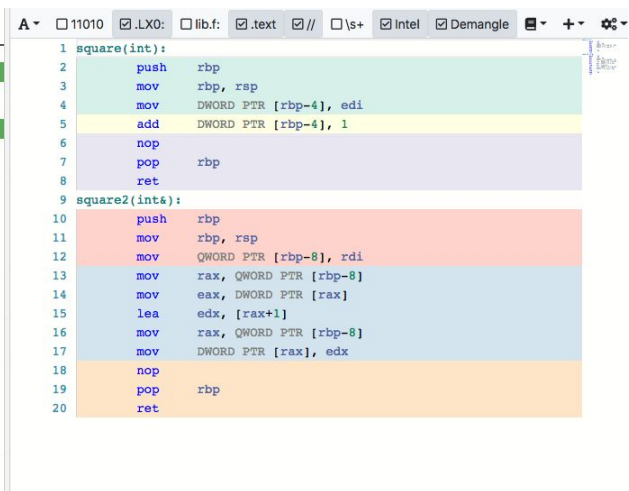


## Parameter Passing

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.

Variables passed by value are actually placed on the stack and mainly rely on the mov instruction. However, when passed by reference, the instruction lea was used, which was copying the variables addresses or arguments instead of the actual variable.

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     num++;
4 }
5 int square2(int &num) {
6     num++;
7 }
```



The screenshot shows a debugger window with assembly code for two functions: `square` and `square2`. The `square` function (lines 1-8) uses `push rbp`, `mov rbp, rsp`, `mov DWORD PTR [rbp-4], edi`, `add DWORD PTR [rbp-4], 1`, `nop`, `pop rbp`, and `ret`. The `square2` function (lines 9-20) uses `push rbp`, `mov rbp, rsp`, `mov QWORD PTR [rbp-8], rdi`, `mov rax, QWORD PTR [rbp-8]`, `mov eax, DWORD PTR [rax]`, `lea edx, [rax+1]`, `mov rax, QWORD PTR [rbp-8]`, `mov DWORD PTR [rax], edx`, `nop`, `pop rbp`, and `ret`. The debugger interface includes a toolbar with options like `.LX0:`, `lib.f:`, `.text`, `//`, `\s+`, `Intel`, and `Demangle`.

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?

The screenshot shows the Compiler Explorer interface. On the left, the C++ source code is displayed, featuring a `pop` function, a `square2` function, and a `main` function. The `pop` function takes a `stack<int>` by reference and calls `s.pop()`. The `square2` function takes a `stack<int>` by reference and also calls `s.pop()`. The `main` function creates a `stack<int>` object `s`, pushes the value 2, and returns 0. On the right, the assembly output for x86-64 gcc 8.3 is shown. The assembly code for `pop` (lines 22-34) and `square2` (lines 35-45) is visible. Both functions use `push rbp`, `mov rbp, rsp`, `sub rsp, 16`, and `mov QWORD PTR [rbp-8], rdi` to set up the stack frame. The `pop` function then calls `std::stack<int, std::deque<int, std::allocator<int>>>::pop()` (line 31). The `square2` function calls `std::stack<int, std::deque<int, std::allocator<int>>>::square2` (line 41). The `main` function (lines 46-48) pushes `rbp` and `rsp` before calling `std::stack<int, std::deque<int, std::allocator<int>>>::pop()`.

They seem to be treated by the assembly code similarly, showing that they are passed in the same way. Each parameter register consists of push, move sub and mov and uses the same registers to do so.

