Maddie Stigler
mgs4ff
11/19/14
postlab9.pdf

1. Inheritance

In order to examine how inheritance is received in x86 assembly code, I created an instance of inheritance through the use of a Contact class inheriting a Name class. Both classes have a constructor, destructor, print method, and a private field. In the main method, I created an instance of a contact, initialized both of its private fields and then called its print method. The data member that the Contact class inherits is the name field, and the data member that it includes on its own is the address field. Both are printed in the print function.

```
public:
  Name() : myName("") { }
  ~Name() { }
  void setName(string theName) {
    myName = theName;
  }
  void print() {
    cout << myName << endl;
  }
private:
  string myName;
};


class Contact: public Name {
public:
  Contact() {
    myAddress = "";
  }
  ~Contact() { }
  void setAddress(string theAddress) {
    myAddress = theAddress;
  }
  void print() {
    Name::print();
    cout << myAddress << endl;
  }
private:
  string myAddress;
};
```

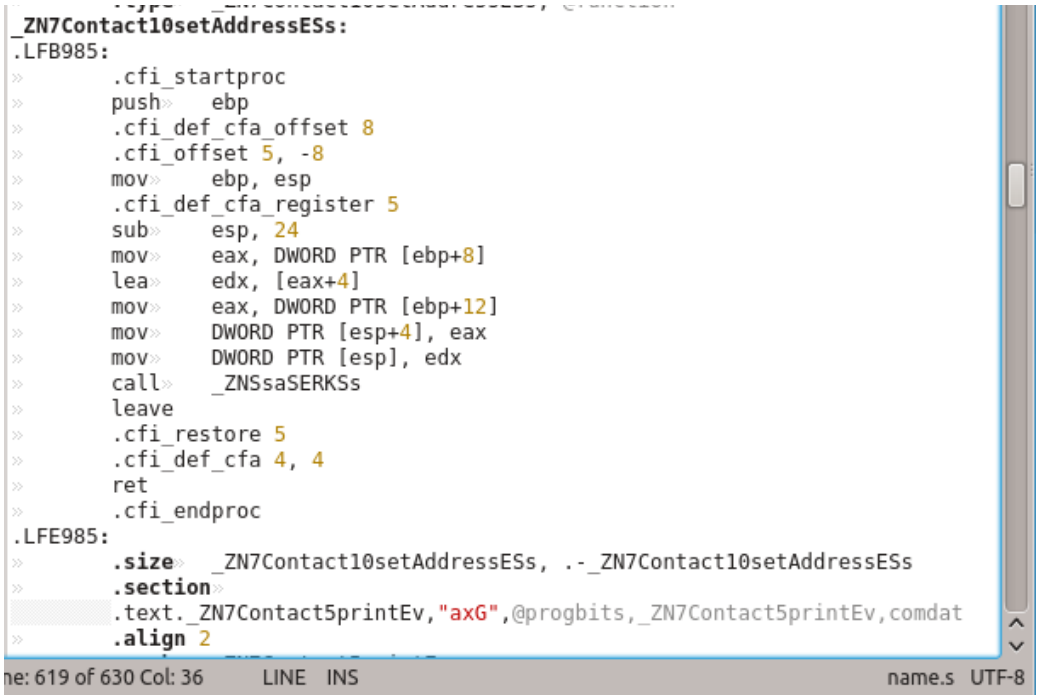
When converted to assembly using the g++ compiler, the resulting assembly code is used to illustrate where in memory data members are stored for the Contact object. First, space is allocated on the stack to store the two data fields initiated in the main method. This space is used to store both the name field inherited from the super class(Name) and the address field defined in the derived class(Contact). This is shown under the Contact label and again in the field label:

```
        .type   _ZN7ContactC2Ev, @function
_ZN7ContactC2Ev:
.LFB980:
        .cfi_startproc
        .cfi_personality 0,__gxx_personality_v0
        .cfi_lsda 0,.LLSDA980
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        push    ebx
        sub     esp, 20
        .cfi_offset 3, -12
        mov     eax, DWORD PTR [ebp+8]
        mov     DWORD PTR [esp], eax
.LEHB2:
        call    _ZN4NameC2Ev
.LEHE2:
        mov     eax, DWORD PTR [ebp+8]
        add     eax, 4
        mov     DWORD PTR [esp], eax
```

