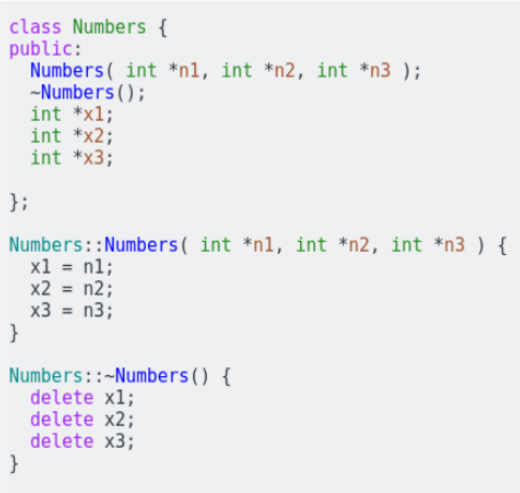
Maria Hito
mh4wt
CS 2150: Post-lab 9
4/13/17

Lab report

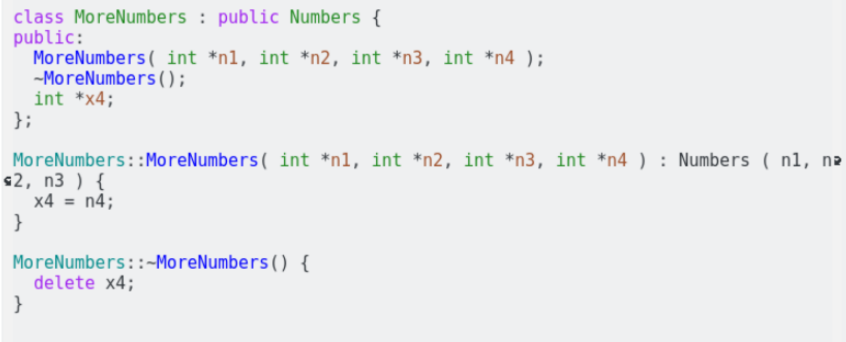
Inheritance

In order to explore the assembly language that is created with classes, I created a simple "Numbers" class which holds three integer pointers, and a subclass called "MoreNumbers" which holds an additional integer pointer. The code for the class Numbers is shown in Figure 1 and the code for the subclass MoreNumbers is shown in Figure 2.



```
class Numbers {
public:
  Numbers( int *n1, int *n2, int *n3 );
  ~Numbers();
  int *x1;
  int *x2;
  int *x3;

};

Numbers::Numbers( int *n1, int *n2, int *n3 ) {
  x1 = n1;
  x2 = n2;
  x3 = n3;
}

Numbers::~Numbers() {
  delete x1;
  delete x2;
  delete x3;
}
```

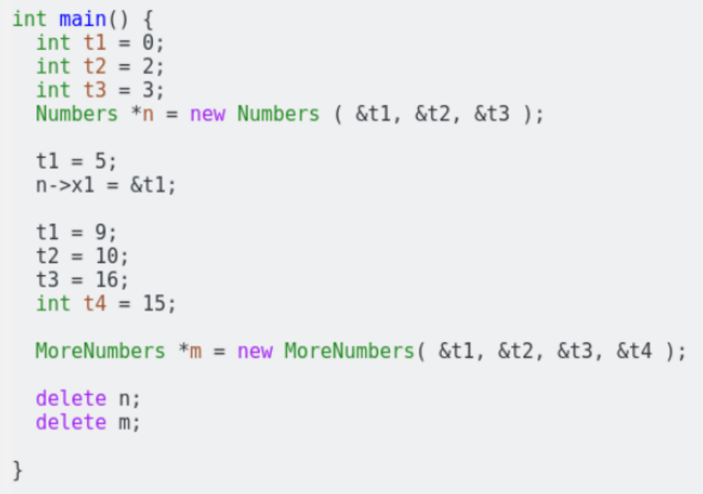
Figure 1



```
class MoreNumbers : public Numbers {
public:
  MoreNumbers( int *n1, int *n2, int *n3, int *n4 );
  ~MoreNumbers();
  int *x4;
};

MoreNumbers::MoreNumbers( int *n1, int *n2, int *n3, int *n4 ) : Numbers ( n1, n
2, n3 ) {
  x4 = n4;
}

MoreNumbers::~MoreNumbers() {
  delete x4;
}
```

