Global Information Assurance Certification Paper

x86 Representation of Object Oriented Programming Concepts for Reverse Engineers

GIAC (GREM) Gold Certification

Author: Jason Batchelor, jxbatchelor@gmail.com

Advisor: Richard Carbone

Accepted: November 23, 2015

Abstract

Modern samples of malicious code often employ object oriented programming techniques in common languages like C++. Understanding the application of object oriented programming concepts, such as data structures, standard classes, polymorphic classes, and how they are represented in x86 assembly, is an essential skill for the reverse engineer to meet today's challenges. However, the additional flexibility object oriented concepts affords developers results in increasingly complex and unfamiliar binaries that are more difficult to understand for the uninitiated. Once proper understanding is applied, however, reversing C++ programs becomes less nebulous and understanding the flow of execution becomes more simplified. This paper presents three custom developed examples that demonstrate common object oriented paradigms seen in malicious code and performs an in-depth analysis of each. The objective is to provide insight into how C++ may be reverse engineered using the Interactive Disassembler software, more commonly known as IDA.

1. Introduction

While object oriented programming is generally understood by developers using higher level languages, such as C++, the reverse engineer is required to understand how these concepts manifest themselves within a compiled binary. A reverse engineer operating on modern malware simply cannot afford to remain ignorant when considering data structures, standard classes, and polymorphic classes, as many of today's specimens invoke one or more of the above.

The remainder of this paper will review some core concepts involving how one derives context from variables identified in x86 disassembly and standard calling conventions. Subsequent sections will then focus specifically on the concepts of data structures, standard classes and polymorphic classes. A discussion on each of the criteria will be provided alongside case study analysis. During each case study, comparisons will be made between true source code and the x86 representation of a specific concept. While the provided source code serves to augment the discussion, the goal is to enable the reader to achieve the same results without its benefit. The exact compiler settings used for each of the provided case studies may be found in the Appendix.

1.1. Assumptions

The examples discussed herein were written in C++ using Microsoft Visual Studio 2010 and compiled on a 32-bit Windows operating system using the Intel x86 architecture. The Interactive Disassembler (IDA) software will be used to identify attributes of general object oriented concepts and derive conclusions on how they are represented within the compiled binary. It is important to note, that all discussion within this paper will be directly tied to the above criteria.

1.2. Core Concepts

The identification of structures, standard classes, and polymorphic classes within a binary file can be a complex task. By understanding how each of these concepts is represented at the lowest human-readable level of code, a key advantage is granted to the reverse engineer. This understanding can oftentimes be defined as a structure with different member elements in IDA. The structure can then be applied to the rest of the disassembled program. Doing so has far

reaching implications on the overall understanding of a program's core capabilities, and how it functions at run time. Oftentimes, this has a domino effect on the overall reverse engineering effort and enriches the context a reverse engineer will be able to pass on to their customer.

To achieve an understanding of structured elements, one needs to be well versed with traditional programming constructs, such as pointers and integers. Programming languages like C aid in this pursuit, because many C constructs correspond at a one to one ratio, to a single line of assembly (Lawlor, 2006). It is also important to understand how these types are used and represented in x86 assembly. The identification of data types relies on a solid understanding of assembly logic, which in itself contains the contextual clues necessary to inform the analysis. Table 1, below, provides a few examples of how the interpretation of basic assembly code can reveal deeper meaning. Consider the following examples in reference to the *EAX* register.

Table 1: Contextual Analysis of Assembly Instructions

Code	Purpose	EAX Context
lea eax, [edx+4] add [eax], 5	Add five to the address of [edx+4].	Eax is likely a pointer to an integer or perhaps another pointer.
mov eax, [ecx+0xc] call eax	Call into the dereferenced value of [ecx+0xc].	Eax is the address of a function.
mov eax, [ecx+0x4] push [eax+0xc] push [eax+8] push [eax+4] call [eax]	Load address to structure at [ecx+0x4] and push its member offsets to the stack before calling a structure function.	Eax is a pointer to a structure containing a pointer to a function and that functions member parameters.

Understanding calling conventions being used within the decompiled binary is also critical to deriving meaning from structured elements that are passed to member functions. Calling conventions are effectively a scheme, or standard, used for function callers (sub routine doing the call) and the callee (sub routine being called) within a program. They are primarily used for defining how parameters are passed to called functions and also how returned values are retrieved from them (Pollard, 2010). Four common types of calling conventions reverse engineers need to be concerned with when considering C++ programming for Windows are *cdecl*, *stdcall*, *fastcall*, and *thiscall* (Trifunovic, 2001).

The *cdecl* convention is the default convention used by Microsoft for C and C++ programs (Microsoft, 2015). For this calling convention, arguments are passed from right to left and placed on the stack and cleanup of the stack is done by the caller (Trifunovic, 2001). 32-bit

integer or memory address values that are returned from the callee are saved to the *EAX* register for the caller (Jönsson, 2005). Figure 1 illustrates this concept with appropriate annotations.

```
00401140                        arguments are pushed to stack from right to left
0040114D
0040114D 50                     push    eax
0040114E E8 AD FE FF FF         call    _cdecl_function
00401153
00401153                        cleanup the stack by adding four (size of arguments) to the ESP register
00401153
00401153 83 C4 04               add     esp, 4
00401156
00401156                        return value from callee is saved to EAX, used in next call
00401156
00401156 50                     push    eax
00401157 68 00 82 40 00         push    offset aTotalDownloade ; "Total downloaded %d bytes \n"
0040115C E8 13 00 00 00         call    sub_401174
```

Figure 1: Example of Cdecl Calling Convention

When considering API calls made using the Windows API, the *stdcall* convention is utilized. *Stdcall* arguments for this calling convention are pushed from the stack from right to left. In contrast to *cdecl* stack cleanup, stack maintenance responsibility is done by the function callee instead of the caller (Microsoft, 2015). Return values from functions are done so using the *EAX* register (Jönsson, 2005). The annotations in Figure 2 depict a common *stdcall* routine.

```
004028B9
004028B9                        arguments are pushed to the stack from right to left
004028B9
004028B9 68 9C 82 40 00         push    offset ProcName ; "CorExitProcess"
004028BE 50                     push    eax             ; hModule
004028BF FF 15 2C 80 40 00      call    ds:GetProcAddress
004028C5
004028C5                        note, no stack cleanup by the caller
004028C5
004028C5                        the returned value from callee is handled by eax register
004028C5
004028C5 85 C0                  test    eax, eax
004028C7 74 05                  jz      short loc_4028CE
```

Figure 2: Example of Stdcall Calling Convention

Fastcall is a convention used to reduce the computational cost of calling a function. This is done primarily by taking the first two arguments of a function and placing them into the *EDX* and *ECX* registers. Remaining arguments are then placed on the stack from right to left. The process makes function calls less expensive because operations being done directly on register values are faster than on the stack (Trifunovic, 2001).

