CS 4910: Intro to Computer Security

Software Security II: function stack

Instructor: Xi Tan

Objectives

- Understand how stack works in Linux x86/64

C/C++ Function in x86

What information do we need to call a function at runtime? Where are they stored?

- Code
- Parameters
- Return value
- Global variables
- Local variables
- Temporary variables
- Return address
- *Function frame pointer*
- *Previous function Frame pointer*

Global and Local Variables in C/C++

Variables that are declared inside a function or block are called **local variables**. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own.

Global variables are defined outside a function. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

In the definition of function parameters which are called **formal parameters**. Formal parameters are similar to local variables.

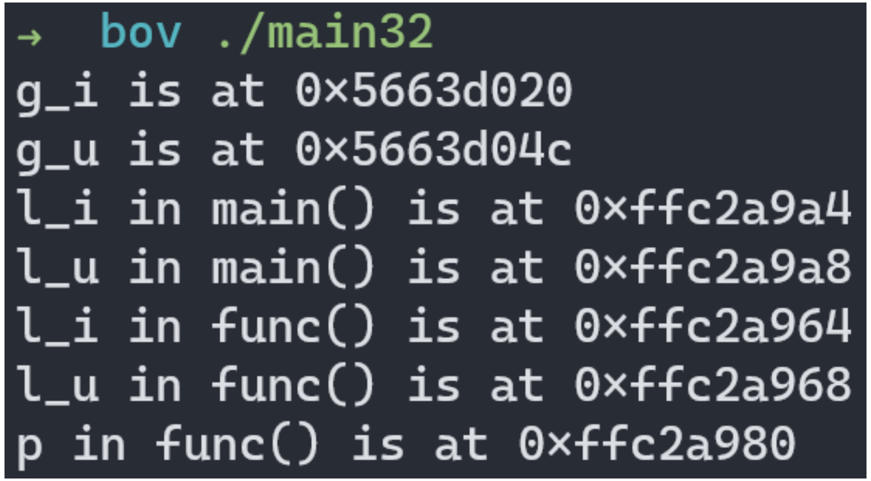
Global and Local Variables (code/globallocalv)

```
char g_i[] = "I am an initialized global variable\n";
char* g_u;

int func(int p)
{
  int l_i = 10;
  int l_u;

  printf("l_i in func() is at %p\n", &l_i);
  printf("l_u in func() is at %p\n", &l_u);
  printf("p in func() is at %p\n", &p);
  return 0;
}
```

```
int main(int argc, char *argv[])
{
  int l_i = 10;
  int l_u;

  printf("g_i is at %p\n", &g_i);
  printf("g_u is at %p\n", &g_u);

  printf("l_i in main() is at %p\n", &l_i);
  printf("l_u in main() is at %p\n", &l_u);

  func(10);
}
```

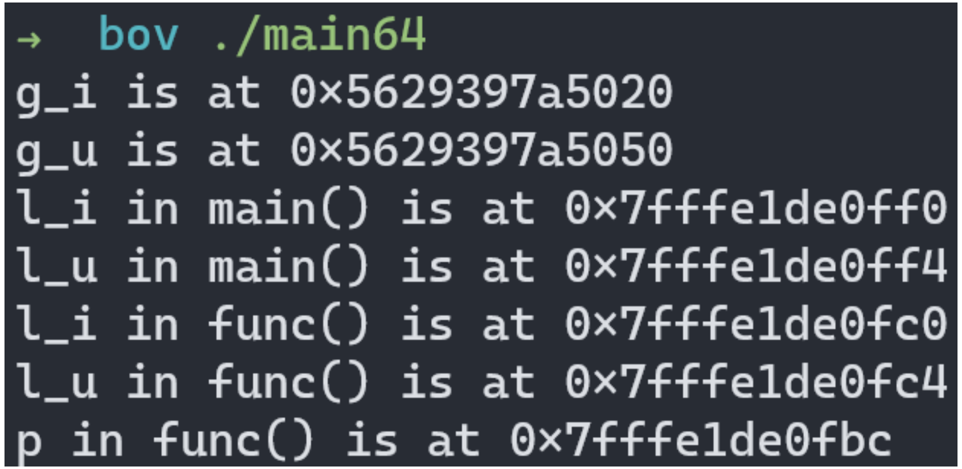
Tools: readelf; nm

Global and Local Variables (code/globallocalv 32bit)



```
→  bov ./main32
g_i is at 0x5663d020
g_u is at 0x5663d04c
l_i in main() is at 0xffc2a9a4
l_u in main() is at 0xffc2a9a8
l_i in func() is at 0xffc2a964
l_u in func() is at 0xffc2a968
p in func() is at 0xffc2a980
```

Global and Local Variables (code/globallocalv 64bit)



```
→  bov ./main64
g_i is at 0x5629397a5020
g_u is at 0x5629397a5050
l_i in main() is at 0x7fffe1de0ff0
l_u in main() is at 0x7fffe1de0ff4
l_i in func() is at 0x7fffe1de0fc0
l_u in func() is at 0x7fffe1de0fc4
p in func() is at 0x7fffe1de0fbc
```

C/C++ Function in x86/64

What information do we need to call a function at runtime? Where are they stored?

