Practical C++ Decompilation

Igor Skochinsky
Hex-Rays

Recon 2011
Montreal
Outline

- Class layouts
- Virtual tables
- Methods, constructors and destructors
- RTTI and alternatives
- Dealing with C++ in IDA and Hex-Rays decompiler
Class layouts

- Class fields are generally placed in memory in the order of declaration.
- Equivalent structure can be produced by removing all methods.
- Example:

```c
class A {
  int a1;
  int a2;
}
```
```
00000000 A       struc
00000000 a1      dd ?
00000004 a2      dd ?
00000008 A       ends
```
Single inheritance

- With simple inheritance, fields of the derived class are placed after the base class
- Example:

```cpp
class B: public A {
    int b3;
}
```

```
00000000 B       struc
00000000 a1      dd ?
00000004 a2      dd ?
00000008 b3      dd ?
0000000C B       ends
```
With multiple inheritance, first the base classes are laid out, then the fields of the derived class.

Example:

```cpp
class C: public A, public B {
   int c4;
};
```

```
00000000 C       struc
00000000 a1      dd ?
00000004 a2      dd ?
00000008 a1      dd ?
0000000C a2      dd ?
00000010 b3      dd ?
00000014 c4      dd ?
00000018 C       ends
```
Virtual inheritance

- In case of virtual inheritance, the place of a virtual base class is not fixed and can change in future derived classes.
- The compiler has to track offset of a virtual base in each specific class inheriting from it.
- MSVC implements it by producing a virtual base table (vbtable) with offsets to each of the virtual bases.
- GCC puts offsets to virtual bases into the virtual function table (vftable).
Virtual inheritance example

class B : public virtual A {
public:
    int b3;
};

00000000 B       struc
00000000 _vbptr   dd ?
00000004 b3       dd ?
00000008 a1       dd ?
0000000C a2       dd ?
00000010 B       ends

class C : public virtual A, public B {
public:
    int c4;
};

00000000 C       struc
00000000 _vbptr   dd ?
00000004 b3       dd ?
00000008 c4       dd ?
0000000C a1       dd ?
00000010 a2       dd ?
00000014 C       ends
Virtual inheritance example

<table>
<thead>
<tr>
<th>MSVC</th>
<th>GCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>const B::`vbtable':</td>
<td>`vtable for'B:'</td>
</tr>
<tr>
<td>dd 0</td>
<td>dd 8</td>
</tr>
<tr>
<td>dd 8</td>
<td>dd 0</td>
</tr>
<tr>
<td>const C::`vbtable':</td>
<td>`vtable for'C:'</td>
</tr>
<tr>
<td>dd 0</td>
<td>dd 0Ch</td>
</tr>
<tr>
<td>dd 0Ch</td>
<td>dd offset <code>typeinfo for'B</code></td>
</tr>
<tr>
<td></td>
<td>dd 0</td>
</tr>
<tr>
<td></td>
<td>dd offset <code>typeinfo for'C</code></td>
</tr>
<tr>
<td></td>
<td>dd 0</td>
</tr>
</tbody>
</table>
Virtual tables

- If class has virtual methods, compiler creates a table of pointers to those methods
- The pointer to the table is placed into a hidden field
- Methods are usually arranged in the order of declaration
- When inheriting, overridden methods are replaced and new ones are added at the end
- If inherited class does not override or add new virtual methods, the table can be reused
- MSVC uses separate tables for virtual functions and virtual bases, GCC combines them
Method calls

- Standard method calls take a hidden **this** parameter
- In MSCV x86, **ecx** is traditionally used (__thiscall)
- In other compilers, it's usually inserted as the first parameter

```assembly
mov    ecx, [edi+0Ch]
call   CUser::IsCachedLogon(void)

cached = m_pUser->IsCachedLogon()
```
Static method calls