Figure 2

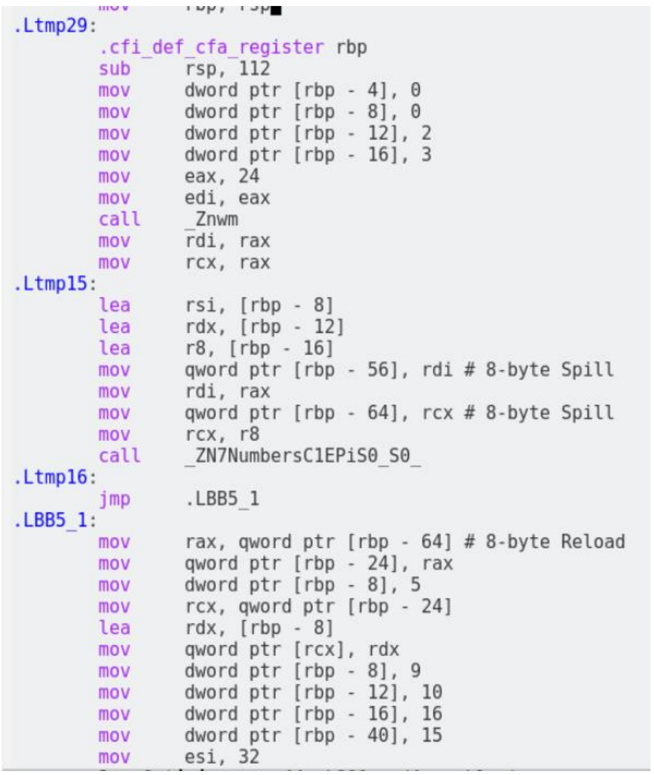
As we can see in Figure 3 the main function of the program simply instantiates various integers, and creates one instance of Numbers and one of MoreNumbers, and then deletes them in order to be able to properly examine the constructor and destructor of both a normal object and an object which inherits some of its data.



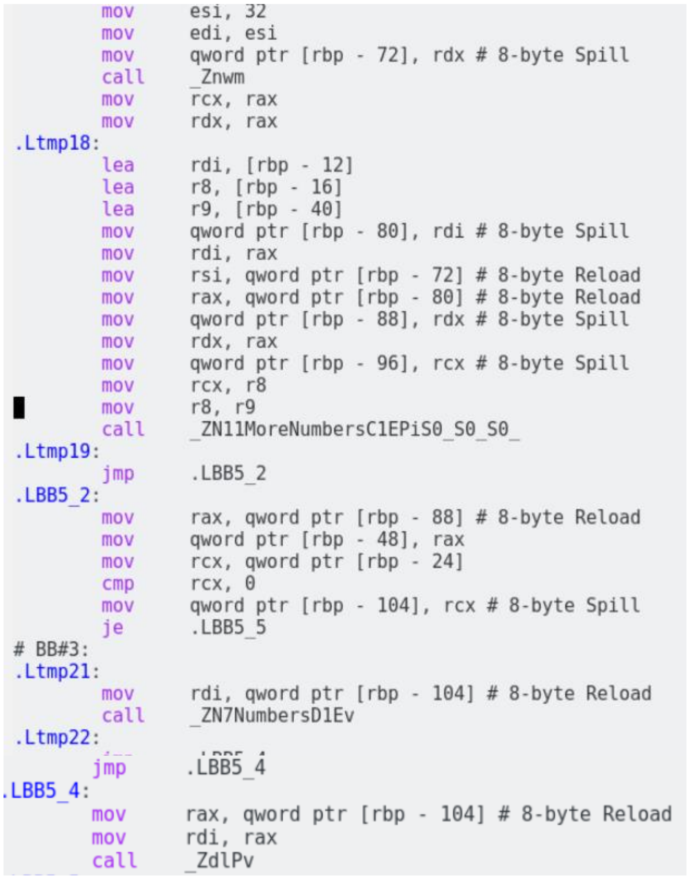
```
int main() {
  int t1 = 0;
  int t2 = 2;
  int t3 = 3;
  Numbers *n = new Numbers ( &t1, &t2, &t3 );

  t1 = 5;
  n->x1 = &t1;

  t1 = 9;
  t2 = 10;
  t3 = 16;
  int t4 = 15;

  MoreNumbers *m = new MoreNumbers( &t1, &t2, &t3, &t4 );

  delete n;
  delete m;

}
```