Author Name, email@address

The *thiscall* calling convention uses the callee to clean the stack and has function arguments pushed from right to left. With the Microsoft Visual C++ compiler the 'this' pointer is passed before a function is called; it is done so using the *ECX* register and not the stack (Microsoft, 2015). It is important to note the *thiscall* convention has a different implementation for the GCC compiler, where it is very similar to *cdecl* except the addition of the 'this' pointer, which is pushed onto the stack last (The Art Of Service, n.d.). For functions that do not have a variable number of arguments, *thiscall* is used by default as the calling convention (Trifunovic, 2001). *Thiscall* cannot be explicitly used, and is reserved for functions that do not specifically request a different convention, and do not take a variable number of arguments (HackCraft, n.d.).

2. Discussion of Object Oriented Concepts and Case Studies

The following section focuses on three core concepts of object-oriented programming; data structures, standard classes, and polymorphic classes. Each concept discussed includes a brief review alongside a case study with emphasis on deriving context of member variables or functions. For each case study, the program is specifically written to demonstrate the topic being presented and deconstructed.

2.1. Data Structures

A data structure is merely a group of variables tied together to a single name defining the group. The name represents a structure type that denotes the beginning of the structure and is usually passed around as a pointer in memory. As an example of this, we have the following defined structure in Figure 3.

```
1.  struct InternetConnection
2.  {
3.      HINTERNET Connect;
4.      HINTERNET OpenAddress;
5.      LPCTSTR Useragent;
6.      LPCTSTR Uri;
7.      char DataReceived[4096];
8.      DWORD NumberOfBytesRead;
9.      DWORD TotalNumberRead;
10. };
```

Figure 3: Example Structure Definition

The structure type would be defined as *InternetConnection*, and members of this structure type include an assortment of handles, C String operators, character, and DWORD variables. Object names of type *InternetConnection*, may be declared in two primary ways. They may show up at the end of the structure type declaration before the final semicolon as seen in Figure 4.

```
1.  struct InternetConnection
2.  {
3.      HINTERNET Connect;
4.      HINTERNET OpenAddress;
5.      LPCTSTR Useragent;
6.      LPCTSTR Uri;
7.      char DataReceived[4096];
8.      DWORD NumberOfBytesRead;
9.      DWORD TotalNumberRead;
10. } nhlDotCom, sansDotCom; // object declarations
```

Figure 4: Example Structure with Declared Objects

Alternatively, in Figure 5, they may also be instantiated separately within the code after the structure type has been declared. In this case, the structure type specifier is an optional attribute (CPlusPlus, n.d.).

```
1.  InternetConnection nhlDotCom;
2.  InternetConnection sansDotCom;
```

Figure 5: Alternate Instantiation of Objects

Incremental offsets to the start of the object are usually dereferenced in x86 to retrieve the member variable of a particular structure. To illustrate this concept from within a debugger, one can use OllyDbg while running a program that uses the structure type defined above.