- Code **[.text]**
- Parameters **[mainly stack (32bit); registers + stack (64bit)]**
- Return value **[eax, rax]**
- Global variables **[.bss, .data]**
- Local variables **[stack; registers]**
- Temporary variables **[stack; registers]**
- Return address **[stack]**
- Function frame pointer **[ebp, rbp]**
- Previous function Frame pointer **[stack]**

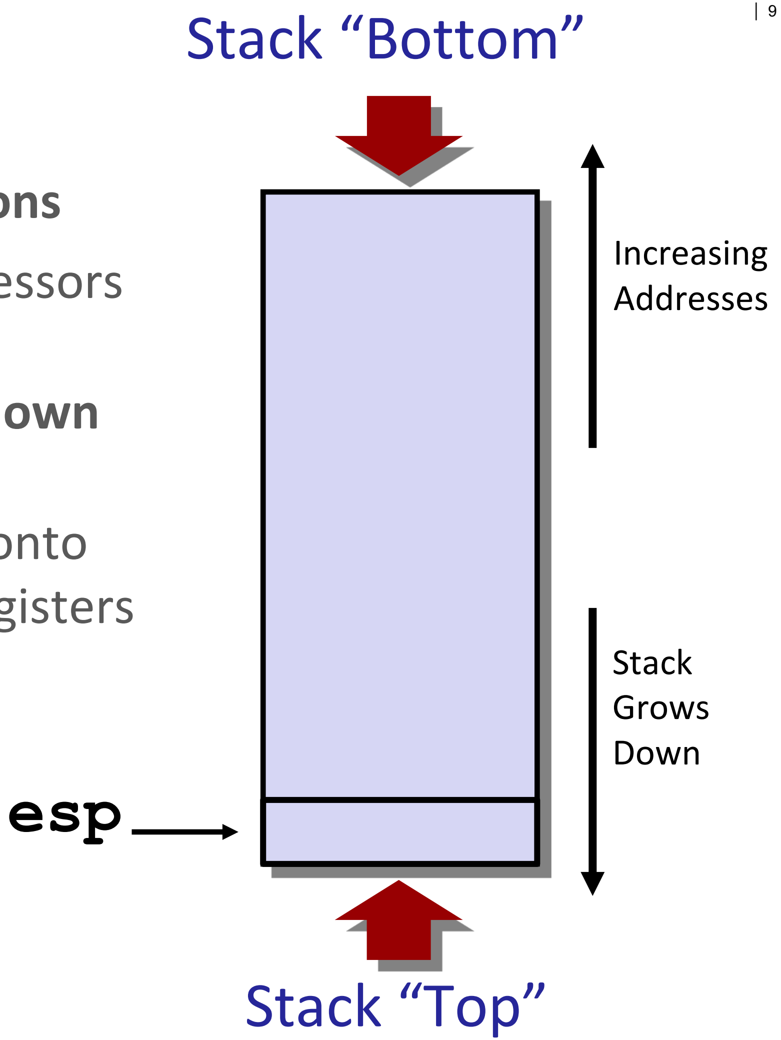
Stack

Stack is essentially **scratch memory for functions**

- Used in MIPS, ARM, x86, and x86-64 processors

Starts at **high** memory addresses, and **grows down**

Functions are free to push registers or values onto the stack, or pop values from the stack into registers



Stack

The assembly language supports this on x86

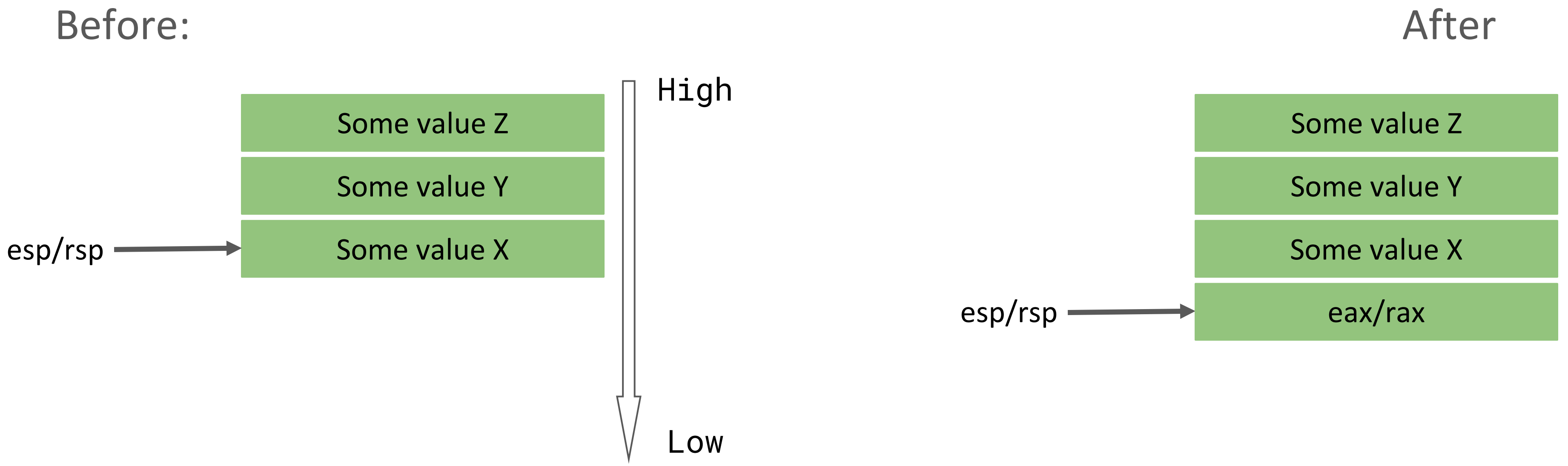
- **esp/rsp** holds the address of the top of the stack
- push eax/rax
  - decrements the stack pointer (esp/rbp) then
  - stores the value in eax/rax to the location pointed to by the stack pointer
- pop eax/rax
  - stores the value at the location pointed to by the stack pointer into eax/rax,
  - increments the stack pointer (esp/rsp)