Figure 3

The first few lines in the snippet are showing the three arguments for the Numbers constructor (0, 2, 3) as written in the C++. Below the lines of argument we can see the command move eax, 24. From the online search, I found that it may be passing in the size of the arguments (24 = 8 * 3, 8 for each pointer). The function that is called is called Znwm doesn't seem to be in any part of the assembly file, and it is called again later in the code to call the constructor for MoreNumbers, I think this is probably to do the inherited functionality from the base-class's constructor. However, it appears to be a function that is called before the constructor, which I believe to be the function ZN7NumbersC1EPisS0_S0_. The next thing I noticed was that it appears that the return value (the object itself) is stored 40 bytes behind the base pointer. I know that 44 bytes behind the base pointer lies the t1 member variable, as command mov dword ptr [rbp – 8], 5 seems to be setting it to five. This means that the first four bytes (40-44 behind the base) are probably the implicit "this", and the following four byte chunks are the data members.



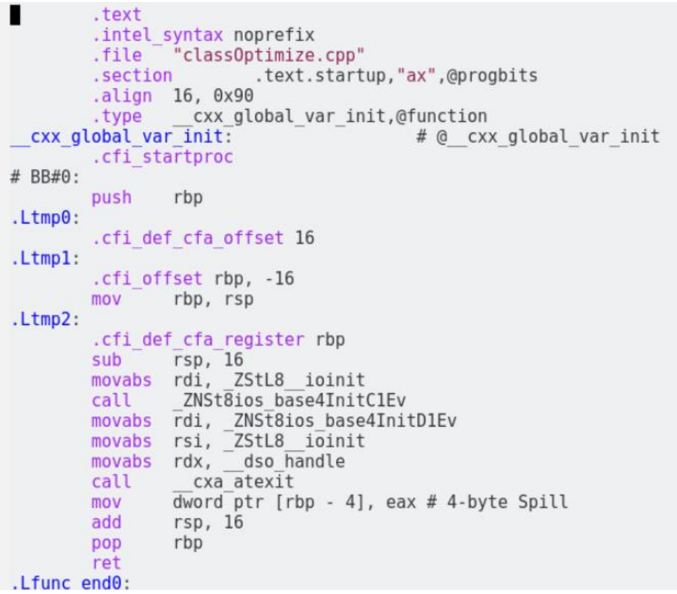
```
        mov     rbp, rsp
.Ltmp29:
        .cfi_def_cfa_register rbp
        sub     rsp, 112
        mov     dword ptr [rbp - 4], 0
        mov     dword ptr [rbp - 8], 0
        mov     dword ptr [rbp - 12], 2
        mov     dword ptr [rbp - 16], 3
        mov     eax, 24
        mov     edi, eax
        call    _Znwm
        mov     rdi, rax
        mov     rcx, rax
.Ltmp15:
        lea     rsi, [rbp - 8]
        lea     rdx, [rbp - 12]
        lea     r8, [rbp - 16]
        mov     qword ptr [rbp - 56], rdi # 8-byte Spill
        mov     rdi, rax
        mov     qword ptr [rbp - 64], rcx # 8-byte Spill
        mov     rcx, r8
        call    _ZN7NumbersC1EPiS0_S0_
.Ltmp16:
        jmp     .LBB5_1
.LBB5_1:
        mov     rax, qword ptr [rbp - 64] # 8-byte Reload
        mov     qword ptr [rbp - 24], rax
        mov     dword ptr [rbp - 8], 5
        mov     rcx, qword ptr [rbp - 24]
        lea     rdx, [rbp - 8]
        mov     qword ptr [rcx], rdx
        mov     dword ptr [rbp - 8], 9
        mov     dword ptr [rbp - 12], 10
        mov     dword ptr [rbp - 16], 16
        mov     dword ptr [rbp - 40], 15
        mov     esi, 32
```



```
        mov     esi, 32
        mov     edi, esi
        mov     qword ptr [rbp - 72], rdx # 8-byte Spill
        call    _Znwm
        mov     rcx, rax
        mov     rdx, rax
.Ltmp18:
        lea     rdi, [rbp - 12]
        lea     r8, [rbp - 16]
        lea     r9, [rbp - 40]
        mov     qword ptr [rbp - 80], rdi # 8-byte Spill
        mov     rdi, rax
        mov     rsi, qword ptr [rbp - 72] # 8-byte Reload
        mov     rax, qword ptr [rbp - 80] # 8-byte Reload
        mov     qword ptr [rbp - 88], rdx # 8-byte Spill
        mov     rdx, rax
        mov     qword ptr [rbp - 96], rcx # 8-byte Spill
        mov     rcx, r8
        mov     r8, r9
        call    _ZN11MoreNumbersC1EPiS0_S0_S0_
.Ltmp19:
        jmp     .LBB5_2
.LBB5_2:
        mov     rax, qword ptr [rbp - 88] # 8-byte Reload
        mov     qword ptr [rbp - 48], rax
        mov     rcx, qword ptr [rbp - 24]
        cmp     rcx, 0
        mov     qword ptr [rbp - 104], rcx # 8-byte Spill
        je      .LBB5_5
# BB#3:
.Ltmp21:
        mov     rdi, qword ptr [rbp - 104] # 8-byte Reload
        call    _ZN7NumbersD1Ev
.Ltmp22:
        jmp     .LBB5_4
.LBB5_4:
        mov     rax, qword ptr [rbp - 104] # 8-byte Reload
        mov     rdi, rax
        call    _ZdlPv
```