- Static methods do not need a class instance
- Because of that, they behave as standard functions
- Can't be easily distinguished from standalone functions

```plaintext
push    eax             ; nLengthNeeded
push    edi             ; hObj
call    CSession::GetDesktopName(HDESK__ *
,* ,ushort * *)

CSession::GetDesktopName(hDesk, &name)
```
Virtual method calls

- Virtual methods can be overridden in derived classes
- The call address has to be calculated dynamically
- The address is loaded from the virtual table
- Virtual methods also expect `this` pointer passed

```
mov     eax, [ebp+pUnk]
mov     ecx, [eax]
push    eax
call    dword ptr [ecx+8]
pUnk->Release();
```
Constructors

- First code executed during an object's lifetime
- Usually performs the following actions:
  a) call constructors of base classes
  b) initialize `vfptrs` if has virtual functions
  c) call constructors of complex members
  d) run initialization list actions
  e) execute the constructor body
- In optimized code, some steps can be shuffled and some calls inlined
- Calling convention same as normal methods (hidden `this` pointer)
- MSVC returns the `this` pointer from constructors
public: __thiscall CMachine::CMachine(void) proc near
    mov    edi, edi
    push   esi
    mov    esi, ecx
    call   CDataStoreObject::CDataStoreObject(void)
    and    dword ptr [esi+24h], 0
    or     dword ptr [esi+28h], 0FFFFFFFFh
    and    dword ptr [esi+2Ch], 0
    or     dword ptr [esi+30h], 0FFFFFFFFh
    mov    dword ptr [esi], offset const CMachine::`vftable'
    mov    eax, esi
    pop    esi
    retn
public: __thiscall CMachine::CMachine(void) endp
Objects can be constructed in different ways
Local (automatic) variables usually get memory allocated on stack

```
push  [ebp+arg_0]
lea   ecx, [ebp+var_7A]
call  CFolderIdString::CFolderIdString(_GUID const &)

CFolderIdString folderString(guid);
```
Objects can be constructed with the `new` operator

The compiler calls generic or class-specific `operator new`

Allocated memory is passed to the constructor

```
push  34h
call  operator new(uint)
pop   ecx
test  eax, eax
jz    short @@nomem
      CMachine *machine = new CMachine();
mov   ecx, eax
call  CMachine::CMachine(void)
jmp   short @@ok
@@nomem:
xor   eax, eax
@@ok:
```
Global objects are constructed at the program start

Usual implementation uses a table of compiler-generated functions that call constructor with memory reserved in the data section

MSCV adds destructors using atexit() calls

GCC uses a separate table of destructors

The table is handled in the runtime start-up code
push    offset ___xc_z
push    offset ___xc_a
call    __initterm

___xc_a dd 0
    dd offset sub_42FAB74B
    dd offset sub_42FAB765
    dd offset sub_42FAB7E1
...
___xc_z dd 0
void __cdecl `dynamic initializer for 'g_PrivateProfileCache'(void) proc near
  mov    ecx, offset g_PrivateProfileCache
  call   CPrivateProfileCache::CPrivateProfileCache(void)
  push   offset `dynamic atexit destructor for 'g_PrivateProfileCache'(void); void (__cdecl *)(
  call   _atexit
  pop    ecx
  retn
void __cdecl `dynamic initializer for 'g_PrivateProfileCache'(void) endp
Global objects: MSVC III

void __cdecl `dynamic atexit destructor for 'g_PrivateProfileCache'(void) proc near
    mov    ecx, offset g_PrivateProfileCache
    jmp    CPrivateProfileCache::~CPrivateProfileCache(void)
void __cdecl `dynamic atexit destructor for 'g_PrivateProfileCache'(void) endp
Global objects: GCC

- .ctors section contains pointers to "global constructor" functions
- .dtors contains pointers to "global destructor" functions
- Sometimes two global arrays (__CTOR_LIST__/__DTOR_LIST__) are used instead
- Both types call a common "initialization_and_destruction" function which either constructs or destructs globals for a module

```c
void __static_initialization_and_destruction_0(int _initialize_p, int _priority)
```
Constructor calls IV

