Adarsh Solanki

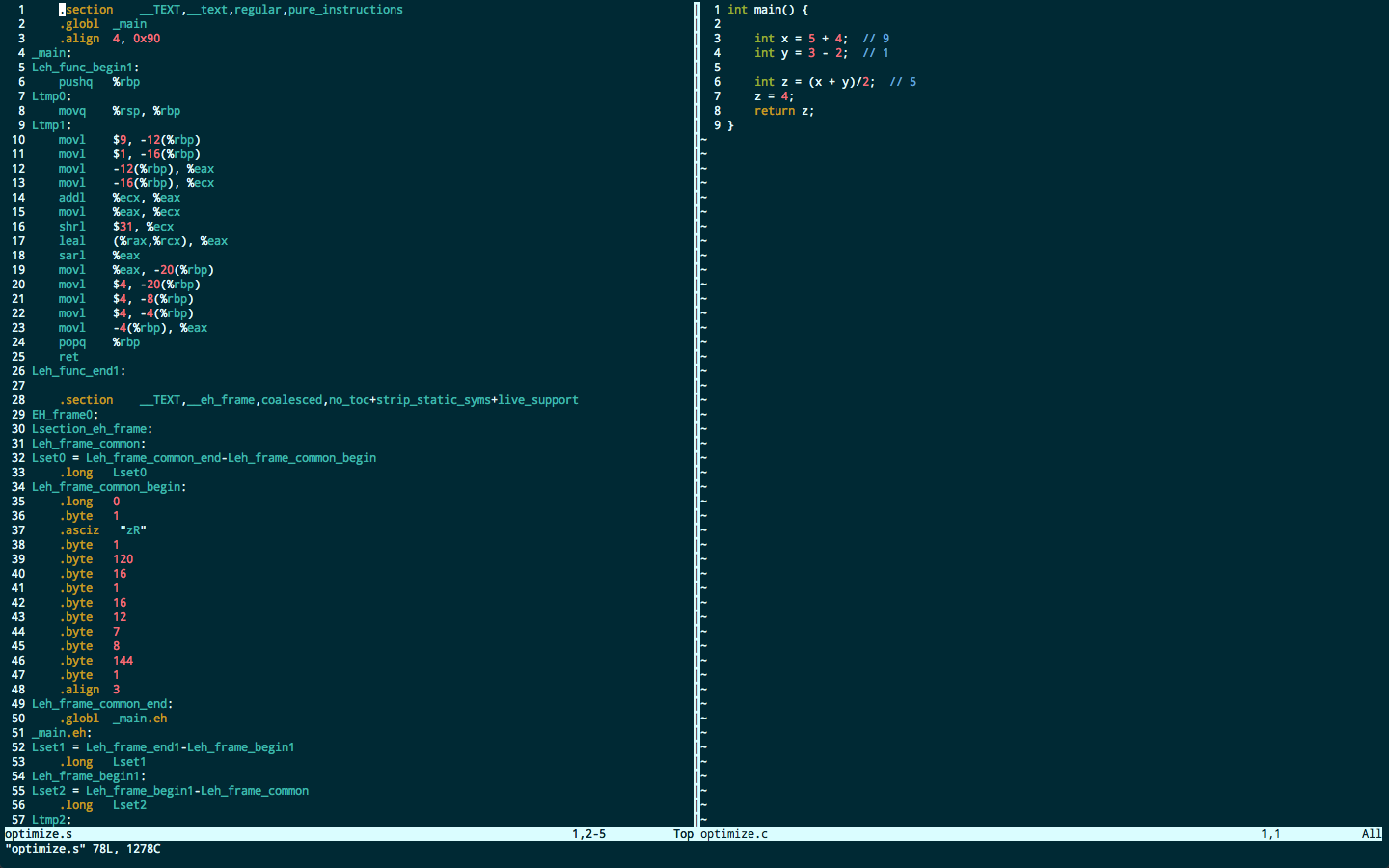
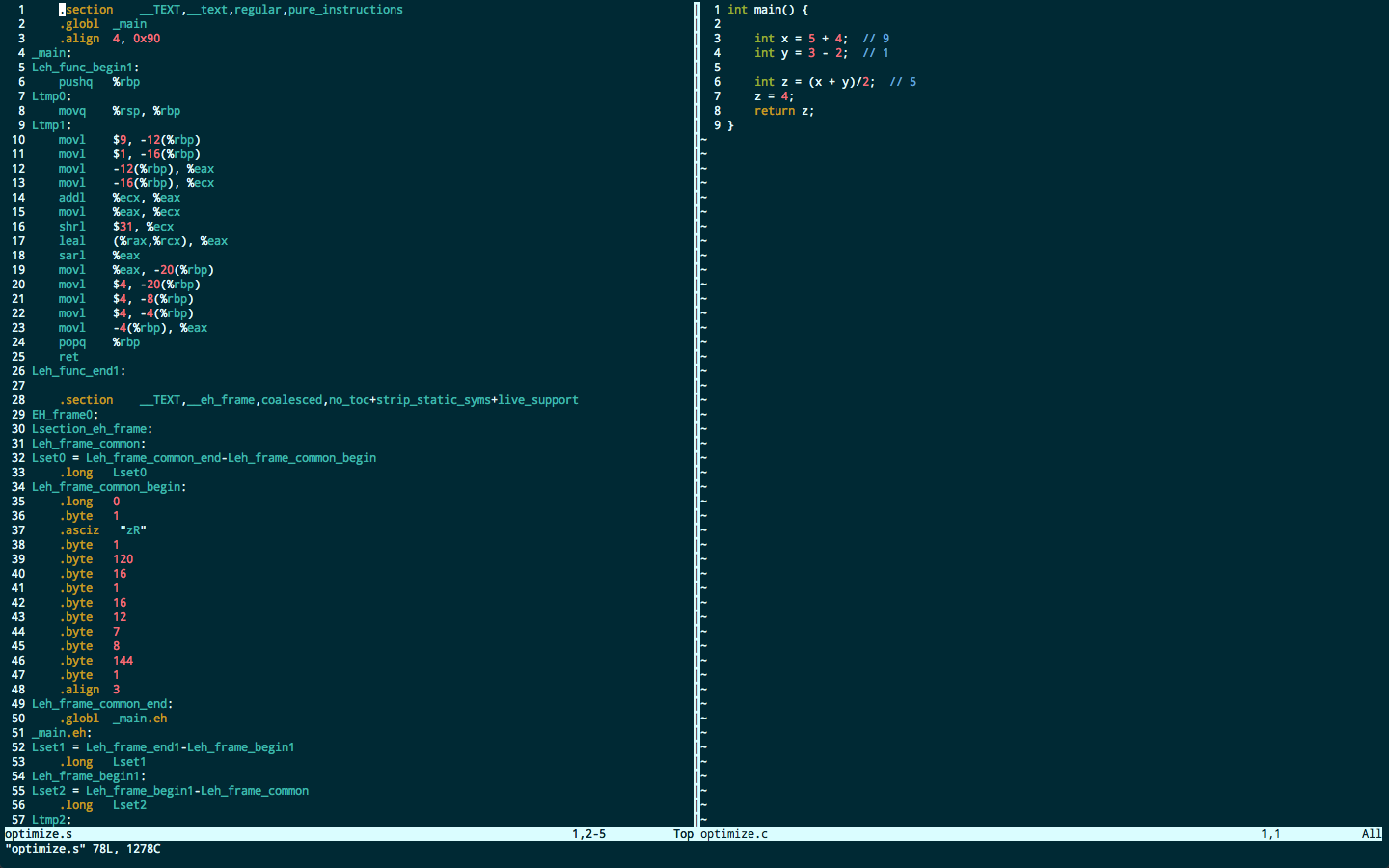
As5nr

CS 2150: In-Lab 9

4/11/12

## Optimized Code

For this section of the lab, I compiled a simple C file (tried C++, but the generated assembly was identical for both according to diff because no C++ specific features were used)