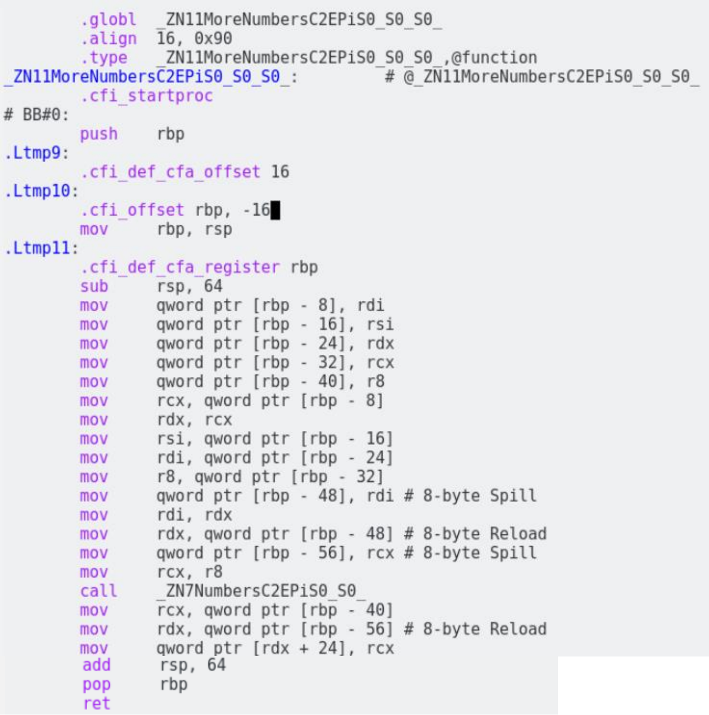
Figure 4

Command line ZN11NumbersC1EPisS0_S0_ S0_ seems to be the call to the constructor of MoreNumbers, and in name alone, it seems to have a lot in common with the constructor of Numbers, except for the addition of “More” and one extra “S0_” suffix. This is probably a result of the inheritance. I was expecting the assembler to automatically call both constructors, perhaps calling the base-class’s constructor as a part of the call to the constructor of the subclass, and I predicted correctly.



```
        .text
        .intel_syntax noprefix
        .file   "classOptimize.cpp"
        .section        .text.startup,"ax",@progbits
        .align  16, 0x90
        .type   __cxx_global_var_init,@function
__cxx_global_var_init:                  # @__cxx_global_var_init
        .cfi_startproc
# BB#0:
        push    rbp
.Ltmp0:
        .cfi_def_cfa_offset 16
.Ltmp1:
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp2:
        .cfi_def_cfa_register rbp
        sub     rsp, 16
        movabs  rdi, _ZStL8__ioinit
        call    _ZNSt8ios_base4InitC1Ev
        movabs  rdi, _ZNSt8ios_base4InitD1Ev
        movabs  rsi, _ZStL8__ioinit
        movabs  rdx, __dso_handle
        call    __cxa_atexit
        mov     dword ptr [rbp - 4], eax # 4-byte Spill
        add     rsp, 16
        pop     rbp
        ret
.Lfunc_end0:
```