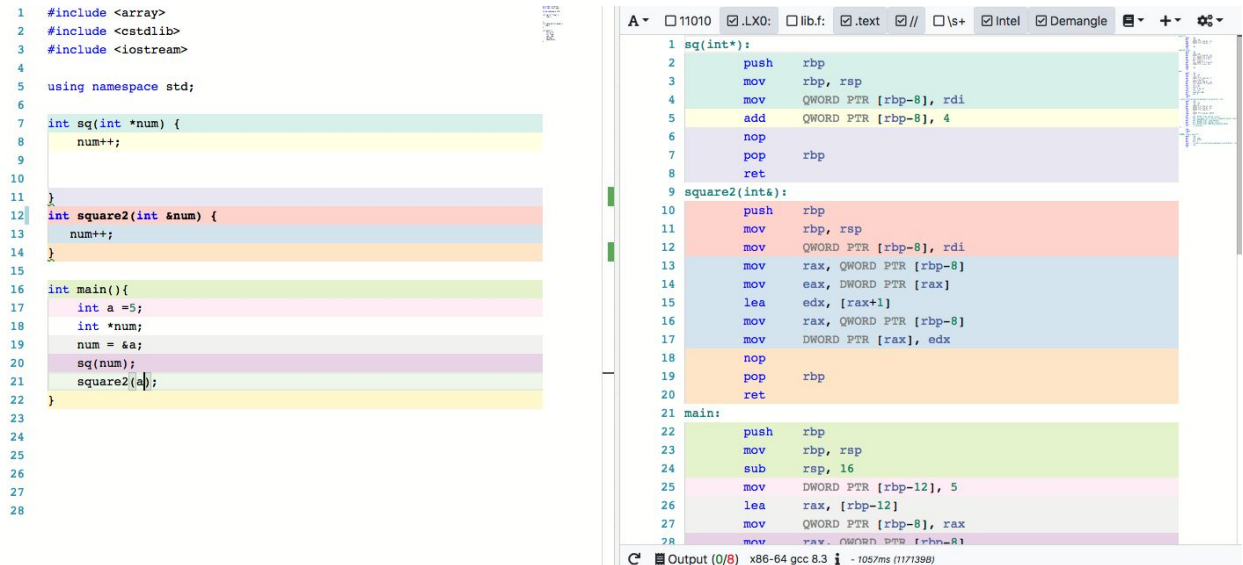
3. Create an array in your main method, and write a function that takes it in as a parameter. How are arrays passed into functions? How does the callee access the parameters? Where are the data values placed? Hint: you will need to determine at least a register-relative address

The screenshot shows the Compiler Explorer interface. On the left, the C++ source code is displayed, featuring a `copy` function and a `main` function. The `copy` function takes an array `a` of type `int a[]` and copies its contents into a local array `copy`. The `main` function creates an array `a` with values {1, 2, 3, 4, 5} and calls `copy(a)`. On the right, the assembly output for x86-64 gcc 8.3 is shown. The assembly code for `copy` (lines 1-10) and `main` (lines 11-28) is visible. The `copy` function uses `push rbp`, `mov rbp, rsp`, `sub rsp, 40`, and `mov QWORD PTR [rbp-40], rdi` to set up the stack frame. It then uses `mov rax, QWORD PTR [rbp-40]` to get the pointer to the array, `mov eax, DWORD PTR [rax+8]` to get the length, and `mov DWORD PTR [rbp-24], eax` to store it. The `main` function uses `push rbp`, `mov rbp, rsp`, `sub rsp, 32`, and `mov DWORD PTR [rbp-32], 1` to set up the stack frame. It then uses `mov DWORD PTR [rbp-28], 2`, `mov DWORD PTR [rbp-24], 3`, `mov DWORD PTR [rbp-20], 4`, and `mov DWORD PTR [rbp-16], 5` to initialize the array `a`. It then uses `lea rax, [rbp-32]` to get the address of `a`, `mov rdi, rax` to pass it to `copy`, and `call copy(int*)` to call the function. Finally, it uses `mov eax, 0` to return 0.

They are passed in as pointers. The callee accesses them as a pointer and the data values are placed using the mov register. The mov register is relative to the rbp

register, and it makes space in the pointer (s shown by DMORD PTR) and then moving each value into the pointer.

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?



The image shows a C++ source code editor on the left and its corresponding assembly output on the right. The C++ code defines a function `sq` that increments a pointer, a function `square2` that increments a pointer and then squares the value at that pointer, and a `main` function that calls `sq` and `square2`.