Author Name, email@address

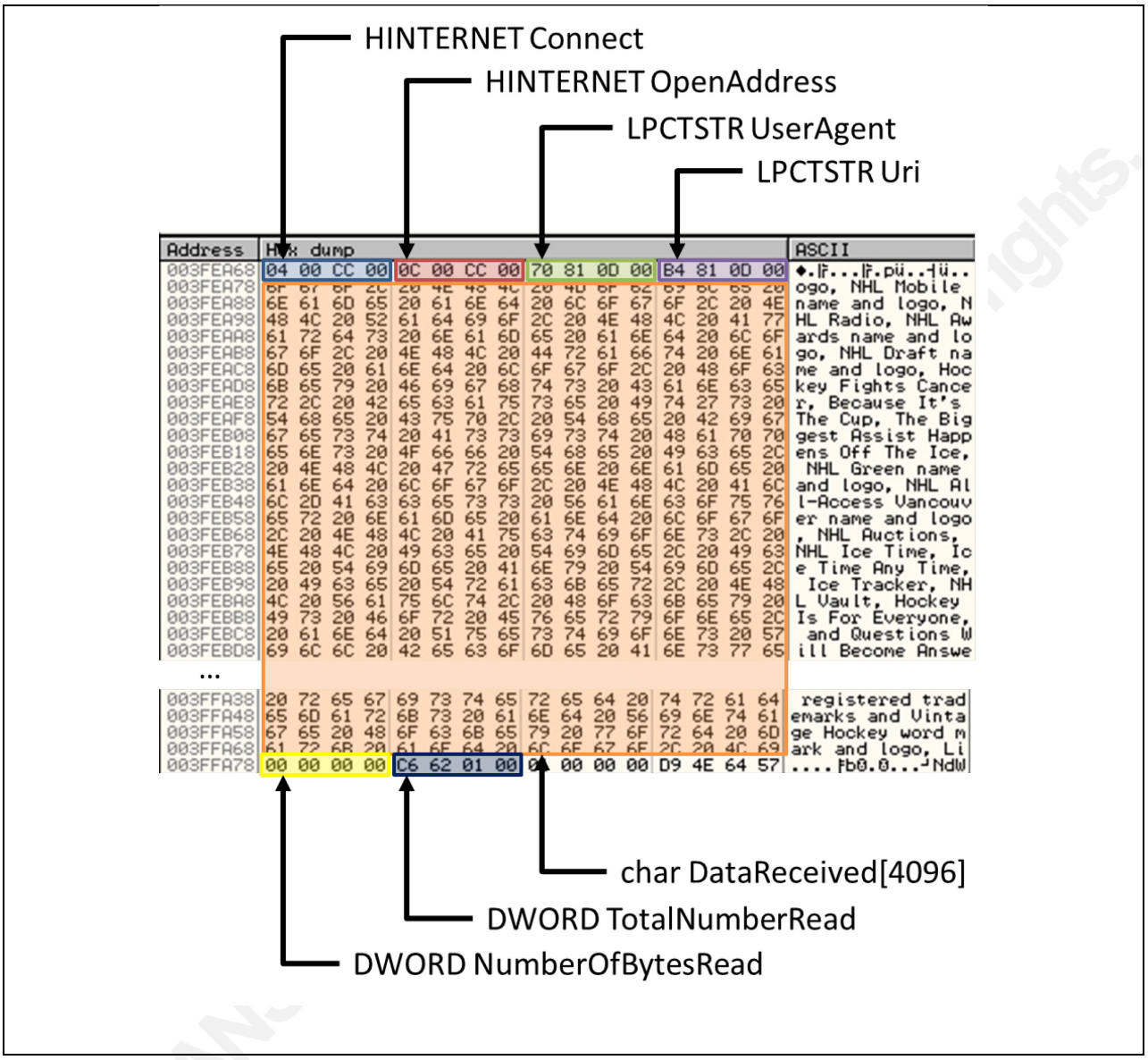


Figure 6: Structure Mapping Using OllyDbg

From the illustration in Figure 6, one can clearly see the address of 0x3FEA68 being used as the start address of our object 'nhlDotCom.' Dereferencing the start address would give one the *HINTERNET* Connect variable. These handle values take up one DWORD, or 4 bytes of memory as can be seen above. The *LPCSTR* values are pointers to their string types and likewise take one DWORD of space. These member values need to be dereferenced in order to be accessed for their contents. Conversely, the *DataReceived* variable is a character array with a maximum size of 4,096 bytes. These bytes can be clearly seen in the address space in the above screenshot. Finally, we have the last two DWORD values representing the number of bytes read

and the total number of bytes read. It is important to remember endianness when interpreting these and all byte values on the heap. The final value for *TotalNumberRead*, while appearing visually to be that of 0xc6620100, is actually 0x0162c6 (90,822 bytes) after endianness is considered.

At this point, a structure of type *InternetConnection* has been declared along with its member values. Analysis was then preformed on how a structure like this would be visually represented in memory of a program making use of this structure. At the assembly level, to reference members of an *InternetConnection* object, we need to first pay special attention to the size of each variable, and understand how much space it will take up on the heap. Doing so will allow us to accurately compute the offset, relative to the start of the *InternetConnection* object. Table 2 below represents how members of *InternetConnection* objects would be accessed in x86 assembly assuming the register *ESI* is the address of our *InternetConnection* object.

Table 2: Assembly Object Member Access Matrix

X86 Instruction	InternetConnection Member
[esi]	HINTERNET Connect
[esi+4]	HINTERNET OpenAddress
[esi+8]	LPCTSTR UserAgent
[esi+0xc]	LPCSTR Uri
[esi+0x10]	Char DataReceived[4096]
[esi+0x1010]	DWORD NumberOfBytesRead
[esi+0x1014]	DWORD TotalNumberRead

2.1.1. Structures in IDA

The IDA software enables analysts to create custom structures and incorporate them into the disassembled program. Using IDA Pro, one would complete the following steps in order to represent the structure type of *InternetConnection* and its members.

1. Select the structures tab and press the insert key.
2. Name the structure *InternetConnection*.
3. Use the 'd' data key after highlighting one's new structure to create member variables.
    a. One may continuously press the 'd' key to adjust the size to either a BYTE, WORD, or DWORD.

b. Special sizes need to be specified by right clicking a BYTE sized representation of the member variable, selecting array, then typing the size.

4. Rename the structure and its members using the 'n' key as one would any variable or subroutine from within IDA (Eagle, 2011).

After completing the steps above, one should have something similar to that depicted in Figure 7.

```
00000000 InternetConnection struc ; (sizeof=0x1018)
00000000 Connect         dd ?
00000004 OpenAddress     dd ?
00000008 UserAgent       dd ?
0000000C Uri             dd ?
00000010 DataReceived    db 4096 dup(?)
00001010 NumberOfBytesRead dd ?
00001014 TotalNumberRead dd ?
00001018 InternetConnection ends
```

Figure 7: Completed IDA Structure

To apply the above structure to an object member of the matching type within IDA, right click on the member in IDA, scroll to structure offset, and select the matching structure type. The illustration below shows the application of the *UserAgent* and *Uri* structure members to a decompiled program that uses the *InternetConnection* structure type.

```
mov     ecx, [ebp+InternetConnection_Object_1]
mov     [ecx+InternetConnection.UserAgent], offset aMozilla5_0Wind ; "Mozilla/5.0 (Windows
mov     edx, [ebp+InternetConnection_Object_1]
mov     [edx+InternetConnection.Uri], offset aHttpWww_nhl_co ; "http://www.nhl.com"
```

Figure 8: Application of Structure in IDA

2.1.2. Case Study #1 – InternetConnection Structure Type

To further demonstrate the concept of data structures as they are encountered at the assembly level, a case study has been compiled that used an *InternetConnection* object. The object is used as part of a program that reaches out to a remote server over an HTTP connection and computes the total number of bytes served up as a webpage to the client. Each member variable of our structure is used as a part of this process, and the decompiled x86 code generated from IDA will be cross referenced with the higher level source code.

Author Name, email@address

In Figure 9, the *InternetConnection* object is first instantiated and then passed as a variable to the *GetHTML* function, which itself returns the number of bytes returned to our *printf* statement.

```
1.  int main()
2.  {
3.      InternetConnection conn; // object instantiation
4.      printf("Total downloaded %d bytes \n", GetHTML(conn)); // passing to GetHTML
5.      return 0;
6.  }
```

Figure 9: Case Study Main Function

Conversely, the passing of the *conn* object is done with IDA, given the following code in Figure 10.

```
lea     eax, [ebp+var_1020]
push    eax
call    f_GetHTML
```

Figure 10: Passing of Conn Object in x86

Reviewing the above x86 code, at first glance, one would have no reason to believe that *var_1020* is an object pointer to our strucutre. A closer look is now needed at how this is used from within the *f_GetHTML* function. Our first major hint is found as various structure elements are initialized.

```
mov     eax, [ebp+arg_0]
mov     dword ptr [eax+1014h], 0
```

Figure 11: Initialization of Structure Member

The illustration in Figure 11 depicts a variable at the 0x1014 offset from the start of the object being set to zero. In the first line, *arg_0* represents the start address of our object in memory and is passed to *EAX*. The register is then used as a reference point to set member values. At this point, one can begin to infer that *arg_0* is a pointer to an object whose type and members are presently unknown. As we continue to work our way forwards, however, some of

the relationships become obvious as plaintext strings are initially set to some of the structure members.