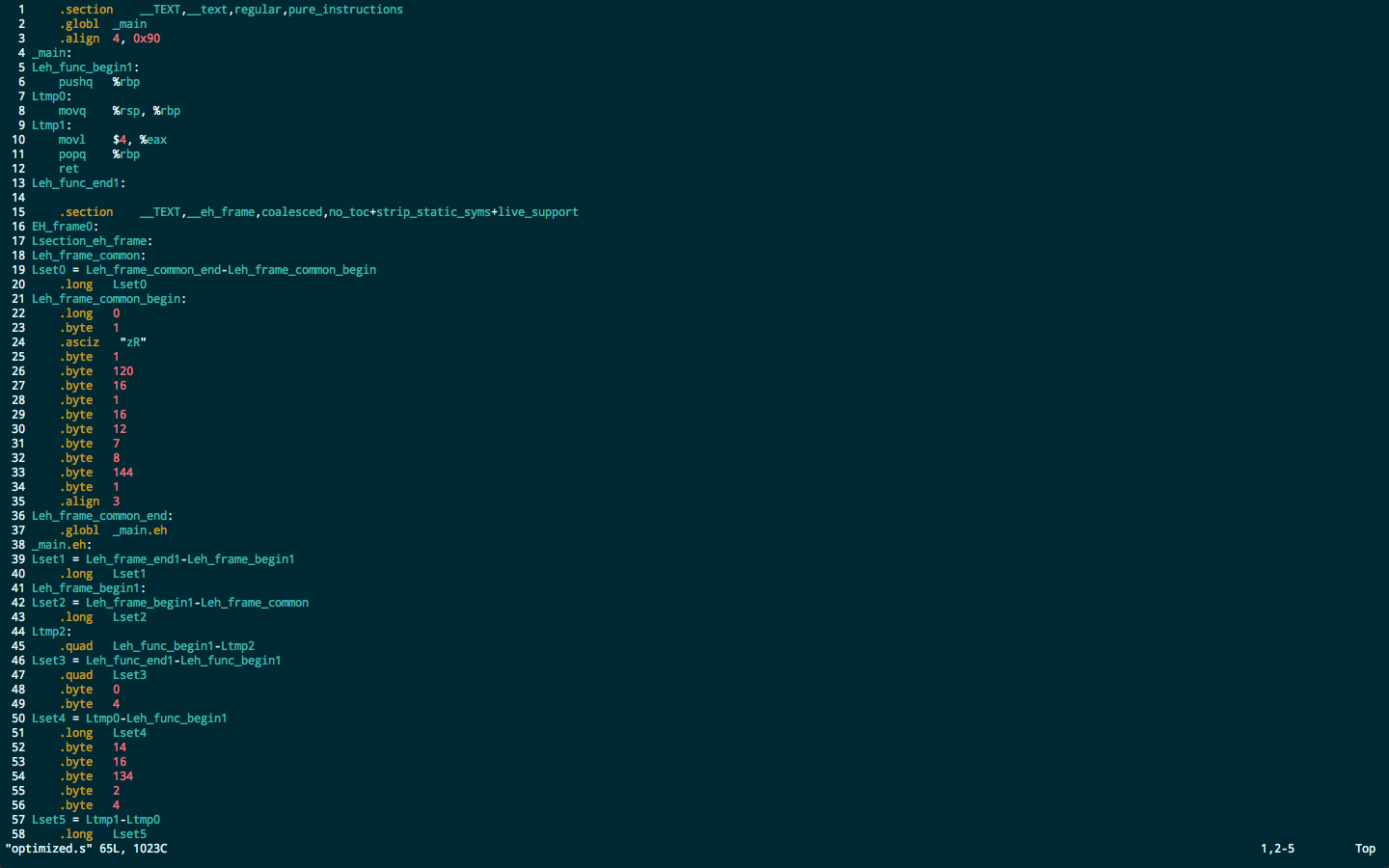


*Above: original C code*

*Right: basic g++ assembly output*

*Below: optimized (O2) g++*

*assembly output*



The difference with the –O2 flag is tremendous! The entire main function was a 17 instruction x86 segment that closely matched the source C code. There are definitely also some inefficiencies, for example at the very end, there are a lot of extraneous 4’s being placed onto the stack and then eventually one is pushed into the return register and the function returns.

Conversely, on the optimized assembly, g++ did whatever it could to minimize the number of instructions required to run the output assembly, so it took the time to pre-process things like the return value. The main function is 5 instructions now, 4 of which are standard boilerplate like saving/restoring the base pointer and calling ret to return. The way I see it, the O2 flag gives g++ the directive to take extra time now to make it more efficient later, pre-compute anything that you already have enough information to compute and cut-out any unnecessary movement/storage of data.

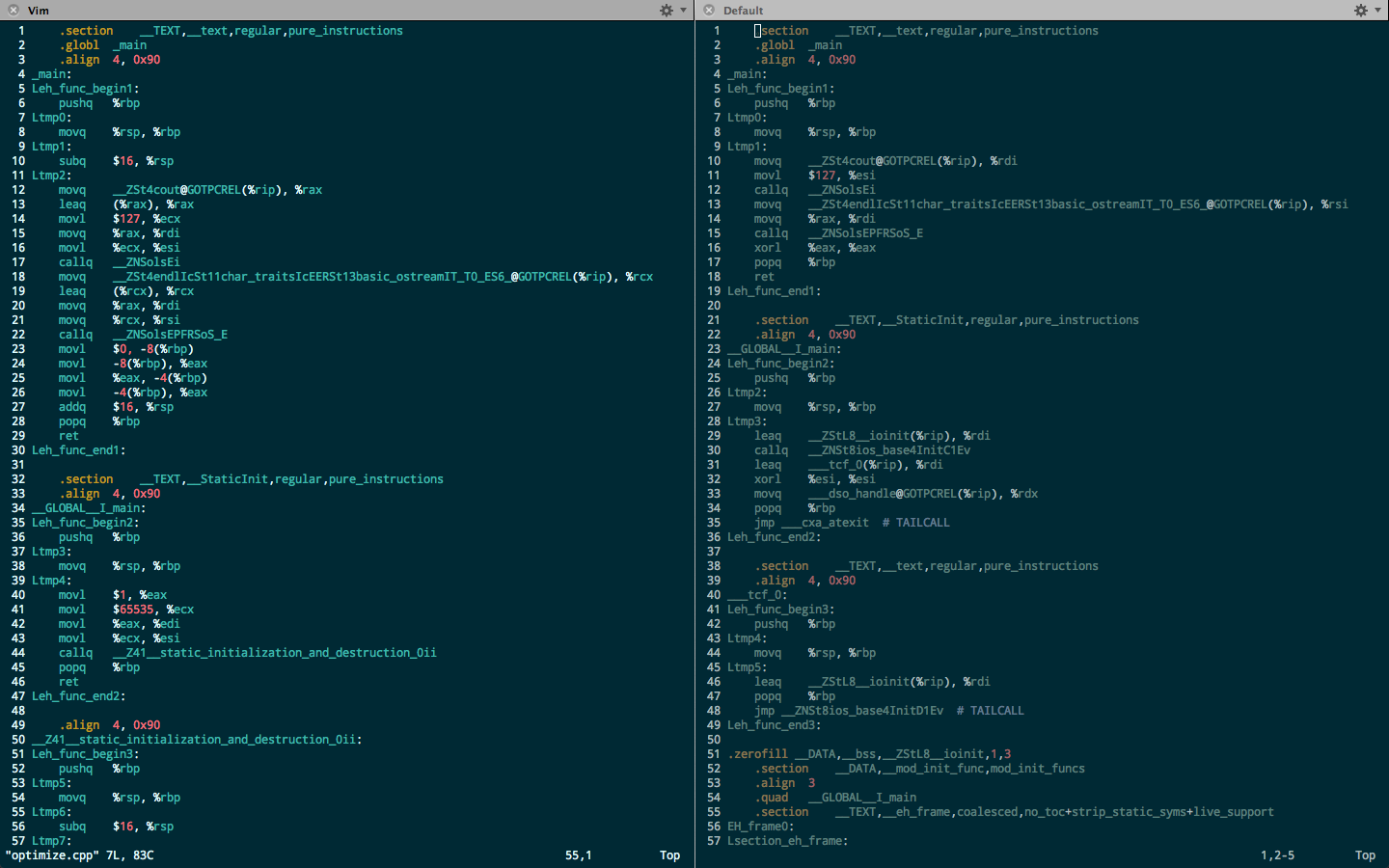
### Cout Function

The next type of optimization I explored was optimization of I/O using standard functions: namely cout. Although, I didn’t expect to really be able to understand most of the generated assembly, as I imagine I/O to be complex on the lower levels, I did gain some valuable insights throughout reading through the assembled programs. I ended up reading a good deal into the differences between the standard x86-64 architecture and that of x86-64 Mac OS X computers.

