

## Parameter passing

- 1) For my program, I used int, double char, pointer, reference to see how parameter passing works. Int, double char, pointer, reference were passed into the edi, xmm0, xmm1, al, rdx, rcx registers respectively. I hadn't seen the xmm0 and xmm1 registers before because we've been working with ints for the prelab and inlab. But after some researching, I discovered that there are several xmm registers. Xmm registers are a part of SSE (Streaming SIMD Extensions)<sup>1</sup>. These xmm registers are where floating point, doubles, anything that is not a whole number is stored because xmm registers are properly able to operate on those numbers and treat them as "decimals" instead of whole numbers. As you can see in the screenshot below, rdx and rcx are the registers that hold the passed int, pointer, and reference. Based on this, ints and other whole number data types like long can be passed in and placed in one of the "general purpose registers" with their hex value but floats and doubles need to be placed in one of the SSE registers. Rdx and rcx store the pointer and reference values passed and based on that, pointers and references are passed by their memory address. The program manipulated RBP to retrieve the parameters and place them in their proper register:

```
mov dword ptr [rbp - 4], edi
movsd qword ptr [rbp - 16], xmm0
movss dword ptr [rbp - 20], xmm1
mov byte ptr [rbp - 21], al
mov qword ptr [rbp - 32], rdx
mov qword ptr [rbp - 40], rcx
```

\*Scroll down for screenshot that was referred to above.\*

---

<sup>1</sup> [https://en.wikibooks.org/wiki/X86\\_Assembly/SSE](https://en.wikibooks.org/wiki/X86_Assembly/SSE)

The screenshot shows a C++ IDE with a file named `postlab.cpp`. The code defines a function `para1` that takes five arguments: an integer `i`, a double `d`, a float `f`, a char `c`, and an integer reference `&r`. The function prints each argument. The `main` function calls `para1` with arguments `i1=3`, `d1=3`, `f1=3`, `c1='3'`, and `&i1`. The debugger window shows the state of the program at the call site. The `rsi` register holds the address of `i1`, `rdi` holds the address of `d1`, `rcx` holds the address of `f1`, `rbx` holds the address of `c1`, and `rax` holds the address of `&i1`. The `rip` register points to the instruction `para1(i1,d1,f1,c1,&i1,i1);` in `postlab.cpp`.

```
1 //Name:Shahroz Imtiaz
2 //Email ID:si6rf
3 //File Name: postlab.cpp
4 //Date:11/6/2018
5 #include <iostream>
6 using namespace std;
7
8 void para1 (int i, double d, float f, char c, int* p, int &r){
9     cout<<i<<endl;
10    cout<<d<<endl;
11    cout<<f<<endl;
12    cout<<c<<endl;
13    cout<<*p<<endl;
14    cout<<r<<endl;
15 }
16
17 int main(){
18     int i1;
19     double d1;
20     float f1;
21     char c1;
22
23     i1=3;
24     d1=3;
25     f1=3;
26     c1 = '3';
27
28     para1(i1,d1,f1,c1,&i1,i1);
29     cout<<"halt"<<endl;
30 }
```

General Purpose Registers:

```
rax = 0x00007ffefbfff9cc
rbx = 0x0000000000000000
rcx = 0x00007ffefbfff9cc
rdx = 0x00007ffefbfff9cc
rdi = 0x0000000000000003
rsi = 0x0000000000000033
rbp = 0x00007ffefbfff9e0
rsp = 0x00007ffefbfff9a8
r8 = 0x0000000000000000
r9 = 0xffffffff00000000
r10 = 0x00007fff936870c8 atexit_mutex + 24
r11 = 0x00007fff936870d0 atexit_mutex + 32
r12 = 0x0000000000000000
r13 = 0x0000000000000000
r14 = 0x0000000000000000
r15 = 0x0000000000000000
rip = 0x0000000100000d30 a.out`para1(int, double, float, char, int*, int
&) at postlab.cpp:8
rflags = 0x0000000000000202
cs = 0x000000000000002b
fs = 0x0000000000000000
gs = 0x0000000000000000
```

- 2) When an object is passed by reference, the memory address of the object is stored in the parameter register, as is the case with RSI in the screenshot below. But when an object is passed by value, after looking at the rdi value and the assembly code, I hypothesize that the object's memory address is offset by some value and is then stored in rdi. In either case, the program uses `qword ptr [rbp - *num*]` to move, access, and manipulate the object.