X86 Disassmebly

```
mov     ecx, [ebp+arg_0]
mov     dword ptr [ecx+8], offset aMozilla5_0Wind ; "Mozilla/5.0 (Windows NT 5.1; rv:28.0) G"
mov     edx, [ebp+arg_0]
mov     dword ptr [edx+0Ch], offset aHttpWww_nhl_co ; "http://www.nhl.com"
push    0               ; dwFlags
push    0               ; lpszProxyBypass
push    0               ; lpszProxy
push    0               ; dwAccessType
mov     eax, [ebp+arg_0]
mov     ecx, [eax+8]
push    ecx             ; lpszAgent
call    ds:InternetOpenA
mov     edx, [ebp+arg_0]
mov     [edx], eax
```

C++ Source Code

```
1.  int GetHTML(InternetConnection &conn)
2.  {
3.      conn.TotalNumberRead = 0;
4.      conn.Useragent = "Mozilla/5.0 (Windows NT 5.1; rv:28.0) Gecko/20100101 Firefox/28.0
    ";
5.      conn.Uri = "http://www.nhl.com";
6.
7.
8.      conn.Connect = InternetOpen(conn.Useragent,INTERNET_OPEN_TYPE_PRECONFIG,NULL, NULL,
     0);
```

Figure 12: Comparative Analysis for Defining Structure Members

From Figure 12, *arg_0* is passed to various register values, which ultimately serves as the starting point for our object. By reviewing the code depicted above, two strings representing a user agent and a URI are set to their respective offsets. Further validation for the user agent variable is given once one sees it pushed to the stack before the call to the API *InternetOpen*. After the call to *InternetOpenA* is made, one can see the returned *HINTERNET* handle stored in another member offset of our structure type. The source code at this point has a very similar flow of execution.

When another API call in the program is made to *InternetOpenUrlA*, one gets more validation concerning the *URI* string that was initialized earlier. In addition, one is able to actualize one more member of our structure using the offset [edx+4]. The member variable is seen storing the *HINTERNET* handle for this call shortly after it is made in Figure 13.

X86 Disassmebly

```
push    0                   ; dwContext
push    400100h             ; dwFlags
push    0                   ; dwHeadersLength
push    0                   ; lpszHeaders
mov     ecx, [ebp+arg_0]
mov     edx, [ecx+0Ch]
push    edx                 ; lpszUrl
mov     eax, [ebp+arg_0]
mov     ecx, [eax]
push    ecx                 ; hInternet
call    ds:InternetOpenUrlA
mov     edx, [ebp+arg_0]
mov     [edx+4], eax
```

C++ Source Code

```
1.  conn.OpenAddress = InternetOpenUrl(conn.Connect, conn.Uri, NULL, 0, INTERNET_FLAG_PRAGM
    A_NOCACHE|INTERNET_FLAG_KEEP_CONNECTION, 0);
```

Figure 13: Accessing Structure Members Before WinAPI Call

It is worth noting at this point that for the user agent and *URI* variables seen initialized earlier in the disassembly, it was not immediately assumed the variables were used as such until it became clear in the code. Doing so would be a novice mistake, as malicious code authors often purposefully initialize variables to something seemingly benign as a simple means to obfuscate its true intent. It is therefore essential that the reverse engineer lets the code do the talking, and lets the true meaning of each variable reveal itself in how it is utilized.

Once a call is made to *InternetReadFile*, one can gain two more definitions for the *InternetConnection* structure type. However, the way the compiler represents the structure offsets is somewhat different than what was have previously observed. Instead of seeing the object offset referenced in open and close braces, one sees the beginning of the object moved to the register, then an add operation done immediately followed by a push to the API function *InternetReadFile*. The behavior can be observed on lines 1-3 and 5-7 of Figure 14 below.

```
mov     ecx, [ebp+arg_0]
add     ecx, 1010h
push    ecx             ; lpdwNumberOfBytesRead
push    1000h           ; dwNumberOfBytesToRead
mov     edx, [ebp+arg_0]
add     edx, 10h
push    edx             ; lpBuffer
mov     eax, [ebp+arg_0]
mov     ecx, [eax+4]
push    ecx             ; hFile
call    ds:InternetReadFile
```

Figure 14: Contextual Insight into Structure Member Role

The usage alongside the API call *InternetReadFile* also gives contextual insight into how they fit into the *InternetConnection* structure. The first example can be correlated back to the *lpdwNumberOfBytesRead*, and the second can be tied directly back to the *lpBuffer* variable. Through MSDN, one can infer what these variables are used for, as well as how much space they take up within our structure. For example, one would know that *lpBuffer* is going to be populated with 0x1000 (4,096) bytes each time based on the *dwNumberOfBytesToRead* attribute.

The final structure member's usage becomes clear after the call to *InternetReadFile* is made. The x86 code enters a small block that seems to compute a summation based on the *InternetConnection* member attribute *NumberOfBytesRead*. The annoted x86 representation in Figure 15 illustrates this concept. The structure member 'NumberOfBytesRead' is applied on the fourth line since previously achieved understanding of that value was done in Figure 14. Doing so assists in deriving meaning for what is ultimately the *TotalNumberRead* structure member.

```
mov     eax, [ebp+arg_0] ; pass object pointer to eax
mov     ecx, [eax+1014h] ; retrieve value at 0x1014 offset
mov     edx, [ebp+arg_0] ; pass object pointer to edx
add number of bytes read to present ecx value
add     ecx, [edx+InternetConnection.NumberOfBytesRead]
mov     eax, [ebp+arg_0] ; pass object pointer to eax
mov     [eax+1014h], ecx ; store summation result at 0x1014 offset
jmp     short loc_4010B1
```

Figure 15: Annotated Usage of Structure Member

2.2. Standard Classes

Classes themselves are merely an expanded concept of data structures. The primary difference between them and a normal data structure is they can contain functions as well as member variables (CPlusPlus, n.d.). Compiled code that uses classes can sometimes lose the 'class' identity if the compiler has better ideas on how the code can be interpreted. This is true for all written code, as all code that is written is open to interpretation by the compiler and is often put through a gauntlet of optimizations. The provided case study examines this phenomenon as it applies to standard classes in greater detail.

2.2.1. Case Study #2 – Square Class to Calculate Area

In order to demonstrate how classes may be represented at the assembly level, a sample proof of concept program was created using a class named 'Square.' The class contains two variables and two functions that utilize them. The program simply initializes an object of type *Square*, computes the area, then exits after printing. Source code of the *Square* class is illustrated in Figure 16.

```
1.  /*
2.  Jason Batchelor
3.  Reverse Engineering
4.  07/20/2015
5.
6.  Rationale: Program for illustrating objects and structures.
7.  Reference: http://www.cplusplus.com/doc/tutorial/classes/
8.  */
9.
10. #include <iostream>
11. using namespace std;
12.
13. class Square {
14.     int width, height;
15.   public:
16.     void set_values(int,int);
17.     int area() {return width*height;}
18. };
19.
20. void Square::set_values (int x, int y) {
21.   width = x;
22.   height = y;
23. }
```

Figure 16: Source Code of Square Class

When the a *Square* object is declared, the memory contents that represent the object will not represent what one might consider to be a *Square* object when looking at the source code. In fact, when reviewing the memory contents of what should be the Square object, one is met with only two variables after it is initialized, both of which are depicted in Figure 17 as 4-byte width and height values.