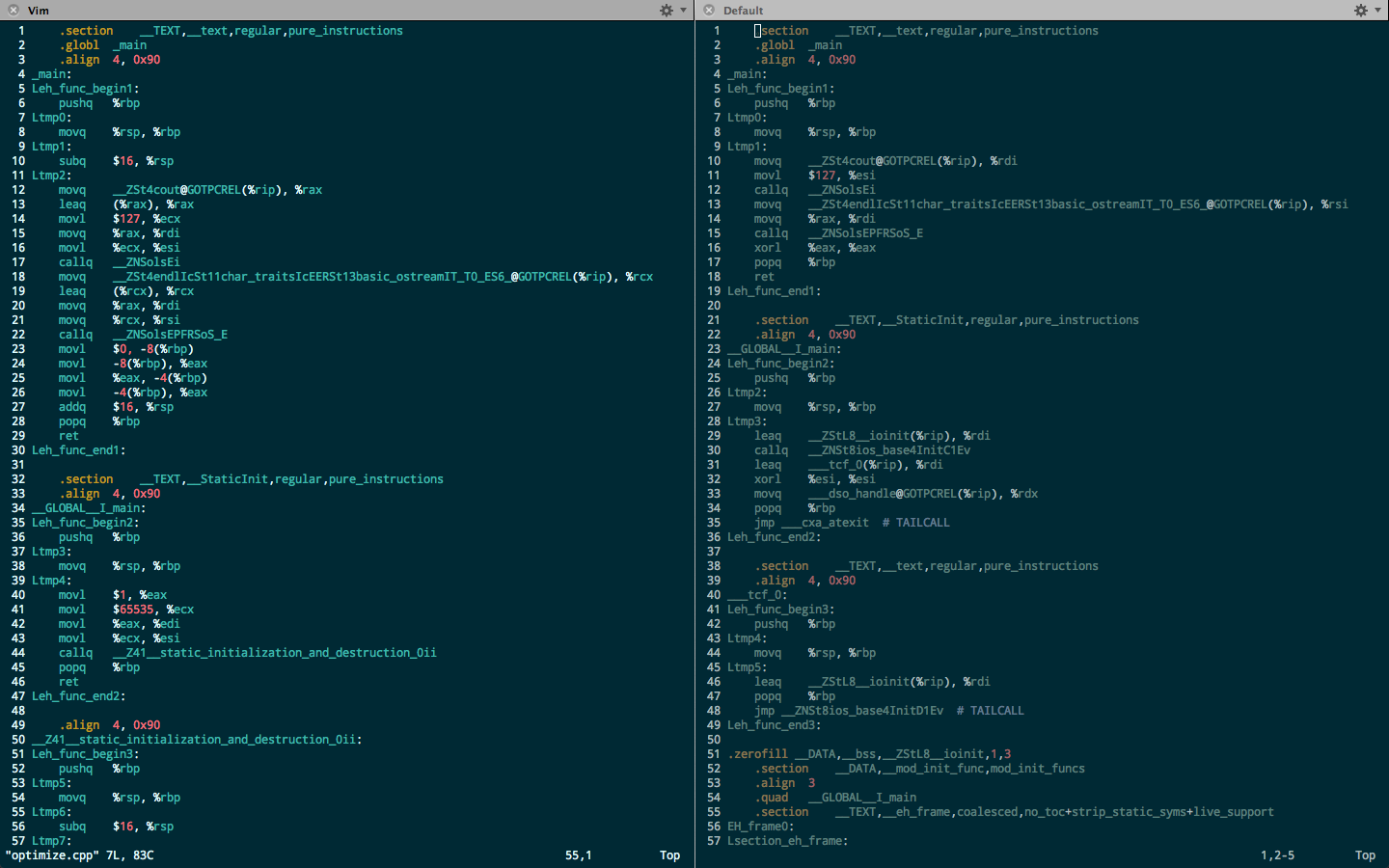
The following are screen captures of the C++ source, the default assembly, and the optimized assembly (in that order):

### Macintosh HD:Users:Adarsh:Desktop:Screen Shot 2012-04-11 at 9.17.24 PM.pngOriginal Source

Truncated assembly

****

Optimized Assembly



The first thing I noticed is that the optimized assembly code passes much less to the \_\_ZNSolsEi function that is called. The original code passes in \_\_ZSt4cout@GOTPCREL($rip) [via $RAX] to the function. This intrigued me, as it seemed like it was passing in a function to the function.

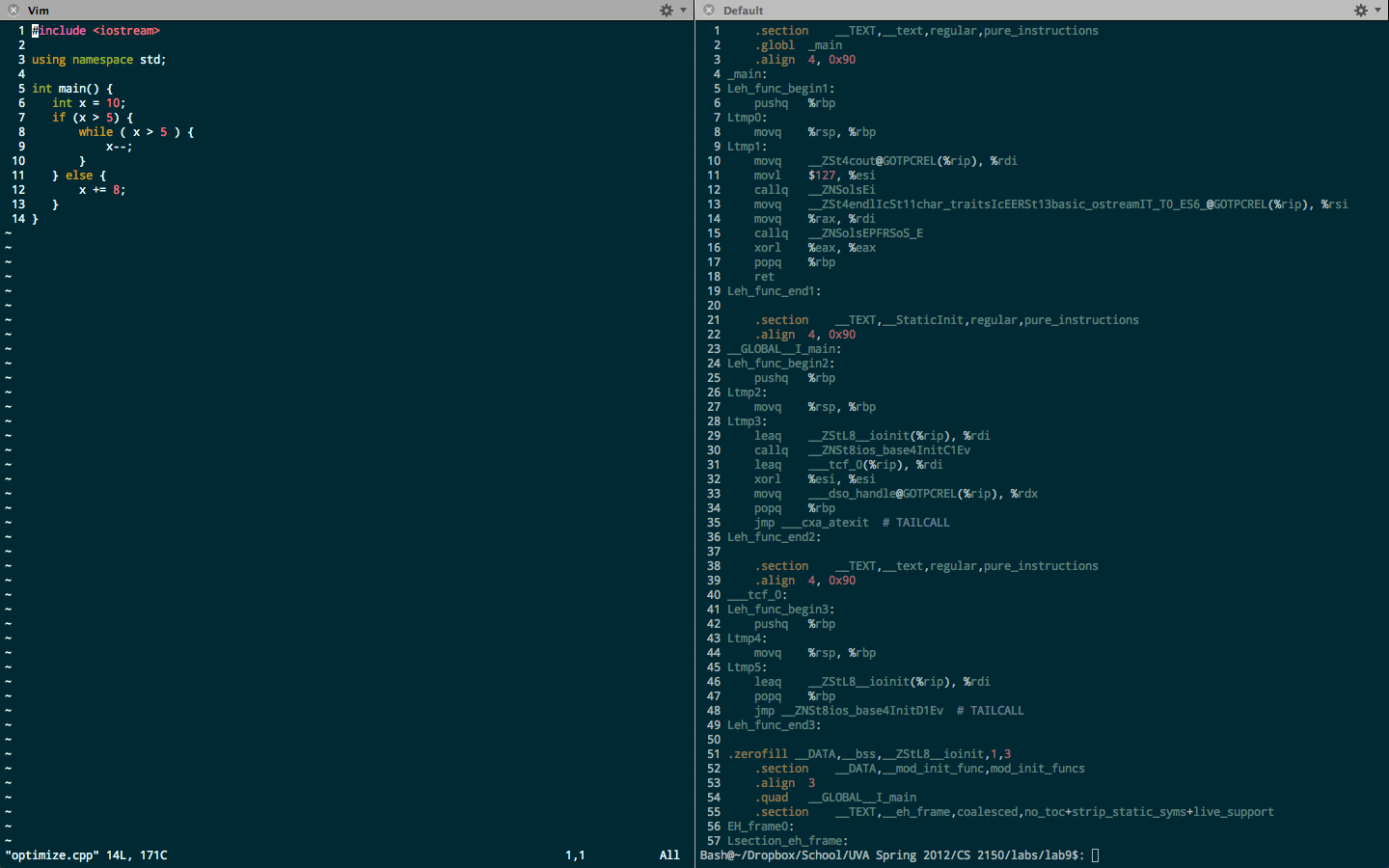
It turns out, the GOTPCREL($rip) is the Mac x86-64 environment’s way of accessing local and small data. (Source: <http://developer.apple.com/library/mac#documentation/DeveloperTools/Conceptual/MachOTopics/1-Articles/x86_64_code.html> )