- Static objects are initialized on first use
- Common way is to use a guard variable

```
mov     eax, ds:_guard_aa
and     eax, 1
jnz     short @@skip
mov     ecx, ds:_guard_aa
or      ecx, 1
mov     ds:_guard, ecx
mov     ecx, offset aa
call    A::A(void)
```

- GCC uses ABI-specified helper functions
  `___cxa_guard_acquire/___cxa_guard_release`. 
Array construction

- Each element of the array has to be constructed separately.
- If any of the constructors throws an exception, all previous elements must be destructed.
- MSVC uses a helper function, "vector constructor iterator".
- Very useful because in one place we get instance size, constructor and (in case of EH iterator) destructor.

```c
push offset ATL::CComTypeInfoHolder::stringdispid::stringdispid(void)
push esi
push 0Ch
push eax
call `vector constructor iterator'(void *,uint,int,void * (*)(void *))

strings = new ATL::CComTypeInfoHolder::stringdispid[count];
```
Destructors

- Unlike constructors, a class can have only one destructor.
- Takes a pointer to instance and reverses actions of the constructor:
  a) initialize \texttt{vfptrs} if has virtual functions
     (this is done so that any virtual calls in the body use the methods of the current class)
  b) execute the destructor body
  c) call destructors of complex class members
  d) call destructors of base classes
- Simple destructors can be inlined, so you can often see the \texttt{vfptr} reloaded many times in the same function.
Destructor example

```assembly
virtual __thiscall CMruLongList::~CMruLongList(void) proc near
    mov  edi, edi
    push esi
    mov  esi, ecx
    mov  eax, [esi+30h]
    mov  dword ptr [esi], offset const CMruLongList::`vftable'
    test eax, eax
    jz    short loc_42D93239
    push eax                     ; hMem
    call ds:LocalFree(x)
    and  dword ptr [esi+30h], 0
loc_42D93239:
    mov  ecx, esi
    pop  esi
    jmp  CMruBase::~CMruBase(void)
virtual __thiscall CMruLongList::~CMruLongList(void) endp
```
Virtual destructors

- When deleting object by pointer, a proper `operator delete` must be called
- It can be different for different classes in hierarchy
- Compiler has to make sure the correct operator regardless of the pointer type
- MSVC uses a helper function (deleting destructor) which is placed into the virtual table instead of the actual destructor
- It calls the actual destructor and then `operator delete`
- GCC emits multiple destructors (in-charge, not-in-charge, in-charge deleting) and calls the corresponding one
Virtual destructor example

virtual void * __thiscall CMruLongList::`scalar deleting destructor'(unsigned int) proc near
push ebp
mov ebp, esp
push esi
mov esi, ecx
mov esi, ecx
call CMruLongList::~CMruLongList(void)
test [ebp+arg_0], 1
jz short loc_42D93260
push esi ; lpMem
call operator delete(void *)
pop ecx
loc_42D93260:
mov eax, esi
pop esi
pop ebp
retn 4
virtual void * __thiscall CMruLongList::`scalar deleting destructor'(unsigned int) endp
RTTI: Run-time type information

- Necessary for `dynamic_cast<>` and `typeid()` operators
- Only required for polymorphic classes (with virtual methods)
- Because of this, usually attached to the virtual table
- MSVC uses a complex set of structures, see my OpenRCE article¹
- GCC puts a pointer to typeid class instance just before the method addresses
- First data member of that instance (after vfptr) is a pointer to the mangled name of the class.