Address	Hex dump
0025FE24	03 00 00 00 04 00 00 00

Figure 17: Heap of Square Object

So, where are the additional function pointers to the 'set_values' and 'area' subroutines? The compiler, in this case, chose not to use them that way. Compiler settings and their underlying logic can change the execution flow of a program dramatically. Adjusting for size or speed can greatly impact how defined classes are interpreted, and ultimately will change how a decompiled program is logically represented versus the original source code. This has an amplifying effect when considering byte-code based signatures for detecting malware and is a simplistic evasion technique employed by malware authors. While the intricacies of compiler theory are far beyond the scope of this paper, it is nonetheless worth mentioning.

Considering the compiler's intentions, and the resulting disassembled code, it is fair to say the 'class' is treated like a 'structure' in so far as the structure used as an example does not contain any void pointers to class functions. The class functions used by *Square* are called as if they are separate elements in Figure 18. Further exploration of each of these functions is necessary to understand the purpose of the two member variables in absence of original source code.

X86 Disassmebly

```
push    4               ; int
push    3               ; int
lea     ecx, [ebp+var_8] ; pointer to our object
call    sub_401020      ; initialize object structure
```

C++ Source Code

```
1.  int main () {
2.    Square sq;
3.    sq.set_values(3,4);
```

Figure 18: Comparison of Calling the set_values Function

When the object is defined a call to its set values method is made. Outside in the main function, the two bits of code seem to be very similar; however, within the x86 example one sees the address of a variable being loaded into the *ECX* register which demonstrates the *thiscall* convention. Observing this behavior also implies this object may be a global in scope (Sabanal & Yason, 2007). Without the benefit of source code, it should be clear to a user that the function being traced into takes two arguments and will likely make use of the 'this' pointer, contained in *ECX* at some point.

```
sub_401020 proc near

var_4= dword ptr -4
arg_0= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+var_4], ecx ; pointer to object
mov     eax, [ebp+var_4] ; eax = object pointer
mov     ecx, [ebp+arg_0] ; arg0, previously pushed int type
mov     [eax], ecx      ; set integer to dereferenced object field
mov     edx, [ebp+var_4] ; pointer to object
mov     eax, [ebp+arg_4] ; arg4, previously pushed int type
mov     [edx+4], eax    ; set integer to dereferenced object field
mov     esp, ebp
pop     ebp
retn    8
sub_401020 endp
```

Figure 19: Initialization of Square Object

When tracing into the *sub_401020* function from Figure 19, the two integer functions are set within the passed object at a predefined offset. The 'this' pointer is moved into *EAX* and *EDX* from *var_4* below, after it is filled with *ECX*, which was the initial object pointer from earlier, set from the function caller. Both *EAX* and *EDX* are dereferenced in order to populate the object with its member values. At this point, it should be clear that the main purpose of this subrountine is to initialize this object.

When tracing back to the original function caller in Figure 20, one sees the this pointer passed again to *ECX*. Before steping into *sub_401000*, it takes zero arguments but likely does something with the previously initialized object. While these inferences inform analysis and gauge expectations, letting the code tell the real story is extremely important.

```
lea     ecx, [ebp+var_8] ; pointer to our object
call    sub_401000
```

Figure 20: Stepping into Function Using Object

Upon stepping into this function one can clearly see it load the this pointer from the *ECX* register into a local function variable. From there, the same pointer is used to populate the *EAX* and *ECX* register. The move from *var_4* to *ECX* is redundant, and it should be noted here the example assembly was compiled using no optimizations to help better illustrate what is going on.

```
sub_401000 proc near

var_4= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+var_4], ecx ; pointer to object moved from ecx
mov     eax, [ebp+var_4] ; pointer moved to eax
mov     ecx, [ebp+var_4] ; pointer moved back to ecx
mov     eax, [eax]       ; eax dereferenced and stored
imul    eax, [ecx+4]     ; eax is multiplied with dereferenced object integer
mov     esp, ebp
pop     ebp
retn
sub_401000 endp
```

Figure 21: Computation Performed Using Structure Members

The object pointers are both dereferenced and used in a multiplication operation to retrieve the calculated area in Figure 21. Even if one did not have the benefit of knowing from previous analysis what the member value types of the object were, observing the *imul* (integer multiplication) operation would be enough to tell us they are integers. The result is stored in *EAX* and when the function returns that value it is pushed to the *printf* API in order to display the result to the end user.

```
push    eax
push    offset aAreaD   ; "Area: %d"
call    ds:printf
```

Figure 22: Final Print of Previously Computed Result

2.3. Polymorphic Classes

The concept of polymorphism is a powerful way to extend a class from its base and still be type compatible. Concepts, such as class inheritance, are fundamental to polymorphism because the derived classes inherit their base member attributes. In this way, a derived class can leverage base class attributes in their own internal functions (CPlusPlus, n.d.). Virtual functions are a related concept to polymorphic classes, in that they are used to extend functions to derived classes in a similar fashion to how base member variables are. Calls to virtual functions are done dynamically, and therefore use of the 'this' pointer is expected (Skochinsky, 2011). The case study presented in the next section goes over this concept and how it may be encountered at the assembly level.

2.3.1. Case Study #3 – Finding the Magic Numbers For Each Shape

The compiled case study presents three different shapes with the base class called 'Polygon'. This base class has two derived member classes called 'Triangle' and 'Rectangle' which share the base class attributes 'width' and 'height.' They also inherit a virtual function called 'magic,' which simply returns a computed integer based on the inherited member variables and its invocation. What makes these derived classes polymorphic is the fact that they

are inheriting a virtual function (CPlusPlus, n.d.). Figure 23 shows a depiction of the source code for this program with the three classes discussed.

```
1.  class Polygon {
2.    protected:
3.      int width, height;
4.    public:
5.      void set_values (int a, int b)
6.        { width=a; height=b; }
7.      virtual int magic ()
8.        { return width + height; }
9.  };
10.
11. class Rectangle: public Polygon {
12.   public:
13.     int magic ()
14.       { return width * height; }
15. };
16.
17. class Triangle: public Polygon {
18.   public:
19.     int magic ()
20.       { return (width - height); }
21. };
```

