

CS 216

Laboratory 7:

x86 Assembly Language Basics¹

Objective:

The purpose of this lab is to familiarize you with the process of writing, assembling, and linking assembly language code. The purpose of the post-lab activity is to investigate how various C++ language features are implemented at the assembly level.

Background:

The Intel x86 assembly language is currently one of the most popular assembly languages on the planet and runs on many architectures from the x86 line through the Pentium 4.

¹ In past semesters, this was often Lab 6.

Procedure

In-lab

Come to class with a printed copy of the checkoff sheet and a functioning version of the pre-lab (factorial). You will be assigned a partner for the in-lab activity.

0. Demonstrate and explain your pre-lab code to a TA for the checkoff.
1. The general activity of this in-lab will be to write small snippets of C++ code, compile them so that you can look at the generated assembly code, then make modifications and recompile as needed in order to deduce the representation of a number of C++ constructs (listed below).

Later in this document are two lists that describe activities to explore how C++ is compiled into assembly. You and your partner must work on the two items in list 1 together. Then together you may want to tackle one of the more complex items from list 2. Keep working on more items as time permits, as you will have to address both items from list 1 and one item from list 2 in your final (individual) post-lab report. Both partners should be prepared to explain both items from list 1 to the TA.

- Before leaving, each partner should explain something from both items from list #1 to a lab TA. (A partner cannot leave until both partners have described both items to a TA.)
- Turn-in your check-off sheet *before* leaving lab.

Post-lab

You may work alone or with your in-lab partner for finishing the post-lab. Although working on the post-lab with others is encouraged, each student will turn in their own individual post-lab report, which should represent their own work.

0. Explore, investigate, and understand both items from List 1 and one item from List 2.
 - Be able to answer “how” and possibly “why” for each item.
 - Use test cases and the debugger as resources.
 - Additionally use resources other than yourself (e.g. books, Web, people).
1. Prepare an individual report that explains your findings. *Follow the guidelines in the Post-lab Report Guideline section.* Address the following:
 - How the compiler implements the construct at the machine and assembly levels.
 - What leads you to this conclusion. You must show evidence of this behavior in the form of assembly code, C++, screenshots, memory dumps, manual quotations, output, etc.
 - Where you found the information that lead to your conclusion. (i.e. your sources)
 - For this lab you will be submitting your *pre-lab code* electronically via the toolkit. Your fully functional code is **due at 11:59 pm Thursday, November 10, 2005**. *Your code will be graded for correctness AND style.* Be sure to include: your name, the date, and the name of the file in a comment at the beginning of each file you submit. Be sure that your text file is readable when printed out on a (normal) 8.5” x 11” piece of paper.

2. Your **Post-lab report** is due **Friday, November 18, 2005 by 11:59pm – electronically**, to be submitted via the toolkit. (Note that this is a Friday deadline. Also, the usual late deadlines are extended by 24 hours.) Your report should be in MS Word, pdf, or ps format. A title page that you should include (electronically) in your report will be posted on the labs web page (and is also included at the end of this handout).

Please note: We are extending the deadline for the x86 *post-lab* since this is slightly more involved than some we have had. The extra time is not to allow you to procrastinate! It is to allow you to ask questions and do a nice job. Note that some extra credit is possible for completing extra items or for doing an exceptional job on the required items.

Tips for Getting Started on the Post-lab

1. Think about how best to investigate the issues you choose. A good starting point is to write a small C++ program that illustrates one of the issues. This program should be as simple as possible.
2. Next you need to take a look at the assembly code associated with your C++ code. To examine the disassembled code you have two main options. 1. You can step through the code in the debugger using the disassembly view, or 2. You can have the assembly code output to a separate file, which you can then browse or edit.
3. To generate an assembly listing in Microsoft Visual C++, on project properties, go to C/C++, select the category output files, click on “Assembler output” then select the desired listing file type. Probably the most useful listing will include source, and assembly code. For some issues it will be of interest to see the machine code as well. Use the MSVC help to figure out what the resulting generated file will be called or how to rename the output file. (Figuring this out for yourself is just a warm-up for the sorts of resourcefulness that will be useful in this project in general.)
4. A couple of things you will notice almost immediately about these assembly files is that a) they can be surprisingly long, b) they contain a bunch of labels, directives, and instructions that at first glance appear to have little to do with your original source program. Don’t despair, with a little perseverance you will be able to make heads or tails of a good bit of this.

