// Dion Niazi dn3gy	29 03 2017 postlab8.pdf

Parameter Passing

I created a function by the name of "addSquares()" to examine for this report. This simple function takes in two integer parameters and sums the squares of each parameter and returns that value as in integer. The function's layout is shown below:

```
int addSquares(int a, int b)
{
  return a*a + b*b;
}
```

Once I placed a function call in the main function and complied the program with the appropriate flags to produce assembly code, I opened the file to see what results were produced. In the assembly file, test.s, the name of the subroutine seems to be really different. The subroutine call name was "_Z10addSquaresii", but the subroutine's name was "addSquares". The first part of the subroutine's name seems to random gibberish or rather maybe some hash value. The middle of the name consists of the original name used earlier. The last part of the name, "ii", represents the parameters of the function and what their types are. The "ii" says that the function has two integer parameters passed in by value. In the main function the first parameter seems to be stored in register edi and the second parameter seems to be stored in register esi. Similar thing happens when the parameters are replaced with different data types, just that the name is slightly modified (the last part) to accommodate the different parameters passed in by value.

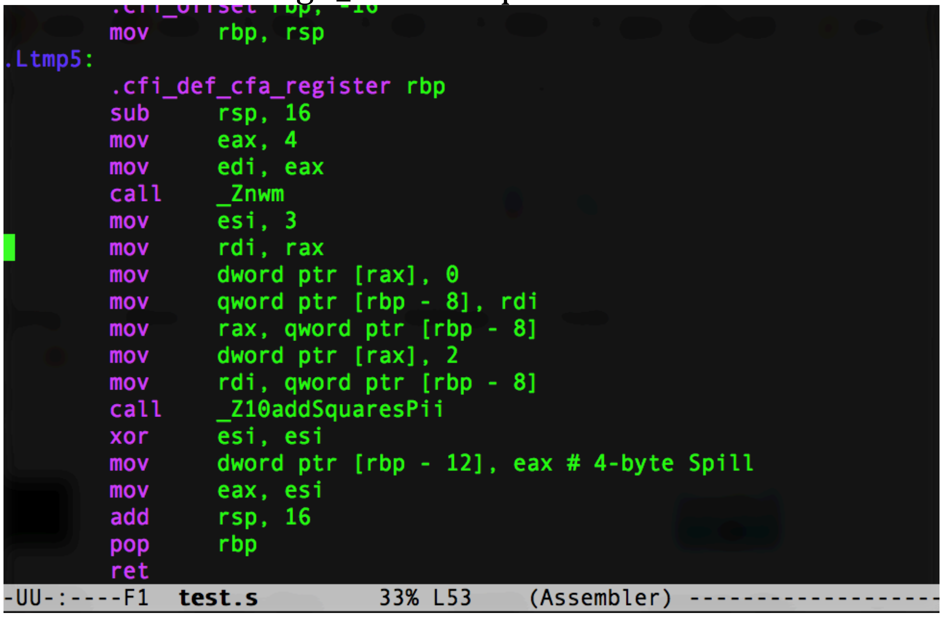
When one of the function's parameter was changed to passing by reference (specifically the first parameter), the assembly code had some slight adjustments. The function's name was relatively the same, except for the fact that the function name this time consisted of an "R" in it, making the name become "_Z10addSquaresRii". I believe the "R" in it means passing by reference and one of the main instructions has changed from mov to lea and this would be because of the moving of memory and changing it right then and there, which is what passing by reference does. When the other parameter is passed by reference, the "R" in the name moves to the parameter that passes a value by reference, but just before it. So from the above example when the first parameter is passed in by reference the name is "_Z10addSquaresRii", the "R" placed right before the first parameter type and when the second

1) http://x86.renejeschke.de/html/file_module_x86_id_205.html

2) https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf
3) http://stackoverflow.com/questions/16585562/where-is-the-this-pointer-stored-in-computer-memory

parameter is passed in by reference the name is "_Z10addSquaresiRi", the "R" placed right before the second parameter type but after the first. When int was passed by reference, the assembly of the caller was different to that of just passing in int because the memory location is being passed in and assembly needs to accommodate for this change. The callee however remained unchanged because it just needs information that the function passes in.