```
1 #include <array>
2 #include <cstdlib>
3 #include <iostream>
4
5 using namespace std;
6
7 int sq(int *num) {
8     num++;
9 }
10
11
12 int square2(int &num) {
13     num++;
14 }
15
16 int main(){
17     int a = 5;
18     int *num;
19     num = &a;
20     sq(num);
21     square2(*num);
22 }
23
24
25
26
27
28
```

The assembly output on the right shows the compiled code for `sq`, `square2`, and `main`. The `sq` function (lines 1-8) pushes `rbp`, moves `rsp` to `rbp`, moves `rdi` to `[rbp-8]`, increments `[rbp-8]` by 4, and then pops `rbp` and returns. The `square2` function (lines 9-20) pushes `rbp`, moves `rsp` to `rbp`, moves `rdi` to `[rbp-8]`, moves `[rbp-8]` to `rax`, increments `rax` by 1, moves `[rbp-8]` to `rax`, and then increments `[rax]` by `edx`. The `main` function (lines 21-28) pushes `rbp`, moves `rsp` to `rbp`, subtracts 16 from `rsp`, moves 5 to `[rbp-12]`, increments `rax` by 12, moves `[rbp-8]` to `rax`, and then moves `rax` to `[rbp-8]`.

Pass by value seems to be the same as passing by pointer. The parameter register consists of instructions push, mov and mov using the registers `rbp` and `rsp`. Therefore, most callee and caller instructions were similar as the caller pushes the value onto the stack, and the callee pops it off the stuck for manipulation.

## Objects

The screenshot displays the Compiler Explorer interface. On the left, the C++ source code for a `Square` class is shown. The class has two public integers, `x` and `y`, a `getArea()` method that returns `x*y`, and a constructor `Square(int l, int w)` that initializes `l` to `x` and `w` to `y`. A `com()` method is also present, which creates a `Square` object `S` with parameters `5` and `4`, and returns the result of `S.getArea()`.

On the right, the assembly code generated by `x86-64 gcc 8.3` is displayed. The assembly shows the `_static_initialization_and_destruction_0` function, which sets up the static data area and calls the `std::ios_base::Init::Init()` function. It then calls the `__cxa_atexit` function. The assembly also shows the `_GLOBAL__sub_I_example.cpp` function, which pushes the `rbp` register, moves `rsp` to `rbp`, and then calls the `_static_initialization_and_destruction_0` function with arguments `int` and `int`.

At the bottom, there is a status bar indicating the output is `0/0` and the compilation took `1239ms (113669B)`. There are also links to read the new cookie policy and buttons for `Consent` and `Don't consent`.

I created the square class then created an instance of it to use. The assembly code looks at my program sequentially, going through it line by line. It is then invoked when a function is called. Thus the object data is looked at sequentially and considering the fact that assembly uses the stack to store data in memory, it is able to keep the fields of an object together.

To look at how data members are actually accessed in assembly, the function `comp` calls instances of `getArea` using the `this` keyword and also the object. None of the assembly code changes, as also emphasised when working with stacks. This answers the question that assembly code accesses the members the same whether from inside or outside the member function, as the code is simply retrieved from the stack.

The pointer, as seen above in one of the screenshots is accessed and stored through the `rsp` register. It is passed to all the member functions by this register, `rsp` which sends the data of the pointer to the other functions to use. Although the pointer is also implemented through the stack, assembly is not as detailed with the updating of the pointer.

## Resources:

- <http://www.drdobbs.com/embedded-systems/object-oriented-programming-in-assembly/184408319>
- [https://en.wikibooks.org/wiki/X86\\_Disassembly/Objects\\_and\\_Classes](https://en.wikibooks.org/wiki/X86_Disassembly/Objects_and_Classes)
-