Note: Printing out these disassembled files is probably not your most useful option. You will most likely find that it is significantly easier to view the files in a browser of your choice: MSVC, emacs, vi, whatever works for you. In this way you can navigate through the file, searching for particular labels or C++ code. Besides, you may want to make a slight modification to your C++ code and recompile often anyway.

5. Still stuck? Some of these issues are non-trivial to figure out. Remember that you can use basically any resource whatsoever to figure these things out. There will almost certainly still remain some things in the disassembled code that you do not fully understand. Don’t let this paralyze you. Focus on devising experiments that will help you learn more about the particular issues in lists 1 and 2. By tracing through some parts of the code and by modifying your C++ code and comparing the

generated assembly code for the two different versions, you should be able to come up with some reasonably good hypotheses about what is happening. Seek out your classmates to see if anyone else is working on the same issue. Seek out books, manuals, and web pages that explain the issue.

Post-lab Report Guidelines

Each student should individually submit a nicely formatted report that explains his or her findings. At a minimum your post-lab report should address *both items in List 1, and one item from List 2*. In your report, label the items according to what list they came from (1 or 2) and their item number within that list (e.g. List 2, Item 2. Dynamic dispatch).

Remember that this is supposed to be a polished project. Code snippets should be imbedded into the document, not just printed out on a page by themselves and stapled in. Similarly, screen shots should be imbedded in the document. Highlighting portions of code or drawing arrows between things may help make your explanations clearer. I would expect the explanation of each item to be at least a couple of pages long, including embedded code snippets and screenshots.

Collaboration: You are encouraged to work with your classmates outside of class, but the report you turn in should be your own work – you may use the same sample C++ program to demonstrate a concept, but the explanation should be your own entirely (and you should credit anyone whose program you use).

Other than your own experiments, feel free to use machine manuals (Intel documents are linked off of the course web page), C++ books, your classmates or other experts, resources you may find on the Internet or elsewhere. Discussing these issues is encouraged, however, remember that your final report must be your own work and that you must credit ANY resources used.

Extra Credit: Exceptional reports or reports that address more than the required number of items may receive an appropriate amount of extra credit. The instructor will determine the amount of extra credit given. This could conceivably be sizable, but it is EXTRA and will not affect others' grades. Basically I want to encourage folks who are interested to really get into this assignment, and I will in turn try to give you some credit for your effort.

**** Be sure to address all issues listed for an item. ****

List 1: (You must do BOTH of these)

1. **Parameter passing:** Show and explain assembly code generated by the compiler for a simple function and function call that passes parameters by a variety of methods. Be sure to show what is happening both in the caller and in the callee. You do not need to describe parts of the C calling convention we described in class (e.g. saving registers, saving the base pointer, how the call instruction works). The focus here is on examining in detail what happens when parameters are passed. You should explain how ints, chars, pointers, floats, and objects that *contain more than one data member* such as user-defined classes are passed by value and by reference. In addition, show how arrays (you may pick any type) are passed in C++. Be sure to

show ***both*** how these values are passed into a function ***and*** how the callee accesses the parameters inside of the function. *To see addresses of variables you can right click in the disassembly window and unselect “Show Symbol Names” which will also convert function names to raw addresses as opposed to their symbolic names. This question asks about exactly where the data values are placed, so you will need to determine at least a register-relative address, just saying the parameter is accessed as [var] is not enough. Be sure to ask if you do not understand.*

2. **Objects:** Explain how data layout, data member access, and method invocation works in C++ objects. Describe where data is laid out for a sample C++ class. You should include at least five data members in your class. Be sure to include data members of different types (ints, chars, ***and*** other user-defined classes) and different access levels (public and private) in your class. To demonstrate data layout in objects be sure to show screenshots of the memory window while your program is running. Next, demonstrate how data members are accessed both from inside a member function and from outside. Finally, show how public member functions are accessed for your sample class. How is the “this” pointer implemented? Where is it stored? When is it accessed? How is it passed to member functions? Please see the note above about the need to find actual addresses: *to see addresses of variables you can right click in the disassembly window and unselect “Show Symbol Names” which will also convert function names to raw addresses as opposed to their symbolic names.*

