Shahroz Imtiaz
si6rf
postlab8.pdf
11/7/2018

Parameter passing

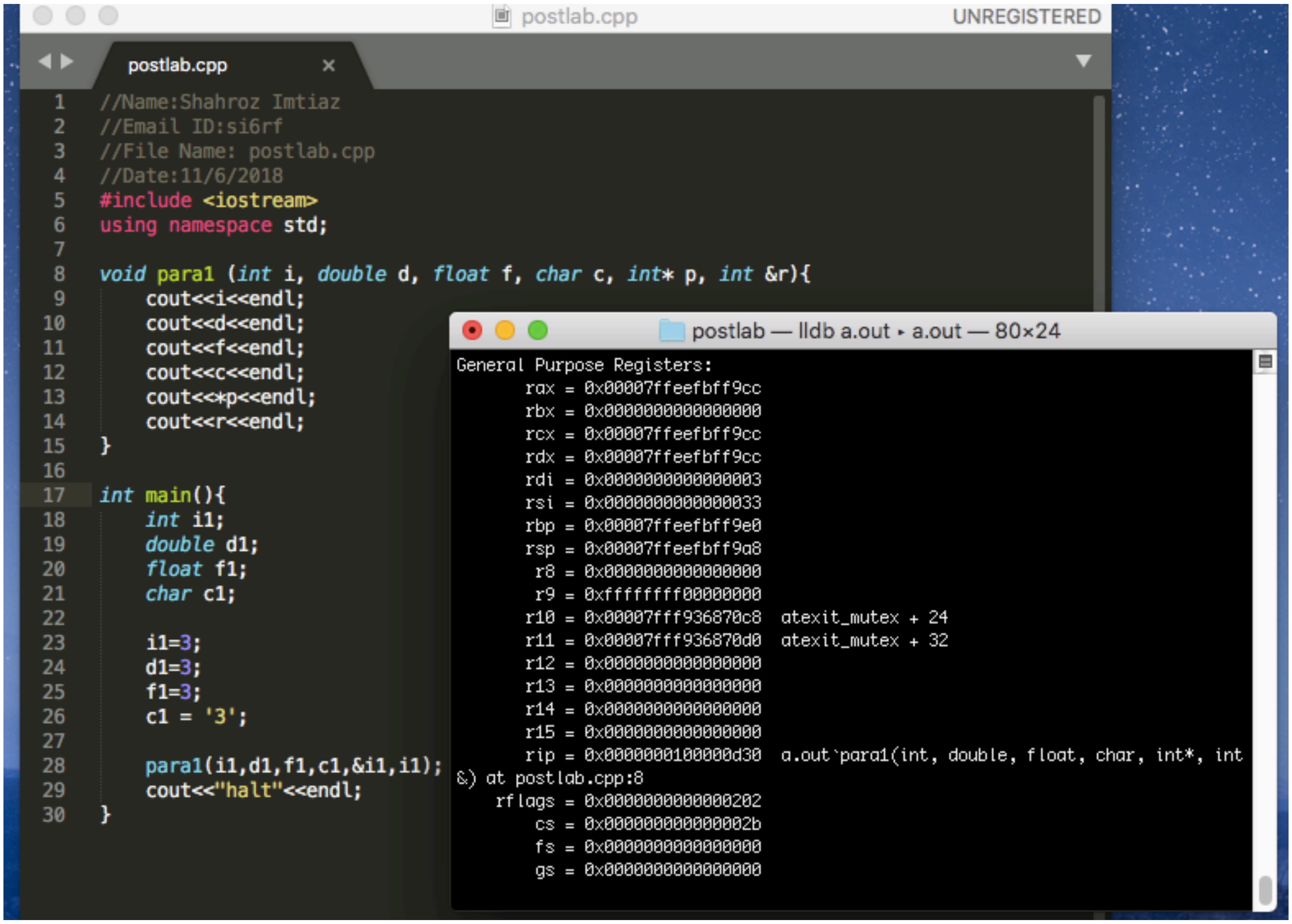
1) For my program, I used int, double char, pointer, reference to see how parameter passing works. Int, double char, pointer, reference were passed into the edi, xmm0, xmm1, a1, rdx, rcx registers respectively. I hadn't seen the xxm0 and xxm1 registers before because we've been working with ints for 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) [1]. 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 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.

