Pat Wongwiset
nw9ca
11/02/17
postlab08.pdf

Parameter Passing

Passing by Value

Passing by integer

```
        .globl  __Z7foo_vali
        .align  4, 0x90
__Z7foo_vali:                           ## @_Z7foo_vali
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp0:
        .cfi_def_cfa_offset 16
Ltmp1:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp2:
        .cfi_def_cfa_register rbp
        mov     eax, 1
        mov     dword ptr [rbp - 4], edi
        pop     rbp
        ret
        .cfi_endproc
```

Passing by Char

```
        .globl  __Z8foo_charc
        .align  4, 0x90
__Z8foo_charc:                          ## @_Z8foo_charc
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp6:
        .cfi_def_cfa_offset 16
Ltmp7:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp8:
        .cfi_def_cfa_register rbp
        mov     al, dil
        mov     edi, 2
        mov     byte ptr [rbp - 1], al
        mov     eax, edi
        pop     rbp
        ret
        .cfi_endproc
```

Passing by float

```
        .globl  __Z9foo_floatf
        .align  4, 0x90
__Z9foo_floatf:                         ## @_Z9foo_floatf
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp12:
        .cfi_def_cfa_offset 16
Ltmp13:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp14:
        .cfi_def_cfa_register rbp
        mov     eax, 4
        movss   dword ptr [rbp - 4], xmm0
        pop     rbp
        ret
        .cfi_endproc
```

Passing by Pointer

```
        .globl  __Z7foo_ptrPi
        .align  4, 0x90
__Z7foo_ptrPi:                          ## @_Z7foo_ptrPi
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp9:
        .cfi_def_cfa_offset 16
Ltmp10:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp11:
        .cfi_def_cfa_register rbp
        xor     eax, eax
        mov     qword ptr [rbp - 8], rdi
        pop     rbp
        ret
        .cfi_endproc
```

Passing by user defined class

```
        .globl  __Z11foo_intChar7intChar
        .align  4, 0x90
__Z11foo_intChar7intChar:               ## @_Z11foo_intChar7intChar
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp27:
        .cfi_def_cfa_offset 16
Ltmp28:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp29:
        .cfi_def_cfa_register rbp
        xor     eax, eax
        mov     qword ptr [rbp - 8], rdi
        pop     rbp
        ret
        .cfi_endproc
```

Passing by Reference

Passing by &integer

```
        .globl  __Z10foo_refIntRi
        .align  4, 0x90
__Z10foo_refIntRi:                      ## @_Z10foo_refIntRi
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp30:
        .cfi_def_cfa_offset 16
Ltmp31:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp32:
        .cfi_def_cfa_register rbp
        xor     eax, eax
        mov     qword ptr [rbp - 8], rdi
        pop     rbp
        ret
        .cfi_endproc
```

Passing by &Char

```
        .globl  __Z11foo_refCharRc
        .align  4, 0x90
__Z11foo_refCharRc:                     ## @_Z11foo_refCharRc
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp30:
        .cfi_def_cfa_offset 16
Ltmp31:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp32:
        .cfi_def_cfa_register rbp
        xor     eax, eax
        mov     qword ptr [rbp - 8], rdi
        pop     rbp
        ret
        .cfi_endproc
```

Passing by &float

```
        .globl  __Z12foo_refFloatPf
        .align  4, 0x90
__Z12foo_refFloatPf:                    ## @_Z12foo_refFloatPf
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp36:
        .cfi_def_cfa_offset 16
Ltmp37:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp38:
        .cfi_def_cfa_register rbp
        xor     eax, eax
        mov     qword ptr [rbp - 8], rdi
        pop     rbp
        ret
        .cfi_endproc
```

Passing by &Pointer

```
        .globl  __Z10foo_refPtrRPi
        .align  4, 0x90
__Z10foo_refPtrRPi:                     ## @_Z10foo_refPtrRPi
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp33:
        .cfi_def_cfa_offset 16
Ltmp34:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp35:
        .cfi_def_cfa_register rbp
        xor     eax, eax
        mov     qword ptr [rbp - 8], rdi
        pop     rbp
        ret
        .cfi_endproc
```

Passing by &(user defined class)

```
        .globl  __Z14foo_refIntCharP7IntChar
        .align  4, 0x90
__Z14foo_refIntCharP7IntChar:           ## @_Z14foo_refIntCharP7IntChar
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp39:
        .cfi_def_cfa_offset 16
Ltmp40:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp41:
        .cfi_def_cfa_register rbp
        xor     eax, eax
        mov     qword ptr [rbp - 8], rdi
        pop     rbp
        ret
        .cfi_endproc
```

Program has a similar pattern when passing by value. The program pushes rbp to continue storing value in rsp. Then, the program would copy (mov) value in first parameter into [rbp - x] in order to plant the integer in subroutine; x = size in bytes of passing parameter. The register in assembly code changes in pattern (rax, eax, al) corresponding to the desired memory it needs to allocate a parameter. Therefore, the [rbp - x] or local variable means the value is located right below rbp which stores rsp.