Figure 5

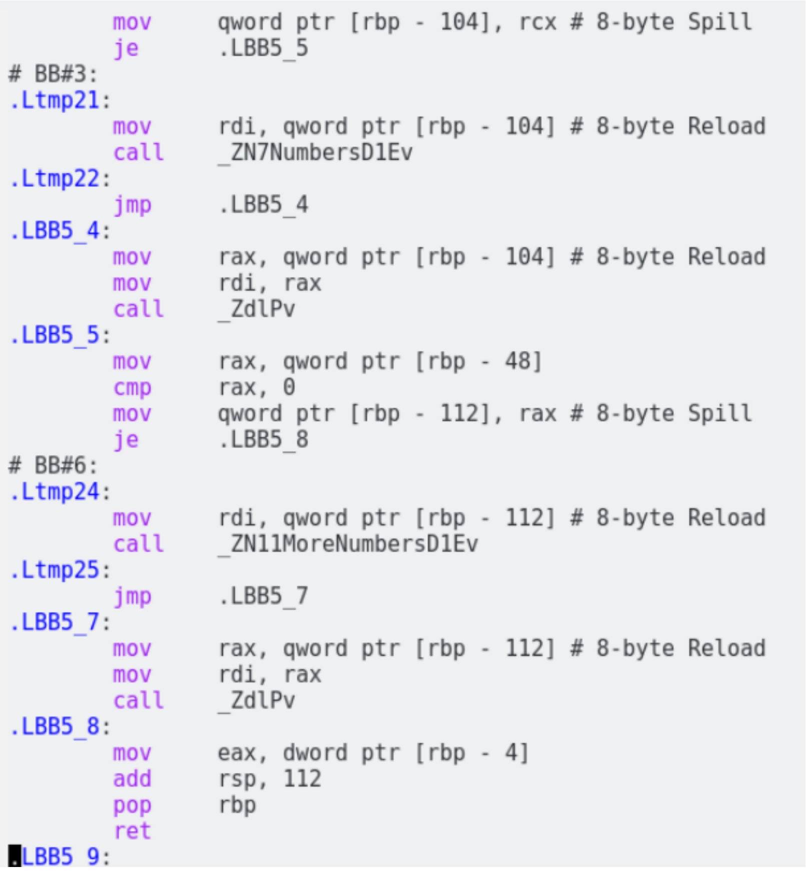


```
        .globl  _ZN11MoreNumbersC2EPiS0_S0_S0_
        .align  16, 0x90
        .type   _ZN11MoreNumbersC2EPiS0_S0_S0_,@function
_ZN11MoreNumbersC2EPiS0_S0_S0_:         # @_ZN11MoreNumbersC2EPiS0_S0_S0_
        .cfi_startproc
# BB#0:
        push    rbp
.Ltmp9:
        .cfi_def_cfa_offset 16
.Ltmp10:
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp11:
        .cfi_def_cfa_register rbp
        sub     rsp, 64
        mov     qword ptr [rbp - 8], rdi
        mov     qword ptr [rbp - 16], rsi
        mov     qword ptr [rbp - 24], rdx
        mov     qword ptr [rbp - 32], rcx
        mov     qword ptr [rbp - 40], r8
        mov     rcx, qword ptr [rbp - 8]
        mov     rdx, rcx
        mov     rsi, qword ptr [rbp - 16]
        mov     rdi, qword ptr [rbp - 24]
        mov     r8, qword ptr [rbp - 32]
        mov     qword ptr [rbp - 48], rdi # 8-byte Spill
        mov     rdi, rdx
        mov     rdx, qword ptr [rbp - 48] # 8-byte Reload
        mov     qword ptr [rbp - 56], rcx # 8-byte Spill
        mov     rcx, r8
        call    _ZN7NumbersC2EPiS0_S0_
        mov     rcx, qword ptr [rbp - 40]
        mov     rdx, qword ptr [rbp - 56] # 8-byte Reload
        mov     qword ptr [rdx + 24], rcx
        add     rsp, 64
        pop     rbp
        ret
```

Figure 6

Figure 5 shows the body of the Numbers constructor, and Figure 6 shows the body of the MoreNumbers constructor. As is notorious on Figure 6, we can see the command ZN7NumbersC2EPisS0_S0_ in the MoreNumbers constructor calls the Numbers constructor within itself. However, I could not exactly tell how the data is manipulated after the constructor is called though.