[1] https://en.wikibooks.org/wiki/X86_Assembly/SSE

Shahroz Imtiaz
si6rf
postlab8.pdf
11/7/2018



postlab.cpp UNREGISTERED

```cpp
//Name:Shahroz Imtiaz
//Email ID:si6rf
//File Name: postlab.cpp
//Date:11/6/2018
#include <iostream>
using namespace std;

void para1 (int i, double d, float f, char c, int* p, int &r){
	cout<<i<<endl;
	cout<<d<<endl;
	cout<<f<<endl;
	cout<<c<<endl;
	cout<<*p<<endl;
	cout<<r<<endl;
}

int main(){
	int i1;
	double d1;
	float f1;
	char c1;

	i1=3;
	d1=3;
	f1=3;
	c1 = '3';

	para1(i1,d1,f1,c1,&i1,i1);
	cout<<"halt"<<endl;
}
```

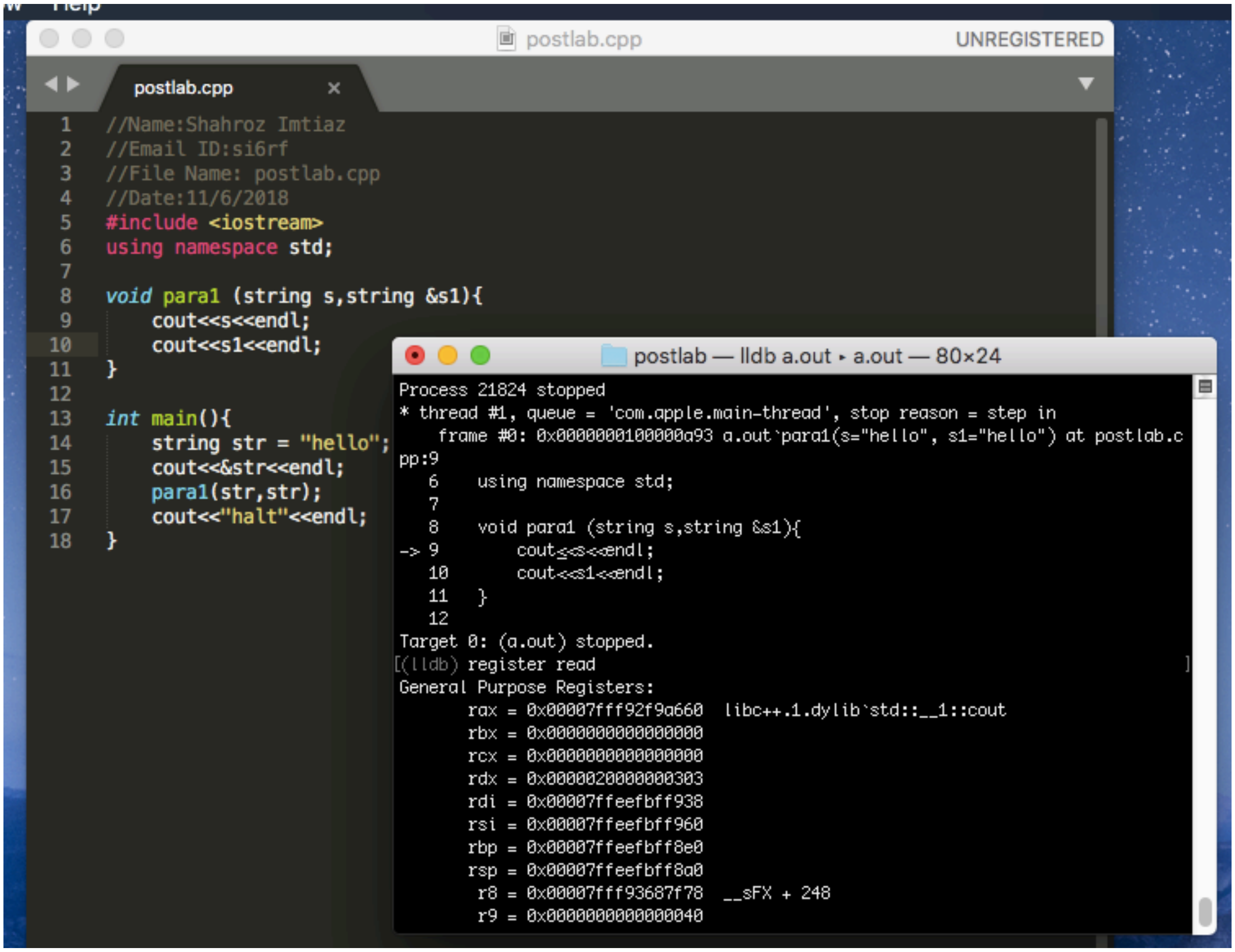
postlab — lldb a.out ▸ a.out — 80×24

```
General Purpose Registers:
       rax = 0x00007ffeefbff9cc
       rbx = 0x0000000000000000
       rcx = 0x00007ffeefbff9cc
       rdx = 0x00007ffeefbff9cc