¹ [http://www.openrce.org/articles/full_view/23](http://www.openrce.org/articles/full_view/23)
RTTI alternatives: MFC

- MFC does not use standard RTTI
- All MFC classes inherit from CObject
- First virtual method of CObject is GetRuntimeClass()
- Returns a pointer to a of CRuntimeClass instance
- The object contains the MFC class name, instance size and functions for dynamic creation
RTTI: MFC example

const CConfirmDriverListPage::`vftable'
  dd offset CConfirmDriverListPage::GetRuntimeClass(void)
  dd offset CConfirmDriverListPage::`vector deleting destructor'(uint)
  dd offset CObject::Dump(CDumpContext &)
...

CConfirmDriverListPage::classCConfirmDriverListPage
  dd offset aCconfirmedriver ; m_lpszClassName
  dd 164h ; m_nObjectSize
  dd 0FFFFh ; m_wSchema
  dd offset CConfirmDriverListPage::CreateObject(void)
  dd offset CTypAdvStatPage::_GetBaseClass(void)
  dd 0
RTTI alternatives: Qt

- Qt uses a completely custom OOP model
- Slots and signals used instead of virtual methods
- Leaves a lot of meta information, including slot method names
- The whole implementation was described by Daniel Pistelli
- See http://www.ntcore.com/files/qtrev.htm for details and some IDC scripts
RTTI alternatives: Apple IOKit

- IOKit is the base framework for implementing drivers for Apple OS X and iOS
- Uses a subset of C++: no exceptions, templates, multiple inheritance or standard RTTI
- Uses its own implementation with support for dynamic object creation
- All classes inherit from OSObject
- One of its methods is `getMetaClass()`
- Metaclass instance contains instance size and class name
- A static instance of metaclass is created for each class
- Names and hierarchy can be tracked from metaclasses
## C++ and Hex-Rays

- IDA type system does not support C++ (yet)
- Hex-Rays is a C decompiler
- C++ constructs have to be emulated using C ones

<table>
<thead>
<tr>
<th>C++</th>
<th>IDA/Hex-Rays</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>structures</td>
</tr>
<tr>
<td>class inheritance</td>
<td>nested structures</td>
</tr>
<tr>
<td>virtual function table</td>
<td>function pointer table</td>
</tr>
<tr>
<td>implicit arguments</td>
<td>explicit arguments</td>
</tr>
</tbody>
</table>
C++ and Hex-Rays: classes and inheritance

class A
{
    int a1;
    int a2;
};
class B: public A
{
    int b3;
};

00 A struc
00 a1 dd ?
04 a2 dd ?
08 A ends

00 B struc
00 _ A ?
08 b3 dd ?
0C B ends
C++ and Hex-Rays: function prototypes

- Some conversion is necessary if C++ prototypes are known (from headers or demangled symbol names)
- Non-static methods need a this pointer added
- Structure/class returns take an additional result pointer
- References to pointers (done by IDA automatically)

<table>
<thead>
<tr>
<th>C++ Prototype</th>
<th>Hex-Rays Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>unsigned long __thiscall CMachine::Initialize(void)</code></td>
<td><code>unsigned long __thiscall CMachine::Initialize(CMachine* this)</code></td>
</tr>
<tr>
<td><code>myStruct MyClass::getStruct()</code></td>
<td><code>myStruct* MyClass::getStruct (myStruct* result, MyClass* this)</code></td>
</tr>
<tr>
<td><code>static unsigned long CFolderRedirector::GetDefaultAttributes(struct _GUID const &amp;)</code></td>
<td><code>static unsigned long CFolderRedirector::GetDefaultAttributes (struct _GUID *)</code></td>
</tr>
</tbody>
</table>
C++ and Hex-Rays: virtual tables

- Table structure can be made manually
- You can use "Create struct from data" to generate initial structure
- Then set types of each member to be a function pointer
- Not very hard to create a script which analyzes vtable and creates a structure
- Hint: add a repeatable comment with the target address and number of purged bytes
- One table per class or share among many classes
- First approach is more universal but recovered prototypes have to be copied to other tables
- Second one doesn't work for tree inheritance
C++ and Hex-Rays: virtual table example

