

1 Parameter Passing

1.1 Passing an int, char

```
1 int foo(int x) {
2     return x+3;
3 }
4
5 int main() {
6     foo(4);
7     return 0;
8 }
```

When compiled into assembly, the above code produces:

```
1 _Z3fooi:
2     push ebp
3     mov  ebp, esp
4     mov  eax, DWORD PTR [ebp+8]
5     add  eax, 3
6     pop  ebp
7     ret
8 main:
9     push ebp
10    mov  ebp, esp
11    push 4
12    call _Z3fooi
13    add  esp, 4
14    mov  eax, 0
15    leave
16    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 than an `int` functions in an almost identical way. The only difference is that, because a `char` is a smaller type 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:

```
1 _Z3fooRi:
2     push ebp
```

```
3     mov     ebp, esp
4     mov     eax, DWORD PTR [ebp+8]
5     mov     eax, DWORD PTR [eax]
6     add     eax, 3
7     pop     ebp
8     ret
9 main:
10    push    ebp
11    mov     ebp, esp
12    sub     esp, 20
13    mov     DWORD PTR [ebp-4], 4
14    lea     eax, [ebp-4]
15    mov     DWORD PTR [esp], eax
16    call    _Z3fooRi
17    mov     eax, 0
18    leave
19    ret
```

Rather than 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 the 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

```
1 int foo(int * x) {
2     return *x+3;
3 }
4
5 int main() {
6     int * c = new int;
7     *c = 3;
8     foo(c);
9     return 0;
10 }
```

Compiles to:

```
1 _Z3fooPi:
2     push    ebp
3     mov     ebp, esp
4     mov     eax, DWORD PTR [ebp+8]
5     mov     eax, DWORD PTR [eax]
6     add     eax, 3
7     pop     ebp
8     ret
9 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 contains 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 proceeds 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.

```
1 struct simple {
2     int x;
3     int y;
```

```
4 } test;
5
6 int foo(simple z) {
7     return z.x;
8 }
9
10 int main() {
11     test.x = 3;
12     test.y = 4;
13     foo(test);
14     return 0;
15 }
```

It compiles to the following assembly.

```
1 test:
2 _Z3foo6simple:
3     push ebp
4     mov  ebp, esp
5     mov  eax, DWORD PTR [ebp+8]
6     pop  ebp
7     ret
8 main:
9     push ebp
10    mov  ebp, esp
11    mov  DWORD PTR test, 3
12    mov  DWORD PTR test+4, 4
13    push DWORD PTR test+4
14    push DWORD PTR test
15    call _Z3foo6simple
16    add  esp, 8
17    mov  eax, 0
18    leave
19    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:

```
1 int foo(int x[]) {
2     return x[0];
3 }
4
5 int foo2(int x[]) {
6     return x[1];
7 }
8
9 int foo3(int x[]) {
```

```
10     return x[2];
11 }
12
13 int main() {
14     int y[] = {1, 2, 3};
15     foo(y);
16     foo2(y);
17     foo3(y);
18
19     return 0;
20 }
```

Compiles to:

```
1  _Z3fooPi:
2      push ebp
3      mov  ebp, esp
4      mov  eax, DWORD PTR [ebp+8]
5      mov  eax, DWORD PTR [eax]
6      pop  ebp
7      ret
8  _Z4foo2Pi:
9      push ebp
10     mov  ebp, esp
11     mov  eax, DWORD PTR [ebp+8]
12     mov  eax, DWORD PTR [eax+4]
13     pop  ebp
14     ret
15 _Z4foo3Pi:
16     push ebp
17     mov  ebp, esp
18     mov  eax, DWORD PTR [ebp+8]
19     mov  eax, DWORD PTR [eax+8]
20     pop  ebp
21     ret
22 main:
23     push ebp
24     mov  ebp, esp
25     sub  esp, 20
26     mov  DWORD PTR [ebp-12], 1
27     mov  DWORD PTR [ebp-8], 2
28     mov  DWORD PTR [ebp-4], 3
29     lea  eax, [ebp-12]
30     mov  DWORD PTR [esp], eax
31     call _Z3fooPi
32     lea  eax, [ebp-12]
33     mov  DWORD PTR [esp], eax
34     call _Z4foo2Pi
35     lea  eax, [ebp-12]
36     mov  DWORD PTR [esp], eax
```

```
37     call _Z4foo3Pi
38     mov  eax, 0
39     leave
40     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 located 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 and 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.