Figure 23: C++ Code Defining Three Classes

Within the main component of the program, we initialize three variables; *rect*, *trgl*, and *poly* respectively, and then for each of them define three more variables that are pointers to *Polygon* and assigned the addresses of a derived class (CPlusPlus, n.d.). The assignments and initilizations are depicted in Figure 24.

```
1.  int main () {
2.    Rectangle rect;
3.    Triangle trgl;
4.    Polygon poly;
5.    Polygon * rectangle = &rect
6.    Polygon * triangle = &trgl;
7.    Polygon * polygon = &poly;
8.    rectangle->set_values (10,6);
9.    triangle->set_values (7,5);
10.   polygon ->set_values (4,5);
```

Figure 24: Assignments Made to Base and Derived Classes

At the assembly level, one can cross reference each setup and initialization of our objects with the different respective subroutines. Figure 25 demonstrates the three pointers to *Polygon* as they are being setup. Note once again, we see the address being loaded into the *ECX* register, falling in line with the *thiscall* convention.

```
lea     ecx, [ebp+var_20] ; load address of variable
call    sub_401130        ; sub to define attributes of object
lea     ecx, [ebp+var_C]
call    sub_401150
lea     ecx, [ebp+var_30]
call    sub_401170
```

Figure 25: x86 Initialization of Classes

By tracing into the top most subroutine, it can clearly be seen that the function override takes place for the derived class in Figure 26.

```
f_initialize_derived_class_1 proc near

ptr_object= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+ptr_object], ecx
mov     ecx, [ebp+ptr_object] ; pointer to object
call    f_initialize_base_class ; must define base class
mov     eax, [ebp+ptr_object]
pointer to new address is overridden with offset to new function
mov     dword ptr [eax], offset off_40215C ; derived classes virtual function
mov     eax, [ebp+ptr_object]
mov     esp, ebp
pop     ebp
retn
f_initialize_derived_class_1 endp
```

Figure 26: Initialization of Derived Class

Within this subroutine two main things happen. First, the pointer to the derived class object is stored in *ECX*, and a subroutine is called which defines the base class. Next, that same object is loaded, and the address is overridden to point to a new offset, the address of the derived classes virtual function. Without the benefit of source code one can still make this assumption

because upon stepping into the function *f_initialize_base_class*, one sees very much the same behavior as compared to the caller.

```
f_initialize_base_class proc near

ptr_object= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+ptr_object], ecx
mov     eax, [ebp+ptr_object] ; move address of object to eax
dereference eax and set to virtual function offset
mov     dword ptr [eax], offset off_402154
mov     eax, [ebp+ptr_object]
mov     esp, ebp
pop     ebp
retn
f_initialize_base_class endp
```

Figure 27: Setup for Base Class Object

As seen in Figure 27, the object pointer is again loaded, and derferenced itself to store the offset to a virtual function at the virtual memory address of 0x402154. However, once this function has completed, the same address from [eax] is loaded and dereferenced again to point to a new virtual function address in Figure 26. The base class definition in Figure 27 is overridden, and instead is the derived classes virtual function at *off_40215C*.

In Figure 28, we take this knowledge combined with our applied annotations and function names thus far, and go back to IDA where the objects were being initialized. It should be now obvious which of the subroutines initializes our *Polygon* base class and which two are derived.

```
lea     ecx, [ebp+var_20] ; load address of variable
call    f_initialize_derived_class_1 ; sub to define attributes of object
lea     ecx, [ebp+var_C]
call    f_initialize_derived_class_2
lea     ecx, [ebp+var_30]
call    f_initialize_base_class
```

Figure 28: Applying Definition of Class Types

Continuing onward through the flow of execution in Figure 29, next is the three separate calls to the *set_values* function. At the x86 level this is called in a very similar fashion to how it is observed in the original source code.

X86 Disassmebly

```
push    6
push    10
mov     ecx, [ebp+var_14]
call    sub_401000
push    5
push    7
mov     ecx, [ebp+var_10]
call    sub_401000
push    5
push    4
mov     ecx, [ebp+var_24]
call    sub_401000
```

C++ Source Code

```
1.  rectangle->set_values (10,6);
2.  triangle->set_values (7,5);
3.  polygon ->set_values (4,5);
```

Figure 29: Comparative Analysis of Code When set_values is Called

Within the subroutine depicted above, one can start to build a structure depiction based on the base class. The figure below contains annotations depicting this in greater detail. It is important to note specifically that earlier, the starting offset for our structure was defined when our object was being initialized, the remaining offsets +4 and +8, respectively, are set here to two separate integers. This is known because the caller's arguments, when pushed to the stack, were exposed as such in Figure 30.

```
; int __stdcall f_set_values(int int_1, int int_2)
f_set_values proc near

ptr_object= dword ptr -4
int_1= dword ptr  8
int_2= dword ptr  0Ch

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+ptr_object], ecx
mov     eax, [ebp+ptr_object]
mov     ecx, [ebp+int_1]
mov     [eax+4], ecx    ; structure member set to int value
mov     edx, [ebp+ptr_object]
mov     eax, [ebp+int_2]
mov     [edx+8], eax    ; structure member set to int value
mov     esp, ebp
pop     ebp
retn    8
f_set_values endp
```

Figure 30: Assigning Class Member Variables

After considering the information thus far, one can define the following structure using IDA in Figure 31.

```
00000000 struc_1_shape   struc ; (sizeof=0xC)
00000000 virtual_func_offset dd ?
00000004 int_1           dd ?
00000008 int_2           dd ?
0000000C struc_1_shape   ends
```

Figure 31: Structure Definition for Class Objects

At the assembly level, the virtual functions are called by referencing the object pointer for the class type and moving the first element of that class into a register that is later dereferenced and called into directly, like any other function. Interestingly, one can see from Figure 32 that the *thiscall* convention operation of moving the object to *ECX* is done seperately from the task of setting up the *EAX* register for the eventual call into the virtual function.