\*Scroll down for screenshot that was referred to above.\*

Shahroz Imtiaz  
si6rf  
postlab8.pdf  
11/7/2018

The screenshot shows a C++ IDE with a file named `postlab.cpp`. The code defines a function `para1` that takes a string `s` and a string reference `&s1`, and prints both. The `main` function calls `para1` with the string "hello". The debugger window shows the program has stopped at line 9 of `para1`. The registers window shows the state of the system, including the `rax` register pointing to the `cout` object.

```
1 //Name:Shahroz Imtiaz
2 //Email ID:si6rf
3 //File Name: postlab.cpp
4 //Date:11/6/2018
5 #include <iostream>
6 using namespace std;
7
8 void para1 (string s,string &s1){
9     cout<<s<<endl;
10    cout<<s1<<endl;
11 }
12
13 int main(){
14     string str = "hello";
15     cout<<&str<<endl;
16     para1(str,str);
17     cout<<"halt"<<endl;
18 }
```

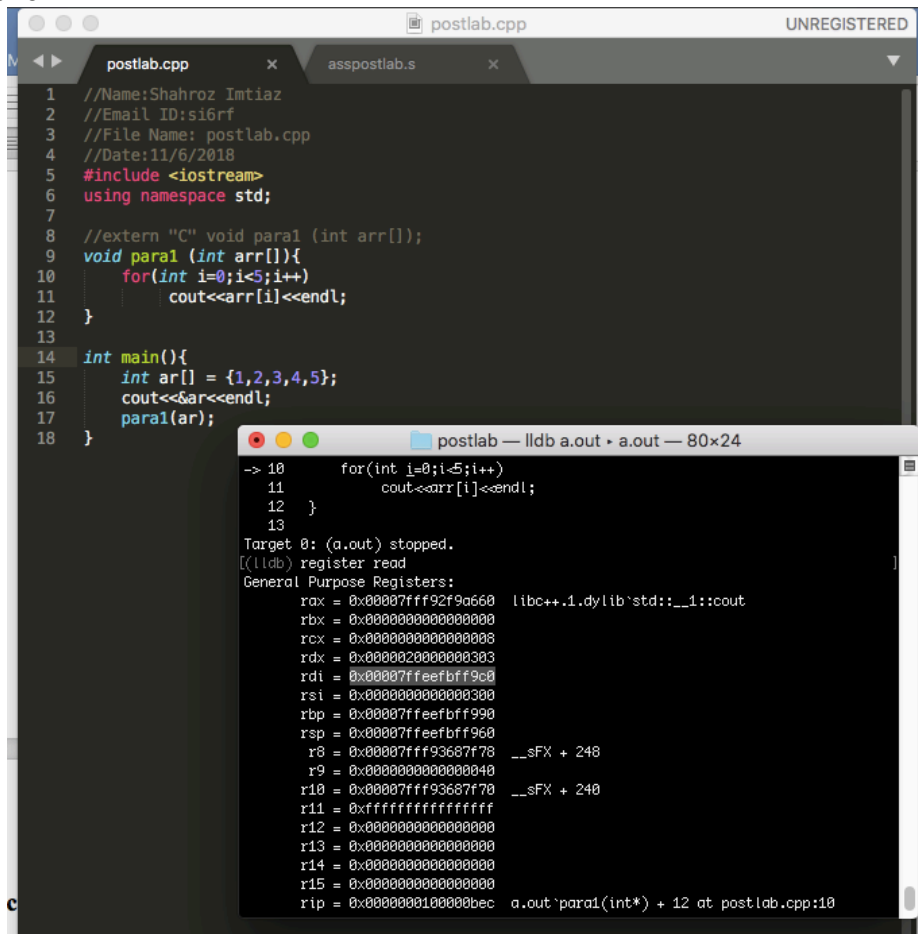
Process 21824 stopped  
\* thread #1, queue = 'com.apple.main-thread', stop reason = step in  
frame #0: 0x0000000100000a93 a.out`para1(s="hello", s1="hello") at postlab.c  
pp:9  
6 using namespace std;  
7  
8 void para1 (string s,string &s1){  
-> 9 cout<<s<<endl;  
10 cout<<s1<<endl;  
11 }  
12  
Target 0: (a.out) stopped.  
[(lldb) register read  
General Purpose Registers:  
rax = 0x00007fff92f9a660 libc++.1.dylib`std::\_\_1::cout  
rbx = 0x0000000000000000  
rcx = 0x0000000000000000  
rdx = 0x00000020000000303  
rdi = 0x00007ffefbfff938  
rsi = 0x00007ffefbfff960  
rbp = 0x00007ffefbfff8e0  
rsp = 0x00007ffefbfff8a0  
r8 = 0x00007fff93687f78 \_\_sFX + 248  
r9 = 0x0000000000000040