*Contact allocates space on the stack and then calls the Name label that contains the Name field.

```
_ZN4Name7setNameESs:
.LFB977:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 24
        mov     eax, DWORD PTR [ebp+8]
        mov     edx, DWORD PTR [ebp+12]
        mov     DWORD PTR [esp+4], edx
        mov     DWORD PTR [esp], eax
        call    _ZNSsaSERKSs
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE977:
```

*The name field is allocated and stored on the stack. This field is inherited by Contact from the Name class



*Space is also allocated for the derived class's(Contact) fields and used to store the address.

When examining the construction and destruction of objects in assembly, I notice it is relative to the class hierarchy. Initially, the base class's constructor is called to initialize the data members inherited from the base class. Then, the derived class's constructor is then called to initialize the data members added in the derived class.

When a user-defined object is instantiated, the constructor is called and space is allocated for the fields defined within the object. When the destructor is called, the assembly destroys the element that the destructor is being called on. The space is reallocated and is now able to be used again. In addition, the destructor will reallocate the space originally allocated by every element it is called on. This means, if a vector were to be initiated and a destructor was called on the vector, it would perform this action on every element contained within the vector.

In terms of class hierarchy, the assembly code lines up fairly well with the c++ code. The object is instantiated and then the constructors for the base class and derived class are called respectively. When the function is over, the destructors are called on the two fields and performed first on the derived class and then on the base class. The general layout of the assembly with respect to class hierarchy is shown below as a snip from the main method.

```
        mov     DWORD PTR [esp], eax
.LEHB9:
        call    _ZN7ContactC1Ev
.LEHE9:
        lea     eax, [esp+20]
        mov     DWORD PTR [esp], eax
        call    _ZNSaIcEC1Ev
        lea     eax, [esp+20]
        mov     DWORD PTR [esp+8], eax
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1
        lea     eax, [esp+16]
        mov     DWORD PTR [esp], eax
.LEHB10:
        call    _ZNSsC1EPKcRKSaIcE
.LEHE10:
        lea     eax, [esp+16]
        mov     DWORD PTR [esp+4], eax
        lea     eax, [esp+24]
        mov     DWORD PTR [esp], eax
.LEHB11:
        call    _ZN4Name7setNameESs
.LEHE11:
        lea     eax, [esp+16]
        mov     DWORD PTR [esp], eax
.LEHB12:
```

The assembly code under the initialization/destruction label holds code for constructors and destructors that are eventually called by the main method. When the static initialization and destruction code is called, the Contact class's destructor is called and then the Name class's destructor is called. This occurs in reverse order than when the constructors were called. The constructors were called in order of base class followed by derived class.