X86 Disassmebly

```
mov     eax, [ebp+ptr_derived_class_obj_1] ; this pointer moved to eax
mov     edx, [eax]      ; eax is dereferenced to locate function offset
mov     ecx, [ebp+ptr_derived_class_obj_1] ; this pointer moving to ecx
mov     eax, [edx]      ; function offset is dereferenced to get address
call    eax             ; function is finally called into via eax register
push    eax
push    offset aRectangleMagic ; "Rectangle Magic Number: %d\n"
call    ds:printf
```

C++ Source Code

```
1.  printf ("Rectangle Magic Number: %d\n", rectangle -> magic());
```

Figure 32: Code Comparison Calling into the Virtual Function

By leveraging earlier analysis concerning how classes were being initialized, it should be clear to us precisely which one of the three virtual functions is chosen to be executed. Concerning the case from above, one can observe the initial offset of the structure to be pointing to the virtual function for the base class at *off_402154*. However, immediately following that, the same offset was overridden to be a pointer to *off_40215c*. Figure 33 shows the virtual table that contains the three functions. If any class has virtual methods, the compiler creates a sequence of pointer entries in a table to those methods (Skochinsky, 2011). Each of them are annotated to point out how they fit together, into the overall picture.

```
off_402154      dd offset virtual_base_class_function
                                        ; DATA XREF: f_initialize_base_class+A↑o
                dd offset unk_4022BC
off_40215C      dd offset virtual_derived_class_function_1
                                        ; DATA XREF: f_initialize_derived_class_1+12↑o
                dd offset unk_402270
off_402164      dd offset virtual_derived_class_function_2
                                        ; DATA XREF: f_initialize_derived_class_2+12↑o
```

Figure 33: Table of Pointer Entries to Virtual Functions

Figure 34 steps into the sub routine of the base class. From within the base class, the object retrieved from the 'this' pointer is used to access member variables, perform some basic arithmatic, and return an integer.

X86 Disassmebly

```
; int __cdecl virtual_base_class_function()
virtual_base_class_function proc near

ptr_object= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
mov     [ebp+ptr_object], ecx
mov     eax, [ebp+ptr_object]
mov     eax, [eax+struc_1_shape.int_1]
mov     ecx, [ebp+ptr_object]
add     eax, [ecx+struc_1_shape.int_2]
mov     esp, ebp
pop     ebp
retn
virtual_base_class_function endp
```

C++ Source Code

```
1.  virtual int magic ()
2.    { return width + height; }
```

Figure 34: Code Comparison for Virtual Base Class Function

3. Conclusion

The outlined case studies illustrated common object oriented concepts with the intent of providing the reader with real examples encountered in reverse engineering C++. Reviewing calling conventions, inferring context of register contents based on assembly code, and following the flow of execution allows one to make informed observations on what a data structure or class ultimately represents. When these conclusions are applied to the broader project, it has an amplifying effect on the completeness of the reversing effort.

Understanding the application of object oriented programming concepts, such as data structures, standard classes, and polymorphic classes, and how it is represented in x86 assembly, is an essential skill for the reverse engineer to meet today's challenges. While these concepts may be challenging for the uninitiated, the application of object oriented principles greatly simplifies the reversing process. When tasked to reverse engineer modern malware it is a mandatory skill to possess.

Author Name, email@address

4. References

The Art Of Service. (n.d.). *Pascal calling convention – thiscall.* Retrieved 10 26, 2015, from The Art Of Service: http://theartofservice.com/pascal-calling-convention-thiscall.html

CPlusPlus. (n.d.). *Data Structures.* Retrieved 07 10, 2015, from cplusplus: http://www.cplusplus.com/doc/tutorial/structures/

CPlusPlus. (n.d.). *Polymorphism.* Retrieved June 15, 2015, from cplusplus: http://www.cplusplus.com/doc/tutorial/polymorphism/

CPlusPlus. (n.d.). *Classes (I).* Retrieved June 15, 2015, from cplusplus: http://www.cplusplus.com/doc/tutorial/classes/

Eagle, C. (2011). *The IDA Pro Book 2nd Edition.* Canada: No Starch Press.

HackCraft. (n.d.). *Calling Convetions in Microsoft Visual C++*. Retrieved August 19, 2015, from HackCraft: http://www.hackcraft.net/cpp/MSCallingConventions/

Jönsson, A. (2005, 02 13). *Calling conventions on the x86 platform*. Retrieved 09 05, 2015, from AngelCode: http://www.angelcode.com/dev/callconv/callconv.html

Lawlor. (2006). *Pointers in C and Assembly*. Retrieved 09 05, 2015, from https://www.cs.uaf.edu/2006/fall/cs301/lecture/10_02_pointer.html

Microsoft. (2015). *__cdecl.* Retrieved June 21, 2015, from Windows Dev Center: https://msdn.microsoft.com/en-us/library/zkwh89ks.aspx

Microsoft. (2015). *__stdcall.* Retrieved June 21, 2015, from Windows Dev Center: https://msdn.microsoft.com/en-us/library/zxk0tw93.aspx

Microsoft. (2015). *__thiscall.* Retrieved June 21, 2015, from Windows Dev Center: https://msdn.microsoft.com/en-us/library/ek8tkfbw.aspx

Microsoft. (2015). *InternetCloseHandle Function.* Retrieved June 15, 2015, from Windows Dev Center: https://msdn.microsoft.com/en-us/library/windows/desktop/aa384350(v=vs.85).aspx

Microsoft. (2015). *InternetOpen Function.* Retrieved June 15, 2015, from Windows Dev Center: https://msdn.microsoft.com/en-us/library/windows/desktop/aa385096(v=vs.85).aspx

Author Name, email@address

Microsoft. (2015). *InternetOpenUrl Function.* Retrieved June 15, 2015, from Windows Dev Center: https://msdn.microsoft.com/en-us/library/windows/desktop/aa385098(v=vs.85).aspx

Microsoft. (2015). *InternetReadFile Function.* Retrieved June 15, 2015, from Windows Dev Center: https://msdn.microsoft.com/en-us/library/windows/desktop/aa385103(v=vs.85).aspx

Pollard, J. d. (2010). *The Gen on Function Calling Conventions.* Retrieved June 21, 2015, from Ntl World: http://homepage.ntlworld.com/jonathan.deboynepollard/FGA/function-calling-conventions.html

Sabanal, P. V., & Yason, M. V. (2007). *Reversing C++.* Retrieved June 15, 2015, from Black Hat: https://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07-Sabanal_Yason-WP.pdf

Skochinsky, I. (2011). *Practical C++ Decompilation.* Retrieved June 15, 2015, from Hexblog: http://www.hexblog.com/wp-content/uploads/2011/08/Recon-2011-Skochinsky.pdf

Trifunovic, N. (2001, September 22). *Calling Conventions Demystified.* Retrieved June 21, 2015, from Code Project: http://www.codeproject.com/Articles/1388/Calling-Conventions-Demystified

