

Adarsh Solanki as5nr
3/30/12
CS 2150: Lab 8 PostLab

Parameter Passing

I first created a C++ function which called a function with the following signature: "int add (int x, int y);" with the goal of examining the passing of the two integer parameters to the subroutine.

The first thing I noticed was the nomenclature of the assembler subroutine corresponding to my add() function. (_Z3addii) I assume the Z3 is some sort of hashed value or perhaps a way to identify the return type. The name 'add' is preserved, and I believe the appended 'ii' represent the two integer parameters. Also, all of the instructions have an 'l' appended to them, functioning on long 32-bit data values.

I am sort of confused about the couple of seemingly redundant instructions that seem to be inverses of each other. For example, there are many instructional pairs of the form:

```
movl    %eax, -12(%rbp)
movl    -12(%rbp), %eax
```

which seemingly does nothing. I tried to Google this to find any discussion on what appears to me to be redundancy, but to no avail.

Next I altered my program to use a float, keeping all of the code the same, simply to see the effects of passing a float primitive value into a subroutine. My results follow:

```
int add( float f )
{
    return f;
}

int main()
{
    float f = 2.0f;
    add (f);
}
~
```

The above C++ code

Yields the following nasm :

```
1. Vim
section __TEXT,__text,regular,pure_instructions
.globl __Z3addf
.align 4,0x90
__Z3addf:
Leh_func_begin1:
    pushq %rbp
    Ltmp0:
        movq %rsp, %rbp
    Ltmp1:
        movss %xmm0, -4(%rbp)
        movss -4(%rbp), %xmm0
        cvttss2si %xmm0, %eax
        movl %eax, -12(%rbp)
        movl -12(%rbp), %eax
        movl %eax, -8(%rbp)
        movl -8(%rbp), %eax
        popq %rbp
        ret
Leh_func_end1:

.globl __main
.align 4,0x90
__main:
Leh_func_begin2:
    pushq %rbp
    Ltmp2:
        movq %rsp, %rbp
    Ltmp3:
        subq $16, %rsp
    Ltmp4:
        movabsq $2, %rax
        cvtsi2ssq %rax, %xmm0
        movss %xmm0, -12(%rbp)
        movss -12(%rbp), %xmm0
        callq __Z3addf
```

To handle a floating point value, the assembler relies on more complex instructions such as “cvtts2si” to translate floating point values into more easily computable signed values. (Source: <http://courses.engr.illinois.edu/ece390/books/labmanual/inst-ref-simd.html>)

Next, I slightly altered the signature of the function to be “int add (int &x, int y);”, using a pass-by-reference for one of the integer parameters. In this case, rather than using the ‘movl’ command to use the parameter values in the subroutine, the assembler instead used the LEAQ instruction that Loads the Effective Address rather than the value. [leaq -12(%rbp), %rax] This makes sense, as the actual bits of data representing the integer should not be passed to the subroutine as in pass-by-value but rather the bits representing the location in memory of the parameter, hence the pass-by-reference. Also, the name of the subroutine was changed from Z3addii to Z3addRii, with the R standing for reference. I predict that if the second parameter were instead the reference, then the name would probably be Z3addiRi.

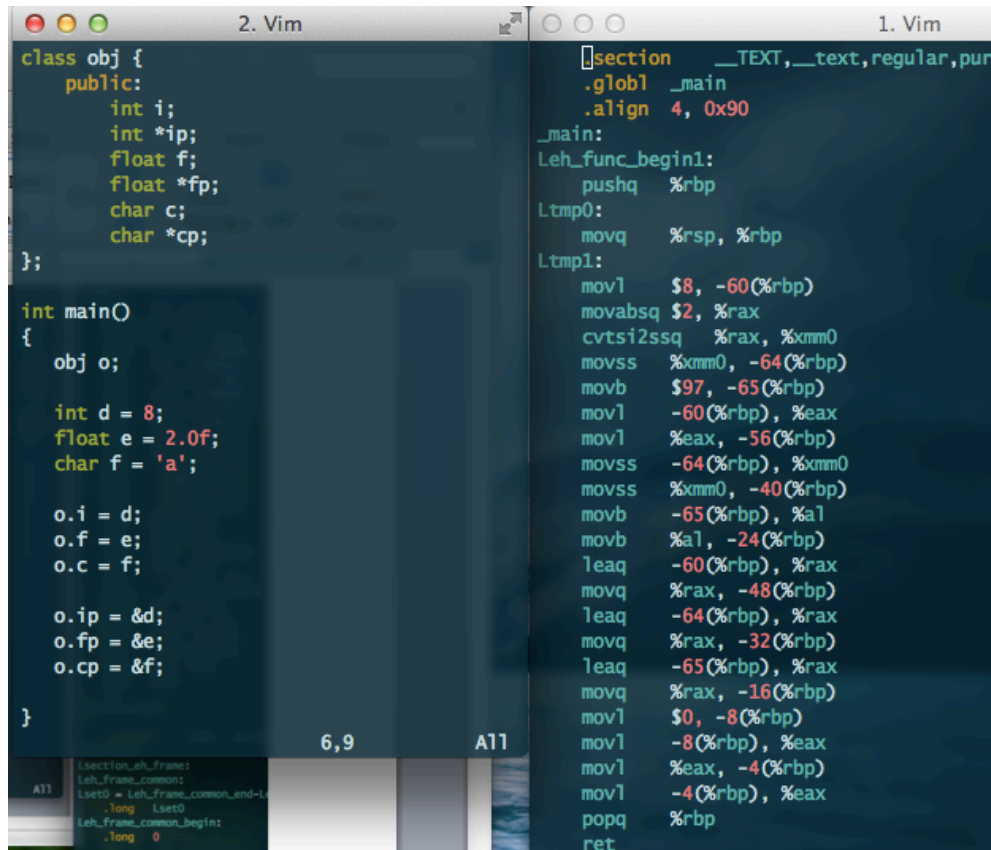
When the first parameter was instead changed to a constant reference, the assembler made very few changes. According to the unix diff command, the only differences between the compiled code with a constant pass-by-reference and a normal pass-by-reference is in the nomenclature. The subroutine’s name is changed from Z3addRii to Z3addRKii with the K standing for ‘constant’ probably.

