The examples and discussion in the following slides have been adapted from a variety of sources, including:

Chapter 3 of Computer Systems 3rd Edition by Bryant and O'Hallaron
x86 Assembly/GAS Syntax on WikiBooks
(http://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax)
Using Assembly Language in Linux by Phillip ??
(http://asm.sourceforge.net/articles/linasm.html)

The C code was compiled to assembly with gcc version 4.8.3 on CentOS 7.
Unless noted otherwise, the assembly code was generated using the following command line:

```
gcc -S -m64 -fno-asynchronous-unwind-tables -mno-red-zone -O0 file.c
```

AT&T assembly syntax is used, rather than Intel syntax, since that is what the gcc tools use.
C Example

int main() {

    int x = 42, y = 99;

    int* p1 = &x;    // p1 stores address of variable x
    int* p2 = &y;    // p2 stores address of variable y

    int** p3 = &p2;   // p3 stores address of variable p2

    int aa = *p1;     // aa stores value of the target of p1, 42
    *p1 = 10;         // the target of p1, which is x, stores 10

    int bb = **p3;    // bb stores value of the target of the
    // target of p3; p3 points to p1 and
    // p1 points to x, so bb gets value 99

    return 0;
}

main:

```
movl    $42, -28(%%rbp)
movl    $99, -32(%%rbp)
leaq   -28(%%rbp), %rax
movq   %rax, -8(%%rbp)
leaq   -32(%%rbp), %rax
movq   %rax, -40(%%rbp)
leaq   -40(%%rbp), %rax
movq   %rax, -16(%%rbp)
movq   -8(%%rbp), %rax
movl   (%rax), %eax
movl   %eax, -20(%%rbp)
movq   -8(%%rbp), %rax
movl   $10, (%rax)
movq   -16(%%rbp), %rax
movq   (%rax), %rax
movl   (%rax), %eax
movl   %eax, -24(%%rbp)
```
main:

# x = 42; y = 99;
movl $42, -28(\%rbp)
movl $99, -32(\%rbp)

the Stack
The code snippet demonstrates how to use pointers in x86-64 assembly. Here's a breakdown of the code:

```assembly
main:
  ; # p1 = &x
  leaq  -28(%rbp), %rax    # rax = rbp - 28 = &x
  movq  %rax, -8(%rbp)     # p1 = *(rbp - 8) = rax

  ; # p2 = &y
  leaq  -32(%rbp), %rax
  movq  %rax, -40(%rbp)
```

**The Stack**

<table>
<thead>
<tr>
<th>rbp</th>
<th>old rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp - 8</td>
<td>p1: rbp - 28</td>
</tr>
<tr>
<td>rbp - 16</td>
<td>p3</td>
</tr>
<tr>
<td>rbp - 20</td>
<td>aa</td>
</tr>
<tr>
<td>rbp - 24</td>
<td>aa</td>
</tr>
<tr>
<td>rbp - 28</td>
<td>x: 42</td>
</tr>
<tr>
<td>rbp - 32</td>
<td>y: 99</td>
</tr>
<tr>
<td>rbp - 40</td>
<td>p2: rbp - 32</td>
</tr>
</tbody>
</table>

In the code:
- `&x` is the address of `x`: `rbp - 28`
- `&y` is the address of `y`: `rbp - 32`
Pointers in x86-64

main:

# p3 = &p2
leaq -40(%rbp), %rax  # rax = rbp - 28 = &p2
movq %rax, -16(%rbp)   # p3 = *(rbp - 16) = eax

&p2 the address of p2: rbp - 40

the Stack

<table>
<thead>
<tr>
<th>rbp</th>
<th>old rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp - 8</td>
<td>p1: rbp - 28</td>
</tr>
<tr>
<td>rbp - 16</td>
<td>p3: rbp - 40</td>
</tr>
<tr>
<td>rbp - 20</td>
<td>aa</td>
</tr>
<tr>
<td>rbp - 24</td>
<td>bb</td>
</tr>
<tr>
<td>rbp - 28</td>
<td>x: 42</td>
</tr>
<tr>
<td>rbp - 32</td>
<td>y: 99</td>
</tr>
<tr>
<td>rbp - 40</td>
<td>p2: rbp - 32</td>
</tr>
</tbody>
</table>
main:
    # aa = *p1
    movq -8(%rbp), %rax  # rax = *(rbp - 8) = p1
    movl ( %rax ), %eax  # rax = *rax = *(*(rbp - 8) = *p1
    movl %eax, -20(%rbp)  # aa = *(rbp - 20) = eax