Author Name, email@address

5. Appendix

5.1. Data Structures Source Code

The following source code was used to create the example used in the first case study presented on data structures:

```
1.  /*
2.  Jason Batchelor
3.  Reverse Engineering
4.  04/13/2014
5.
6.  Rationale: Program for illustrating objects and structures.
7.  */
8.
9.  #pragma comment(lib,"wininet.lib")
10. #include<iostream>
11. #include<Windows.h>
12. #include<wininet.h>
13. #include<cstring>
14. using namespace std;
15.
16.
17. struct InternetConnection
18. {
19.     HINTERNET Connect;
20.     HINTERNET OpenAddress;
21.     LPCTSTR Useragent;
22.     LPCTSTR Uri;
23.     char DataReceived[4096];
24.     DWORD NumberOfBytesRead;
25.     DWORD TotalNumberRead;
26. };
27.
28. int GetHTML(InternetConnection &conn)
29. {
30.     conn.TotalNumberRead = 0;
31.     conn.Useragent = "Mozilla/5.0 (Windows NT 5.1; rv:28.0) Gecko/20100101 Firefox/28.0
    ";
32.     conn.Uri = "http://www.nhl.com";
33.
34.
35.     conn.Connect = InternetOpen(conn.Useragent,INTERNET_OPEN_TYPE_PRECONFIG,NULL, NULL,
     0);
36.
37.     if(!conn.Connect){
38.         printf("Connection Failed or Syntax error\n");
39.         return 0;
40.     }
41.
42.     conn.OpenAddress = InternetOpenUrl(conn.Connect, conn.Uri, NULL, 0, INTERNET_FLAG_P
    RAGMA_NOCACHE|INTERNET_FLAG_KEEP_CONNECTION, 0);
43.
44.     if ( !conn.OpenAddress )
```

Author Name, email@address

```
45.     {
46.         printf("Failed to open URL\n");
47.         InternetCloseHandle(conn.Connect);
48.         return 0;
49.     }
50.
51.     conn.NumberOfBytesRead = 0;
52.     while(InternetReadFile(conn.OpenAddress, conn.DataReceived, 4096, &conn.NumberOfBytesRead) && conn.NumberOfBytesRead)
53.     {
54.         conn.TotalNumberRead += conn.NumberOfBytesRead;
55.     }
56.
57.     InternetCloseHandle(conn.OpenAddress);
58.     InternetCloseHandle(conn.Connect);
59.
60.     return conn.TotalNumberRead;
61. }
62.
63. int main()
64. {
65.     InternetConnection conn;
66.     printf("Total downloaded %d bytes \n", GetHTML(conn));
67.     return 0;
68. }
```

5.2. Standard Classes Source Code

The following code was used to compile the second presented case study on standard classes:

```
1.  /*
2.  Jason Batchelor
3.  Reverse Engineering
4.  07/10/2015
5.
6.  Rationale: Program for illustrating objects and structures.
7.  Reference: http://www.cplusplus.com/doc/tutorial/classes/
8.  */
9.
10. #include <iostream>
11. using namespace std;
12.
13. class Square {
14.     int width, height;
15.   public:
16.     void set_values(int,int);
17.     int area() {return width*height;}
18. };
19.
20. void Square::set_values (int x, int y) {
21.   width = x;
22.   height = y;
23. }
```

Author Name, email@address

```
24.
25. int main () {
26.   Square sq;
27.   sq.set_values(3,4);
28.   printf ("Area: %d", sq.area());
29.   return 0;
30. }
```

5.3. Polymorphic Classes Source Code

The following source code was used to illustrate the third case study on polymorphic classes, and their representation in x86:

```
1.   /*
2.  Jason Batchelor
3.  Reverse Engineering
4.  07/21/2015
5.
6.  Rationale: Program for illustrating virtual tables.
7.  Reference: http://www.cplusplus.com/doc/tutorial/polymorphism/
8.  */
9.
10. // virtual members
11. #include <iostream>
12. using namespace std;
13.
14. class Polygon {
15.   protected:
16.     int width, height;
17.   public:
18.     void set_values (int a, int b)
19.       { width=a; height=b; }
20.     virtual int magic ()
21.       { return width + height; }
22. };
23.
24. class Rectangle: public Polygon {
25.   public:
26.     int magic ()
27.       { return width * height; }
28. };
29.
30. class Triangle: public Polygon {
31.   public:
32.     int magic ()
33.       { return (width - height); }
34. };
35.
36. int main () {
37.   Rectangle rect;
38.   Triangle trgl;
39.   Polygon poly;
```

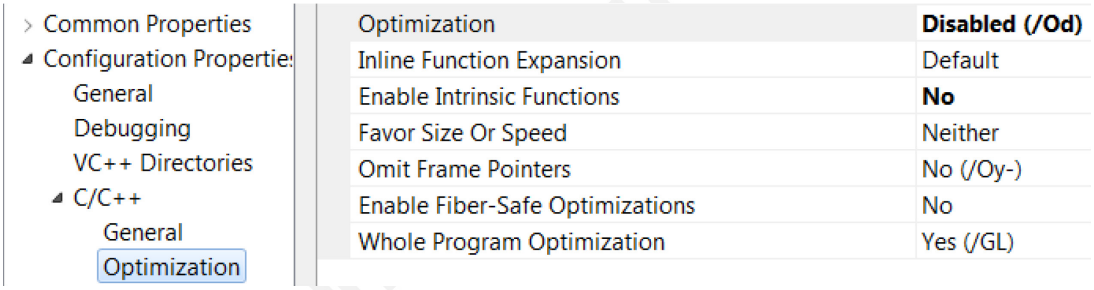
Author Name, email@address

```
40.   Polygon * rectangle = &rect
41.   Polygon * triangle = &trgl;
42.   Polygon * polygon = &poly;
43.   rectangle->set_values (10,6);
44.   triangle->set_values (7,5);
45.   polygon ->set_values (4,5);
46.   printf ("Rectangle Magic Number: %d\n", rectangle -> magic());
47.   printf ("Triangle Magic Number: %d\n", triangle -> magic());
48.   printf ("Polygon Magic Number: %d\n", polygon -> magic());
49.   return 0;
50. }
```

5.4. Compiler Options

All of the case study examples were compiled using the following settings. To produce a binary similar to what was reviewed in this paper, please ensure the settings below are applied.

5.4.1. Optimization Settings for Microsoft Visual Studio 2010



5.4.2. Compiler Flags

/Zi /nologo /W3 /WX- /Od /Oy- /D "_MBCS" /Gm- /EHsc /MT /GS /Gy /fp:precise /Zc:wchar_t /Zc:forScope /Fp"Release\[project name].pch" /Fa"Release\" /Fo"Release\" /Fd"Release\vc100.pdb" /Gd /analyze- /errorReport:queue