1 Parameter Passing

1.1 Passing an int, char

```
int foo(int x) {
    return x+3;
}

int main() {
    foo(4);
    return 0;
}
```

When compiled into assembly, the above code produces:

```
_Z3fooi:
   push  ebp
   mov   ebp, esp
   mov   eax, DWORD PTR [ebp+8]
   add   eax, 3
   pop   ebp
   ret
main:
   push  ebp
   mov   ebp, esp
   push  4
   call  _Z3fooi
   add   esp, 4
   mov   eax, 0
   leave
   ret
```

The caller, main(), pushes the argument onto the stack and then calls the foo subroutine. After the standard prologue, the callee then copies the argument into the eax register. The callee accesses the argument by a constant difference from the ebp pointer.

The same function modified to take a char rather that an int functions in an almost identical way. The only difference is that, because a char is a smaller tpye than an int, precautions are taken to make sure that when the argument is copied into the eax register no extra data is accidentally used.

1.2 Passing int by reference

When modified slightly to take a reference to an int as an argument, the code compiles to:

```
_Z3fooRi:
   push  ebp
   mov   ebp, esp
   mov   eax, DWORD PTR [ebp+8]
   mov   eax, DWORD PTR [eax]
   add   eax, 3
   pop   ebp
   ret
main:
   push  ebp
   mov   ebp, esp
   sub   esp, 20
   mov   DWORD PTR [ebp-4], 4
   lea   eax, [ebp-4]
   mov   DWORD PTR [esp], eax
   call  _Z3fooRi
   mov   eax, 0
   leave
   ret
```

Rather that pushing the number four to the stack to be used directly as an argument like before, it now puts four on the stack and then pushes the address of 4 which is taken as the argument to foo. Then foo dereferences the argument in order to use to value 4. This approach of passing and then dereferencing an address is how passing any data-type by reference works.

1.3 Passing a pointer

```
int foo(int * x) {
    return *x+3;
}

int main() {
    int * c = new int;
    *c = 3;
    foo(c);
    return 0;
}
```

Compiles to:

```
_Z3fooPi:
   push  ebp
   mov   ebp, esp
   mov   eax, DWORD PTR [ebp+8]
   mov   eax, DWORD PTR [eax]
   add   eax, 3
   pop   ebp
   ret
main:
```

```
10    lea   ecx, [esp+4]
11    and   esp, -16
12    push  DWORD PTR [ecx-4]
13    push  ebp
14    mov   ebp, esp
15    push  ecx
16    sub   esp, 20
17    sub   esp, 12
18    push  4
19    call  _Znwj
20    add   esp, 16
21    mov   DWORD PTR [ebp-12], eax
22    mov   eax, DWORD PTR [ebp-12]
23    mov   DWORD PTR [eax], 3
24    sub   esp, 12
25    push  DWORD PTR [ebp-12]
26    call  _Z3fooPi
27    add   esp, 16
28    mov   eax, 0
29    mov   ecx, DWORD PTR [ebp-4]
30    leave
31    lea   esp, [ecx-4]
32    ret
```

Here, the call to _Znwj creates a pointer to the int 4, and returns it in the eax register. In line 21 it takes the value of eax, which is an address that contians the value 4, and stores it in [ebp-12] which is space allocated by main for local variables. Then, [ebp-12] is pushed on the stack so it can be used as an argument when the call to foo() is made in the next line. Within foo, the variable [ebp-12] (an address pointing to the integer 4) is moved to the eax register. Then eax is dereferenced to get the value being pointed to and the function procedes from there on the same as when passing a normal int.

1.4 Float

When the function is modified to take a floating point number, the argument is passed and accessed in the same way as an int. There is additional complexity involved with creating the floating point number but the general procedure of pushing an argument to the stack then accessing it by an offset from [ebp] is unchanged.

1.5 Object

The following code passes a simple user defined object to a function.

```
struct simple {
  int x;
  int y;
} test;

int foo(simple z) {
    return z.x;
}

int main() {
    test.x = 3;
    test.y = 4;
    foo(test);
    return 0;
}
```

It compiles to the following assembly.

```
test:
_Z3foo6simple:
   push ebp
   mov  ebp, esp
   mov  eax, DWORD PTR [ebp+8]
   pop  ebp
   ret
main:
   push ebp
   mov  ebp, esp
   mov  DWORD PTR test, 3
   mov  DWORD PTR test+4, 4
   push DWORD PTR test+4
   push DWORD PTR test
   call _Z3foo6simple
   add  esp, 8
   mov  eax, 0
   leave
   ret
```

As can be seen, the members of the object are pushed to the stack in reverse order and then accessed as normal arguments by an offset from [ebp].

The same procedure is used for passing arrays as arguments. For example, the following c++ code:

```
int foo(int x[]) {
    return x[0];
}

int foo2(int x[]) {
    return x[1];
}

int foo3(int x[]) {
    return x[2];
}

int main() {
    int y[] = {1, 2, 3};
    foo(y);
    foo2(y);
    foo3(y);

    return 0;
}
```

Compiles to:

```
_Z3fooPi:
   push ebp
   mov  ebp, esp
   mov  eax, DWORD PTR [ebp+8]
   mov  eax, DWORD PTR [eax]
   pop  ebp
   ret
_Z4foo2Pi:
   push ebp
   mov  ebp, esp
   mov  eax, DWORD PTR [ebp+8]
   mov  eax, DWORD PTR [eax+4]
   pop  ebp
   ret
_Z4foo3Pi:
   push ebp
   mov  ebp, esp
   mov  eax, DWORD PTR [ebp+8]
   mov  eax, DWORD PTR [eax+8]
   pop  ebp
   ret
main:
   push ebp
   mov  ebp, esp
   sub  esp, 20
   mov  DWORD PTR [ebp-12], 1
   mov  DWORD PTR [ebp-8], 2
   mov  DWORD PTR [ebp-4], 3
   lea  eax, [ebp-12]
   mov  DWORD PTR [esp], eax
   call _Z3fooPi
   lea  eax, [ebp-12]
   mov  DWORD PTR [esp], eax
   call _Z4foo2Pi
   lea  eax, [ebp-12]
   mov  DWORD PTR [esp], eax
```

```
	call _Z4foo3Pi
	mov  eax, 0
	leave
	ret
```

The base of the array is accessed as [ebp+8]. In order to access the second element of the array (i.e. index 1), the base is found and then offset by the size of a single element. So the second element is at [ebp+8], the third is the base offset by two elements and is locate at [ebp+12], and so on.

1.6 Pointers vs. References

Passing values by reference works by pushing an argument to the stack that contains the address of the actual object. The argument is then dereferenced inside the callee to access the object itself. The implementation of pointers and references are identical in assembly.

1.7 Summary

In general there are two ways that arguments are passed to functions. The first, by value, involves simply pushing the argument to the stack. It is then accessed from within the callee by a memory offset from [ebp] calculated based on the size of the argument. The second, by reference, involves storing the argument somewhere in memory ad then pushing the address of its location to the stack before making the subroutine call. Then inside the callee the address of the argument is located by an offset from [ebp]. Then, the address is dereferenced using "[ ]" in order to obtain the actual value of the argument.