- 3) Arrays are passed into functions via a pointer pointing to the base address of the array. The callee accesses the parameters which are on the stack by manipulating the RBP pointer. The data values are also on the stack and are accessed by adding `*num*` to the base address. For my code, the program did the following:

```
para1(int*): //as evident, the array is passed in as a pointer pointing to the base address
movsxd rax, dword ptr [rbp - 12]
mov rcx, qword ptr [rbp - 8]
mov esi, dword ptr [rcx + 4*rax]
```

\*Scroll down for screenshot \*

Shahroz Imtiaz  
si6rf  
postlab8.pdf  
11/7/2018



The screenshot shows a code editor with two tabs: 'postlab.cpp' and 'asspostlab.s'. The 'postlab.cpp' tab is active, displaying the following C++ code:

```
1 //Name:Shahroz Imtiaz
2 //Email ID:si6rf
3 //File Name: postlab.cpp
4 //Date:11/6/2018
5 #include <iostream>
6 using namespace std;
7
8 //extern "C" void para1 (int arr[]);
9 void para1 (int arr[]){
10     for(int i=0;i<5;i++)
11         cout<<arr[i]<<endl;
12 }
13
14 int main(){
15     int ar[] = {1,2,3,4,5};
16     cout<<ar<<endl;
17     para1(ar);
18 }
```

Below the code editor, a debugger window titled 'postlab - lldb a.out - a.out - 80x24' is open. It shows the execution of the 'para1' function. The code being executed is:

```
-> 10     for(int i=0;i<5;i++)
11         cout<<arr[i]<<endl;
12     }
13
```

