Ashley Nguyen - apn2my
PostLab08 - postlab8.pdf
3/30/16

Parameter Passing

1. Passing ints by value as a parameter, as per the code to the right, uses caller-saved registers (EDX for passByVal) so that contents of the register are unmanipulated across the subroutine call. EDX is pushed onto the stack by moving the first parameter (DWORD PTR [ebp + 8]) into the EDX register. Passing the parameter to the subroutine first pushes parameters BEFORE the subroutine is called. The first parameter (C++ int a), which is located 8 bytes above EBP and 4 bytes above the return address, is pushed to the bottom of the stack. The second parameter (C++ int b), which is 4 bytes above the first parameter [EBP + 12], is pushed on top of the first parameter and so on. To perform the subtraction, int b is the most recent load to eax and int a is the first load edx. The sub command subtracts eax - edx, subtracting b from a which is a - b as desired in the C++ code. Essentially, the parameters are passed on the activation record in reverse order to account for a variable number of parameters that the record will be able to anticipate by growing the stack downwards and adding the most recent parameters to the bottom of the stack at higher addresses. Passing more than one data member by reference is described in #3 below.

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

using namespace std;

int passByRef( int &a, int &b ){
  return ( a * b );
}

int passByVal( int a, int b ){
  return (a - b);
}

int main(){
  int x = 3;
  int y = 4;

  passByRef(x, y);
  cout << passByRef(x , y) << endl;

  int p = 18;
  int q = 4;

  passByVal(p,q);
  cout << passByVal(p ,q) << endl;
  return 0;
}
```

```
_Z9passByRefRiS_:
.LFB975:
        .cfi_startproc
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR [ebp+8]
        mov     edx, DWORD PTR [eax]
        mov     eax, DWORD PTR [ebp+12]
        mov     eax, DWORD PTR [eax]
        imul    eax, edx
        pop     ebp
        ret
        .cfi_endproc
```

```
_Z9passByValii:
.LFB976:
        .cfi_startproc
        push    ebp
        mov     ebp, esp
        mov     eax, DWORD PTR [ebp+12]
        mov     edx, DWORD PTR [ebp+8]
        sub     edx, eax
        mov     eax, edx
        pop     ebp
        ret
        .cfi_endproc
```

2. For an array of ints, my program looped through the array and placed the value of the average from the previous program added to the index of the array in an array of 10 elements. When the callee is called, the parameters were accessed by first creating the whole array on the stack. To begin accessing the elements, the EBP pointer begins at the first element in the array. Then, lea is called to load a pointer to the array which is then saved in the register which holds the array. The callee accesses the parameters inside the function since the assembler iteratively moves each subsequent value ([EBX + 4], [EBX + 8] for integers) onto EAX in backwards order to be called by LEA (load effective address) to load the array values. LEA creates a pointer to the array that is saved in the array variable (another register). Thus, the array values are pushed onto the stack which the caller can access.

3. The second subroutine I wrote, passByRef performs parameter passing using memory addresses of caller-saved registers rather than values inside it. The first parameter (C++ int &a = DWORD PTR [ebp

+ 8]) is moved to the EAX register. Then the memory address of EAX (since the parameter is a reference to a memory address) is moved to EDX. The second parameter (C++ int &b = DWORD PTR [ebp + 12]) is moved to EAX and again the memory address of EAX is moved into the actual register of EAX. EDX holds the memory address of the first parameter int &a, while EAX holds the memory address of the second parameter int &b. The purpose of this function is to multiply the two parameters, doing so using imul opcode, pushing the result onto EAX register which is returned after restoring the base pointers value.

The main difference between passing by value and passing by reference in assembly is that the caller-saved registers hold the values in pass by value while the caller-saved registers contain memory addresses in pass by reference. This is because the parameters in pass by reference, reference memory addresses that hold the values, not the values themselves. Thus, in assembly, their memory addresses must be used to refer to the values as well. Based on my tester program shown to the right, passing by pointer is identical to pass by reference in that the memory addresses are treated as values in the caller-saved registers as shown in the code on the right (Tutorials Point).