User Defined Class: The assembly code will write to allocate the passing parameter in the same way as the pointer regardless of the size or types of parameter in the class.

Passing by Reference: All of the snippet code in different types look very similar such that it locates as a local parameter at right below rsp. (-8 in rap -8 means that the memory used to store parameter is 8 bytes. The number of bytes (8) is equal to the memory used to store pointer).

Float: movss means moving a scalar single-precision floating-point value from the source operand (second operand) to the destination operand (first operand). The source and destination operands of the floating number can be XMM registers or 32-bit memory locations.

(http://x86.renejeschke.de/)

Array

C++ Code **Assembly Code**

```
int foo_arr(int x[4]){ // pass by array
  int i = 0;
  int j = 0;
  while(i < 4){
    j += x[i];
    i++;
      }

  return j;
}
```

```
Dump of assembler code for function foo_arr(int*):
=> 0x000000100000c20 <+0>:     push   rbp
   0x000000100000c21 <+1>:     mov    rbp,rsp
   0x000000100000c24 <+4>:     mov    QWORD PTR [rbp-0x8],rdi
   0x000000100000c28 <+8>:     mov    DWORD PTR [rbp-0xc],0x0
   0x000000100000c2f <+15>:    mov    DWORD PTR [rbp-0x10],0x0
   0x000000100000c36 <+22>:    cmp    DWORD PTR [rbp-0xc],0x4
   0x000000100000c3d <+29>:    jge    0x100000c65 <foo_arr(int*)+69>
   0x000000100000c43 <+35>:    movsxd rax,DWORD PTR [rbp-0xc]
   0x000000100000c47 <+39>:    mov    rcx,QWORD PTR [rbp-0x8]
   0x000000100000c4b <+43>:    mov    edx,DWORD PTR [rcx+rax*4]
   0x000000100000c4e <+46>:    add    edx,DWORD PTR [rbp-0x10]
   0x000000100000c51 <+49>:    mov    DWORD PTR [rbp-0x10],edx
   0x000000100000c54 <+52>:    mov    edx,DWORD PTR [rbp-0xc]
   0x000000100000c57 <+55>:    add    edx,0x1
   0x000000100000c5d <+61>:    mov    DWORD PTR [rbp-0xc],edx
   0x000000100000c60 <+64>:    jmp    0x100000c36 <foo_arr(int*)+22>
   0x000000100000c65 <+69>:    mov    eax,DWORD PTR [rbp-0x10]
   0x000000100000c68 <+72>:    pop    rbp
   0x000000100000c69 <+73>:    ret
End of assembler dump.
```

The base address of the array will be located at rcx, and has rdi as a pointer to rcx. When the C++ code increments in the loop, the loop in assembly code will increment the index of the array by +rax*4 as shown on the line which the red arrow points to. rax which stores incrementing number, is multiplied by 4 because the value in the array is 'int'. It would define the next number which is in the next 4 bytes when incrementing i (value of rax). Since the address of each value in array is coded in form of a pointer, the code has to dereference in order to access and operate it.

The loop intends to show how callee accesses parameters and how parameters are passed in the function.

Passing by Pointer VS Passing by Reference

```
        .globl  __Z7foo_ptrPi
        .align  4, 0x90
__Z7foo_ptrPi:                          ## @_Z7foo_ptrPi
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp9:
        .cfi_def_cfa_offset 16
Ltmp10:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp11:
        .cfi_def_cfa_register rbp
        mov     qword ptr [rbp - 8], rdi
        mov     rdi, qword ptr [rbp - 8]
        mov     eax, dword ptr [rdi]
        pop     rbp
        ret
        .cfi_endproc
```

```
        .globl  __Z7foo_refRi
        .align  4, 0x90
__Z7foo_refRi:                          ## @_Z7foo_refRi
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp15:
        .cfi_def_cfa_offset 16
Ltmp16:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp17:
        .cfi_def_cfa_register rbp
        mov     qword ptr [rbp - 8], rdi
        mov     rdi, qword ptr [rbp - 8]
        mov     eax, dword ptr [rdi]
        pop     rbp
        ret
        .cfi_endproc
```

The assembly code looks exactly similar (i.e. parameter passing, dereference to return) when passing by pointer and reference, so there is no difference between passing by pointer and passing by reference in assembly code. When passing by reference, the parameter which stores value will be allocated the same way as the pointer. The memory address of parameter will be stored as a local variable in the stack at [rbp -8] which is the position right after rsp.