       rdi = 0x0000000000000003
       rsi = 0x0000000000000033
       rbp = 0x00007ffeefbff9e0
       rsp = 0x00007ffeefbff9a8
        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

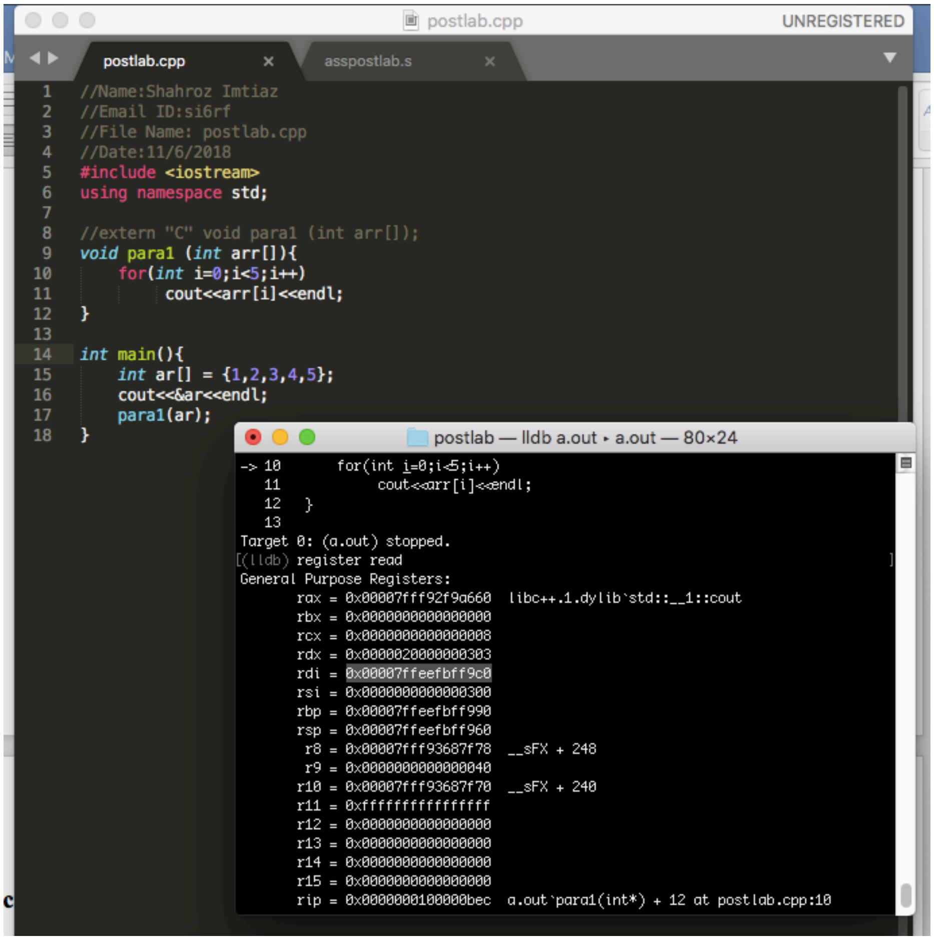


3) Arrays are passed into functions via a pointer pointing to the base address of the array. The callee accesses the parameters which are on the stack by manipulating the RBP pointer. The data values are also on the stack and are accessed by adding *num* to the base address. For my code, the program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

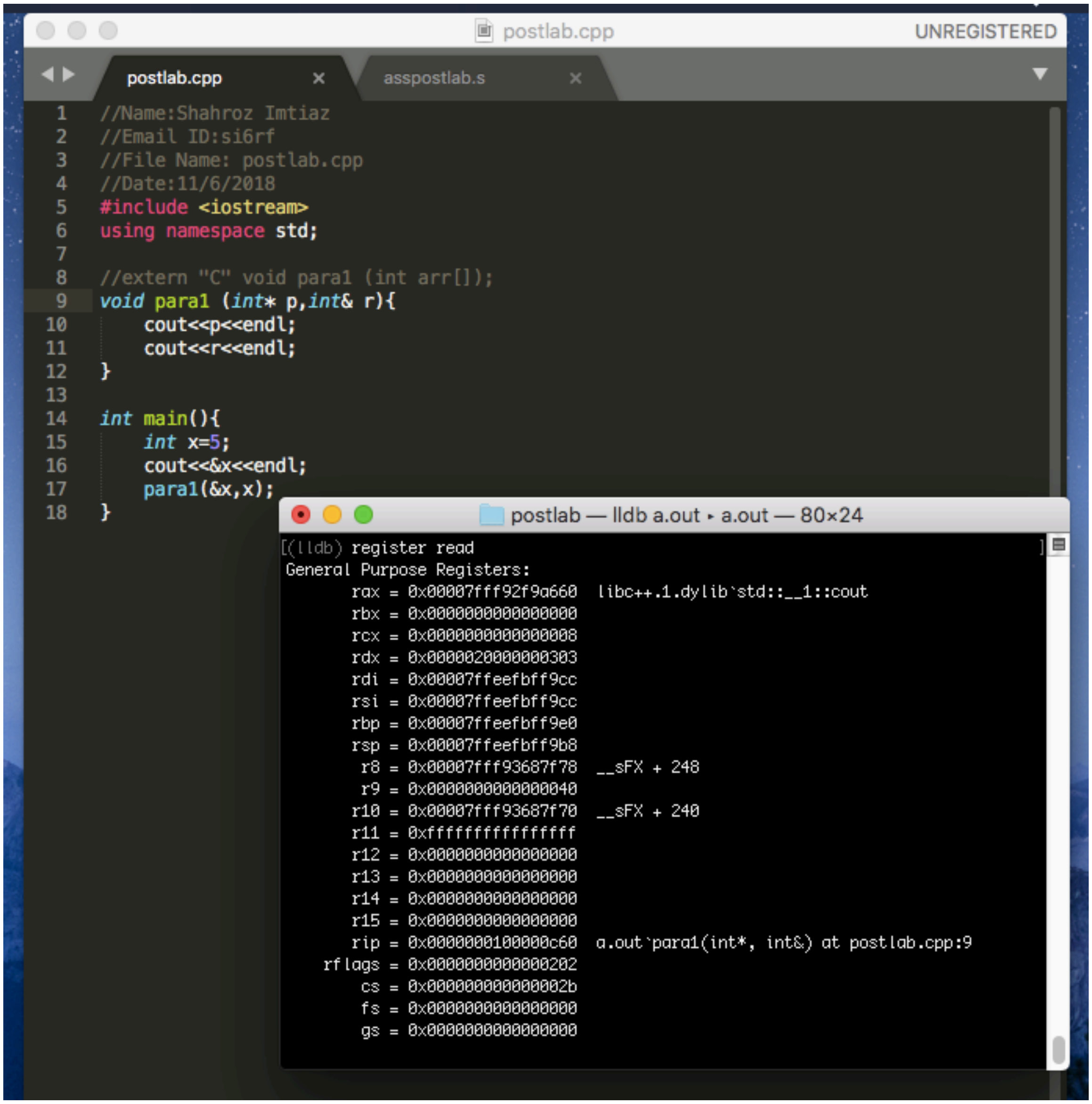
Shahroz Imtiaz
si6rf
postlab8.pdf
11/7/2018



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



Shahroz Imtiaz
si6rf
postlab8.pdf
11/7/2018