When a character is passed, the mov function is movb instead, moving the byte-sized character rather than the ‘long’ integer.

This lab has enlightened me to the intense ‘plumbing’ going down underneath the shiny façade of high-level programming languages, as it is impossible to skip the step of assembly to machine code. This strange land of extremely truncated function names and obscure acronyms is behind all of the digital computation that exists, and the importance of a good/efficient assembler is suddenly much more clear to me.

Object Passing

Object passing was much more difficult for me to decipher. Once the assembly code was created, I had to take it slowly, line-by-line to attempt to make a correspondence between the C++ code I wrote and the output. See images:



The image shows two side-by-side Vim editor windows. The left window, titled '2. Vim', contains C++ code for a class 'obj' and its 'main' function. The right window, titled '1. Vim', contains the corresponding assembly code. The assembly code starts with a section declaration for text, followed by a global symbol for '_main'. It then shows the setup of the stack frame with 'pushq %rbp' and 'movq %rsp, %rbp'. The assembly code then initializes local variables 'd', 'e', and 'f' using 'movl', 'movss', and 'movb' instructions respectively. It then initializes the object 'o' by setting its members 'i', 'f', 'c', 'ip', 'fp', and 'cp' to the addresses of 'd', 'e', and 'f' using 'movl' and 'leaq' instructions. The assembly code ends with 'popq %rbp' and 'ret'.

```
class obj {
public:
    int i;
    int *ip;
    float f;
    float *fp;
    char c;
    char *cp;
};

int main()
{
    obj o;

    int d = 8;
    float e = 2.0f;
    char f = 'a';

    o.i = d;
    o.f = e;
    o.c = f;

    o.ip = &d;
    o.fp = &e;
    o.cp = &f;
}

section __TEXT,__text,regular,pure_instructions
.globl _main
.align 4, 0x90
_main:
Leh_func_begin1:
    pushq    %rbp
Ltmp0:
    movq     %rsp, %rbp
Ltmp1:
    movl     $8, -60(%rbp)
    movabsq  $2, %rax
    cvtsi2ssq %rax, %xmm0
    movss    %xmm0, -64(%rbp)
    movb     $97, -65(%rbp)
    movl     -60(%rbp), %eax
    movl     %eax, -56(%rbp)
    movss    -64(%rbp), %xmm0
    movss    %xmm0, -40(%rbp)
    movb     -65(%rbp), %al
    movb     %al, -24(%rbp)
    leaq     -60(%rbp), %rax
    movq     %rax, -48(%rbp)
    leaq     -64(%rbp), %rax
    movq     %rax, -32(%rbp)
    leaq     -65(%rbp), %rax
    movq     %rax, -16(%rbp)
    movl     $0, -8(%rbp)
    movl     -8(%rbp), %eax
    movl     %eax, -4(%rbp)
    movl     -4(%rbp), %eax
    popq     %rbp
    ret
```

As the main method begins, the assembler creates each of the values, using `movl` for the integer, `cvtsi2ssq` and `movss` for the float, and `movb` for the byte per usual.

Object member values seem to be allocated beneath the base pointer, as can be seen in the following as the integer, float, and char are all set to the public data members:

```

movl    -60(%rbp), %eax
movl    %eax, -56(%rbp)
movss   -64(%rbp), %xmm0
movss   %xmm0, -40(%rbp)
movb    -65(%rbp), %al
movb    %al, -24(%rbp)

```

The integer takes 4 bytes (-60 to -56), whereas the float seems to take 8. The character takes the 1 byte that it should take given the ASCII max values, all placed adjacently underneath the base pointer.

This continues as the pointers are added to the 'object space' (which is what I am going to call the space below the base pointer being used in a current frame of reference) until the last most 4 byte pointer is added just 4 bytes before the base pointer.

```

int g = 0.i + 3;

```

I inserted the above line to the end of the above C++ file to add an exterior variable accessing the data member integer of the object. The assembler adds a further variable underneath the rbp and before the epilogue:

```

movl    -56(%rbp), %eax
addl    $3, %eax
movl    %eax, -72(%rbp)

```

The 'this' member is implicitly passed in every call stack. This is why every variable starts at at least an off set of 8, leaving a space for the 'this' memory address. Every time a data member is accessed from a class, the this pointer is 'automatically used' in a manner of speaking (Source:

<http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8l.doc%2Flanguage%2Fref%2Fcplr035.htm>)

This makes me want to believe that the 'this' pointer is almost treated as the implicit first parameter of every call stack. I can see how this is very useful, as having a self-referential pointer allows easy access to the object and its data members/functions.

Next I created a private member function 'doubleInt()' which returns 2 x the integer value. When I appended the function call to the main method, a subroutine called `__ZN3obj9doubleIntEv` was created.

Within the subroutine, the multiplication works as expected, by saving the integer value into register EAX, and then multiplying it with "imull \$2, %eax, %eax". I'm surprised it wasn't assembled down into a bit shift operation for efficiency reasons. On the other side, within the main method when the subroutine is called, the preamble is the following:

```
movq    %rcx, %rdi  
callq   __ZN3obj9doubleIntEv
```