x86/64 Instructions that affect Stack

push, pop, call, ret, enter, leave

x86/64 Instructions that affect Stack

push eax/rax

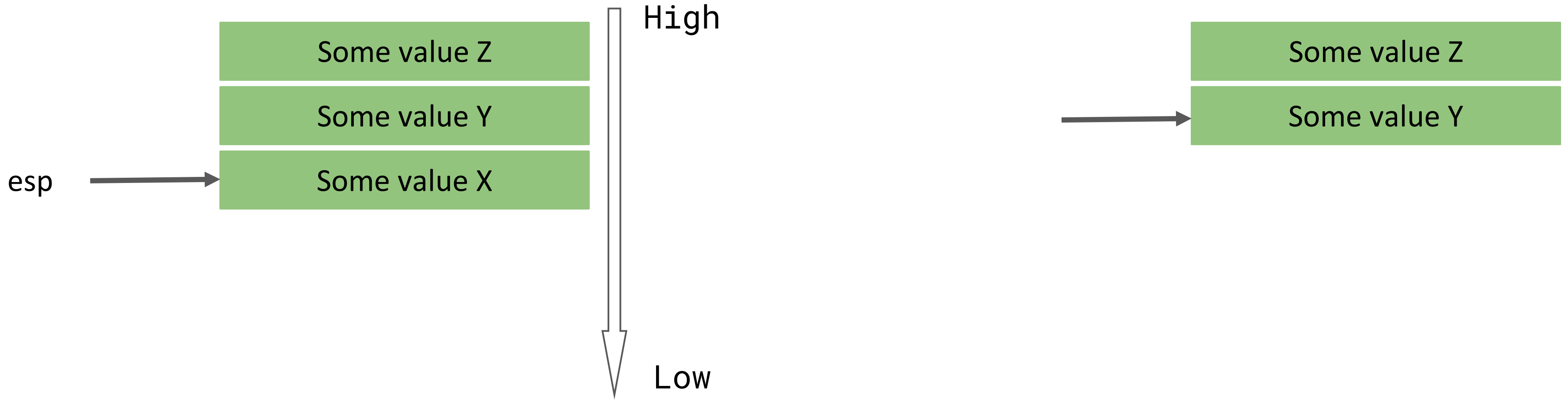


x86 Instructions that affect Stack

pop eax

Before:

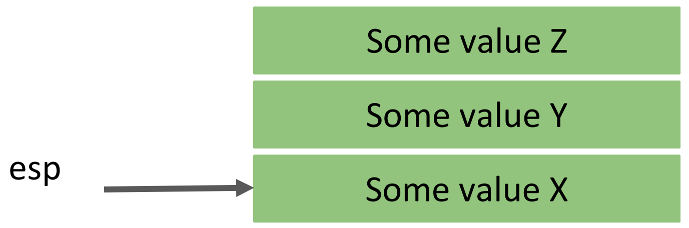
After: eax = X



x86 Instructions that affect Stack

call eax

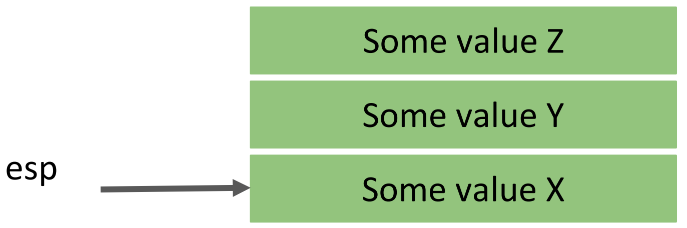
Before:

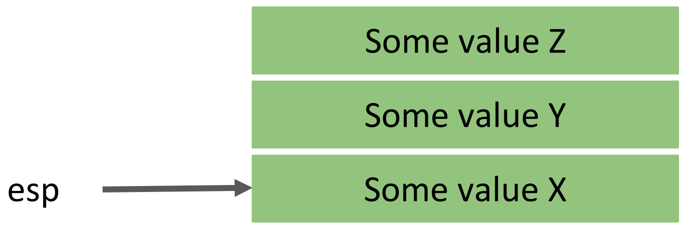


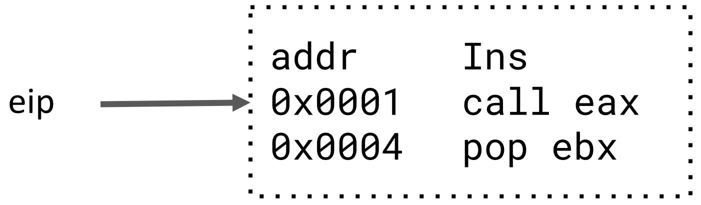
x86 Instructions that affect Stack

call eax

Before:



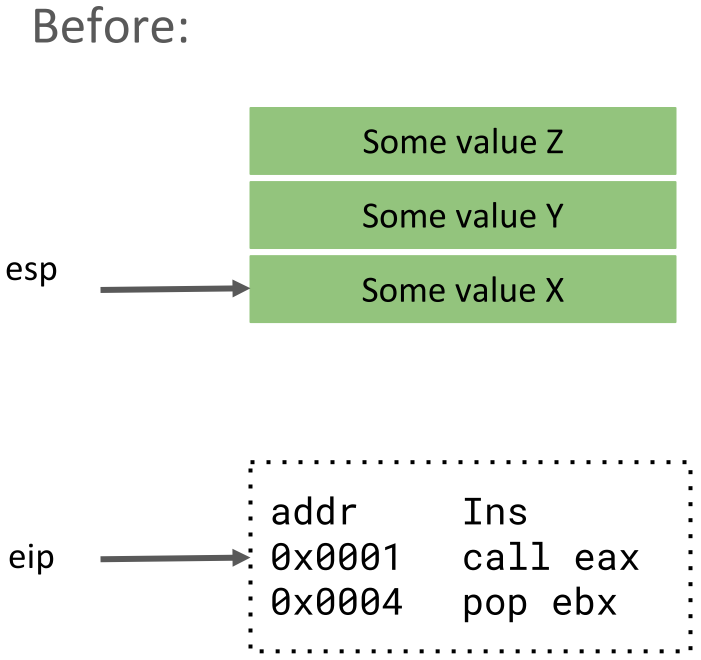




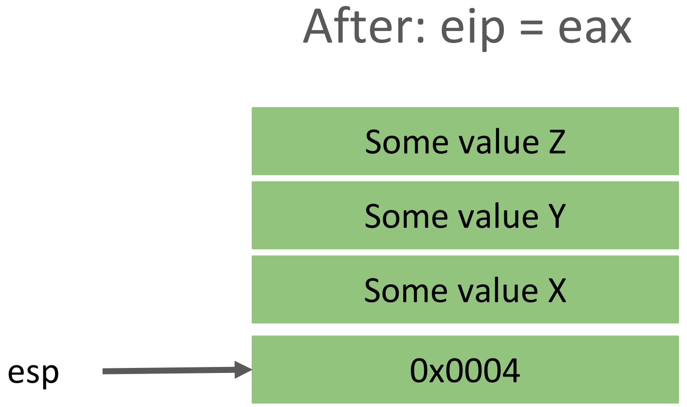
```
addr      Ins
0x0001    call eax
0x0004    pop ebx
```

x86 Instructions that affect Stack

call eax



```
addr     Ins
0x0001   call eax
0x0004   pop ebx
```



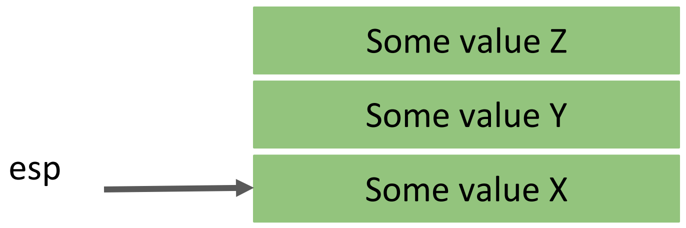
The call instruction does two things:

1. Push the address of next instruction to the stack
2. Move the dest address to %eip

x86 Instructions that affect Stack

ret

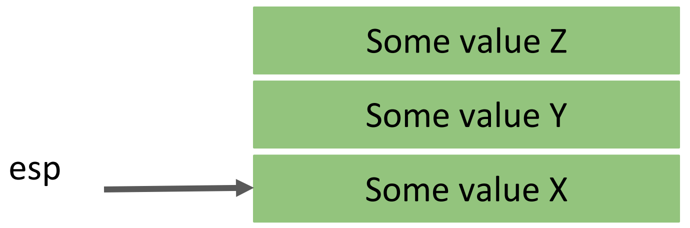
Before:

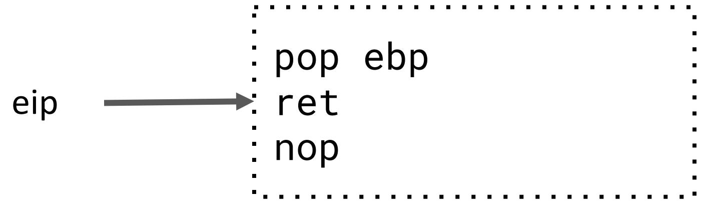


x86 Instructions that affect Stack

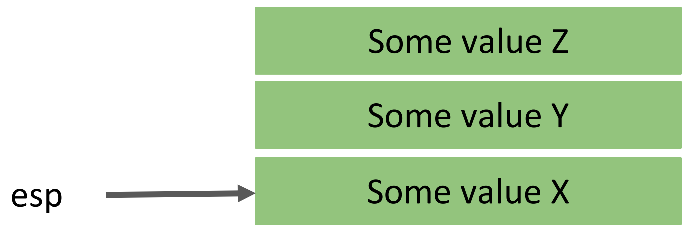
ret

Before:





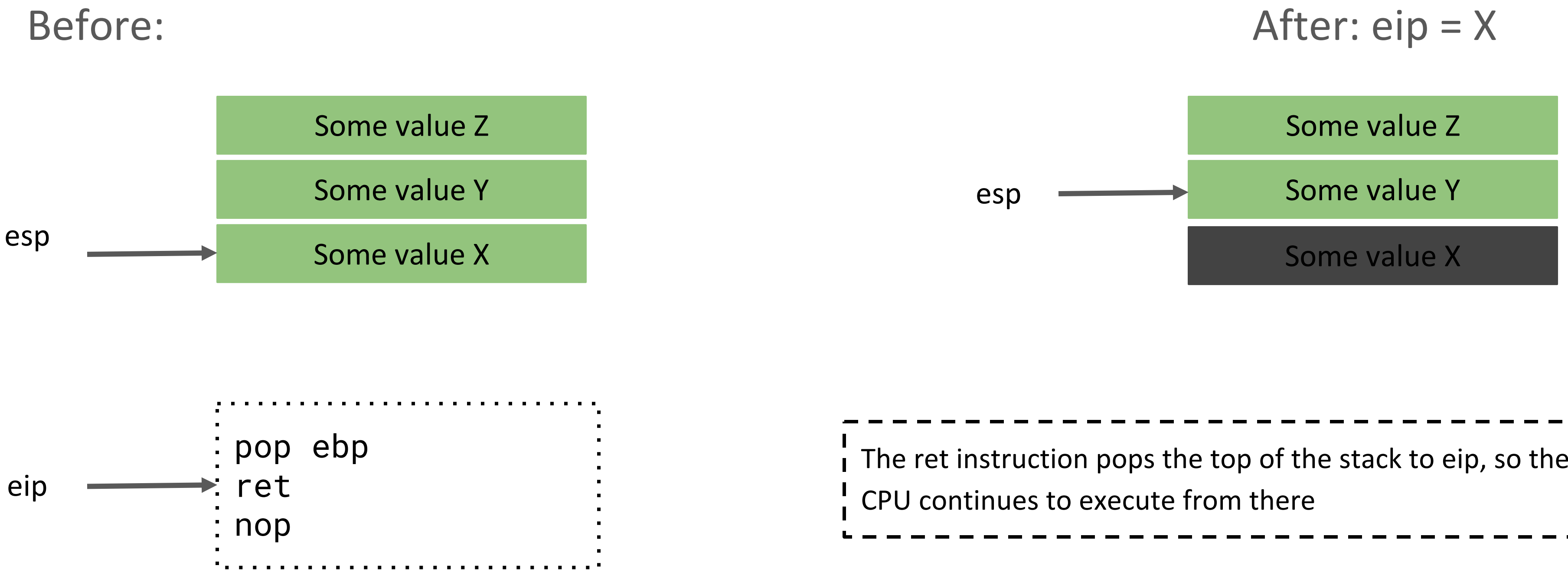
```
pop  ebp
ret
nop
```



The ret instruction pops the top of the stack to eip, so the CPU continues to execute from there

x86 Instructions that affect Stack

ret



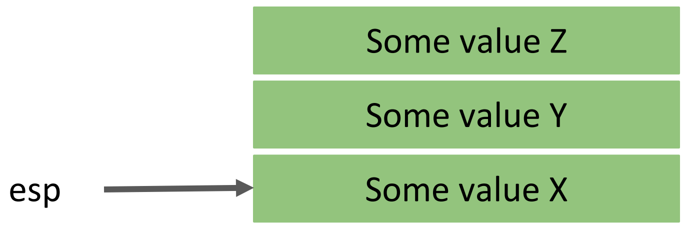
```
pop  ebp
ret
nop
```

The ret instruction pops the top of the stack to eip, so the CPU continues to execute from there

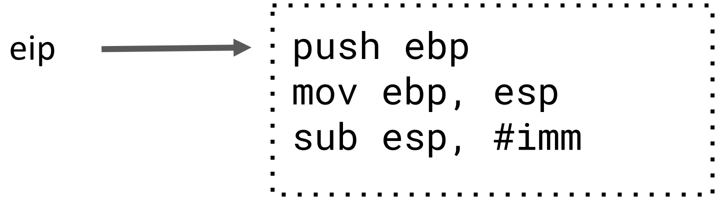
x86 Instructions that affect Stack

Before:

enter



x86 Instructions that affect Stack

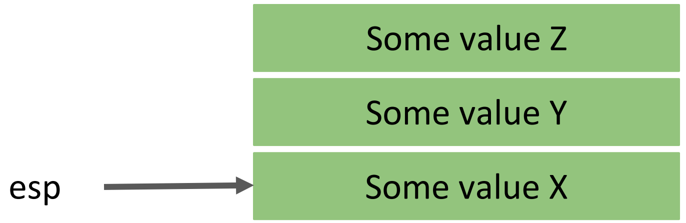


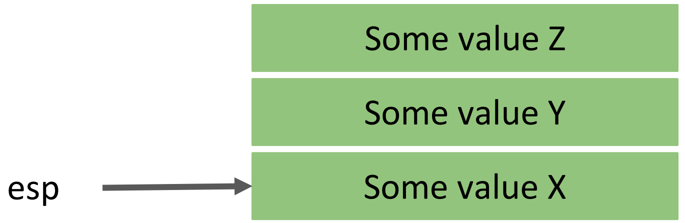
```
push ebp
mov ebp, esp
sub esp, #imm
```

Before:

enter

After:





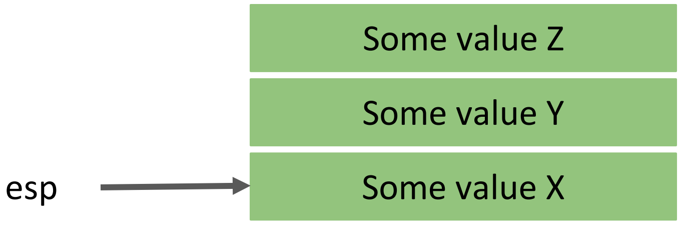
x86 Instructions that affect Stack

eip →

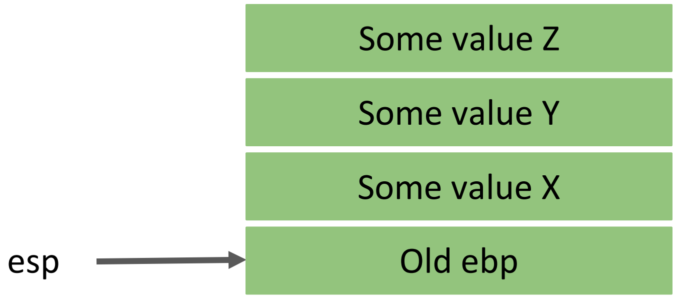
```
push ebp
mov ebp, esp
sub esp, #imm
```

enter

Before:

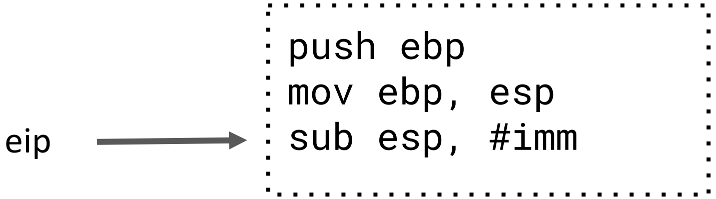


After:



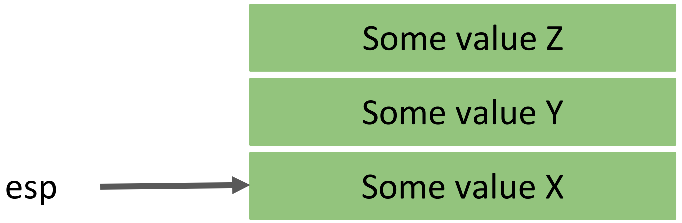
x86 Instructions that affect Stack

```
push ebp
mov ebp, esp
sub esp, #imm
```

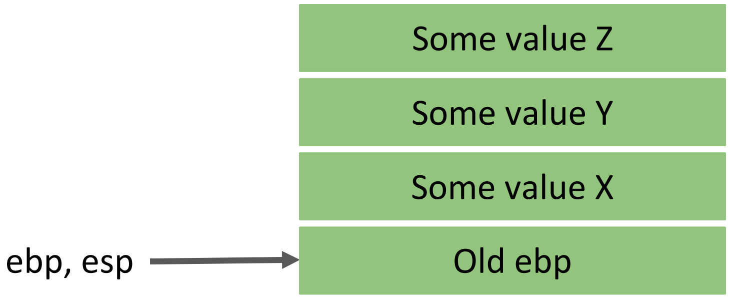


enter

Before:



After:

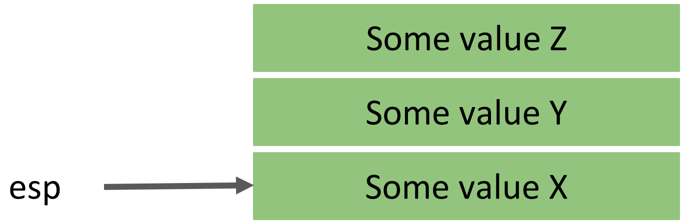


x86 Instructions that affect Stack

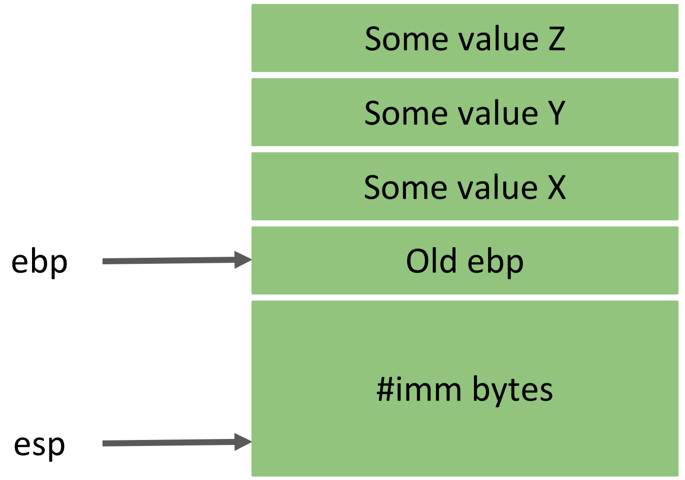
```
push ebp
mov ebp, esp
sub esp, #imm
```

enter

Before:



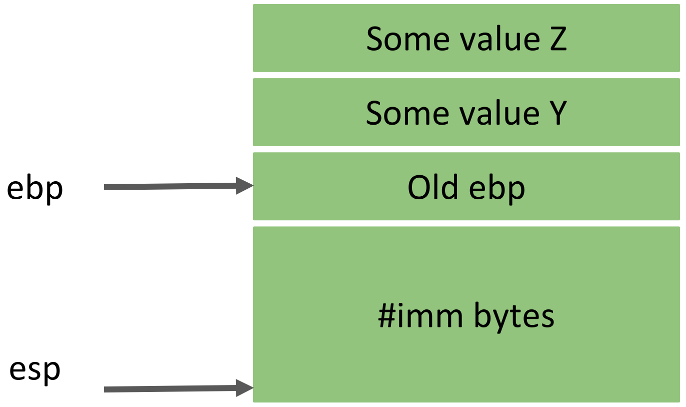
After:



x86 Instructions that affect Stack

Before:

leave



x86 Instructions that affect Stack

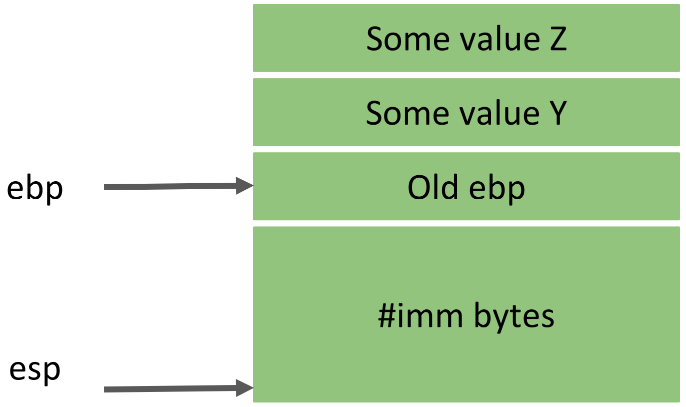
eip →

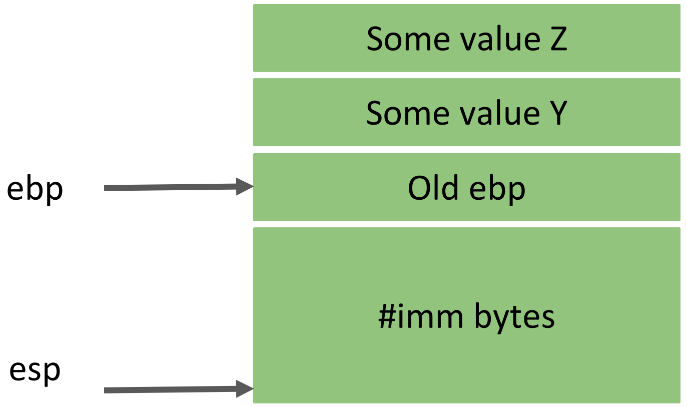
```
mov esp, ebp
pop ebp
```

Before:

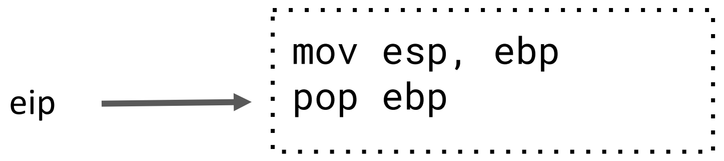
leave

After





x86 Instructions that affect Stack

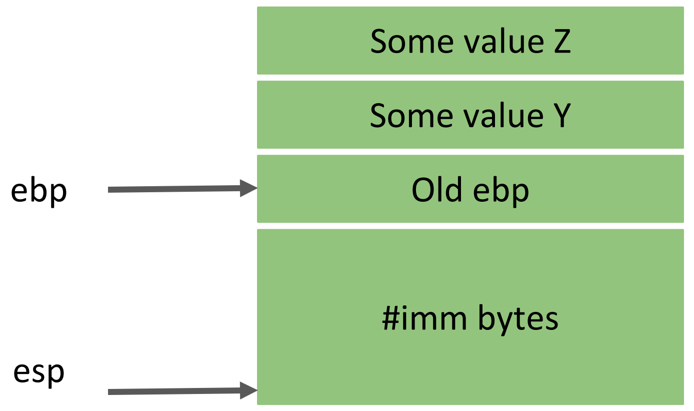


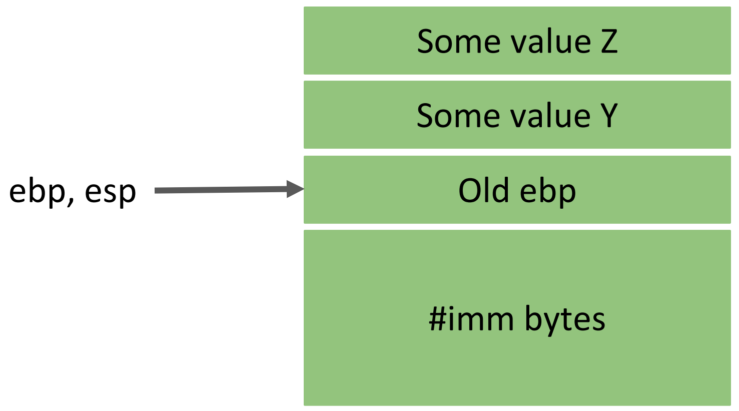
```
mov esp, ebp
pop ebp
```

Before:

leave

After





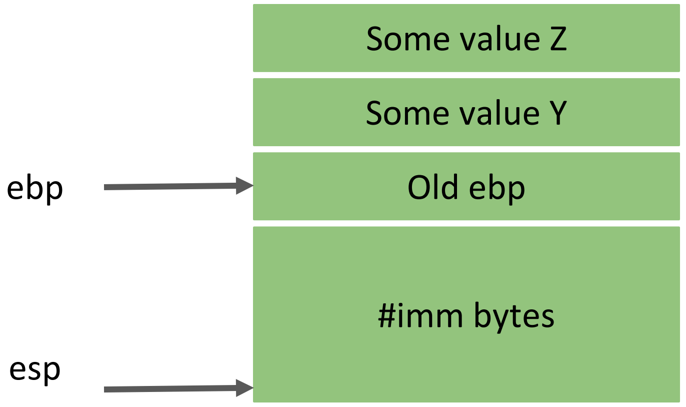
x86 Instructions that affect Stack

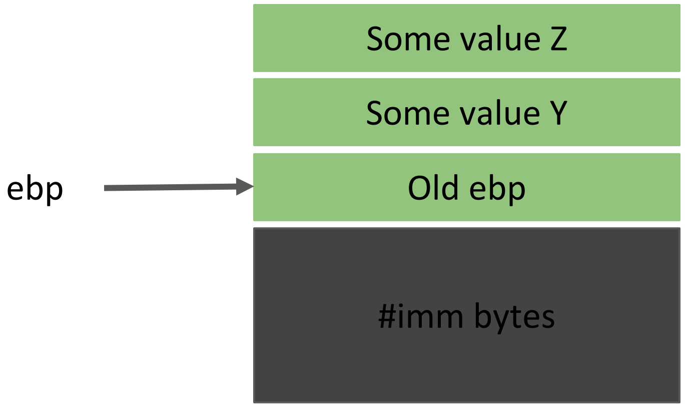
```
mov esp, ebp
pop ebp
```

Before:

leave

After: ebp = old ebp





Function Frame

Functions would like to use the stack to allocate space for their local variables.
Can we use the stack pointer (esp/rsp) for this?

- Yes, however stack pointer can change throughout program execution

Frame pointer points to the start of the function's frame on the stack

- Each local variable will be (different) **offsets** of the frame pointer
- In x86/64, frame pointer is called the base pointer, and is stored in **ebp/rbp**

Function Frame

A function's Stack Frame

- Starts with **where ebp/rbp points to**
- Ends with **where esp/rsp points to**

Calling Convention

Information, such as parameters, must be stored on the stack in order to call the function. Who should store that information? Caller? Callee?

Thus, we need to define a convention of who pushes/stores what values on the stack to call a function

- Varies based on processor, operating system, compiler, or type of call

x86 (32 bit) Linux Calling Convention (cdecl)

Caller (in this order)

- Pushes arguments onto the stack (in right to left order)
- Execute the call instruction (pushes address of instruction after call, then moves dest to eip)

Callee

- Pushes previous frame pointer onto stack (ebp)
- Setup new frame pointer (`mov ebp, esp`)
- Creates space on stack for local variables (`sub esp, #imm`)
- Ensures that stack is consistent on return
- Return value in eax register

Callee Allocate a stack (Function prologue)

Three instructions:

```
push  ebp;         (pushes previous frame pointer onto stack)
mov  ebp,  esp;  (change the base pointer to the stack)
sub  esp,  10;    (allocating a local stack space)
```

enter

Callee Deallocate a stack (Function epilogue)

```
mov esp, ebp (the bottom of the current function stack is the top of the caller’s stack)
pop ebp (restore the bottom of caller’s stack)
ret
```

Leave

Global and Local Variables (code/globallocalv)

```
char g_i[] = "I am an initialized global variable\n";
char* g_u;

int func(int p)
{
  int l_i = 10;
  int l_u;

  printf("l_i in func() is at %p\n", &l_i);
  printf("l_u in func() is at %p\n", &l_u);
  printf("p in func() is at %p\n", &p);
  return 0;
}
```

```
int main(int argc, char *argv[])
{
  int l_i = 10;
  int l_u;

  printf("g_i is at %p\n", &g_i);
  printf("g_u is at %p\n", &g_u);

  printf("l_i in main() is at %p\n", &l_i);
  printf("l_u in main() is at %p\n", &l_u);

  func(10);
}
```

Tools: readelf; nm

Global and Local Variables (code/globallocalv)

```
int func(int p)
{
  int l_i = 10;
  int l_u;

  printf("l_i in func() is at %p\n", &l_i);
  printf("l_u in func() is at %p\n", &l_u);
  printf("p in func() is at %p\n", &p);
  return 0;
}
```

Function main()

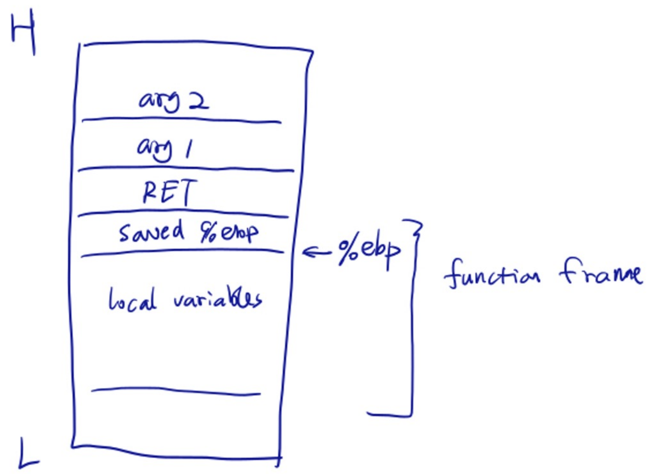
```
657:   83 ec 0c          sub    esp,0xc
65a:   6a 0a             push   0xa
65c:   e8 3c ff ff ff    call   59d <func>
661:   83 c4 10          add    esp,0x10
```

Function func()

```
59d:   55                      push   ebp
59e:   89 e5                   mov    ebp,esp
5a0:   83 ec 18                sub    esp,0x18
5a3:   c7 45 f4 0a 00 00 00    mov    DWORD PTR [ebp-0xc],0xa
5aa:   83 ec 08                sub    esp,0x8
5ad:   8d 45 f4                lea    eax,[ebp-0xc]
5b0:   50                      push   eax
5b1:   68 00 07 00 00          push   0x700
5b6:   e8 fc ff ff ff          call   5b7 <func+0x1a>
5bb:   83 c4 10                add    esp,0x10
5be:   83 ec 08                sub    esp,0x8
5c1:   8d 45 f0                lea    eax,[ebp-0x10]
5c4:   50                      push   eax
5c5:   68 18 07 00 00          push   0x718
5ca:   e8 fc ff ff ff          call   5cb <func+0x2e>
5cf:   83 c4 10                add    esp,0x10
5d2:   83 ec 08                sub    esp,0x8
5d5:   8d 45 08                lea    eax,[ebp+0x8]
5d8:   50                      push   eax
5d9:   68 30 07 00 00          push   0x730
5de:   e8 fc ff ff ff          call   5df <func+0x42>
5e3:   83 c4 10                add    esp,0x10
5e6:   b8 00 00 00 00          mov    eax,0x0
5eb:   c9                      leave
5ec:   c3                      ret
```

Draw the stack (x86 cdecl)

x86 , cdel in a function



(%ebp) : saved %ebp

4(%ebp) : RET

8(%ebp) : first argument

-8(%ebp) : maybe a local variable

x86 Stack Usage (32bit)

- Negative indexing over ebp

```
mov eax, [ebp-0x8]
lea eax, [ebp-24]
```

- Positive indexing over ebp

```
mov eax, [ebp+8]
mov eax, [ebp+0xc]
```

- Positive indexing over esp

x86 Stack Usage (32bit)

- Accesses local variables (negative indexing over ebp)

`mov eax, ebp-0x8`	value at ebp-0x8
`lea eax, ebp-24`	address as ebp-0x24

- Stores function arguments from caller (positive indexing over ebp)

`mov eax, ebp+8`	1st arg
`mov eax, ebp+0xc`	2nd arg

- Positive indexing over esp

Function arguments to callee

Stack example: code/factorial

```
int fact(int n)
{
  printf("---In fact(%d)\n", n);
  printf("&n is %p\n", &n);

  if (n <= 1)
    return 1;

  return fact(n-1) * n;
}
```

```
int main(int argc, char *argv[])
{
  if (argc != 2)
  {
    printf("Usage: fact integer\n");
    return 0;
  }

  printf("The factorial of %d is %d\n.",
atoi(argv[1]), fact(atoi(argv[1])));
}
```

Stack example: code/fivepara

```
int func(int a, int b, int c, int d, int e)
{
  return a + b + c + d + e;
}

int main(int argc, char *argv[])
{
  func(1, 2, 3, 4, 5);
}
```

X86 disassembly

globallocalv_fast_32

fastcall

On x86-32 targets, the fastcall attribute causes the compiler to pass the first argument (if of integral type) in the register ECX and the second argument (if of integral type) in the register EDX. Subsequent and other typed arguments are passed on the stack. The called function pops the arguments off the stack. If the number of arguments is variable all arguments are pushed on the stack.

```
int __attribute__ ((fastcall)) func(int p)
```

x86-64 (64 bit) Linux Calling Convention

Caller

- Use registers to pass arguments to callee. Register order (1st, 2nd, 3rd, 4th, 5th, 6th, etc.) rdi, rsi, rdx, rcx, r8, r9, ... (use stack for more arguments)

Stack example: code/fivepara

```
int func(int a, int b, int c, int d, int e)
{
  return a + b + c + d + e;
}

int main(int argc, char *argv[])
{
  func(1, 2, 3, 4, 5);
}
```

X86-64 disassembly

X86-64 Stack Usage

- Access local variables (negative indexing over rbp)

```
mov rax, [rbp-8]
lea rax, [rbp-0x24]
```

- Access function arguments from caller

```
mov rax, rdi
```

- Setup parameters for callee

```
mov rdi, rax
```