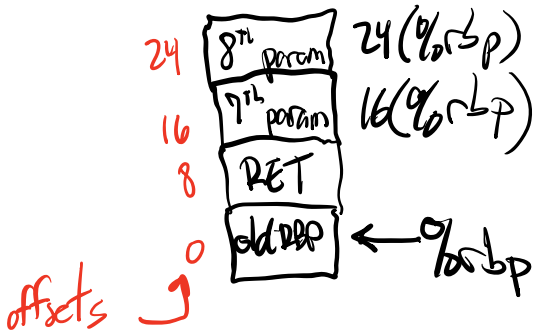
# The stack

## MacOS + Linux




## Windows

Calling convection says that an extra 32 bytes must be allocated on the stack before a function call.

- after any parameters are pushed

```
push %rsi
push %r8
subq $32, %rsp
call foo
addq $32, %rsp
add $16, %rsp
```

pushing parameters (6th & 5th params)

← required by Microsoft

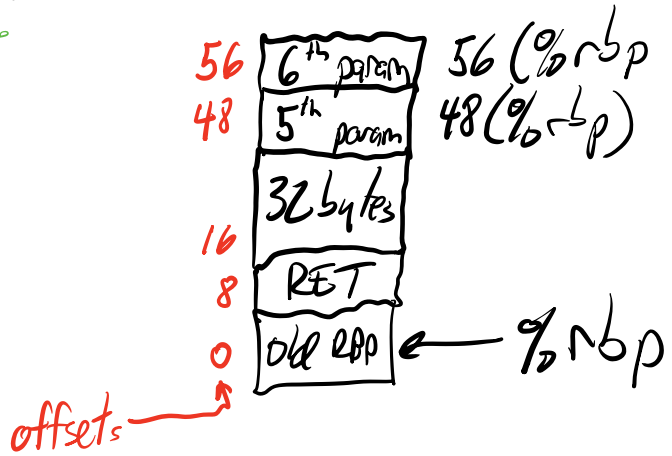
← function call

← remove the 32 bytes

← remove the parameters

Can be combined as addq $48, %rsp

Within the function, the parameter offsets will be different than in macOS:




only in Windows

A register I haven't mentioned yet:

%rip - instruction pointer (aka "program counter")

The %rip register always points to the next instruction in memory to execute.

- after each mov, add, etc. instruction (not a jump or call), %rip is automatically incremented by the size of the instruction.
  - so that it points to the next instruction in memory.

- for a jump, conditional jump, or call instruction, the address of the target instruction is written into %rip.
    - for a conditional jump, this only happens if the condition is satisfied.
- for a return instruction, "ret", the return address is popped off the stack and written to %rip

---

## Allocating local variables on the stack:

C:
```
void f()
{ int x = 10;
  int y = 15;
  :
}
```

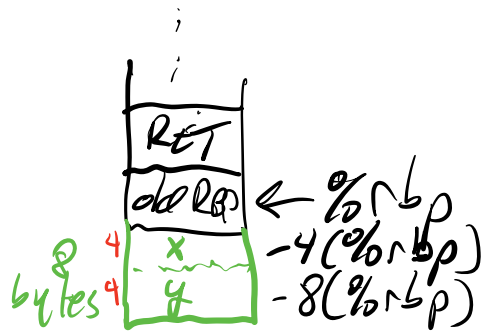
Assembly:

Local variables are allocated below where %rbp points:

```
_f:
        pushq   %rbp
        movq    %rsp, %rbp
        subq    $8, %rsp
        movl    $10, -4(%rbp)
        movl    $15, -8(%rbp)
        :
        addq    $8, %rsp
        popq    %rbp
        ret
```

allocates space for x and y.

remove the space for the local variables



**Important:** If you are calling a C library function (printf, malloc, etc.), keep %rsp aligned to 16-byte boundaries

- subtract/add from %rsp in multiples of 16.
- this includes when pushing. For example, if you only need to save one register (say, %rcx) before calling printf, do:

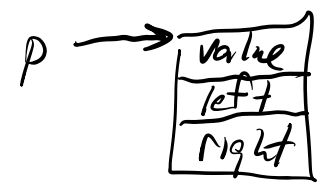
decrements %rsp by a total of 16

```
push %rcx      # pushes 8 bytes
subq $8, %rsp  # adjust %rsp by 8 more.
:
call printf
:
addq $8, %rsp  # adjust stack back
pop %rcx       # restore %rcx
```

- This is only if you are calling a C function (anywhere in your program)

# Revisting structs in C

```
typedef struct node {
    int value;
    struct node *left;
    struct node *right;
} NODE;
```




How do you figure out the offset from the start of the struct for each field?

- it's up to the C compiler.

(1) Write some code, compile it into assembly ("gcc -S ..."), and read the assembly code to figure it out:

```
int main()
{ NODE *p = malloc(sizeof(NODE));
    int x = p->value;
    NODE *q = p->left;
      q = p->right;
}
```

Could compile this into assembly and then read the assembly code to see what the offsets are.

(2) Easier solution: Take addresses of the fields of the NODE p points to and subtract the value of p from them:

```
int main()
{ NODE *p = malloc(sizeof(NODE));
  printf("offset of p->value = %ld\n",
      (void *) &(p->value) - (void *)p );
  printf("offset of p->left = %ld\n",
      (void *) &(p->left) - (void *)p );
```

to stop the compiler from complaining

---

An important instruction:

lea - "load effective address"

- analogous to the "address of" operator in C (&).

Suppose a local variable is at -24(%rbp)

- its address is %rbp - 24 obviously!

lea -24(%rbp), %rax

- just puts %rbp - 24 into %rax

Exactly equivalent to:

```
mov %rbp, %rax
sub $24, %rax
```

You can use lea to take the address of an array element: &(a[i])

lea (%rcx, %r8, 4), %rbx

%rcx + (%r8 * 4)

lea just computes addresses, it doesn't access memory!

```
mov (%rcx, %r8, 4), %rbx
```

this computes the address and fetches the content at that address in memory.

Suppose, in assembly, I want call a C function that takes a pointer as a parameter.

```
void increment(int *p)
{
  (*p)++;
}
```

How do we call increment from assembly, passing the address of a local variable?

```
-g:
        push %rbp
        mov %rsp, %rbp
        #allocate space for a local variable
        subq $16, %rsp  #multiple of 16
                #4-byte local variable will
                #be at -4(%rbp)
        mov $27, -4(%rbp)  # x=27
            #want to call increment(&x)
        leq -4(%rbp), %rdi  #parameter
                            # = &x
        call _increment
            #x would have the value 28
```