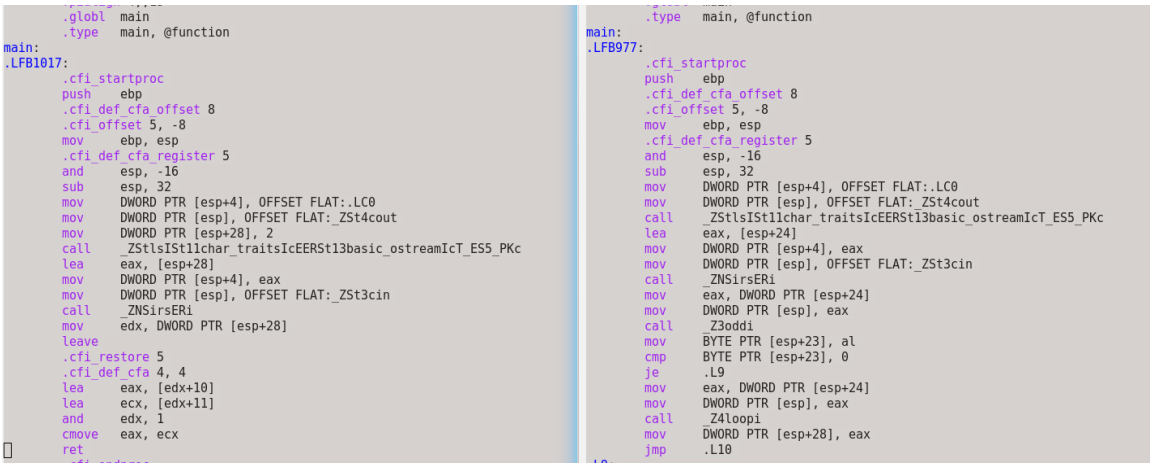
```
_Z41__static_initialization_and_destruction_0ii:
.LFB1036:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 24
        cmp     DWORD PTR [ebp+8], 1
        jne     .L35
        cmp     DWORD PTR [ebp+12], 65535
        jne     .L35
        mov     DWORD PTR [esp], OFFSET FLAT:_ZStL8__ioinit
        call    _ZNSt8ios_base4InitC1Ev
        mov     DWORD PTR [esp+8], OFFSET FLAT:__dso_handle
        mov     DWORD PTR [esp+4], OFFSET FLAT:_ZStL8__ioinit
        mov     DWORD PTR [esp], OFFSET FLAT:_ZNSt8ios_base4InitD1Ev
        call    __cxa_atexit
.L35:
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
.LFE1036:
        .size   _Z41__static_initialization_and_destruction_0ii, .-
        _Z41__static_initialization_and_destruction_0ii
        .type   _GLOBAL__sub_I_main, @function
_GLOBAL__sub_I_main:
.LFB1037:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
```

2. Optimized Code

In order to assess the difference between compiler optimized assembly code and un-optimized assembly, I wrote a simple c++ program that implements a for loop as well as a conditional statement. The main method makes function calls to the for loop function (int loop(int x)) and the conditional statement (Boolean parity(int x)) and returns a value of x depending on the parity of the user defined value. The c++ program is shown below:

```
/*Maddie Stigler
 *mgs4ff
 *11/19/14
 *optimize.cpp
 */

#include <cstdlib>
#include <iostream>

using namespace std;

int loop(int x) {
  for(int i = 0; i<10; i++) {
    x++;
  }
  return x;
}

bool odd(int x) {
  if(x%2 == 0) return false;
  else return true;
}

int main() {
  int x = 2;
  int xAdd;
  cout << "Enter a number: ";
  cin >> x;
  bool parity = odd(x);
  if(parity == true) {
    xAdd = loop(x);
  }
  else xAdd = loop(x)+1;
  return xAdd;
}
```

After compiling regularly and saving the resulting assembly file, I then compiled the optimize.cpp program with the –O2 flag. The first optimization I immediately notice is that the optimized assembly code is that the same routines are produced using less x86 code. You can see this just within the main method. While the un-optimized code uses 22 x86 instructions to run the main method, the optimized code uses only 19. Rather than being a direct translation from the c++ code to assembly, the –O2 flag produces an assembly code of only what is essential to making the program run and compile. The two versions are shown below. The optimized is on the left and un-optimized to the right.