When the first parameter was changed to "int addSquares(int* a, int b)", changes in assembly had occurred. The caller function (int main()) had two subroutine calls instead of one, one with a really random name being "_Znwm" and the other being "_Z10addSquaresPii".



```
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp5:
        .cfi_def_cfa_register rbp
        sub     rsp, 16
        mov     eax, 4
        mov     edi, eax
        call    _Znwm
        mov     esi, 3
        mov     rdi, rax
        mov     dword ptr [rax], 0
        mov     qword ptr [rbp - 8], rdi
        mov     rax, qword ptr [rbp - 8]
        mov     dword ptr [rax], 2
        mov     rdi, qword ptr [rbp - 8]
        call    _Z10addSquaresPii
        xor     esi, esi
        mov     dword ptr [rbp - 12], eax # 4-byte Spill
        mov     eax, esi
        add     rsp, 16
        pop     rbp
        ret
-UU-:----F1  test.s          33% L53     (Assembler) ------------------
```

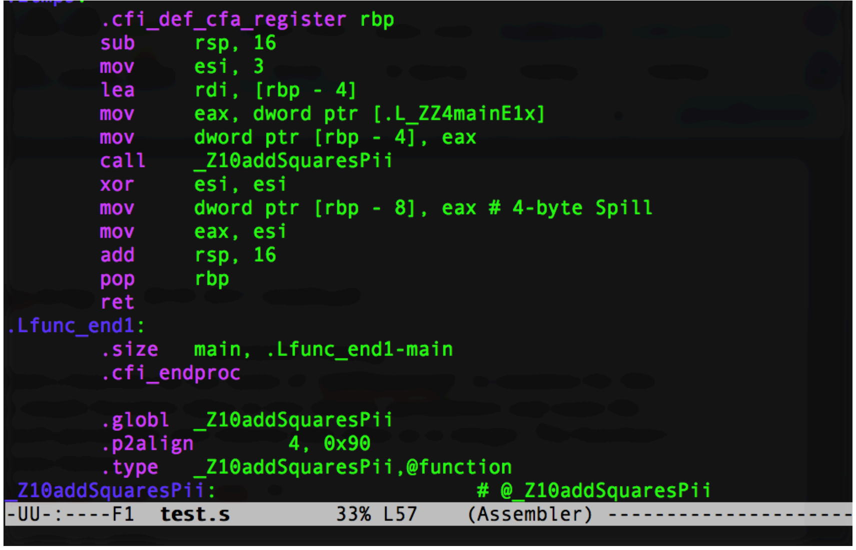
The second subroutine call was the original function that contained a "P" in it saying the function has a pointer passed in one of its parameters.

When passing in an int[] in the subroutine, the name of the function remains the same as a pointer being passed in. That is because an array is really a pointer itself. Also the caller function has only to call to just the subroutine and does lea to load up the values from the parameter's pointer, which is where the base pointer is pointing to in the stack minus the element

1) http://x86.renejeschke.de/html/file_module_x86_id_205.html

2) https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf
3) http://stackoverflow.com/questions/16585562/where-is-the-this-pointer-stored-in-computer-memory

to be accessed and for this example it was the first element so it was [rbp – 4], to the rdi register.



```
        .cfi_def_cfa_register rbp
        sub     rsp, 16
        mov     esi, 3
        lea     rdi, [rbp - 4]
        mov     eax, dword ptr [.L_ZZ4mainE1x]
        mov     dword ptr [rbp - 4], eax
        call    _Z10addSquaresPii
        xor     esi, esi
        mov     dword ptr [rbp - 8], eax # 4-byte Spill
        mov     eax, esi
        add     rsp, 16
        pop     rbp
        ret
.Lfunc_end1:
        .size   main, .Lfunc_end1-main
        .cfi_endproc

        .globl  _Z10addSquaresPii
        .p2align        4, 0x90
        .type   _Z10addSquaresPii,@function
_Z10addSquaresPii:                              # @_Z10addSquaresPii
-UU-:----F1  test.s          33% L57     (Assembler) ---------------------
```