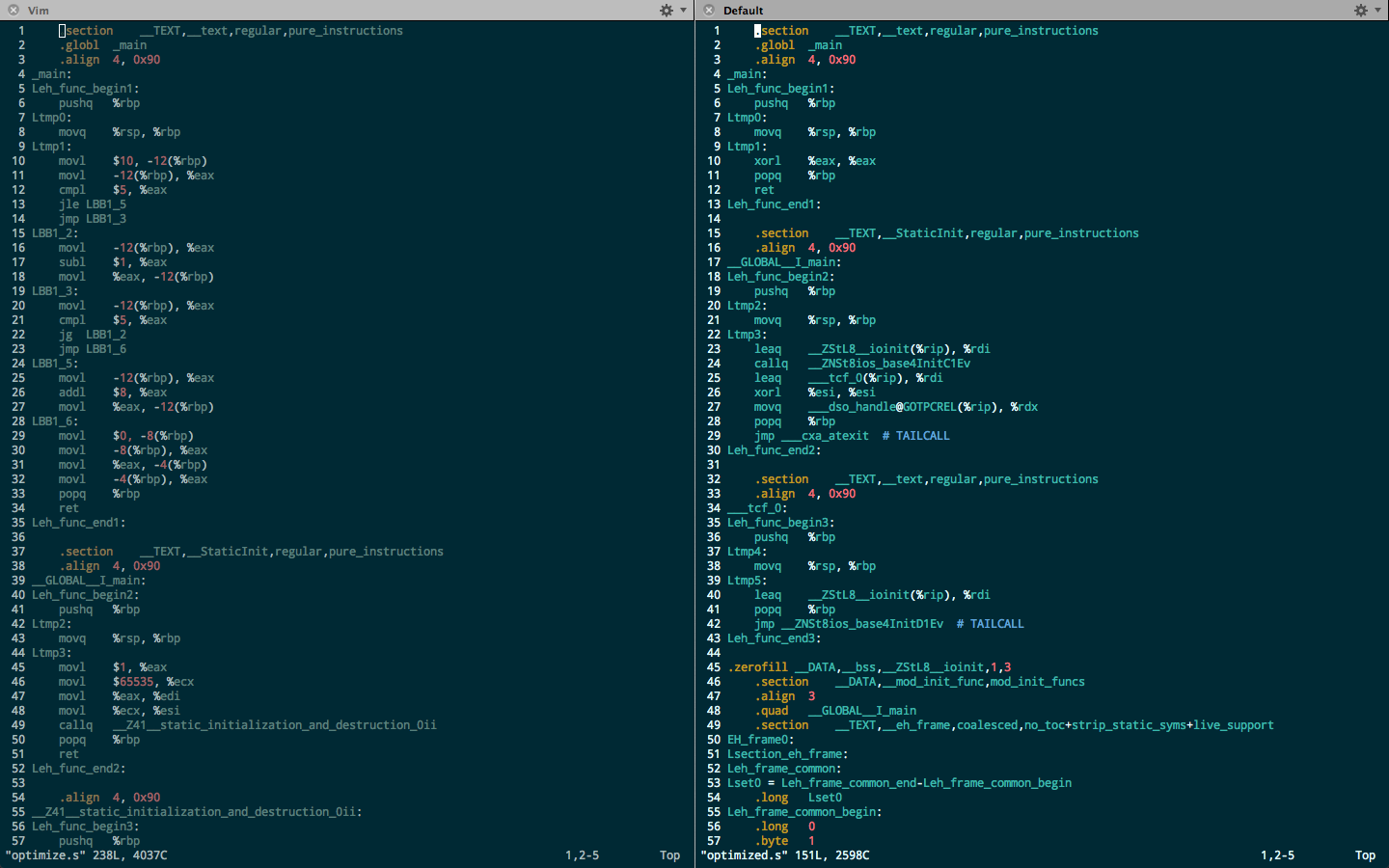
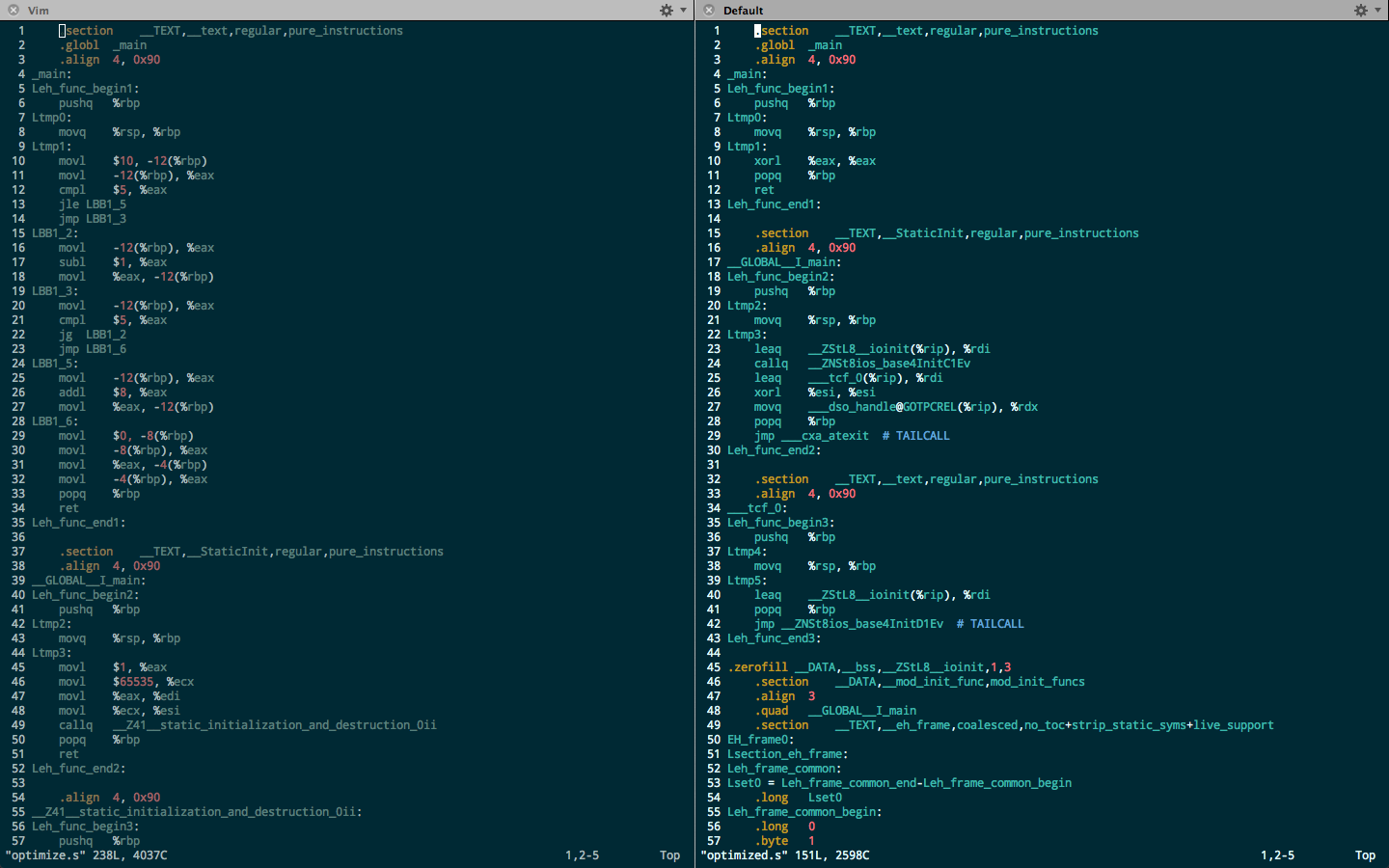
My interpretation of this is that the prefix \_\_ZSt4cout is a label for one of the functions that exists in the iostream library (the cout function, perhaps a specific one to print a number vs. printing another data type) and is represented in the program space’s “global offset table” (GOT) which uses “RIP-relative addressing” which means addressing relative to the instruction pointer.