I believe that per the x86 naming conventions of clang++, the constructors are always the name of the class followed by a C, and the destructors with a D. The destructor functions are the next called in the main function, as we see in Figure 7 the two command lines are: _ZN7NumbersD1Ev and _ZN11NumbersD1Ev.



```
        mov     qword ptr [rbp - 104], rcx # 8-byte Spill
        je      .LBB5_5
# BB#3:
.Ltmp21:
        mov     rdi, qword ptr [rbp - 104] # 8-byte Reload
        call    _ZN7NumbersD1Ev
.Ltmp22:
        jmp     .LBB5_4
.LBB5_4:
        mov     rax, qword ptr [rbp - 104] # 8-byte Reload
        mov     rdi, rax
        call    _ZdlPv
.LBB5_5:
        mov     rax, qword ptr [rbp - 48]
        cmp     rax, 0
        mov     qword ptr [rbp - 112], rax # 8-byte Spill
        je      .LBB5_8
# BB#6:
.Ltmp24:
        mov     rdi, qword ptr [rbp - 112] # 8-byte Reload
        call    _ZN11MoreNumbersD1Ev
.Ltmp25:
        jmp     .LBB5_7
.LBB5_7:
        mov     rax, qword ptr [rbp - 112] # 8-byte Reload
        mov     rdi, rax
        call    _ZdlPv
.LBB5_8:
        mov     eax, dword ptr [rbp - 4]
        add     rsp, 112
        pop     rbp
        ret
.LBB5_9:
```

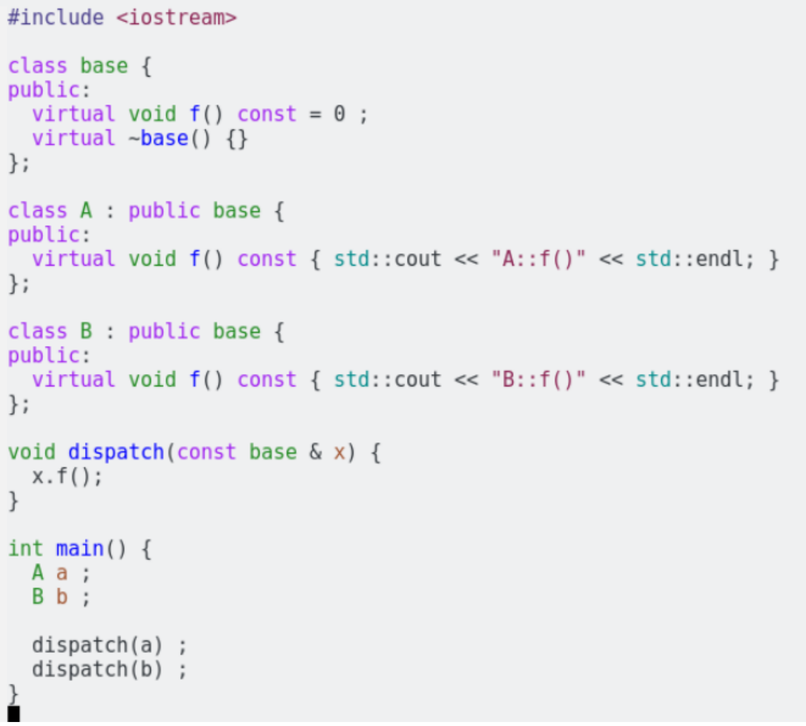
Figure 7

The prologue for the first destructor seems to be data passed in through the RDI register from behind the base pointer. This is probably the "this" pointer for the object that is allocated somewhere on the heap. The next function called with "_ZdlPv" is kind of confusing, but from purely the name it sounds like it deletes a pointer variable, and perhaps it is responsible for deallocating the data members of the instance of the Numbers class. From searching through the Internet, I found references to this strange function on gnu.org and realized that it is a part of GCC and stands for "operator delete (void*)" meaning it deletes a void pointer (a point of any type). It seems to me like this is probably the function used to delete all pointers in every single destructor.

As we can see in the picture, the function is called again after the MoreNumbers destructor is called, this makes me believe that this Zdlpv is the function to actually free the memory itself, whereas the other functions that share the namesake of their respective classes exist simply to perform any other functionality that might need to exist for an objects destruction.

Dynamic Dispatch

To explore how Dynamic dispatch works in Assembly code I created a program were I had to use virtual functions, as we can see in Figure 1, to help the compiler determine the runtime of the function. The line x.f () gets executed twice, but a different function gets called each time. Dynamic dispatch is implemented by means of a virtual function table that has the address of the final overrider for the complement object.