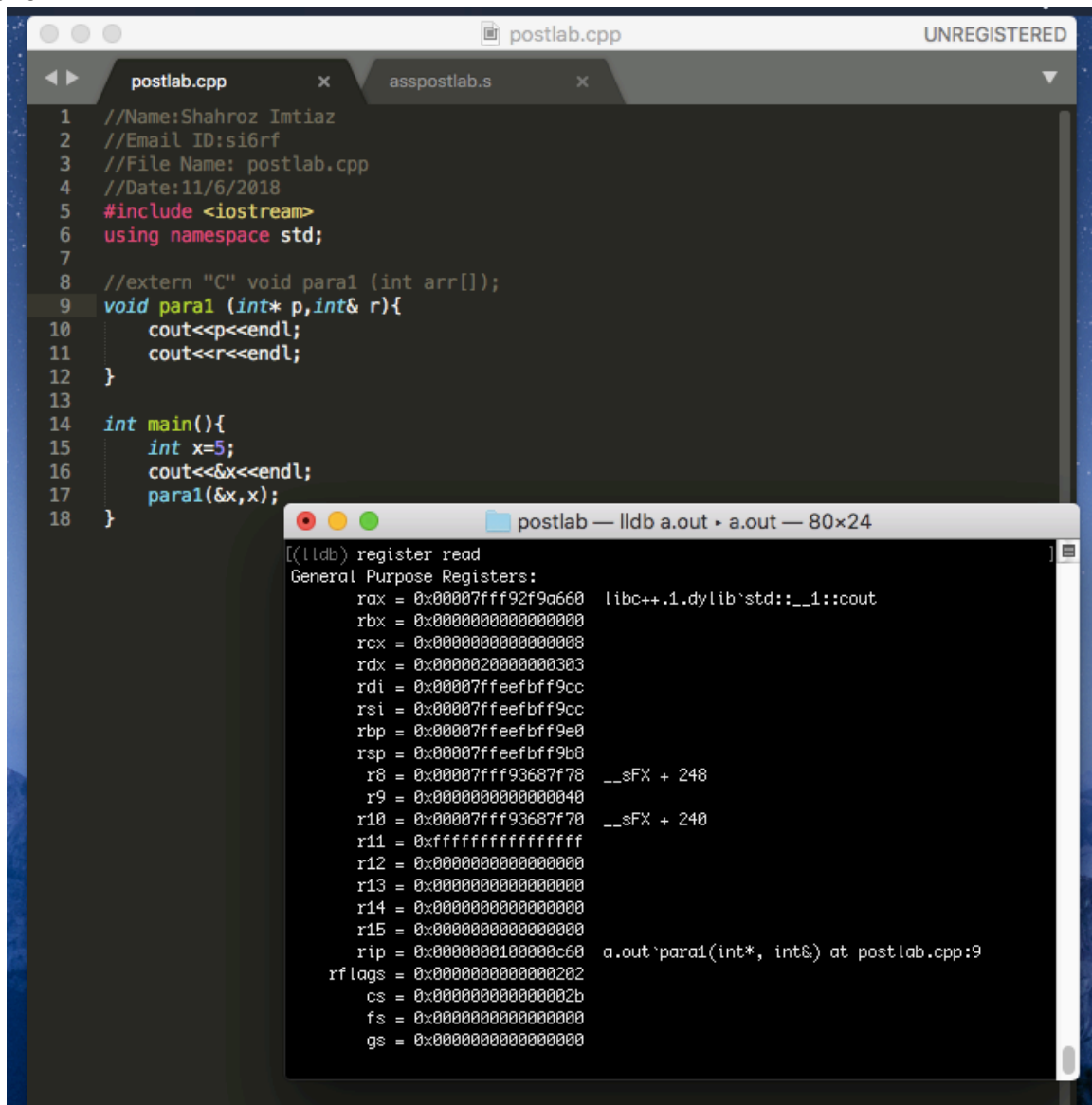
The debugger indicates that the target process '(a.out)' has stopped. It shows the register read for the 'para1' function. The General Purpose Registers are listed as follows:

Register	Value	Comment
rax	0x00007fff92f9a660	libc++.1.dylib`std::__1::cout
rbx	0x0000000000000000	
rcx	0x0000000000000000	
rdx	0x00000020000000303	
rdi	0x00007ffefbfff9c0	
rsi	0x00000000000000300	
rbp	0x00007ffefbfff990	
rsp	0x00007ffefbfff960	
r8	0x00007fff93687f78	__sFX + 248
r9	0x0000000000000040	
r10	0x00007fff93687f70	__sFX + 240
r11	0xffffffffffffffff	
r12	0x0000000000000000	
r13	0x0000000000000000	
r14	0x0000000000000000	
r15	0x0000000000000000	
rip	0x0000000100000bec	a.out`para1(int*) + 12 at postlab.cpp:10

- 4) Passing by reference and passing by pointer made no difference. RDI (which was used to hold the pointer value) stored the memory address of the variable being passed in. RSI (which was used to hold the reference value) held the same memory address. The assembly code for both was also almost identical in instructions and length. This suggests that one isn't more efficient than the other and the reason for using one of over the other depends on what you're doing in your program.