CMachine::`vftable'
    dd offset CMachine::`scalar deleting destructor'(uint)
    dd offset CMachine::Initialize(void)

00 CMachine_vtable struc
00 __delDtor dd ? ; 0101A9C3
04 Initialize dd ? ; 0101A6C0
08 CMachine_vtable ends

struct CMachine_vtable
{
    int (__thiscall *__delDtor)(CMachine *, int);
    int (__thiscall *Initialize)(CMachine *);
};
C++ and Hex-Rays: virtual table example

public: __thiscall CMachine::CMachine(void) proc near
    push    esi
    mov     esi, ecx
    call    CDataStoreObject::CDataStoreObject(void)
    and     dword ptr [esi+24h], 0
    or      dword ptr [esi+28h], 0FFFFFFFFh
    and     dword ptr [esi+2Ch], 0
    or      dword ptr [esi+30h], 0FFFFFFFFh
    mov     dword ptr [esi], offset const CMachine::`vftable'
    mov     eax, esi
    pop     esi
    retn

public: __thiscall CMachine::CMachine(void) endp
C++ and Hex-Rays: virtual table example

```cpp
struct CMachine
{
    CDataStoreObject _;
    _DWORD dword24;
    _DWORD dword28;
    _DWORD dword2C;
    _DWORD dword30;
};
```

```cpp
00 CMachine struc
00 _ CDataStoreObject ?
24 dword24 dd ?
28 dword28 dd ?
2C dword2C dd ?
30 dword30 dd ?
34 CMachine ends
```
C++ and Hex-Rays: virtual table example

public: __thiscall CMachine::CMachine(void) proc near
push    esi
mov     esi, ecx
call   CDataStoreObject::CDataStoreObject(void)
and    [esi+CMachine.dword24], 0
or     [esi+CMachine.dword28], 0FFFFFFFFh
and    [esi+CMachine.dword2C], 0
or     [esi+CMachine.dword30], 0FFFFFFFFh
mov     [esi+CMachine._._vtable], offset const CMachine::`vftable'
mov     eax, esi
pop     esi
retn
public: __thiscall CMachine::CMachine(void) endp
CMachine *__thiscall CMachine::CMachine(CMachine *this)
{
    CDataStoreObject::CDataStoreObject(&this->_);
    this->dword24 = 0;
    this->dword28 = -1;
    this->dword2C = 0;
    this->dword30 = -1;
    this->_._vtable = (CMachine_vtable *)&CMachine::_vftable_
    return this;
}
C++ and Hex-Rays: virtual table example

```c
mov     ecx, [esi+4]                ; CMachine *
cmp     ecx, edi
jz      short loc_100C57C
mov     eax, [ecx+CMachine._._vtable]
push    1                           ; int
call    [eax+CMachine_vtable.__delDtor] ; 0101A9C3
mov     [esi+4], edi

_machine = context->m_machine;
if ( _machine )
{
    _machine->_.vtable->__delDtor(_machine, 1);
    context->m_machine = 0;
}
```
Nesting of complete class structures works for simple cases, but not good for complex inheritance

We cannot set different types for the vtable pointer shared between classes

Nesting breaks down in case of virtual inheritance because virtual bases are shuffled around

A different approach is to use two structures: one for just the fields, and one for the complete class

The complete class structure includes the fields structure and adds virtual table pointers
C++ and Hex-Rays: vtable redux example

```c
00 CDataStoreObject_fields struc
  00 dword4 dd ?
  04 m_cs _RTL_CRITICAL_SECTION ?
  1C dword20 dd ?
20 CDataStoreObject_fields ends

00 CMachine_fields struc
  00 dword24 dd ?
  04 dword28 dd ?
  08 dword2C dd ?
  0C dword30 dd ?
10 CMachine_fields ends

00 CDataStoreObject struc
  00 _vtable dd ?
  04 __f CDataStoreObject_fields ?
24 CDataStoreObject ends

00 CMachine struc
  00 _vtable dd ?
  04 __f1 CDataStoreObject_fields ?
  24 __f2 CMachine_fields ?
34 CMachine ends

00 CSession struc
  00 _vtable dd ?
  04 __f1 CDataStoreObject_fields ?
  24 __f2 CSession_fields ?
84 CSession ends
```
mov    eax, [ecx+CMachine._vtable]
push   1                           ; int
call   [eax+CMachine_vtable.__delDtor] ; 0101A9C3