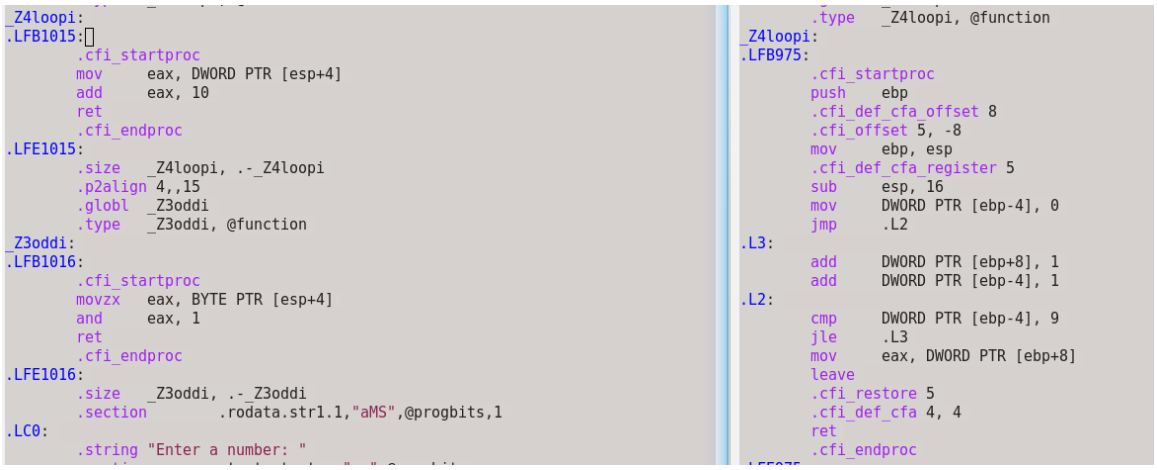


```
        .globl  main
        .type   main, @function
main:
.LFB1017:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        and     esp, -16
        sub     esp, 32
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], OFFSET FLAT:_ZSt4cout
        mov     DWORD PTR [esp+28], 2
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
        lea     eax, [esp+28]
        mov     DWORD PTR [esp+4], eax
        mov     DWORD PTR [esp], OFFSET FLAT:_ZSt3cin
        call    _ZNSirsERi
        mov     edx, DWORD PTR [esp+28]
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        lea     eax, [edx+10]
        lea     ecx, [edx+11]
        and     edx, 1
        cmove   eax, ecx
        ret
```

```
        .type   main, @function
main:
.LFB977:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        and     esp, -16
        sub     esp, 32
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC0
        mov     DWORD PTR [esp], OFFSET FLAT:_ZSt4cout
        call    _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
        lea     eax, [esp+24]
        mov     DWORD PTR [esp+4], eax
        mov     DWORD PTR [esp], OFFSET FLAT:_ZSt3cin
        call    _ZNSirsERi
        mov     eax, DWORD PTR [esp+24]
        mov     DWORD PTR [esp], eax
        call    _Z3oddi
        mov     BYTE PTR [esp+23], al
        cmp     BYTE PTR [esp+23], 0
        je      .L9
        mov     eax, DWORD PTR [esp+24]
        mov     DWORD PTR [esp], eax
        call    _Z4loopi
        mov     DWORD PTR [esp+28], eax
        jmp     .L10
```

One way the –O2 flag did this was by using more efficient x86 opcodes. For instance, in the main method, the un-optimized code uses a lot of mov and conditional jmp commands. These take time as they perform one mov at a time and the conditional commands rely on previous values. The optimized assembly code uses lea commands to optimize multiple mov commands. It also makes use of bitwise operations instead of conditional jumps.