\*Scroll down for evidence\*

Shahroz Imtiaz  
si6rf  
postlab8.pdf  
11/7/2018



The image shows a code editor window with two tabs: 'postlab.cpp' and 'asspostlab.s'. The 'postlab.cpp' tab is active, displaying the following C++ code:

```
1 //Name:Shahroz Imtiaz
2 //Email ID:si6rf
3 //File Name: postlab.cpp
4 //Date:11/6/2018
5 #include <iostream>
6 using namespace std;
7
8 //extern "C" void para1 (int arr[]);
9 void para1 (int* p,int& r){
10     cout<<p<<endl;
11     cout<<r<<endl;
12 }
13
14 int main(){
15     int x=5;
16     cout<<&x<<endl;
17     para1(&x,x);
18 }
```

Below the code editor, a debugger window titled 'postlab — lldb a.out • a.out — 80x24' is open. It displays the output of the 'register read' command, showing the values of various registers:

```
[(lldb) register read]
General Purpose Registers:
rax = 0x00007fff92f9a660  libc++.1.dylib`std::__1::cout
rbx = 0x0000000000000000
rcx = 0x0000000000000000
rdx = 0x00000020000000303
rdi = 0x00007ffeefbff9cc
rsi = 0x00007ffeefbff9cc
rbp = 0x00007ffeefbff9e0
rsp = 0x00007ffeefbff9b8
r8 = 0x00007fff93687f78  __sFX + 248
r9 = 0x0000000000000040
r10 = 0x00007fff93687f70  __sFX + 240
r11 = 0xfffffffffffffff
r12 = 0x0000000000000000
r13 = 0x0000000000000000
r14 = 0x0000000000000000
r15 = 0x0000000000000000
rip = 0x0000000100000c60  a.out`para1(int*, int&) at postlab.cpp:9
rflags = 0x0000000000000202
cs = 0x000000000000002b
fs = 0x0000000000000000
gs = 0x0000000000000000
```

```
5 #include <iostream>
6 using namespace std;
7
8 //extern "C" void paral (int arr[]);
9 void paral (int* p,int& r){
10     cout<<p<<endl;
11     cout<<r<<endl;
12 }
13
14 int main(){
15     int x=5;
16     cout<<&x<<endl;
17     paral(&x,x);
18 }
```

```
21     mov     qword ptr [rbp - 8], rdi
22     mov     qword ptr [rbp - 16], rsi
23     mov     rsi, qword ptr [rbp - 8]
24     mov     rdi, rax
25     call    std::basic_ostream<char, std::char_traits<char>>::operator<<(std::basic_ostream<char, std::char_traits<char>>*, const char*)@plt
26     movabs  rsi, std::basic_ostream<char, std::char_traits<char>>::operator<<(std::basic_ostream<char, std::char_traits<char>>*, const char*)@plt
27     mov     rdi, rax
28     call    std::basic_ostream<char, std::char_traits<char>>::operator<<(std::basic_ostream<char, std::char_traits<char>>*, const char*)@plt
29     movabs  rdi, std::cout
30     mov     rsi, qword ptr [rbp - 16]
31     mov     esi, dword ptr [rsi]
32     mov     qword ptr [rbp - 24], rax # 8-byte Spill
33     call    std::basic_ostream<char, std::char_traits<char>>::operator<<(std::basic_ostream<char, std::char_traits<char>>*, const char*)@plt
34     movabs  rsi, std::basic_ostream<char, std::char_traits<char>>::operator<<(std::basic_ostream<char, std::char_traits<char>>*, const char*)@plt
35     mov     rdi, rax
36     call    std::basic_ostream<char, std::char_traits<char>>::operator<<(std::basic_ostream<char, std::char_traits<char>>*, const char*)@plt
37     mov     qword ptr [rbp - 24], rax # 8-byte Spill
```

