Maddie Stigler
mgs4ff
11/4/14
inlab8.pdf

For the purpose of the inlab portion of the assignment, I examined the Parameter Passing questions associated with the cpp code for passing different data types different ways.

In order to examine how ints, chars, pointers, floats and objects that contain more than one data member are passed into subroutines, I created different functions that took in these data types as parameters and performed small transitions to them before returning back to the main method.

Passing ints, chars, and floats into the callee all functioned very similarly. The caller saved the stack pointer before pushing the parameters on the stack and calling the callee. However, there were some differences within the callee subroutine that differed due to whether the parameter was passed by value or by reference. There was a difference between passing by value and passing by reference when passing char data types. When passing a char by value, more of the x86 code pushes parameters onto the stack using the mov and push calls. When passing by reference, addresses of the parameters are stored on the stack and The code for passing by reference tends to be longer and spends time looking up the address for the referenced variable. In fact, this is also the case when passing ints and floats as well. Another difference just between passing the different data types was that the subroutine spent more time on the float parameter using fladd, fld, and fst. These instructions did not appear in the int and char methods because it didn't need to convert anything for them.

I also examined the difference between passing a pointer and passing a reference. When comparing passing a pointer into the subroutine with passing a reference into the subroutine, both consume the same amount of code and use the same code to achieve the same result. This suggests that neither is more efficient than the other. Both rely on the eax and edx registers to store DWORD PTR values and move them before exiting and returning the result to the main method. This suggests that determining which way to pass a value depends on the context of the code, not that one is more or less efficient than the other.

To demonstrate how objects that contain more than one data member are passed into subroutines, I used the c++ list. When passing by value, the x86 code used a DWORD PTR to store the values of the list and move them into the eax register and manipulate them from there. When passing by reference, the x86 code delivers the same result. They are both stored as DWORD PTR in the eax register and manipulated the same way.

In order to replicate passing an array in C++ to x86, I created an array of 10 elements and passed it into a function that simply assigned 5 to array at 0. The x86 code representing the callee accessed the array parameters similarly to how it would access a list or multiple parameters. It adjusted the ebp register to the start of the array and began accessing parameters from there. When calling the callee, the caller simultaneously pushed the array values onto the stack, thus making them accessible to the callee.

As for the interaction between the caller and callee within this program, the caller invokes the callee by using lea and mov before calling it. This saves the parameters and allows the callee access to them. After the callee returns, the caller saves the variables again and alternates between using lea with the eax register and using mov with the DWORD PTR and esp to increment the stack pointer. This remains pretty consistent throughout all data types. However, with the list parameter, the caller simultaneously calls the callee while performing a push_back, storing the list's parameters on the list.

Ultimately, while the x86 code is certainly not as easily understood as the c++ code, when comparing the two, it is easy to tell that one stems from the other. There were a couple of instructions I didn't understand, but by looking at my cpp

program and comparing to the assembly, I was able to piece together what the instructions meant. Most of the caller and callee conventions are similar. While there are slight differences between certain data types, the caller still continues to push the stack point and mov values onto the stack before invoking the callee and the callee continues to pop its' stack values before returning to the main method.