The optimized code performs very similar things, but in fewer instructions. For example, the default code stores the ZSt4cout function pointer in $rax and then moves it from $rax to $rdi. This is probably due to convention. The optimized code on the other hand stores the information directly into $rdi. I don’t understand why modern compilers don’t always optimize little steps like these, as it doesn’t even seem to be an optimization, but rather an improvement on the inefficient assembly produced by default. I’m sure there is a reason for this that involves things beyond the scope of my understanding.

## Control Flow

The final optimization that I tried to study was that of control flow. I made a C++ program that included a very primitive demonstration of if/else and a while loop.





default optimized

The loop iteration in the non-optimized code is pretty similar to how I would code it myself. It creates a few labels for each of the control flow “states” and then uses the basic cmp, jle, j, jge instructions that I would use, were I to code this myself.

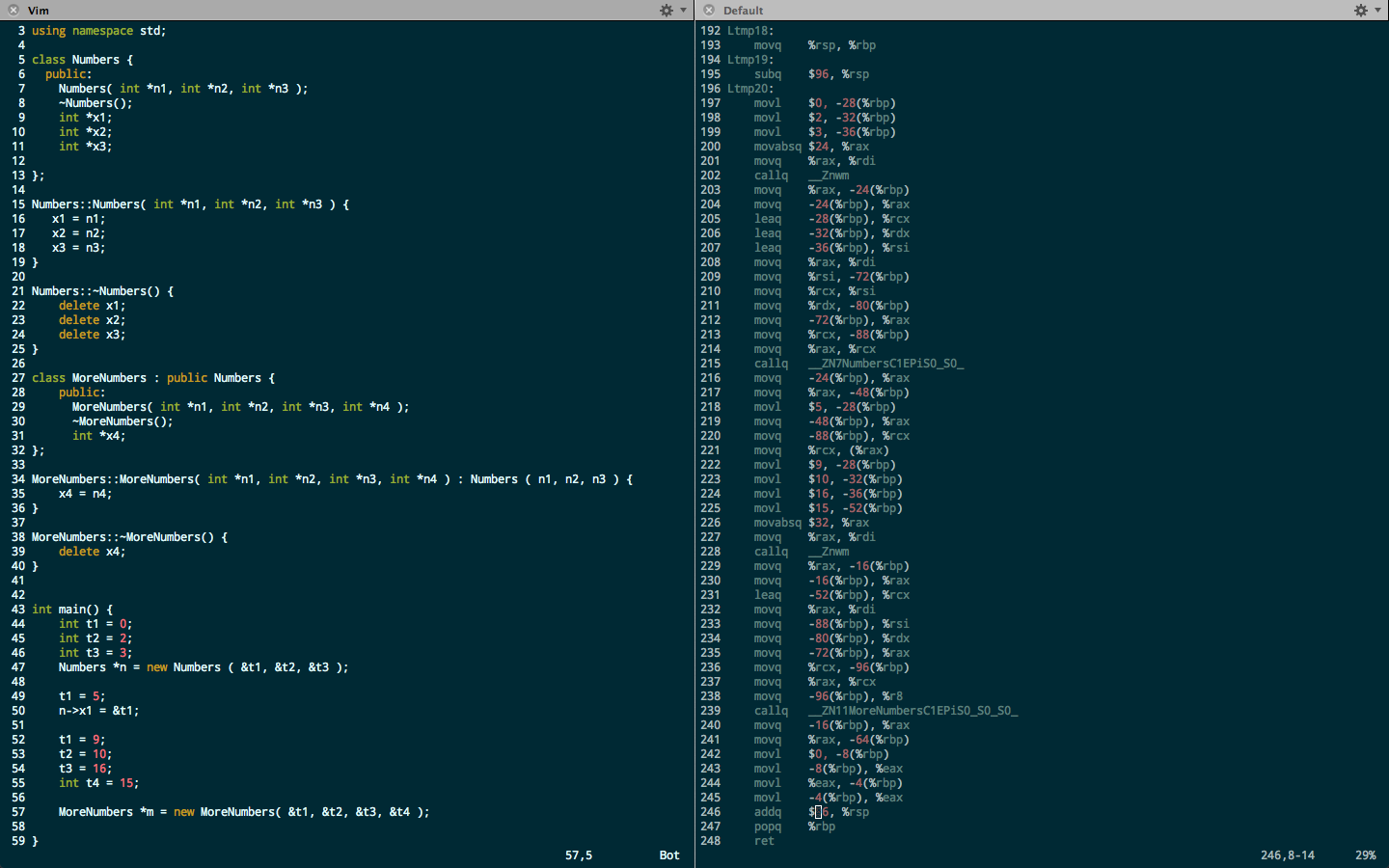
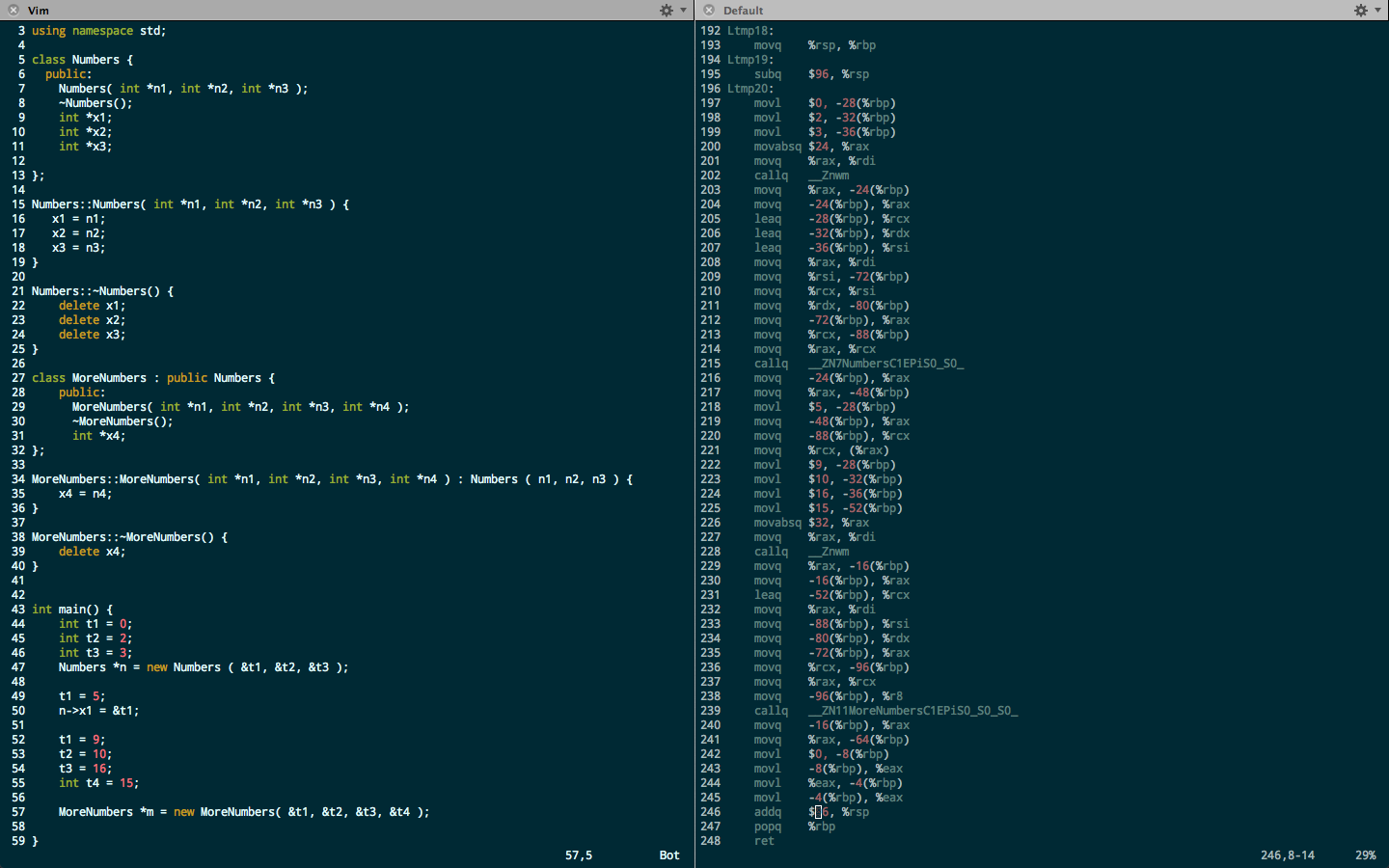
The optimized code on the other hand, is very different. It is much harder to draw similarities between the C++ code and the optimized assembly code. I tried to search for some common instructions like cmp or any jump, but to no avail, so I decided to dig around in the code and see what I could find through the Internet.

