Lab 08 | Assembly

Parameter Passing

1. When an int, char, or float is passed by value in C++, the assembly code first pushes rbp to the stack. Then, the rsp register is moved to rbp. Then, the assembly code takes the parameter value and puts it into a spot at [rbp - <type size>]. All of my functions just returned the parameter that was passed in, so that value was then put into eax and then returned. When a pointer is passed into a function, the code takes the pointer and places into a spot that is always [rbp(the same as rsp) - 8]. This [rbp-8] is then also moved into rdi. When the object (a stack from the STL class) was passed in, the stack pointer was first moved to the rbp register and then rdi, the parameter register, was passed to the position [rbp-8]. The rbp register was then popped and the program returned nothing. This makes me think that really a pointer to that object is passed into the function instead of an actual copy of that object as it is very similar to the pointer assembly code.

```
_Z4intVi:                               # @_Z4intVi
        .cfi_startproc
# BB#0:
        push    rbp
.Ltmp74:
        .cfi_def_cfa_offset 16
.Ltmp75:
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp76:
        .cfi_def_cfa_register rbp
        mov     dword ptr [rbp - 4], edi
        mov     eax, dword ptr [rbp - 4]
        pop     rbp
        ret
```

Generated assembly for a function that takes an int as a parameter

2. For an array passed by value into a method, rdi, the parameter register, a pointer that points to the start of the array, is passed into the spot [rbp-8] where rbp is the same as the stack pointer. Then, to access a member of the array, assembly takes rdi and adds the required number to get to that member (e.g. [rdi+4] will get the second element in an array of ints). This array is treated just like a pointer was treated in assembly which aligns with the way in which C++ deals with arrays in general.

```
_Z6arrayVPi:                            # @_Z6arrayVPi
        .cfi_startproc
# BB#0:
        push    rbp
.Ltmp95:
        .cfi_def_cfa_offset 16
.Ltmp96:
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp97:
        .cfi_def_cfa_register rbp
        mov     qword ptr [rbp - 8], rdi
        mov     rdi, qword ptr [rbp - 8]
        mov     eax, dword ptr [rdi + 4]
        pop     rbp
        ret
```

Generated assembly for function that takes array as parameter and returns second element

3. When passing by reference in assembly, the code takes the memory address of the value and places into a spot that is always [rbp(the same as rsp) - 8]. This makes sense because memory addresses are all the same size and are not dependent on the data type at that memory address. This [rbp-8] is then also moved into rdi. This is done in the exact same way as when a pointer is passed by value. I would guess the only difference is what assembly code is allowed to be generated and what is automatically generated when using a reference versus a pointer. A reference and a pointer are basically the same thing with a few key differences, and I would guess that these differences are only represented in the assembly code when necessary.

```
_Z4intRRi:                              # @_Z4intRRi
        .cfi_startproc
# BB#0:
        push    rbp
.Ltmp98:
        .cfi_def_cfa_offset 16
.Ltmp99:
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp100:
        .cfi_def_cfa_register rbp
        mov     qword ptr [rbp - 8], rdi
        mov     rdi, qword ptr [rbp - 8]
        mov     eax, dword ptr [rdi]
        pop     rbp
        ret
```

Generated assembly of a function that takes an int by reference as a parameter

```
_Z6floatRRf:                            # @_Z6floatRRf
        .cfi_startproc
# BB#0:
        push    rbp
.Ltmp107:
        .cfi_def_cfa_offset 16
.Ltmp108:
        .cfi_offset rbp, -16
        mov     rbp, rsp
.Ltmp109:
        .cfi_def_cfa_register rbp
        mov     qword ptr [rbp - 8], rdi
        mov     rdi, qword ptr [rbp - 8]
        movss   xmm0, dword ptr [rdi]   # xmm0 = mem[0],zero,zero,zero
        pop     rbp
        ret
```

Generated assembly of a function that takes a float by reference as a parameter

Objects

The class I am using to test:

```
class objects{
 public:
  objects();
  objects(int d, int e, char f, long g);
  int add();
  float add2(float x);
  char char1();
  long lg;        // public data member

 private:
  int a;
  int b;
  char c;
  float fl;
  //long lg;
};
```