Objects

Data Layout:

```
diver::diver(){
  yes = true;
  j = 0;
  intch = NULL;
  text = '?';
  fl = 0;
}
```

```
        .globl  __ZN5diverC2Ev
        .align  4, 0x90
__ZN5diverC2Ev:                         ## @_ZN5diverC2Ev
Lfunc_begin2:
        .loc    6 42 0                  ## postlab.cpp:42:0
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp11:
        .cfi_def_cfa_offset 16
Ltmp12:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp13:
        .cfi_def_cfa_register rbp
        xorps   xmm0, xmm0
        mov     qword ptr [rbp - 8], rdi
        mov     rdi, qword ptr [rbp - 8]
        .loc    6 43 7 prologue_end     ## postlab.cpp:43:7
Ltmp14:
        mov     byte ptr [rdi], 1
        .loc    6 44 5                  ## postlab.cpp:44:5
        mov     dword ptr [rdi + 16], 0
        .loc    6 45 9                  ## postlab.cpp:45:9
        mov     qword ptr [rdi + 8], 0
        .loc    6 46 8                  ## postlab.cpp:46:8
        mov     byte ptr [rdi + 20], 63
        .loc    6 47 6                  ## postlab.cpp:47:6
        movss   dword ptr [rdi + 24], xmm0
Ltmp15:
        .loc    6 48 1                  ## postlab.cpp:48:1
        pop     rbp
        ret
```

The assembly does not have ‘pre-definition’ (as *.h file does in C++), but the value will be kept in memory when parameters are allocated as shown in the picture. The constructor initializes all parameters that are defined in the class. The first parameter will be put in the address, which register rdi points to. Regardless of size of fields, the next parameter will continue storing by incrementing pointer (+8) and so on for the next parameter. Different types can be stored “together” in one class such that a class acts as a base pointer which each field has its specific “pointer” by incrementing from the base pointer in order to define its position when it was called.

In other words, the pointer can be observed as padding of data alignment such that the program will put extra unused memory in order to optimize the performance of the program. Since the data in C++ is aligned naturally (think as an array/table), the more members in one class with different types can cause the program run slower if each member compacts exactly to its size. However, padding can cause problem in term of lots of memory usage.

.loc is Compiler-Use-Only Directives.

.loc file_name line_number optional_column

Data Member Access:

```
int diver:: plus(){
  return intch->i + j;
}
```

```
        .globl  __ZN5diver4plusEv
        .align  4, 0x90
__ZN5diver4plusEv:                      ## @_ZN5diver4plusEv
Lfunc_begin4:
        .loc    6 50 0                  ## postlab.cpp:50:0
        .cfi_startproc
## BB#0:
        push    rbp
Ltmp22:
        .cfi_def_cfa_offset 16
Ltmp23:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp24:
        .cfi_def_cfa_register rbp
        mov     qword ptr [rbp - 8], rdi
        mov     rdi, qword ptr [rbp - 8]
        .loc    6 51 10 prologue_end    ## postlab.cpp:51:10
Ltmp25:
        mov     rax, qword ptr [rdi + 8]
        .loc    6 51 17 is_stmt 0       ## postlab.cpp:51:17
        mov     ecx, dword ptr [rax]
        .loc    6 51 19                 ## postlab.cpp:51:19
        add     ecx, dword ptr [rdi + 16]
        .loc    6 51 3                  ## postlab.cpp:51:3
        mov     eax, ecx
        pop     rbp
        ret
Ltmp26:
Lfunc_end4:
        .cfi_endproc
```

When program calls public member without function, it will go to find the data from from member alignment. Otherwise, if the program accesses member via member function, the program will pass the local variable which defines that class into the callee. The private data member cannot be accessed when calling outside of the member function in C++. (It causes compile error). However, the assembly code does not differentiate between private or public member, so the program can access private member in the same way as public member. The data member will be accessed by specific pointer that explained in the data layout. For example, intch uses qword ptr[rdi +8] (local var 1)to indicate its address while j used qword ptr[rdi + 16] (local var 2).