The first thing that jumped out to me was the line

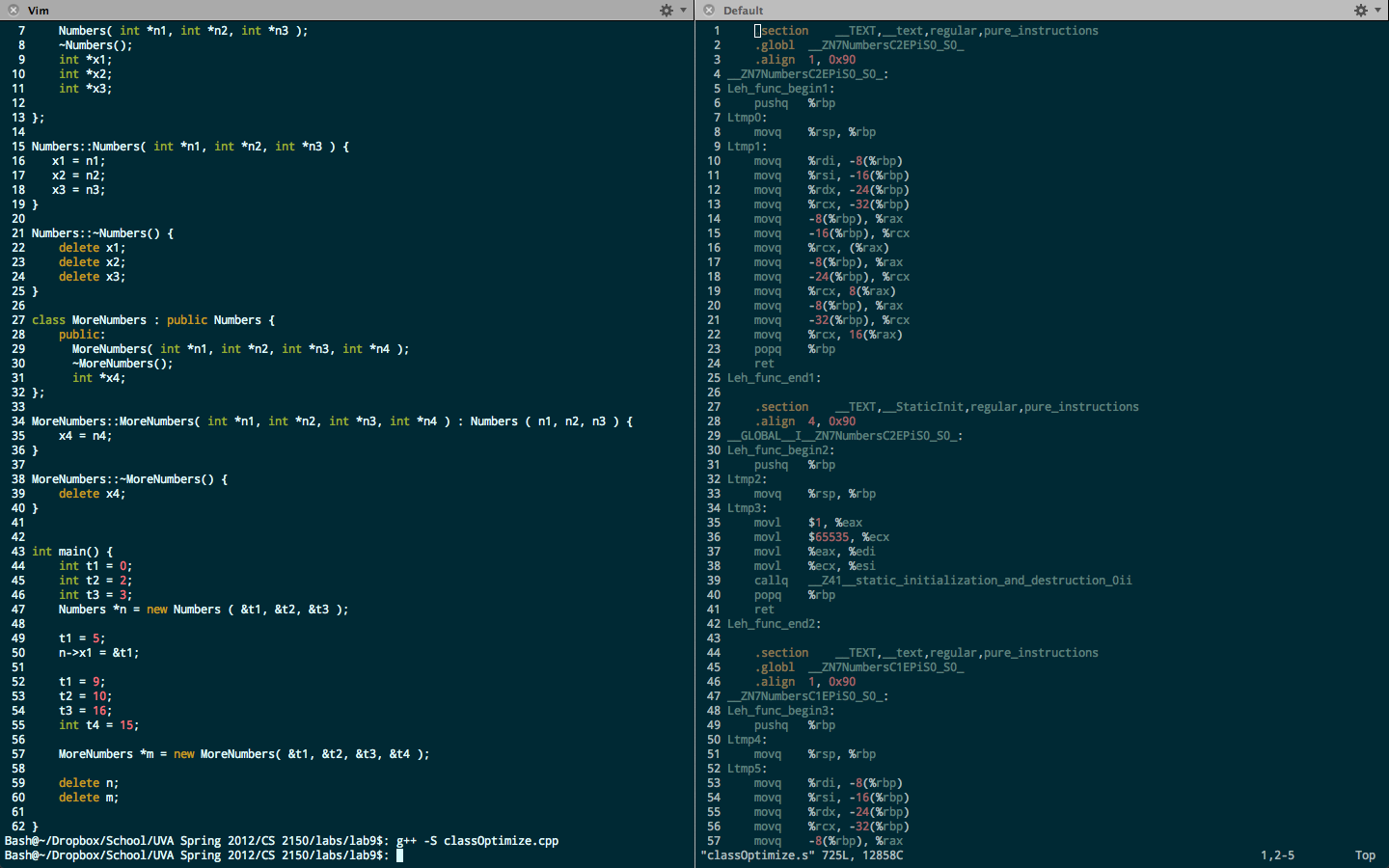
jmp \_\_cxa\_atexit # TAILCALL

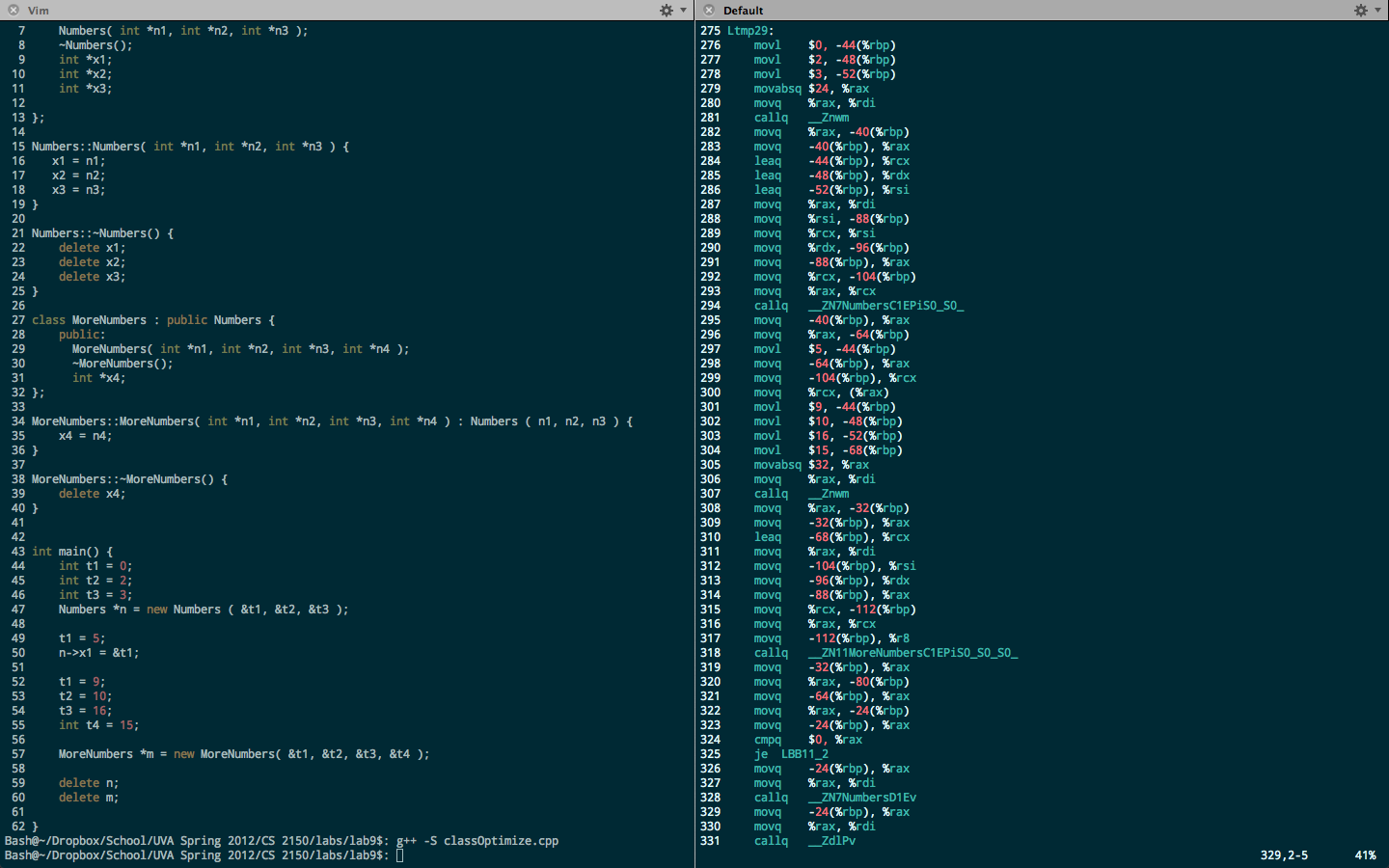
This seemed odd, because I hadn’t yet seen g++ generate comments in assembled code. It turned out that this was actually present in the default assembly code, as well as every other assembled program I had generated with g++. It turned out to be g++’s way of doing constructors, so this must have been some sort of initialization of the main() object? This leads me to believe that this program simply does nothing, as the first (main) function simply xor’s EAX with itself (zero’s it) and then returns. The compiler rendered my function equivalent to “return 0;” and come to