```
_Z8passByPtPiS_:
.LFB977:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        mov     eax, DWORD PTR [ebp+8]
        mov     edx, DWORD PTR [eax]
        mov     eax, DWORD PTR [ebp+12]
        mov     eax, DWORD PTR [eax]
        add     eax, edx
        pop     ebp
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Further investigation:

Char

Refer to the x86 code to the right. When passing in two chars as parameters in the passChar() function, the assembler allocates 8 bytes in the activation record for these parameters. The first passed parameter, char a (DWORD PTR [ebp + 8]) is moved to the caller-saved register EDX. The second passed parameter char b (DWORD PTR [ebp + 12]) is moved to EAX. Then the least significant 8 bits in EDX (dl) are moved to the BYTE PTR [ebp - 4], the properly sized local variable allocated one spot below the base pointer. Similarly, the least significant 8 bits in EAX (al) are moved to the next local variable spot below the first parameter. The order in which the first parameter is moved to edx, and the second parameter is moved to eax is the same when DL and AL are moved. This is so that the activation record still holds the local variables in reverse order. Since char are 1 byte in size, the parameters passed are treated as local variables in least significant 8-bit form to perform the function action, comparing the value of chars (CMP).

```
bool passChar( char a, char b ){
  return (a == b);
}
```

```
_Z8passCharcc:
.LFB976:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        sub     esp, 8
        mov     edx, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [ebp+12]
        mov     BYTE PTR [ebp-4], dl
        mov     BYTE PTR [ebp-8], al
        movzx   eax, BYTE PTR [ebp-4]
        cmp     al, BYTE PTR [ebp-8]
        sete    al
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Objects

1 & 4. Global variables, function and main data members are kept in memory at low memory in the data section of the stack in the virtual address space of a C process (Danielian, S). The data members of a public field allows the members to be accessed outside the class, within the scope of the class object. Private field data members can typically only be accessed by friend function or the class itself. C++ can lump together different fields of an object based on accessibility within the scope of that object. The assembler knows which data member to access based on what was loaded into the register at initialization. If the assembler could not access that section or segment of data then a segmentation fault would occur (Danielian, S). Private and public data fields are accessed nearly the same way in assembly, by reading through the public or private respective fields and creating global functions for them in assembly. This is intriguing because further research would show that memory in public and private fields can be accessed equally in the assembler, whereas C++ displays differentiation in accessing. As for method invocation, the assembler knows which object it is being called out of because the local variables have a unique spot in the stack. Notice for the student_getName method to the right, the assembler subtracts 24 bytes from the stack pointer to create 6 spots in the activation record for the local variables. Each "object" is essentially designated a unique spot below the stack pointer which can be referenced to by incrementing/decrementing the bytes by for to access that certain object.

```
student_getName:
.LFB975:
        .cfi_startproc
        push    ebp
        mov     ebp, esp
        sub     esp, 24
        mov     eax, DWORD PTR [ebp+12]
        mov     DWORD PTR [esp+4], eax
        mov     eax, DWORD PTR [ebp+8]
        mov     DWORD PTR [esp], eax
        call    _ZNSsC1ERKSs
        mov     eax, DWORD PTR [ebp+8]
        leave
        ret     4
        .cfi_endproc
```

2. I created a student class that created a Student object with the parameters of name, id and size (of the student), details on data members are shown in code to the right and below. Since a c string byte size is dependent on the number of letters (chars) in the string plus the 1 byte null pointer at the end of the string, the string's byte size could vary based on the length of the string. However, the assembler assumes the space required for a string is 4 bytes (DWORD PTR usage in getName() above). In total, 4 bytes (string) + 4 bytes (int) + 1 byte (char) = 9 bytes total. The public field methods allocate enough bytes for their respective return types. The compiler interprets the data members in both of the fields then assigns string name for example a hexadecimal address. Since string contains 4 bytes, the next 4 spots are reserved for string name to hold its memory. Next, the compiler will read in an int id, assign it to a spot in memory below string name allocated spots, and allocate the next 4 memory addresses for the int. Finally, the char size will be read in, assigned a spot below int id allocated memory and char's memory will be contained at THAT memory address since char is 1 byte in size (CS Utah).

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

using namespace std;

class student{
public:
  string getName();
  int getId();
  student(string n, int i, char s);

private:
  string name;
  int id;
  char size;
};

string student::getName(){
  return this->name;
}

int student::getId(){
  return this->id;
}
```

```
student::student(string n, int i, char s){
  this->name = n;
  this->id = i;
  this->size = s;
  cout << n << " || " << i << " || " << s << endl;
}

int main(){
  student ash = student( "Ashley", 6281996, 'S' );
  return 0;
}
```