Passing in floats is the same process as passing in an int, but has a couple more processes to occur to make it a clean representative to use throughout the function. Floats are complex, so the assembly code has to compensate to make floats simpler. Passing in char is similar to an int, but since memory is smaller its uses one byte to hold memory. Passing in an object is the same as passing an int, except the memory being held depends on what the object contains.

Objects

1. Now I how I tested objects was by using a class I made called Obj, which takes in an int, char, and a float and has a method called "int sub(int x)" which subtracts what is passed in by variable a. The assembly code seems to show that data is laid out in memory by first setting the rbp register up and then subtract 32 to allow space for memory to hold the object. The memory of [rbp – 16] was placed in rdi to be used later. Then it calls a weird function that goes to initialize the object and the initialization of the object has a weird mmnemoic called movss. It seems movss is to move scalar singular-precision floating-point numbers into some other registers called xmm.[1] Up to eight floating point arguments can be passed in XMM registers.[2] C++ keeps the different fields of an object together by allocating enough space in memory to hold all the objects fields together. The way assembly accesses the correct data member is based on its location in memory. It also seems assembly looks at the

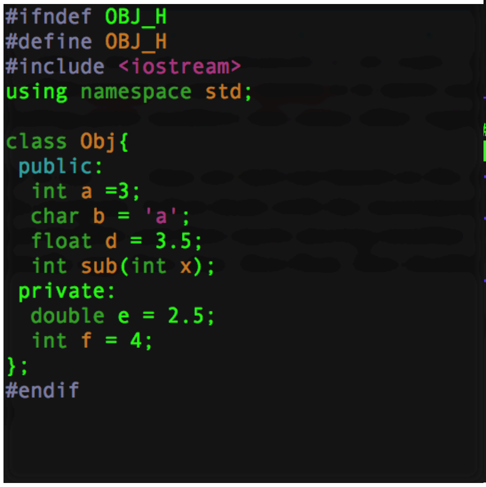
1) http://x86.renejeschke.de/html/file_module_x86_id_205.html

2) https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf

3) http://stackoverflow.com/questions/16585562/where-is-the-this-pointer-stored-in-computer-memory

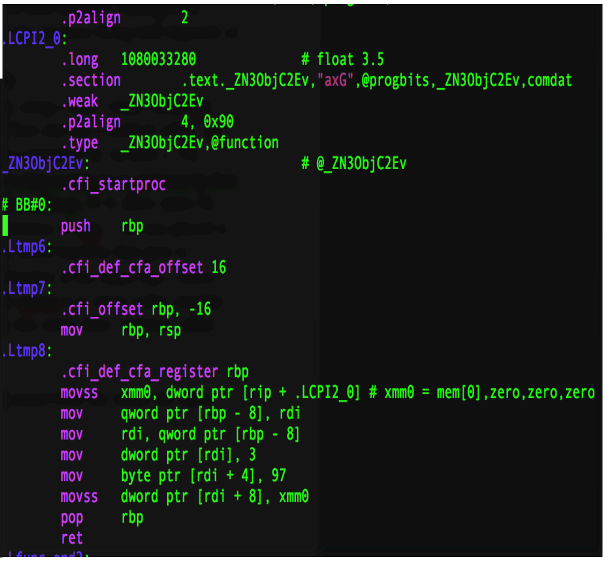
obj.s file as well to look at the data fields stored. Assembly looks at the names of the methods to know which method to access.

