní funkci `__RTDynamicCast`.

```
const CRefcountedRect* pRefCountedRect2 = dynamic_cast<const CRefcountedRect*>(pRect);
  00402C58  push  0                                                    // 0 pro ukazatele, 1 pro ref.
  00402C5A  push  offset CRefcountedRect 'RTTI Type Descriptor' (432110h) // Cílový typ
  00402C5F  push  offset Rect 'RTTI Type Descriptor' (4320FCh)          // Zdrojový typ
  00402C64  push  0                                                    // Offset VMT uvnitř objektu
  00402C66  mov   eax,dword ptr [ebp-0A8h]                             // Objekt na přetypování
  00402C6C  push  eax
  00402C6D  call  __RTDynamicCast (40331Eh)
  00402C72  add   esp,14h
  00402C75  mov   dword ptr [ebp-0ACh],eax                             // Ulož přetypovaný výsledek
```

Jak vidíme, samotný `_RTTITypeDescriptor*` postačuje k ověření, zda `CRefcountedRect` dědí z `Rect`. Jak je toto ověření provedeno?

Dynamic_cast na ne-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 ...
      // Zde jsou větve pro vícenásobnou dědičnost
```

Dynamic_cast na ne-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 na ne-void* IV

Každá VMT[−1] obsahuje ukazatel `_RTTICompleteObjectLocator*` pLoc; jeho pole `pLoc->pTypeDescriptor->d_name` nese jméno typu! Znalost jména je jen první krok. Můžeme pokračovat zkoumáním struktur uvedených níže [2, 3] a odhalit celou hierarchii!

```
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 v g++

První pohled

Podívejme se, kde jsou VMT a metadata objektů uložena v g++. Následující kód pochází z libstdc++, konkrétně z 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;
  ...
```

Vidíme, že i zde je ukazatel na VMT prvním prvkem strukturovaného typu a že metadata jsou uložena před VMT. Proti MSVC můžeme pozorovat jeden rozdíl: pokud je do `dynamic_cast` předán ukazatel `NULL`, MSVC vrátí `NULL` zatímco g++ spadne při dereferencování ukazatele `NULL`.

Typová informace v reverzním inženýrství

Typová informace je dalším užitečným zdrojem informace o objektu reverzního inženýrství. Můžeme získat:

- jméno třídy;
- hierarchii tříd.

K dispozici jsou skripty [1] pro IDA Pro, které tuto práci udělají za nás a vypíšou celou hierarchii.

Pokud bychom chtěli hledat tuto informaci sami, museli bychom projít kódovou sekci a hledat typický kód konstruktoru — přiřazení VMT do prvního datového prvku objektu. Z ukazatele VMT[−1] se pak dostaneme k typové informaci a můžeme ji vyextrahovat.

Literatura


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.