the Stack

<table>
<thead>
<tr>
<th>rbp</th>
<th>old rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp - 8</td>
<td>p1: rbp - 28</td>
</tr>
<tr>
<td>rbp - 16</td>
<td>p3: rbp - 40</td>
</tr>
<tr>
<td>rbp - 20</td>
<td>aa: 42</td>
</tr>
<tr>
<td>rbp - 24</td>
<td>bb</td>
</tr>
<tr>
<td>rbp - 28</td>
<td>x: 42</td>
</tr>
<tr>
<td>rbp - 32</td>
<td>y: 99</td>
</tr>
<tr>
<td>rbp - 40</td>
<td>p2: rbp - 32</td>
</tr>
</tbody>
</table>
main:

# *p1 = 10
movq -8(%rbp), %rax  # rax = *(rbp - 4) = p1
movl $10, (%rax)     # *p1 = *rax = 10

the Stack

<table>
<thead>
<tr>
<th>rbp</th>
<th>old rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp</td>
<td>p1: rbp - 28</td>
</tr>
<tr>
<td>rbp - 8</td>
<td></td>
</tr>
<tr>
<td>rbp - 16</td>
<td>p3: rbp - 40</td>
</tr>
<tr>
<td>rbp - 20</td>
<td></td>
</tr>
<tr>
<td>rbp - 24</td>
<td>aa: 42</td>
</tr>
<tr>
<td>rbp - 28</td>
<td></td>
</tr>
<tr>
<td>rbp - 32</td>
<td>bb</td>
</tr>
<tr>
<td>rbp - 40</td>
<td></td>
</tr>
<tr>
<td>rbp - 40</td>
<td>x: 10</td>
</tr>
<tr>
<td>rbp - 40</td>
<td>y: 99</td>
</tr>
<tr>
<td>rbp - 40</td>
<td>p2: rbp - 32</td>
</tr>
</tbody>
</table>
main:

# bb = **p3
movq -16(%rbp), %rax  # rax = *(rbp - 16) = p3
movq (%rax), %rax     # rax = *rax = *(*(rbp - 16)) = *p3
movl (%rax), %eax     # rax = *(*(rbp - 16) ) = y = **p3
movl %eax, -24(%rbp)  # bb = **p3

the Stack

<table>
<thead>
<tr>
<th>rbp</th>
<th>old rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>rbp</td>
<td></td>
</tr>
<tr>
<td>rbp - 8</td>
<td>p1: rbp - 28</td>
</tr>
<tr>
<td>rbp - 16</td>
<td>p3: rbp - 40</td>
</tr>
<tr>
<td>rbp - 20</td>
<td>aa: 42</td>
</tr>
<tr>
<td>rbp - 24</td>
<td>bb: 99</td>
</tr>
<tr>
<td>rbp - 28</td>
<td>x: 10</td>
</tr>
<tr>
<td>rbp - 32</td>
<td>y: 99</td>
</tr>
<tr>
<td>rbp - 40</td>
<td>p2: rbp - 32</td>
</tr>
</tbody>
</table>
# C and x86-64 Array Example

```c
uint32_t ArrayExample(int * array, uint32_t size)
{
    if (array == NULL) return 0;

    for (uint32_t x = 0; x < size; x++)
    {
        array[x] = rand() % 1024;
    }

    return size;
}
```
ArrayExample:

```
  cmpq  $0, -40(%%rbp)
jne   .L2
movl  $0, %eax
jmp   .L3
.L2:
  movl  $0, -20(%%rbp)
jmp   .L4
.L4:
  movl  -20(%%rbp), %eax
cmpl  -44(%%rbp), %eax
jb     .L5
movl  -44(%%rbp), %eax
.L3:
```

```
  rbp     old rbp
  rbp - 4      ...
  rbp - 20     loop variable x
  rbp - 40     ...
  rbp - 44     array

the Stack
```

