Parameter Passing

1. Passing an int, char, etc.

```
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.

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.

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.

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.

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].

2. Arrays

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.

3. 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.

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.

Objects

1. Overview

The memeber variables of an object are pushed to the stack in consecutive memory locations. The compiler may insert buffer space between the members in order to make access more efficient but must preserve order within access blocks. Data fields are then accessed like elements of an array, by offsetting from the address of the base of the object.

The following c++ code defines a simple class that contains two integers, a char, a bool, and a user defined struct as member variable. The class has a single public function that changes the value of the first int. The main method then creates an instance of the class and calls the function. It will be used as an example to demonstrate data layout, access of data members, and access of member functions.

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

class Test {
    int x;
    bool b;
    char c;
    simple s;

  public:
    void setX(int a);
    int y;
};

void Test::setX(int a) {
    this->x = a;
}

int main() {
    Test myTest = Test();
    myTest.setX(3);
    myTest.y = 2;
    return 0;
}
```

It compiles to the following.

```
_ZN4Test4setXEi:
   push  ebp
   mov   ebp, esp
   mov   eax, DWORD PTR [ebp+8]
   mov   edx, DWORD PTR [ebp+12]
   mov   DWORD PTR [eax], edx
   pop   ebp
   ret
main:
   push  ebp
   mov   ebp, esp
   sub   esp, 32
   mov   DWORD PTR [ebp-20], 0
   mov   BYTE PTR [ebp-16], 0
   mov   BYTE PTR [ebp-15], 0
   mov   DWORD PTR [ebp-12], 0
   mov   DWORD PTR [ebp-8], 0
   mov   DWORD PTR [ebp-4], 0
   push  3
   lea   eax, [ebp-20]
```

```
21    push eax
22    call _ZN4Test4setXEi
23    add  esp, 8
24    mov  DWORD PTR [ebp-4], 2
25    mov  eax, 0
26    leave
27    ret
```

2. Data Layout Sample

As can be seen above, data is stored in a very familiar way. The data members are moved to the stack in memory locations reserved for local variables so that they can be popped in the same order as they are listed in the class definition. The struct is placed on the stack, just as it would be as an argument, by pushing each of its components in reverse order. In this example [ebp-20] is the address of test.x, [ebp-16] of b, [ebp-15] of c, [ebp-12] of simple.x, [ebp-8] of simple.y, and [ebp-4] of test.y. Notice that although b only requires one byte of memory, the compiler is allowed to leave space between it and test.x. There is no fundamental difference between how the private and public variables are stored in memory.

4. Public Member Function Sample

The member method 'setX(int)' is stored in memory just like a global function. The method needs to know what instance of the class it is being called on. In order to make this possible, a 'this' pointer is passed as an argument. The pointer is the memory address of the memory location where the particular instance is stored. The pointer itself doesn't need to be stored seperately because it is simply the address of the first member of the object. Every time a method is called on a different instance the this pointer is different.

3. Data Access Sample

In the example above, data is accessed in two ways. First, the private variable test.x is accessed through the method 'setX(int).' Second, the public variable test.y is set directly. The second case is the simpler one, in order to set test.y the assembly simply moves the desired value into the memory location allocated for y ([ebp-4] as mentioned previously). This happend in line 24. The first case involves calling a method of the class which is discussed in the previous paragraph. It uses the 'this' pointer to access the location of the class instance. In line 4 it loads this address into the register eax. In line 5 it loads the explicit argument 3 into the register edx. It then moves the value of edx (3) into the location pointed to by eax (test.x.). This requires three steps in order to avoid accessing memory twice in once cycle.

References

The following links were used when learning about their respective topics.

- http://stackoverflow.com/questions/405112/how-are-objects-stored-in-memory-in-c
- http://stackoverflow.com/questions/12378271/what-does-an-object-look-like-in-memory
- http://stackoverflow.com/questions/1632600/memory-layout-c-objects
- http://docs.oracle.com/cd/E19455-01/806-3773/6jct9o0af/index.html
- http://cs.lmu.edu/ ray/notes/nasmexamples/