```cpp
#include <iostream>
using namespace std;

//extern "C" void paral (int arr[]);
void paral (int* p,int& r){
    cout<<p<<endl;
    cout<<r<<endl;
}

int main(){
    int x=5;
    cout<<&x<<endl;
    paral(&x,x);
}
```

```asm
        mov     qword ptr [rbp - 8], rdi
        mov     qword ptr [rbp - 16], rsi
        mov     rsi, qword ptr [rbp - 8]
        mov     rdi, rax
        call    std::basic_ostream<char, std::char_trai
        movabs  rsi, std::basic_ostream<char, std::char
        mov     rdi, rax
        call    std::basic_ostream<char, std::char_trai
        movabs  rdi, std::cout
        mov     rsi, qword ptr [rbp - 16]
        mov     esi, dword ptr [rsi]
        mov     qword ptr [rbp - 24], rax # 8-byte Spil
        call    std::basic_ostream<char, std::char_trai
        movabs  rsi, std::basic_ostream<char, std::char
        mov     rdi, rax
        call    std::basic_ostream<char, std::char_trai
```

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")[2]. 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

[2] https://www.cs.uaf.edu/2011/fall/cs301/lecture/10_07_class.html

Shahroz Imtiaz
si6rf
postlab8.pdf
11/7/2018

```cpp
//Name:Shahroz Imtiaz
//Email ID:si6rf
//File Name: shoes.cpp
//Date:11/6/2018
#include <iostream>
using namespace std;

class shoes{

public:
    int quantity;
    double size;
    char color;

shoes(int quantity, double size, char color, long price, string brand){
    this->quantity=quantity;
    this->size=size;
    this->color=color;
    this->price=price;
    this->brand=brand;
}
int getPrice(){
    return this->price;
}
int isBetter(shoes &other){
    if(this->getPrice()>other.getPrice())
        return 0;
    else return 1;
}
void para1 (int* p,int& r){
    cout<<p<<endl;
    cout<<r<<endl;
}
private:
    long price;
    string brand;


};
int main(){
    shoes runners = shoes(2,9.5,'B',8,"Nike");
    shoes runners2 = shoes(1,8.5,'W',6,"Adidas");

    cout<<runners.isBetter(runners2)<<endl;
}
```

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[3].
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".

[3] http://www.drdobbs.com/embedded-systems/object-oriented-programming-in-assembly/184408319