2. Data laid out in my C++ class are laid out based on the order they appear in the program. It first deals with the movement of decimals (like float and double) and then starts storing member. It stores the data sequentially (int, then char, then float, then double, and then int). The way the data is stored depending on whether it is public or private fields doesn't seem to matter, but the way programs work seems to be that it should matter and written somewhere in assembly to tackle this problem, but I can't find where this is happening. I believe though that the offset call seems to be the key here for accessing private or public data.



```
#ifndef OBJ_H
#define OBJ_H
#include <iostream>
using namespace std;

class Obj{
 public:
  int a =3;
  char b = 'a';
  float d = 3.5;
  int sub(int x);
 private:
  double e = 2.5;
  int f = 4;
};
#endif
```



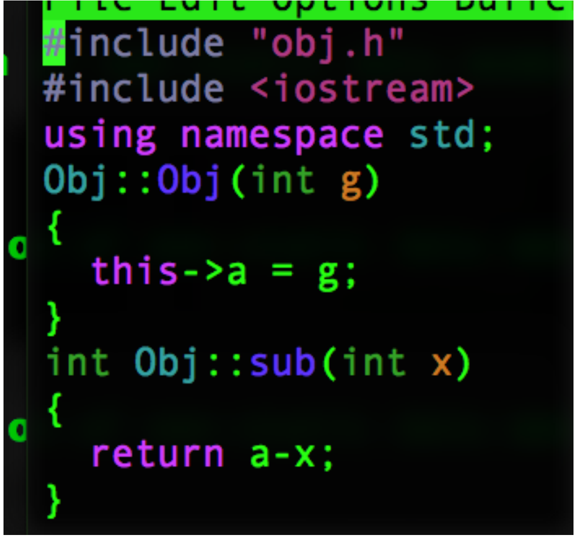
```
        .p2align        2
.LCPI2_0:
        .long   1080033280              # float 3.5
        .section        .text._ZN3ObjC2Ev,"axG",@progbits,_ZN3ObjC2Ev,comdat
        .weak   _ZN3ObjC2Ev
        .p2align        4, 0x90
        .type   _ZN3ObjC2Ev,@function
_ZN3ObjC2Ev:                            # @_ZN3ObjC2Ev
        .cfi_startproc
# BB#0:
        push    rbp
.Ltmp6:
        .cfi_def_cfa_offset 16
.Ltmp7:
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp8:
        .cfi_def_cfa_register rbp
        movss   xmm0, dword ptr [rip + .LCPI2_0] # xmm0 = mem[0],zero,zero,zero
        mov     qword ptr [rbp - 8], rdi
        mov     rdi, qword ptr [rbp - 8]
        mov     dword ptr [rdi], 3
        mov     byte ptr [rdi + 4], 97
        movss   dword ptr [rdi + 8], xmm0
        pop     rbp
        ret
```

3. Data members seem to be accessed the same whether inside or outside the function. Outside the function is by using the this pointer and inside the function can be accessed the regular way. From outside seems to be stored on a stack and has an offset.
4. I implemented the this pointer in the constructor of the obj.cpp program and set variable a equal to g. The this pointer seems to not be visibly in the assembly code so I assume its stored on the stack. It also seems to be accessed and implemented through the stack as well. The this pointer doesn't appear to be a pointer, but rather an

1) http://x86.renejeschke.de/html/file_module_x86_id_205.html

2) https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf
3) http://stackoverflow.com/questions/16585562/where-is-the-this-pointer-stored-in-computer-memory

expression. It doesn’t seem to be updated in the assembly code. [3]



```
#include "obj.h"
#include <iostream>
using namespace std;
Obj::Obj(int g)
{
  this->a = g;
}
int Obj::sub(int x)
{
  return a-x;
}
```

1) http://x86.renejeschke.de/html/file_module_x86_id_205.html

2) https://www.cs.cmu.edu/~fp/courses/15213-s07/misc/asm64-handout.pdf
3) http://stackoverflow.com/questions/16585562/where-is-the-this-pointer-stored-in-computer-memory