John Zheng
jz5pt
3/28/19
postlab8

Parameter passing

1. How are variables (ints, chars, pointers, floats, etc.) passed by value? How are they passed by reference? Create several functions and examine the parameter registers to help you answer this question.

   C++:

   ```
   int square(int num) {
      return num * num;
   }
   ```

   Assembly:

   ```
   square(int):
           push rbp
           mov rbp, rsp
           mov DWORD PTR [rbp-4], edi
           mov eax, DWORD PTR [rbp-4]
           imul eax, DWORD PTR [rbp-4]
           pop rbp
           ret
   ```

In this simple function above, the parameter is passed by value and the function returns the square of the integer passed in. We can see in the assembly code converted from godbolt.org that the assembly code pushes the base pointer onto the stack to save its address. It then moves the stack pointer to the base pointer. The next line moves edi, the register for the first parameter, into the 4 bytes of memory allocated. It then moves that parameter into eax, the return register. Then it multiplies the return register and the parameter and stores it in eax. It then pops the base pointer and returns.

   C++:

   ```
   int square(int& num) {
      return num * num;
   }
   ```

   Assembly:

   ```
   square(int&):
           push rbp
           mov rbp, rsp
           mov QWORD PTR [rbp-8], rdi
           mov rax, QWORD PTR [rbp-8]
           mov edx, DWORD PTR [rax]
           mov rax, QWORD PTR [rbp-8]
           mov eax, DWORD PTR [rax]
           imul eax, edx
           pop rbp
           ret
   ```

in the simple function above, the parameter is passed in by reference and the function returns the square of the value. Most of the code is pretty similar to the original function where the parameter is passed by value. The difference is that in the function above uses a pointer, which is 8 bytes instead of 4.

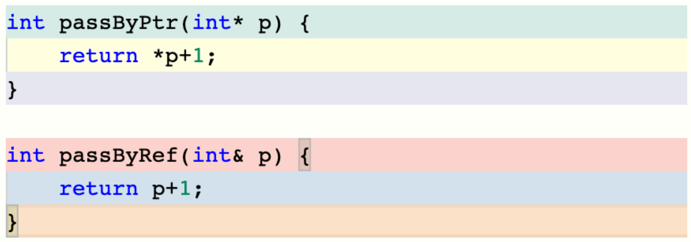
2. Create a simple function that takes in an object. How are objects passed by value? How are they passed by reference? Specifically, what is contained in the parameter registers in each case?
   C++:
   ```
   #include <string>
   using namespace std;
   int stringLength(string str) {
      return str.length();
   }
   ```
   Assembly:
   ```
   stringLength(std::__cxx11::basic_string<char, std::char_traits<char>,
   std::allocator<char> >):
           push    rbp
           mov     rbp, rsp
           sub     rsp, 16
           mov     QWORD PTR [rbp-8], rdi
           mov     rax, QWORD PTR [rbp-8]
           mov     rdi, rax
           call    std::__cxx11::basic_string<char, std::char_traits<char>,
   std::allocator<char> >::length() const
           leave
           ret
   ```
   The function above returns the length of the string passed in by value. In the assembly code, we can see that the setup is standard; push the base pointer onto the stack and move the stack pointer into the base pointer. Then it subtracts 16 bytes from the base pointer to allocate memory and moves the value passed into the space 8 bytes from the base pointer. It then moves it again into the return register. Before the function calls str.length(), it moves what's in the return register into rdi, the first parameters register. Then it pops the base pointer and returns. The assembly code is the same for the same function that passes by reference. ("Passing Values in C++.")