## Objects

1. Before I started thinking about how is object data laid out in memory, I did some research to get the basic idea of it all. What I found out was that the parts of a class get laid out in memory at higher and higher address (similar to arrays). But the big difference is that the different fields of an object get accessed at a fixed offset. A pointer to a class is a pointer to the first byte of the first field of the class (just like arrays and their “base address”)<sup>2</sup>. Now to test out my findings, I created a small class called shoes. I then created an instance of said class (i.e. object), assigned its data members values and looked at the assembly code. The assembly code revealed that my suspicions and research were correct. C++ keeps the different fields of an object “together” by assigning it sequentially higher memory addresses and keeping track of said memory addresses, so that it can access/manipulate them with an offset to the rbp.

\*Scroll down for sample class code\*

<sup>2</sup> [https://www.cs.uaf.edu/2011/fall/cs301/lecture/10\\_07\\_class.html](https://www.cs.uaf.edu/2011/fall/cs301/lecture/10_07_class.html)

Shahroz Imtiaz  
si6rf  
postlab8.pdf  
11/7/2018



```
1 //Name:Shahroz Imtiaz
2 //Email ID:si6rf
3 //File Name: shoes.cpp
4 //Date:11/6/2018
5 #include <iostream>
6 using namespace std;
7
8 class shoes{
9
10 public:
11     int quantity;
12     double size;
13     char color;
14
15     shoes(int quantity, double size, char color, long price, string brand){
16         this->quantity=quantity;
17         this->size=size;
18         this->color=color;
19         this->price=price;
20         this->brand=brand;
21     }
22     int getPrice(){
23         return this->price;
24     }
25     int isBetter(shoes &other){
26         if(this->getPrice()>other.getPrice())
27             return 0;
28         else return 1;
29     }
30     void para1 (int* p,int& r){
31         cout<<p<<endl;
32         cout<<r<<endl;
33     }
34 private:
35     long price;
36     string brand;
37
38 };
39
40 int main(){
41     shoes runners = shoes(2,9.5,'B',8,"Nike");
42     shoes runners2 = shoes(1,8.5,'W',6,"Adidas");
43
44     cout<<runners.isBetter(runners2)<<endl;
45 }
```

The following code snippet shows how assembly accesses one of my objects field variables and assigns it a value:

```
this->quantity=quantity; //This is the C++ version, added for clarity
```

```
mov edx, dword ptr [rbp - 12]
mov rcx, qword ptr [rbp - 80] # 8-byte Reload
mov dword ptr [rcx], edx
```

Shahroz Imtiaz

si6rf

postlab8.pdf

11/7/2018

2. Data members for an object are arranged sequentially. Similar to how parameters and variables are accessed, an offset to the base pointer is used. This offset, as stated before, is dependent on when you declared the data field in your class.
3. After doing some research and looking at the assembly code for my sample class, I conclude that if you had an object S1 and a method S1.geti, the compiler passes the memory address of your object S1 as a parameter into a register (like you would with any other parameter) and that becomes the “this” pointer<sup>3</sup>.
4. To test how are data members accessed both from inside a member function and from outside I created a isBetter() method that compares the prices of two shoe objects. In my main method outside of my sample class, the assembly code revealed that my code used the “call” command to call my isBetter() method that it had already declared. So, it jumped to isBetter() accessed the data members like it would have normally and proceeded to return and continue with the program. So, there’s no difference on how data members are accessed from inside and outside a member function because of the call command.

Code snippet from assembly:

```
call shoes::isBetter(shoes&)
```

5. Based on the assembly code for my isBetter() method, the “this” pointer is implemented by passing the address of the “this” object into a register and then storing that address in a rbp. This address in rbp is then used to adjust the rbp, so you can simple offsetting to access the data members for the “this object”.

---

<sup>3</sup> <http://www.drdobbs.com/embedded-systems/object-oriented-programming-in-assembly/184408319>