think of it, it technically was, as none of the variables or register values have any importance outside of the scope of the function itself.

## 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 then I created a subclass called “MoreNumbers” which holds an additional integer pointer. The code for the two classes are shown to the right and below:

The main function of the program simply instantiates various integers, and creates one instance of Numbers and one of 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.

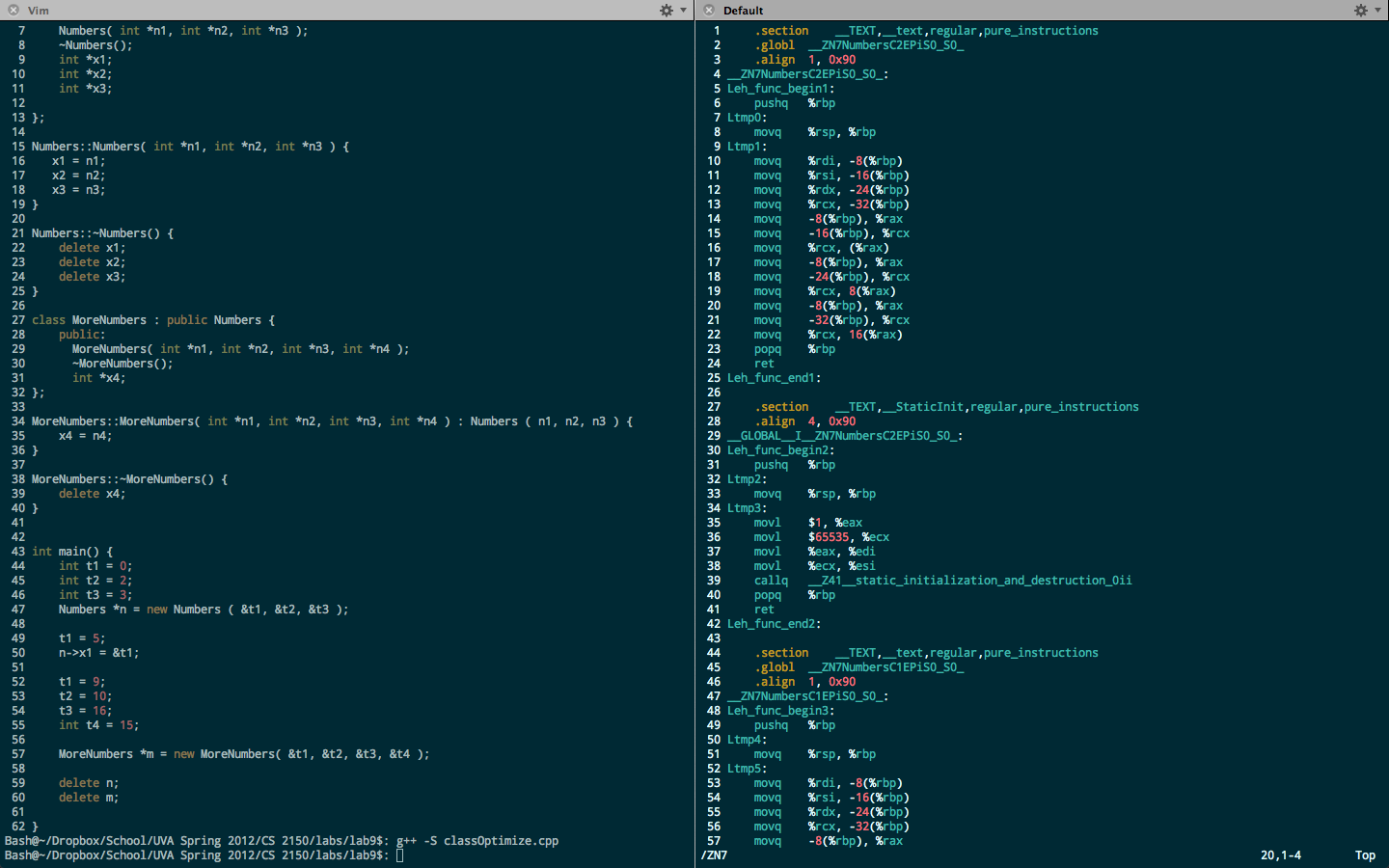
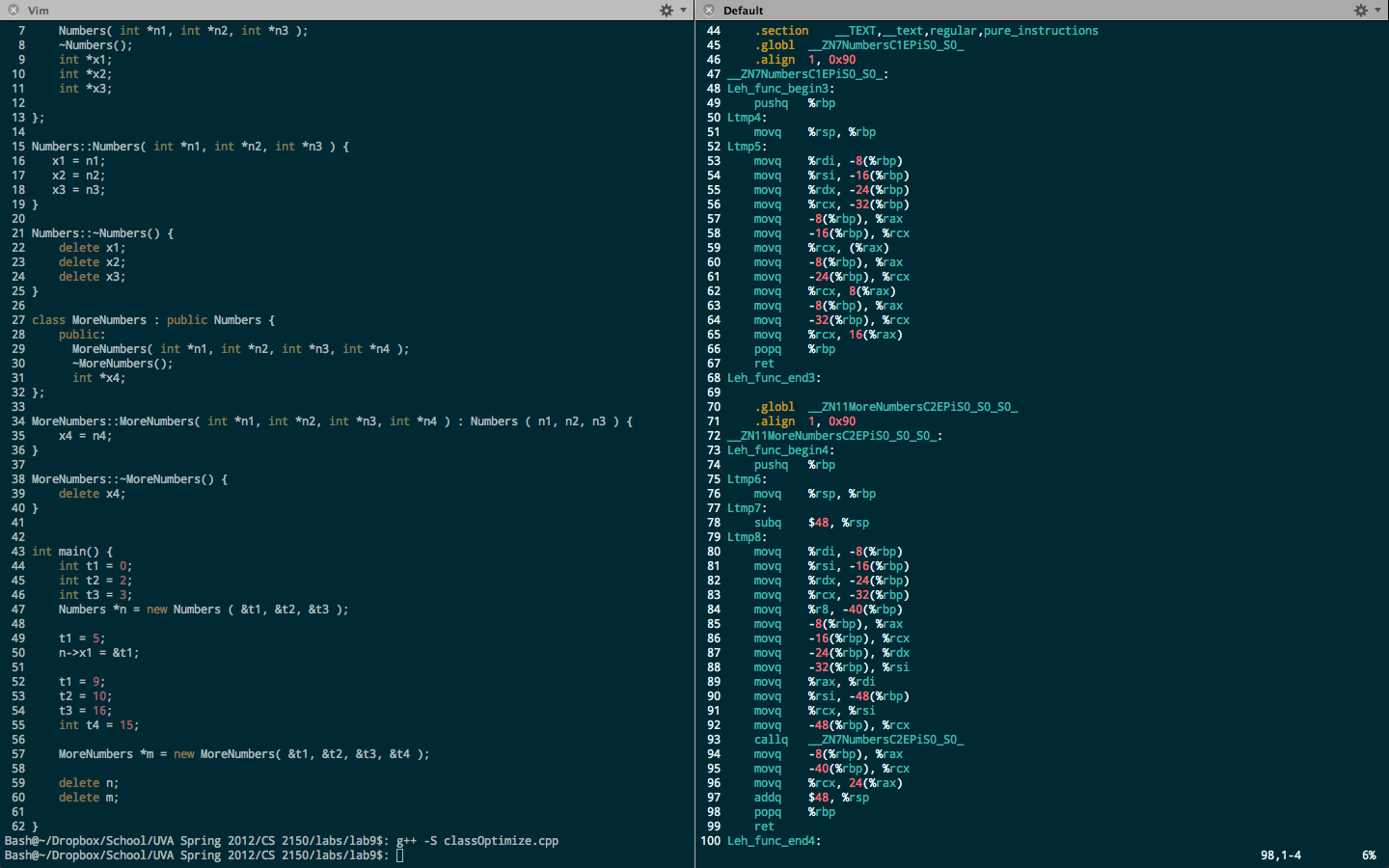
To the left is a screen capture of the main function, showing its exact functionality, followed by the associated assembly code.