1&2. When objects are constructed in assembly, the first thing that happens is that the contents of the stack pointer register (rsp) are moved to rbp. Then, rdi, which contains data (the memory address of the current object) about the object itself is moved onto the stack. When doing something object oriented, the resulting assembly code always passes some metadata about the object into the rdi register which is used to find that instance of the object on the stack. This metadata from something called "_ZNSt8ios_base4InitD1Ev" which is an address that includes the pointer that points to the current object, this is the "this" pointer. Then, each parameter of the object constructor is moved to a position on the stack behind the stack pointer by the size of the data type, in descending order by size. So, if you are passing in an int, a float, and a char, in that order, the float is put in a spot 4 behind the stack pointer [rbp-12], the int in a spot 4 behind that [rbp-12], and the char in a spot 1 behind that [rbp-13]. These values, once placed on the stack, are then placed in a position just after rdi so they can be accessed later (e.g. [rdi+4]). Public members, such as that held in the r8 register, are placed directly into the position held at rdi so that they can be differentiated from the private members.

```
push   rbp
mov    rbp,rsp
mov    al,cl
movss  xmm0,DWORD PTR ds:0x4009a4
mov    QWORD PTR [rbp-0x8],rdi
mov    DWORD PTR [rbp-0xc],esi
mov    DWORD PTR [rbp-0x10],edx
mov    BYTE PTR [rbp-0x11],al
mov    QWORD PTR [rbp-0x20],r8
mov    rdi,QWORD PTR [rbp-0x8]
mov    ecx,DWORD PTR [rbp-0xc]
mov    DWORD PTR [rdi+0x8],ecx
mov    ecx,DWORD PTR [rbp-0x10]
mov    DWORD PTR [rdi+0xc],ecx
mov    al,BYTE PTR [rbp-0x11]
mov    BYTE PTR [rdi+0x10],al
movss  DWORD PTR [rdi+0x14],xmm0
mov    r8,QWORD PTR [rbp-0x20]
mov    QWORD PTR [rdi],r8
pop    rbp
ret
```

Constructing an object in x86

3. When accessing a public data member in the main method, the assembly code pulls the data directly from the beginning of the object class which, in this case, is at the location [rbp-0x30]. Within the object, this will take us to the address at rdi which is where our public data was stored. When accessing a data member within a method of the object class, the assembly offsets the base of the object (stored at the rdi register) by however many needed to get to the required data member. (e.g. [rdi+0x10]). This is then stored in a register and calculations can be done with it.

```
mov    rdi,QWORD PTR [rbp-0x30]
mov    QWORD PTR [rbp-0x48],rdi
add    rsp,0x50
pop    rbp
ret
```

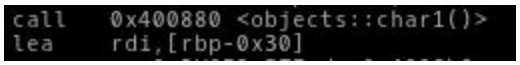
This bit of x86 is the main method accessing the public data member lg.

```
push   rbp
mov    rbp,rsp
mov    QWORD PTR [rbp-0x8],rdi
mov    rdi,QWORD PTR [rbp-0x8]
movsx  eax,BYTE PTR [rdi+0x10]
pop    rbp
ret
```

This bit of x86 is the char1() method accessing the private data member c.

4. Within the main method, public member functions are accessed using the “call” instruction with a memory address pointing to the desired function. This call goes to the assembly of the desired function and executes what is there. After, rdi is reset to a position offset from the base pointer (e.g. [rbp-0x30]). The rdi register always resets to this position on the stack as it allows for the space necessary to store the object and all of the local variables. This offset is calculated before the x86 code is generated and then used throughout the life of the main method. When executing a member function,

the “this” pointer is stored in the rdi register. So, when a function of the class is called, the pointer to that object is treated as the first parameter and is passed to the rdi register. This pointer is updated whenever the object is physically moved to a different memory address. All data members placement on the stack are based in relation to the “this” pointer.



Example of calling a function and resetting rdi in main

Example of rdi being used to store “this” pointer of object

Sources

https://www.cs.uaf.edu/2011/fall/cs301/lecture/10_07_class.html
https://www.cs.uaf.edu/courses/cs301/2014-fall/notes/float-asm/
https://en.wikibooks.org/wiki/X86_Disassembly/Objects_and_Classes