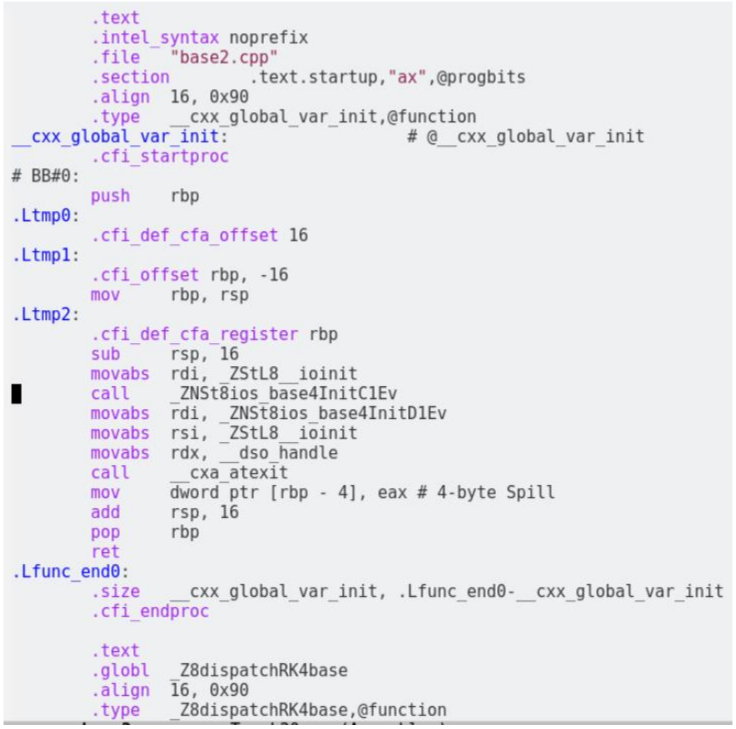


```
#include <iostream>

class base {
public:
  virtual void f() const = 0 ;
  virtual ~base() {}
};

class A : public base {
public:
  virtual void f() const { std::cout << "A::f()" << std::endl; }
};

class B : public base {
public:
  virtual void f() const { std::cout << "B::f()" << std::endl; }
};

void dispatch(const base & x) {
  x.f();
}

int main() {
  A a ;
  B b ;

  dispatch(a) ;
  dispatch(b) ;
}
```

Figure 1

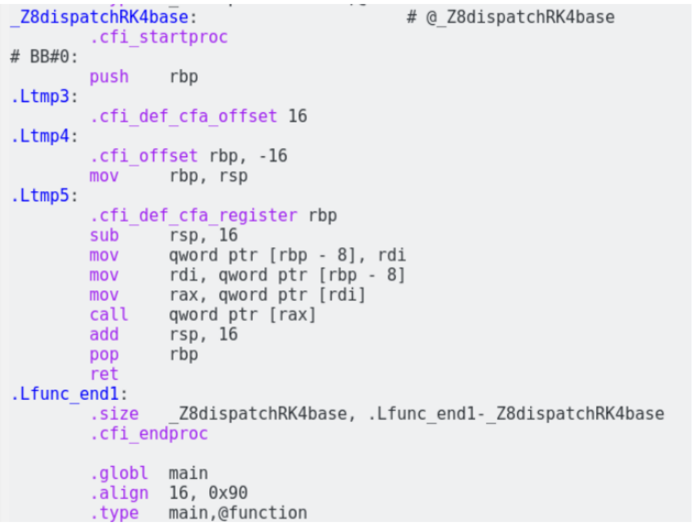
After creating the Assembly code the first thing I noticed was the opcode movabs, which was something I have not seem before in any of my programs, was being used for the passing values of the function _ZNSt8ios_base4InitC1Ev. The searching about the meaning of this opcode I found that is just a GAS specific way to enforce encoding in a 64 bit memory offset. Movabs should perform the same way as the standard move opcode.

Figure 2 and Figure 3 show how the line x.f () gets called twice in the dispatch function. The lines were virtual dispatch takes place in Assembly are shown in Figure 3. From the picture we can see that the line mov rdi, qword ptr [rdi – 8] is loading the vtable from the object, then loading the function address from the vtable to the rax register in the following line: mov rax, qword ptr [rdi] (rdi represents the value that ‘this’ is pointing to) , then calling the function directly (call qword ptr [rax]). So we see that the address is fetched and called all in one statement. Finally the stack pointer is moved back up to clean off the arguments that were pushed before the call. Depending on compiler implementations, in C++ is common to the see the caller clean off the arguments.



```
        .text
        .intel_syntax noprefix
        .file   "base2.cpp"
        .section        .text.startup,"ax",@progbits
        .align  16, 0x90
        .type   __cxx_global_var_init,@function
__cxx_global_var_init:                  # @__cxx_global_var_init
        .cfi_startproc
# BB#0:
        push    rbp
.Ltmp0:
        .cfi_def_cfa_offset 16
.Ltmp1:
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp2:
        .cfi_def_cfa_register rbp
        sub     rsp, 16
        movabs  rdi, _ZStL8__ioinit
        call    _ZNSt8ios_base4InitC1Ev
        movabs  rdi, _ZNSt8ios_base4InitD1Ev
        movabs  rsi, _ZStL8__ioinit
        movabs  rdx, __dso_handle
        call    __cxa_atexit
        mov     dword ptr [rbp - 4], eax # 4-byte Spill
        add     rsp, 16
        pop     rbp
        ret
.Lfunc_end0:
        .size   __cxx_global_var_init, .Lfunc_end0-__cxx_global_var_init
        .cfi_endproc

        .text
        .globl  _Z8dispatchRK4base
        .align  16, 0x90
        .type   _Z8dispatchRK4base,@function
```