The Stack

```
  rbp     old rbp
  rbp - 4      ...
  rbp - 20     loop variable x
  rbp - 40     ...
  rbp - 44     array
```

```
the Stack
```

```
  rbp     old rbp
  rbp - 4      ...
  rbp - 20     loop variable x
  rbp - 40     ...
  rbp - 44     array
```

```
the Stack
```

```
  rbp     old rbp
  rbp - 4      ...
  rbp - 20     loop variable x
  rbp - 40     ...
  rbp - 44     array
```

```
the Stack
```

```
  rbp     old rbp
  rbp - 4      ...
  rbp - 20     loop variable x
  rbp - 40     ...
  rbp - 44     array
```
ArrayExample:

# Test if int* array (i.e. %rbp - 40) is NULL.
cmpq $0, -40(%rbp)

# If array is not NULL jump to the body of the function.
jne .L2

# We return 0 if array == NULL, so we put 0 in %eax then
# jump to the very end of the function.
movl $0, %eax
jmp .L3
.L4:

# After the body, do the loop test

# Put the loop variable in %eax
movl  -20(%rbp), %eax

# The loop test, compare x and size.
# The condition in the C code was x < size

cmpl   -44(%rbp), %eax

# Performs the operation x - size, discarding the result.

rbp   old rbp
------  ------
rbp - 4 ...          
rbp - 20 loop variable x
...    ...
rbp - 40 ...    array
rbp - 44 ...    size
# The function body, after the if statement...

.L2:

```
movl $0, -20(%rbp)    # initialize loop variable x
jmp .L4              # jump to the loop test.
```

.L5:

```
movl -20(%rbp), %eax # get x for this iteration
leaq 0(,%rax,4), %rdx # rdx = 4*rax,
            # i.e. x*sizeof(uint32_t)
movq -40(%rbp), %rax # get the pointer, array
leaq (%rdx,%rax), %rbx # add array + x*sizeof(uint32_t)
            # giving us a pointer to the
            # current element in the array

# snip. . . %eax stores the result of rand() % 1024

movl %eax, (%rbx)    # Remember: rbx is a pointer
            # to the current array element.
            # i.e. array + x*sizeof(uint32_t)
            # So we are storing the random
            # number into array index x

addl $1, -20(%rbp)   # update the loop variable x
```

.L4

...
Aside: leal

You also noticed another use of the `leal` instruction:

```assembly
leal 0(,%rax,4), %rdx  # rdx = 4*rax
   # i.e. x*sizeof(uint32_t)
```

The particular form of the instruction used here is:

```assembly
leal Imm1(src1, src2, Imm2), dst
   dst = Imm2*src2 + src1 + Imm1
```

The execution of the instruction offers some additional performance advantages and is often used when indexing arrays, both for the “sizeof” calculation and to compute the pointer to the corresponding element.
#include <stdint.h>

int main() {
    uint32_t X = 100;
    uint32_t Y = 200;

    Swap(&X, &Y);

    return 0;
}

void Swap(uint32_t* A, uint32_t* B) {
    uint32_t Temp = *A; // Temp = 100
    *A = *B; // X = 200
    *B = Temp; // Y = 100
}

The *pass-by-pointer* protocol provides a called function with the ability to modify the value of the caller's variable.
x86-64 Assembly View

```assembly
Swap:
pushq %rbp
movq %rsp, %rbp
subq $32, %rsp
movq %rdi, -24(%rbp)
movq %rsi, -32(%rbp)
movq -24(%rbp), %rax
movl (%rax), %eax
movl %eax, -4(%rbp)
movq -32(%rbp), %rax
movl (%rax), %edx
movq -24(%rbp), %rax
movl %edx, (%rax)
movq -32(%rbp), %rax
movl -4(%rbp), %edx
movl %edx, (%rax)
leave
ret
```

(Pointers in x86-64 17)