The first few lines in the snippet are showing the three arguments for the

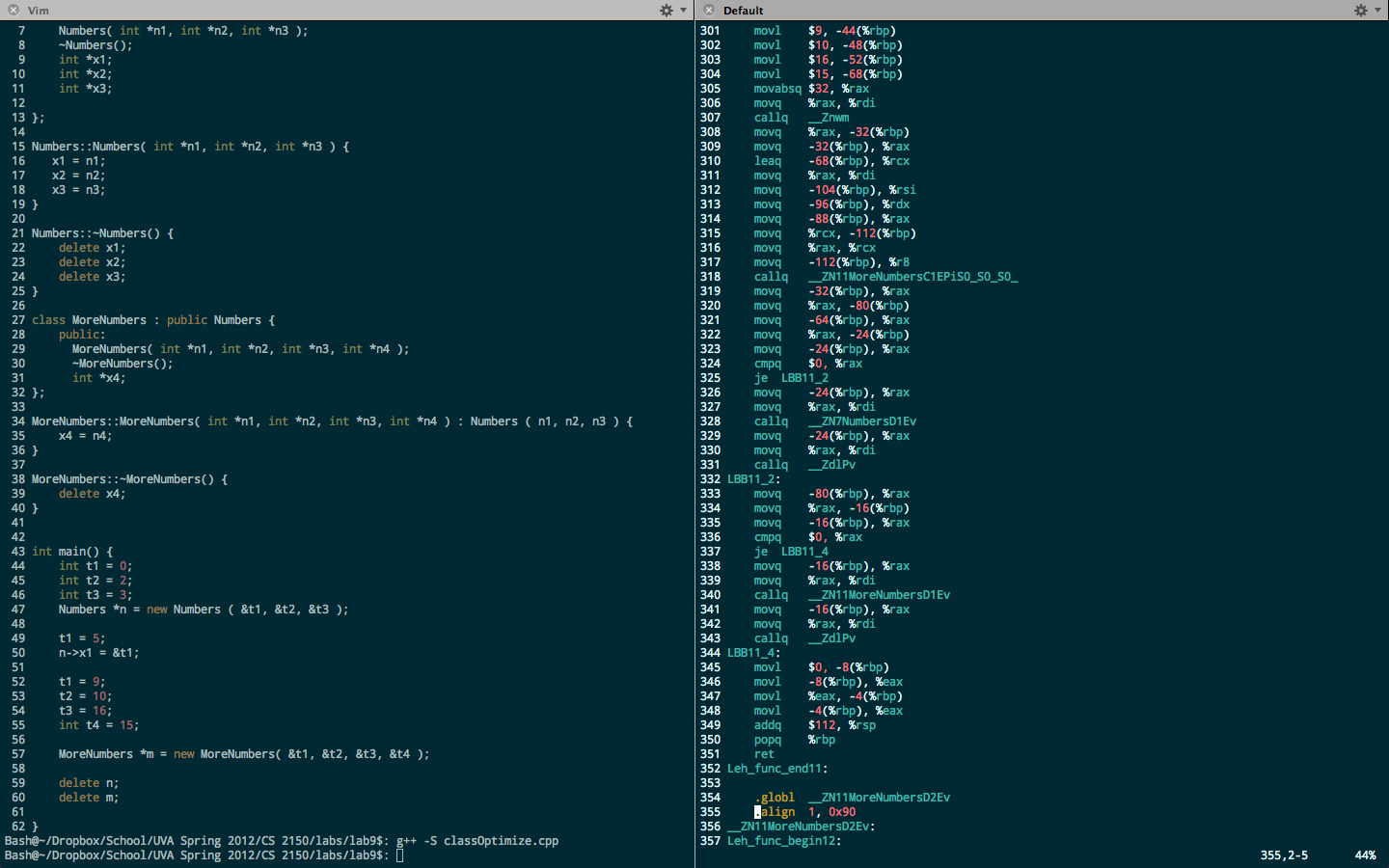
Numbers constructor (0, 2, 3) as written in the C++, but next there is an instruction that I have never seen before, movabsq $24, %rax.

From an online search, I found that this is meant to move an absolute quadword, and I think 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, this is probably to do the inherited functionality from the base-class’s constructor. On second thought, it appears to be a function that is called before the constructor, which I believe to be the function on line 294.

Next, 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 line 297 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.

Line 318 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.

Above is the body of the Numbers constructor, and to the right is the body of the MoreNumbers constructor. As is apparent on line 93, the MoreNumbers constructor calls the Numbers constructor within itself. I cannot exactly tell how the data is manipulated after the constructor is called though.



I believe that per the x86 naming conventions of g++, the constructors are always the name of the class followed by a C, and the destructors with a D. That being said, the destructors are the next functions called in the main function (right, lines 328 and 340).

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 at 331 “ZdlPv” is kind of confusing, but from purely the name it sounds like it **d**e**l**etes a **p**ointer **v**ariable, 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\*)” (source: <http://lists.gnu.org/archive/html/bug-binutils/2010-01/msg00047.html> )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.

Predictably, the function is called again after the MoreNumbers destructor is called, leading me to 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.