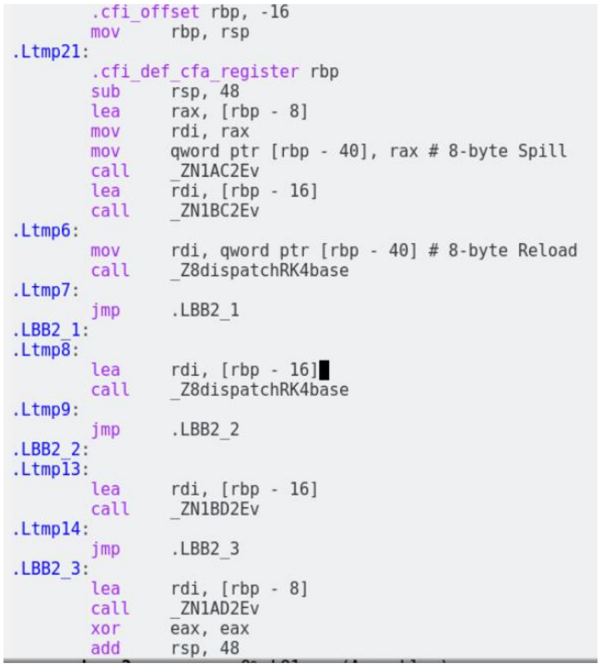
Figure 2



```
_Z8dispatchRK4base:                     # @_Z8dispatchRK4base
        .cfi_startproc
# BB#0:
        push    rbp
.Ltmp3:
        .cfi_def_cfa_offset 16
.Ltmp4:
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp5:
        .cfi_def_cfa_register rbp
        sub     rsp, 16
        mov     qword ptr [rbp - 8], rdi
        mov     rdi, qword ptr [rbp - 8]
        mov     rax, qword ptr [rdi]
        call    qword ptr [rax]
        add     rsp, 16
        pop     rbp
        ret
.Lfunc_end1:
        .size   _Z8dispatchRK4base, .Lfunc_end1-_Z8dispatchRK4base
        .cfi_endproc

        .globl  main
        .align  16, 0x90
        .type   main,@function
```

Figure 3

The code below shows the part of the main method. The end of the dispatch function ends with “sub rsp, 48” and the epilog starts with the instruction “add rsp, 48”. As we see in the picture, the instructions between the prolog and epilog access stack content using RSP as a reference without having any push and pop instructions intervening in the function body.



```
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp21:
        .cfi_def_cfa_register rbp
        sub     rsp, 48
        lea     rax, [rbp - 8]
        mov     rdi, rax
        mov     qword ptr [rbp - 40], rax # 8-byte Spill
        call    _ZN1AC2Ev
        lea     rdi, [rbp - 16]
        call    _ZN1BC2Ev
.Ltmp6:
        mov     rdi, qword ptr [rbp - 40] # 8-byte Reload
        call    _Z8dispatchRK4base
.Ltmp7:
        jmp     .LBB2_1
.LBB2_1:
.Ltmp8:
        lea     rdi, [rbp - 16]
        call    _Z8dispatchRK4base
.Ltmp9:
        jmp     .LBB2_2
.LBB2_2:
.Ltmp13:
        lea     rdi, [rbp - 16]
        call    _ZN1BD2Ev
.Ltmp14:
        jmp     .LBB2_3
.LBB2_3:
        lea     rdi, [rbp - 8]
        call    _ZN1AD2Ev
        xor     eax, eax
        add     rsp, 48
```

Figure 4

Sources

http://lists.gnu.org/archive/html/bugbinutils/2010-01/msg00047.html

http://www.codemachine.com/article_x64deepdive.html

http://www.stackoverflow.com/questions/20147054/hot-does-dynamic-dispatch-happen-in-assembly