3. Create an array in your main method, and write a function that takes it in as a parameter. How are arrays passed into functions? How does the callee access the parameters? Where are the data values placed? Hint: you will need to determine at least a register-relative address.
   C++:
   ```
   int foo [] = { 1, 4, 3 };
   int arrayLength(int a[]) {
     return a[1] + 1;
   }
   ```
   Assembly:

```
foo:
        .long   1
        .long   4
        .long   3
arrayLength(int*):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        add     rax, 4
        mov     eax, DWORD PTR [rax]
        add     eax, 1
        pop     rbp
        ret
```

The function above simply returns the second value of the array plus one. The set up is standard, push base pointer to the stack and moves the stack pointer to the base pointer. It then moves the array into memory location 8 bytes from the base pointer and then moves that into the return register. It adds 4 to the return register because the program wants to locate the array at index 1, which is 4 bytes from the original memory location because it is an int array. It dereferences that pointer and adds one. Then it pops the base pointer and returns. ("12.4. Arrays in Assembly Language,")

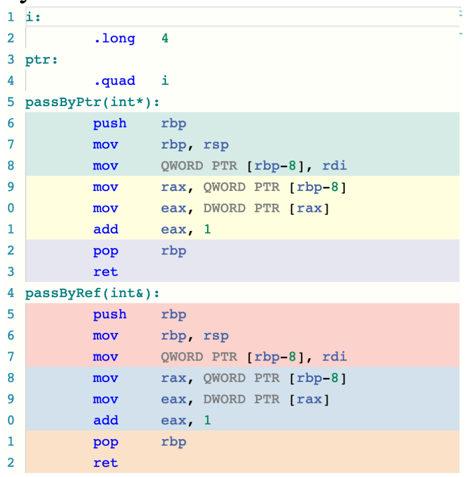
4. Is passing values by reference different than passing by pointer? If they are the same, what exactly is passed in the parameter register? If they are different, how so?
   C++:



```cpp
int passByPtr(int* p) {
    return *p+1;
}

int passByRef(int& p) {
    return p+1;
}
```

Assembly:



```
i:
        .long   4
ptr:
        .quad   i
passByPtr(int*):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        mov     eax, DWORD PTR [rax]
        add     eax, 1
        pop     rbp
        ret
passByRef(int&):
        push    rbp
        mov     rbp, rsp
        mov     QWORD PTR [rbp-8], rdi
        mov     rax, QWORD PTR [rbp-8]
        mov     eax, DWORD PTR [rax]
        add     eax, 1
        pop     rbp
        ret
```

The functions above take in an int by pointer and by reference, respectively. The setup of assembly code for the two functions are identical and so is the rest of the code. This leads to the conclusion that the two are the same in assembly.

Objects

1. How is object data laid out in memory? Create an object in your main method, and view where each data member is located in memory. How does C++ keep different fields of an object "together"?

It puts them in order in memory from the initial address. To access each object data, you add the correct offset to the initial address of the object. This is similar to how you access elements of an array in memory. You know the size of each cell and the index you want to access, you then multiply the size of the cell by the index and add the initial address of the array to get to the element you want.

2. Explain how data member access works for objects. How does the assembly know which data member to access? We know how local variables and parameters are accessed (offsets from the base pointer) -- describe how this is done for data fields.

The assembler knows the initial address of the object and each data member is in order. To access the data members, it just goes from initial address as much as it needs. For example, if an object has member variables x and y, respectively, it would access y by adding the size of the data type by 2.

3. How does method invocation work for objects? Specifically, how does the assembly know which object it is being called out of? Remember that assembly is not object oriented.

Before the object calls a method, the assembler loads the address of the object into the parameter registers and calls the method. It knows which object it is being called out of that way.

4. How are data members accessed both from inside a member function and from outside? In other words, describe what the assembly code does to access data members in both of these situations.

In C++, there is a distinction between public and private. In assembly, however, there is no distinction between the two. The code in both cases is the same, by adding the correct offset to the initial address of the object.

5. How are public member functions accessed for your class? Call some of the public member functions for your class and examine the parameters. How is the "this" pointer implemented? Where is it stored? When is it accessed? How is it passed to member functions?

This pointer is just the pointer obtained by retrieving the proper pointer location from the stack using the correct offset, and then moving that address into the first parameter register to be used by the method call. This pointer is pushed onto the stack for storage. And is accessed when either a method is called.

Work sited

"12.4. Arrays in Assembly Language," 28 Mar. 2019
http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch12s04.html

"Passing Values in C++." CUED Department of Engineering, University of Cambridge, www.h.eng.cam.ac.uk/help/tpl/anguages/C++/argsC++.html