List 2: (Pick one of these)

1. **Inheritance: (Data layout and construction and destruction)** (Only do this if you know about C++ and inheritance.) (a) Create an instance of an object that inherits data members from another class and also includes data members of its own. Show in memory where data members are laid out in that object. (b) Then explain how construction and destruction happens in this class hierarchy. Explain what happens when a user-defined object is instantiated and what happens when it goes out of scope. What if anything is “destroyed” by the destructor? Show this process happening in the assembly code using a simple class hierarchy. Point out in the assembly code exactly where the destructors and constructors are getting called.
2. **Dynamic dispatch:** (Only do this if you know how C++ uses virtual functions with inheritance.) Describe how dynamic dispatch is implemented. Show this using a simple class hierarchy that includes virtual functions. Use more than one virtual function per class.
3. **Other compilers:** Compare x86 code generated by Microsoft Visual Studio to x86 code generated by g++ or another compiler, describe the similarities/differences you find. One choice here is to use Olsson 001, start up the Cygwin UNIX-like shell, and then run g++ from that command-line interface. (Note that compiling with g++ on one of ITC’s computers is not only a different compiler, but also generates code for a different *architecture!* See next item.) This item requires you to use two different compilers on the *same* architecture. Describe at least four (non-trivial) differences you see between the code generated by the two compilers.

4. **Other architectures:** Compare code generated by Visual Studio for the x86 architecture to code generated by g++ or another compiler for a different architecture, describe the differences you find. (Go back and look at the info on the web for Lab 2, where we compiled and executed your postfix calculator application on one of ITC's machines.) Identify as many similarities to x86 instructions as you can. Compare things such as: labels, op codes, number of operands, addressing modes. Describe at least four (non-trivial) differences you see between the two listings.
5. **Optimized code:** Compare code generated using debug mode (the default) to optimized code. To do this you will need to change your project configuration from debug to release by selecting "Win32 Release" from the "Set Active Configuration" drop-down list on the Build menu. Then select: project-> settings->C++ and select an optimization level. Can you make any guesses as to why the optimized code looks as it does? What is being optimized? Be sure to show your original sample code as well as the optimized version. Try loops, function calls, to see what "optimizing" does. Be aware that if instructions are "not necessary" to the final output of the program then they may be optimized away completely! (this does not lead to very interesting comparisons) Describe at least four (non-trivial) differences you see between code generated in debug mode vs. optimized code.
6. **Templates:** What does the code look like for the instantiation of a simple templated class you wrote? What if you instantiate the class for different data types, what code is generated then? Is it the same or different? If the same, why? If different, why? Compare code for a user-defined templated class or function to a templated class from the STL (e.g. classes such as vectors or functions such as sort).
7. **Dynamic Memory and Linked Lists:** Using C++ code that uses a dynamic data structure such as a linked list, explore the location and values of list-node objects and associated pointers. Can you say something about where a call to the operator *new* creates nodes? What effect does operator *delete* have on the node-values stored in memory? It's not enough for you to study this for one or two calls to *new* and *delete*; make sure your code inserts and removes a number of list-nodes so that you can see what is happening. Mix in these calls with calls to *new* that dynamically allocate other kinds of objects, e.g. strings or doubles. Can you make some kind of prediction about where each new node or new string or new double will occur? Does it appear that *new* re-uses previously freed memory? Also, add code to carry out two common pointer errors: a memory leak, and a dangling pointer (i.e. one that points to a node that has been deleted). Using your examination of memory, explain exactly what bad thing the computer tries to do in this situation. **Note:** You may re-use some previously written code that uses linked-lists for this, but you might find it easier to write a very minimal linked-list implementation (perhaps singly-linked lists that use structs rather than objects). Either way is fine as long as you can explore what's happening and explain it.

CS 216: Program and Data Representation Checkoff

Date:	Circle: Your 2 Tue 7:30 – 9:15 PM Section: 3 Tue 12:15 – 2 PM
My <u>Email</u> Address: (UVa ID please, no aliases)	My Name:
Partner's <u>Email</u>:	My Partner's Name:

<u>Lab 7: x86 Assembly Language Basics</u>	
Demonstrated and explained pre-lab functionality (factorial)	
Explained List 1, Item 1	
Explained List 1, Item 2	

Pledge:
