

# **Lecture 5. Using stack to allocate subroutine parameters and local variables. Recursive functions.**

Dmitry Zaitsev

<http://daze.ho.ua>

# Compile C code into assembler

- `gcc -S prog.c`
- consider code in
- `prog.s`
- gcc can process assembler files as well
- `gcc -o prog prog.s`

# Calling function in C

- load up to 6 parameters into registers:
- `%rdi, %rsi, %rdx, %rcx, %r8, %r9`
- push other parameters into stack
- call function
- take result from `%rax`

# Write a function in assembler

- as far as your function does not call other functions, especially from libc or written in C, use only agreement on passage of parameters trying to preserve values of registers you use
- compile function separately
- `gcc -c my_fun.s`
- link it
- `gcc -o my_app my_app.c my_fun.o`

# Example – function $x^y$

```
.globl zpow
zpow: mov $1, %rax
      mov %esi, %ecx
      inc %ecx
l:     loop next
      ret
next:  imul %rdi
      jmp l
```

- Compile:  
gcc -c zpow.s
- or  
as -o zpow.o zpow.s

# Example – call zpow function from C

```
#include <stdio.h>
#include <stdlib.h>
extern long zpow(int x,int y);
int main(int argc, char *argv[])
{
    int a=atoi(argv[1]);
    int p=atoi(argv[2]);
    long w=zpow(a,p);
    printf("%ld\n",w);
}
```

- Compile:  
gcc -c mzpow.c
- Link:  
gcc -o mzpow mzpow.o zpow.o
- Run:  
./mpow 2 9  
512
- Or compile and link  
gcc -o mzpow mzpow.c zpow.o  
gcc -o mzpow mzpow.c zpow.s

```

.globl      dyck
dyck:      pushw $'*'      # stack bottom symbol
loop:      movb (%rdi),%al
           cmpb $0, %al   # end of string
           je chk
           cmpb $('(', %al
           jne cls
           pushw %ax      # push '(' into stack
           jmp nxt
cls:       popw %ax        # pop '(' from stack when read ')'
           cmpb $('(', %al
           jne ff         # early stack bottom
nxt:       inc %rdi
           jmp loop
chk:       popw %ax
           cmpb $'*',%al
           jne fp
           movq $1, %rax  # stack bottom - succes
           jmp p
fp:        popw %ax       # clear stack
           cmpb $'*',%al
           jne fp
ff:        xor %rax, %rax
p:         ret

```

# Example – function dyck – parenthesis language recognition

**Returns 1 on success and 0 otherwise**

# Example – call dyck function from C

```
#include <stdio.h>
#include <stdlib.h>

extern int dyck(char *s);

int main(int argc, char *argv[])
{
    printf("%d\n", dyck(argv[1]));
}
```

- Compile:  
gcc -c mdyck.c
- Link:  
gcc -o mdyck mdyck.o dyck.o
- Run:  
./mdyck '(()(())())'  
1
- Or compile and link  
gcc -o mdyck mdyck.c dyck.o  
gcc -o mzpov mdyck.c dyck.s

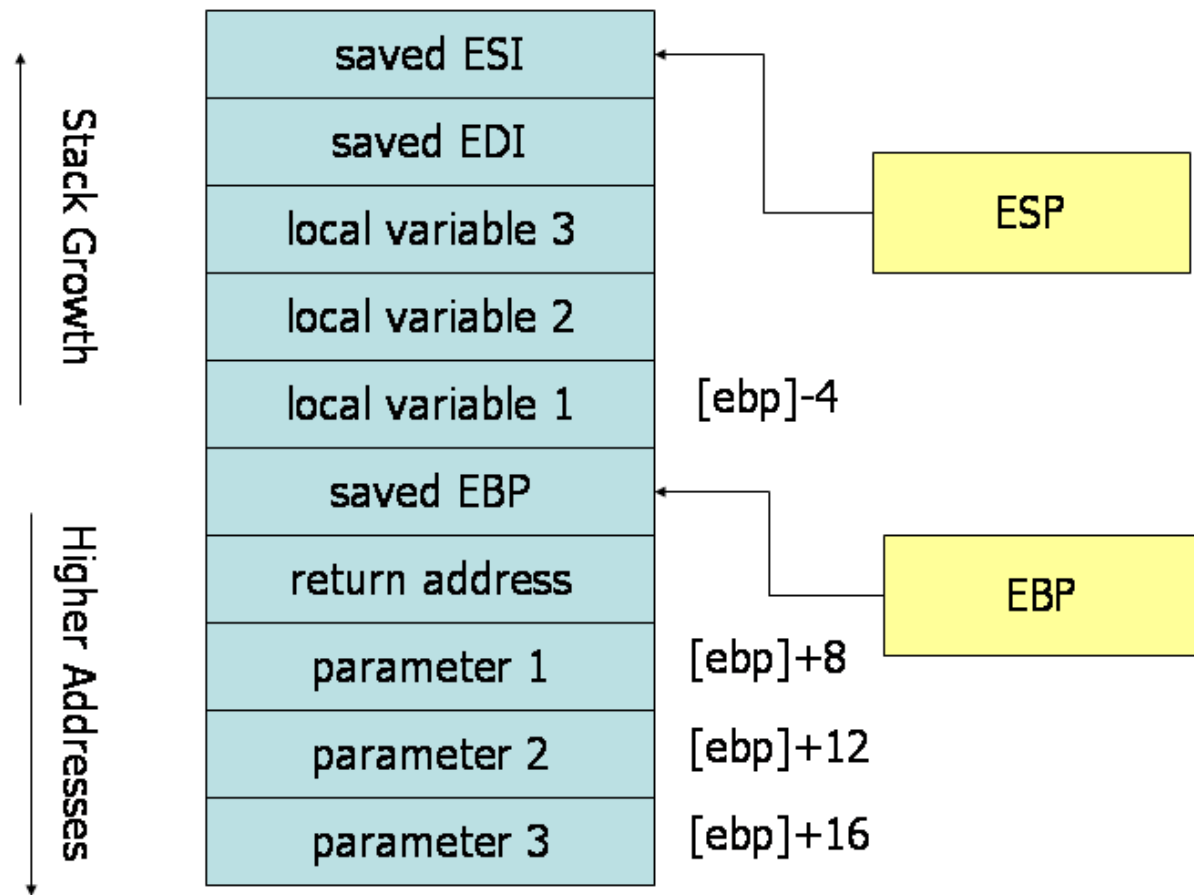


# Allocate local variables

- `pushq %rbp` # store %rbp of caller
- `movq %rsp, %rbp` # set stack bottom
- `subq $16, %rsp` # space for local variables
- ...
- `addq $16, %rsp` # de-allocate local variables
- `popq %rbp` # restore %rbp of caller
- `ret`

leave

# Using stack



# Addressing variables

- Local variables:  
-4(%rbp), -8(%rbp), ...
- Save call parameters on top of local variables
- Save register you use in stack
- Address extra call parameters in stack of caller  
8(%rbp), 12(%rbp), ...

# Passing 16 parameters example

```
main:  pushq    %rbp                                movl    $6, %r9d
        movq    %rsp, %rbp                        movl    $5, %r8d
        pushq   $16                               movl    $4, %ecx
        pushq   $15                               movl    $3, %edx
        pushq   $14                               movl    $2, %esi
        pushq   $13                               movl    $1, %edi
        pushq   $12                               # sum up numbers
        pushq   $11                               call     sumo
        pushq   $10                               addq     $80, %rsp
        pushq   $9                                leave
        pushq   $8                                ret
        pushq   $7
```

# Get 16 parameters

sumo:      endbr64

```
pushq    %rbp
movq     %rsp, %rbp
movl     %edi, -20(%rbp)
movl     %esi, -24(%rbp)
movl     %edx, -28(%rbp)
movl     %ecx, -32(%rbp)
movl     %r8d, -36(%rbp)
movl     %r9d, -40(%rbp)
movl     -20(%rbp), %edx
movl     -24(%rbp), %eax
addl     %eax, %edx
movl     -28(%rbp), %eax
addl     %eax, %edx
movl     -36(%rbp), %eax
addl     %eax, %edx
movl     -40(%rbp), %eax
addl     %eax, %edx
```

```
movl     16(%rbp), %eax
addl     %eax, %edx
movl     24(%rbp), %eax
addl     %eax, %edx
movl     32(%rbp), %eax
addl     %eax, %edx
movl     40(%rbp), %eax
addl     %eax, %edx
movl     48(%rbp), %eax
addl     %eax, %edx
movl     56(%rbp), %eax
addl     %eax, %edx
movl     64(%rbp), %eax
addl     %eax, %edx
movl     72(%rbp), %eax
addl     %eax, %edx
movl     80(%rbp), %eax
addl     %eax, %edx
movl     88(%rbp), %eax
addl     %edx, %eax
```

```
#include <stdio.h>
#include <stdlib.h>
```

```
long fact(int x)
{
    if(x==0) return 1;
    else return x*fact(x-1);
}
```

```
int main(int argc, char *argv[])
{
    int n=atoi(argv[1]);
    long f=fact(n);
    printf("%ld\n",f);
}
```

## Recursive functions

Allocation of local variable  
and call parameters in stack  
(for each call) allows a  
function to call itself

```

fact:      endbr64
           pushq   %rbp
           movq    %rsp, %rbp
           pushq   %rbx
           subq    $24, %rsp
           movl    %edi, -20(%rbp)
           cmpl    $0, -20(%rbp)
           jne     .L2
           movl    $1, %eax
           jmp     .L3
.L2:       movl    -20(%rbp), %eax
           movslq   %eax, %rbx
           movl    -20(%rbp), %eax
           subl    $1, %eax
           movl    %eax, %edi
           call    fact
           imulq   %rbx, %rax
.L3:       addq    $24, %rsp
           popq    %rbx
           popq    %rbp
           ret

```

## Function fact

- save %rbx in stack
- allocate 24 bytes in stack
- save call parameter from %edi into -20(%rbp)

# Code optimization

- -O0. (default). No optimization is performed. Each source code command relates directly to the corresponding instructions in the executable file. This gives the clearest view for source level debugging.
- -O1. Most common forms of optimization that requires no size versus speed decisions, including function inlining.
- -O2. Additional optimizations, such as instruction scheduling.
- -O3. Additional optimizations, such as aggressive function inlining and can therefore increase the speed at the expense of image size.