Alex Yang (ahy3nz)
inlab8

```
#include <iostream>
#include <cstdlib>


using namespace std;


int arrayop(int[]);
int numop(int);
int refop(int&);


int main() {
 //int array[3] = {1,2,3};
 int x = 3;
        //  int num = arrayop(array);
 //  int num = numop(x);
 int num = refop(x);
 cout << num << endl;
}


int arrayop(int array[]) {
 int ret = array[0] + array[1] + array[2];
 return ret;
}


int numop(int x) {
 int ret = x*3;
 return ret;
}


int refop(int &x) {
 int ret = x*3;
 return ret;
}
```

This is the code I performed my operations on and the assembly code I then studied. Numop deals with passing by value since we are just passing the integer x. Arrayop deals with passing by pointer since passing an array means passing a pointer to the first element. Refop deals with passing by reference.

NumOp:

```
push   ebp
mov    ebp, esp
sub    esp, 16
mov    edx, dword ptr [ebp+8]
mov    eax, edx
add    eax, eax
add    eax, edx
mov    dword ptr [ebp-4], eax
mov    eax, dword ptr[ebp-4]
```

In numop, we see the standard prologues (both for main and for the numop subroutine because main can be a subroutine also, which would mean we still need to save the base pointer). The next step is allocating space on the stack. Even though we will only be pushing one value onto the stack, the assembler will allocate padding on the stack. This is observed by subtracting more than 4 bytes from the stack pointer. we move the value that the address [ebp+8] specifies into edx, and then we move that value into eax. The rest of the code deals with adding the value 3 times, moving it to the local variables, and returning eax. In Numop, we passed-by-value. In doing so, we were dealing with the actual values (into the registers, we moved values that were pointed to by esp and ebp offsets).

```
arrayop:
push   ebp
mov    ebp, esp
sub    esp, 16
mov    eax, dword ptr [ebp+8]
mov    edx, dword ptr [eax]
mov    eax, dword ptr [ebp+8]
add    eax, 4
mov    eax, dword ptr[ eax]
add    edx, eax
mov    eax, dword ptr[ebp+8]
add    eax, 8
mov    eax, dword ptr[eax]
add    eax, edx
mov    dword ptr[ebp-4], eax
mov    eax, dword ptr[ebp-4]
```

In arrayop, we see the standard prologues (both for main and for the arrayop subroutine because main can also be a subroutine, which would mean we still need to save the base pointer). The next step is allocating space on the stack. Even though we will only be pushing one value onto the stack, the assembler will allocate padding on the stack. This is observed by subtracting more than 12 bytes from the stack pointer. Ebp offsets will specify addresses, so we load those addresses into the register eax. Since arrays are just pointers to the first element, the value in eax is an address to the first element, so we then move the value at address eax into register edx. We repeat this process by where the values on the stack are addresses, and we load those addresses into register eax and offset it to get addresses of other elements in the array. We follow the pointer eax to get the actual value in the array, and then add it to the register edx. In this operation, we are passing values by pointer. On the stack, we are pushing addresses, and these addresses

are loaded into register eax. To access the array elements, we access the address specified by eax.

Refop works very similarly to arrayop in that we are passing and accessing addresses.

```
push   ebp
mov    ebp, esp
sub    esp, 16
mov    eax, dword ptr [ebp+8]
mov    edx, dword ptr [eax]
mov    eax, edx
add    eax, edx
add    eax, edx
mov    dword ptr [ebp-4], eax
mov    eax, dword ptr [ebp-4]
```

The code in refop is similar to numop since we are still multiplying an integer x by 3. However, there is an additional move operation. First, we have to move the value stored at address [ebp +8] into eax. The value at eax is an address because we passed a parameter in by reference. To actually obtain a value, we access the value stored at address eax (and store this value in edx). From there we perform the same mathematical operations in numop to multiply by 3

In pass by value, values are added to the stack. Passing by pointer and passing by reference are similar to each other, but different from pass by value. In both cases (pass by pointer and pass by reference), you are pushing an address of a variable on the stack. Pass by reference works by passing in addresses, so we will be accessing address stored as values in the stack, and then accessing the addresses of the values we just obtained. This means we need to follow a pointer specified by an ebp offset and store into a register, and then follow the pointer specified by the register to obtain a value. In the subroutine, we will need to dereference these values on the stack to get the values at the addresses referred to by the stack. Similarly, passing by pointer means we have to follow (dereference) pointers based on the address in the pointer's value. With pass by pointer, we are also pushing addresses onto the stack, so we need to access the values at the addresses on the stack. This makes sense compared to how we learned about parameter passing in C++; we don't need to copy the variables like passing by value (pushing the values onto the stack). Instead in pass by reference and pass by pointer, we put addresses onto the stack (pushing addresses onto the stack.)