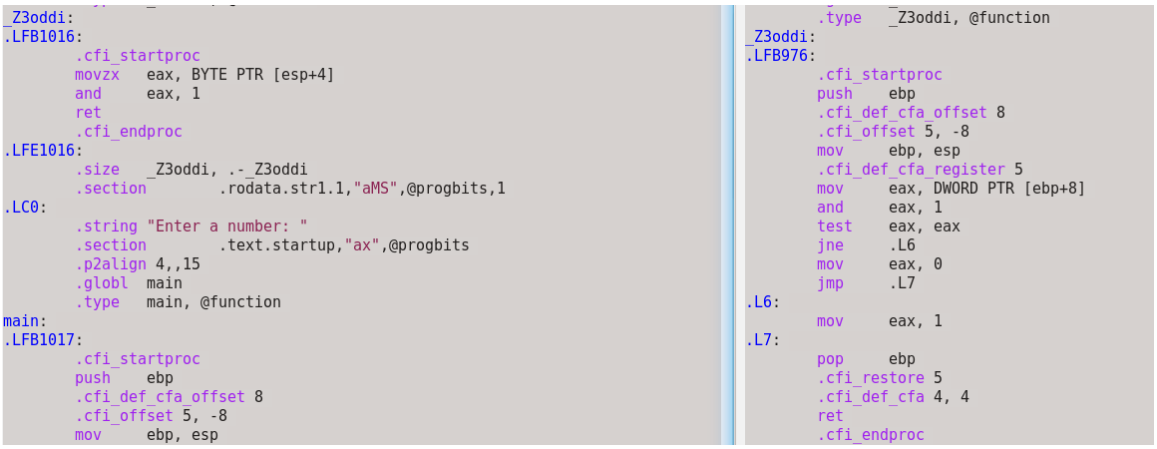
Another optimization the –O2 flag uses is within the loop function. The function loops through 10 times, incrementing the value of x by 1 each time. Once again the code is optimized by producing less x86 instructions. However, this is not due to the addition of more efficient opcodes but an examination of the purpose of the code. While the optimized code recognizes the for loop is simply adding 10 to x, the un-optimized code takes the time to actually go through the for loop adding one each time based on conditional jmp commands. The optimized code adds 10 to x and returns. This is shown below (optimized – left).



```
_Z4loopi:
.LFB1015:
        .cfi_startproc
        mov     eax, DWORD PTR [esp+4]
        add     eax, 10
        ret
        .cfi_endproc
.LFE1015:
        .size   _Z4loopi, .-_Z4loopi
        .p2align 4,,15
        .globl  _Z3oddi
        .type   _Z3oddi, @function
_Z3oddi:
.LFB1016:
        .cfi_startproc
        movzx   eax, BYTE PTR [esp+4]
        and     eax, 1
        ret
        .cfi_endproc
.LFE1016:
        .size   _Z3oddi, .-_Z3oddi
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "Enter a number: "
```

```
        .type   _Z4loopi, @function
_Z4loopi:
.LFB975:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 16
        mov     DWORD PTR [ebp-4], 0
        jmp     .L2
.L3:
        add     DWORD PTR [ebp+8], 1
        add     DWORD PTR [ebp-4], 1
.L2:
        cmp     DWORD PTR [ebp-4], 9
        jle     .L3
        mov     eax, DWORD PTR [ebp+8]
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Looking at the Boolean parity(x) function, it is also significantly shorter in the optimized assembly. The optimized assembly looks at the function as a whole and omits the if else statements determining x's parity and instead uses the bitwise and operator to return the parity of x. The un-optimized code uses mov and jmp commands paired with cmp opcodes to determine the parity of x and then return. While the un-optimized x86 code provides a literal translation of the c++ code, the optimized looks at the goal first and then returns the most efficient means of achieving that goal. The following code illustrates this function:



```
_Z3oddi:
.LFB1016:
        .cfi_startproc
        movzx   eax, BYTE PTR [esp+4]
        and     eax, 1
        ret
        .cfi_endproc
.LFE1016:
        .size   _Z3oddi, .-_Z3oddi
        .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
        .string "Enter a number: "
        .section        .text.startup,"ax",@progbits
        .p2align 4,,15
        .globl  main
        .type   main, @function
main:
.LFB1017:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
```

```
        .type   _Z3oddi, @function
_Z3oddi:
.LFB976:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        mov     eax, DWORD PTR [ebp+8]
        and     eax, 1
        test    eax, eax
        jne     .L6
        mov     eax, 0
        jmp     .L7
.L6:
        mov     eax, 1
.L7:
        pop     ebp
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Something else to consider when comparing these two assembly files is the calling procedure used. In the un-optimized code, the odd function makes use of the calling procedure, thus updating the stack by pushing on a base pointer and local variables. This is not only time-consuming but also uses memory. The optimized code doesn't bother updating the base pointer and allocating room on the stack. This is something we also discussed in lecture. If the code is simple enough and the parameters are known, it is not necessary to push ebp on the stack. This doesn't violate the calling convention because the subroutine is still set up correctly. The –O2 flag recognizes there is no need for this and omits the callee prologue from the loop and parity functions.

In general, some common optimizations that were illustrated here and produced using the –O2 flag include avoiding redundancy, using less code, avoiding jumps by using straight line code, and strength reduction. These techniques focus on using less code and more efficient code. Demonstrated in the optimization.cpp file that I optimized using the flag was especially the strength reduction technique that replaces complex operations with simpler ones. This is shown in both function calls by replacing mov and jmp commands with lea and bitwise commands. This is also an example of straight line code/branch free code. This optimizes assembly by limiting the use of loops and conditional branches.

http://en.wikipedia.org/wiki/Optimizing_compiler