struct CMachine_vtable {
    int (__thiscall *__delDtor)(CMachine *, int);
    int (__thiscall *Initialize)(CMachine *);
};

result = _machine->_vtable->__delDtor(_machine, 1);

mov    eax, [ecx+CSession._vtable]
push   1
call   [eax+CSession_vtable.__delDtor] ; 0101B90F

struct CSession_vtable {
    int (__thiscall *__delDtor)(CSession *, int);
    int (__thiscall *Initialize)(CSession *);
};

result = _session->_vtable->__delDtor(_session, 1);
C++ decompiling workflow

- Identify constructors and destructors from the global init tables or code patterns (stack construction, heap/new allocation, unwind funclets)
- Drill down to the most base constructors/destructors
- Make initial structures (e.g. using "Create new struct type")
- If has vtable, make a vtable stucture and set the vfptr type to be a pointer to it
- Follow cross-references to identify other methods of the class
- Fix up the structures and vtable function pointer types as necessary
Conclusion

- C++ decompilation is somewhat difficult but doable
- A lot of information can be extracted from RTTI and vtables/vbtables
- Many common tasks can be automated
- There is a lot of room for improvement

Links

MSVC: http://www.openrce.org/articles/full_view/23

Bonus matter: Hex-Rays 1.6 preview

Spoiler Alert
Hex-Rays 1.6: variable mapping

v1 = this;
lock_mtx_lock(this->__b.field_228);
if ( !OSIncrementAtomic(&v1->__b.field_19C) )
{
    v1->_vtbl->virt380(v1);
    ...
}

[map v1 to this]

lock_mtx_lock(this->__b.field_228);
if ( !OSIncrementAtomic(&this->__b.field_19C) )
{
    this->_vtbl->virt380(this);
Hex-Rays 1.6: support for unions

if ( StackLocation->Parameters.Read.ByteOffset.LowPart == 315396 
    || StackLocation->Parameters.Read.ByteOffset.LowPart == 315412 ) 
{
    if ( StackLocation->Parameters.Create.Options >= 0x2C )

    [choose correct union field]

    if ( StackLocation->Parameters.DeviceIoControl.IoControlCode == 0x4D004 
        || StackLocation->Parameters.DeviceIoControl.IoControlCode == 0x4D014 )
    {
        if ( StackLocation->Parameters.DeviceIoControl.InputBufferLength >= 0x2C )
Hex-Rays 1.6: kernel idioms support

deviceInfo->ListEntry.Blink = &deviceInfo->ListEntry;
deviceInfo->ListEntry.Flink = deviceInfo->ListEntry.Blink;

InitializeListHead(&deviceInfo->ListEntry);

while ( (LIST_ENTRY *)ListHead.Flink != &ListHead )

while ( !IsListEmpty(&ListHead) )

deviceInfo = (_DEVICE_INFO *)((char *)&thisEntry[-131] - 4);
if ( *p_serial == *((_DWORD *)thisEntry - 1) )
    break;

deviceInfo = CONTAINING_RECORD(thisEntry, _DEVICE_INFO, ListEntry);
if ( *p_serial == CONTAINING_RECORD(thisEntry, _DEVICE_INFO, ListEntry)->SerialNo )
    break;
We're hiring!

We're looking for someone to help us improve the decompiler

If you liked this talk and would like to work on it, let us know

info@hex-rays.com
Thank you!

Questions?