3 & 4. Inside a member function:

The private data member, int id, is accessed from inside a member function getId. As seen in the code snippet to the right, the DWORD PTR [ebp + 8] is where the data member, int id, was pushed onto the stack and where is can be reached by offsetting the base pointer. Thus the variable int id is pushed onto the result stack eax. Since getId reaches the private member contents by "**this->id**", the value is held 4 bytes below the variable (8). This is done by moving DWORD PTR[EAX + 4], to EAX, which fully provides access to int id. The data member is now in the EAX result register. This is similarly done for getName() to access the private data member string name; code is above.

```
_ZN7student5getIdEv:
.LFB976:
        .cfi_startproc
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        mov     eax, DWORD PTR [ebp+8]
        mov     eax, DWORD PTR [eax+4]
        pop     ebp
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
```

Outside a member function:

The main function I created initialized a student object with the appropriately constructed parameters. In assembly, main accesses these parameters/data members by first calling the student function. Once the function is called, the student function accesses the data members string name, int id, and char size through internal member function access (described above). However, I will note that since student accesses the char data member, the assembler accounts for the size of this object by using a BYTE PTR. Investigating this topic further, once main can access the student function by call, the contents are not simply pushed onto the stack but rather loaded by LEA. By looking at the main assembly code to the bottom right, three LEA commands were used to load the effect addresses of the three parameters of the student object. These three parameters are private data members of the student class. This demonstrates that when functions access data members from the outside, the addresses must be loaded onto the caller-saved register (EAX) to preserve the value and memory address for each data member to be accessed by main.

```
_ZN7studentC2ESsic:
.LFB978:
        .cfi_startproc
        .cfi_personality 0,__gxx_personality_v0
        .cfi_lsda 0,.LLSDA978
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        push    ebx
        sub     esp, 20
        .cfi_offset 3, -12
        mov     eax, DWORD PTR [ebp+20]
        mov     BYTE PTR [ebp-12], al
        mov     eax, DWORD PTR [ebp+8]
        mov     DWORD PTR [esp], eax
```

```
main:
.LFB980:
        .cfi_startproc
        .cfi_personality 0,__gxx_personality_v0
        .cfi_lsda 0,.LLSDA980
        push    ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        mov     ebp, esp
        .cfi_def_cfa_register 5
        push    ebx
        and     esp, -16
        sub     esp, 48
        .cfi_offset 3, -12
        lea     eax, [esp+31]
        mov     DWORD PTR [esp], eax
        call    _ZNSaIcEC1Ev
        lea     eax, [esp+31]
        mov     DWORD PTR [esp+8], eax
        mov     DWORD PTR [esp+4], OFFSET FLAT:.LC1
        lea     eax, [esp+32]
        mov     DWORD PTR [esp], eax
.LEHB3:
        call    _ZNSsC1EPKcRKSaIcE
```

Works Cited

- http://www.tutorialspoint.com/cplusplus/cpp_passing_pointers_to_functions.htm
- Bucky - https://www.youtube.com/watch?v=53VYYMy-LBo&list=PLAE85DE8440AA6B83&index=45
- Sergei Danielian - http://www.gahcep.com/cpp-internals-memory-layout/
- http://www.tutorialspoint.com/cplusplus/cpp_classes_objects.htm
- http://www.cs.utah.edu/~germain/PPS/Topics/memory_layout.html