Method Invocation:

```
int main(){
  intChar x;
  intChar y;
  x.setInt(2,3);
  int k = x.plus();
}
```

```
Lfunc_begin0:
        .loc    6 72 0 is_stmt 1        ## postlab.cpp:72:0
        .cfi_startproc
## %bb.0:
        push    rbp
Ltmp42:
        .cfi_def_cfa_offset 16
Ltmp43:
        .cfi_offset rbp, -16
        mov     rbp, rsp
Ltmp44:
        .cfi_def_cfa_register rbp
        sub     rsp, 48
        lea     rdi, [rbp - 16]
Ltmp45:
        ##DEBUG_VALUE: main:x <- RDI
        .loc    6 73 11 prologue_end    ## postlab.cpp:73:11
        call    __ZN7intCharC1Ev
        lea     rdi, [rbp - 32]
Ltmp46:
        ##DEBUG_VALUE: main:y <- RDI
        .loc    6 74 11                 ## postlab.cpp:74:11
        call    __ZN7intCharC1Ev
        lea     rdi, [rbp - 16]
Ltmp47:
        mov     esi, 2
        mov     edx, 3
        .loc    6 75 3                  ## postlab.cpp:75:3
        call    __ZN7intChar6setIntEii
        lea     rdi, [rbp - 16]
        .loc    6 76 11                 ## postlab.cpp:76:11
        call    __ZN7intChar4plusEv
        xor     edx, edx
        .loc    6 76 7 is_stmt 0        ## postlab.cpp:76:7
        mov     dword ptr [rbp - 36], eax
        .loc    6 77 1 is_stmt 1        ## postlab.cpp:77:1
        mov     eax, edx
        add     rsp, 48
        pop     rbp
        ret
```

Since the main method set the offset in order to create local variables in the function. The address at rbp - 16 and at rbp - 32 are used to allocate x and y respectively after rsp created 48-byte space in the prologue. [rbp -b], is used to reserved space for the local variables such that b is the number of bytes to store itself or its pointer. Thus, method invocation can call different objects from the pointer [rbp -b] that is specified by the caller. For example, x is a local variable in main() and will be called and stored at rbp - 16, so the setInt method will copy the memory address at rbp-16 to rdi before operating in the subroutine.

Note: The user-defined class has 16-byte offset because the program creates padding to optimize itself.

Public Function Access

```
.loc    17 86 0 is_stmt 1       ## postlab.cpp:86:0
mov     rdi, qword ptr [rbp - 32]
mov     qword ptr [rbp - 104], rdi
mov     edx, dword ptr [rbp - 24]
mov     dword ptr [rbp - 96], edx
.loc    17 86 10 is_stmt 0      ## postlab.cpp:86:10
mov     rdi, qword ptr [rbp - 48]
mov     qword ptr [rbp - 120], rdi
mov     edx, dword ptr [rbp - 40]
mov     dword ptr [rbp - 112], edx
.loc    17 86 3                 ## postlab.cpp:86:3
mov     rdi, qword ptr [rbp - 104]
mov     qword ptr [rbp - 136], rdi
mov     edx, dword ptr [rbp - 96]
mov     dword ptr [rbp - 128], edx
mov     rdi, qword ptr [rbp - 136]
mov     al, byte ptr [rbp - 128]
mov     r8, qword ptr [rbp - 120]
mov     qword ptr [rbp - 152], r8
mov     edx, dword ptr [rbp - 112]
mov     dword ptr [rbp - 144], edx
mov     rdx, qword ptr [rbp - 152]
movzx   esi, al
movzx   ecx, byte ptr [rbp - 144]
call    __Z4join7intCharS_
```

Public function can access member in the class from stack pointer register which will point to member (local variable) exactly without pointing to the class variable. It also shows that the data does not stores in between the alignment when the class variables are initialized since the local variable of members in the class are not in between rbp - 16 or rbp - 32 as shown from the assembly code above.

data alignment and structure padding
-http://www.geeksforgeeks.org/structure-member-alignment-padding-and-data-packing/
-https://software.intel.com/en-us/articles/coding-for-performance-data-alignment-and-structures
*(.loc)*http://ftp.icm.edu.pl/packages/linux-uk/alpha/alpha/asm6.html