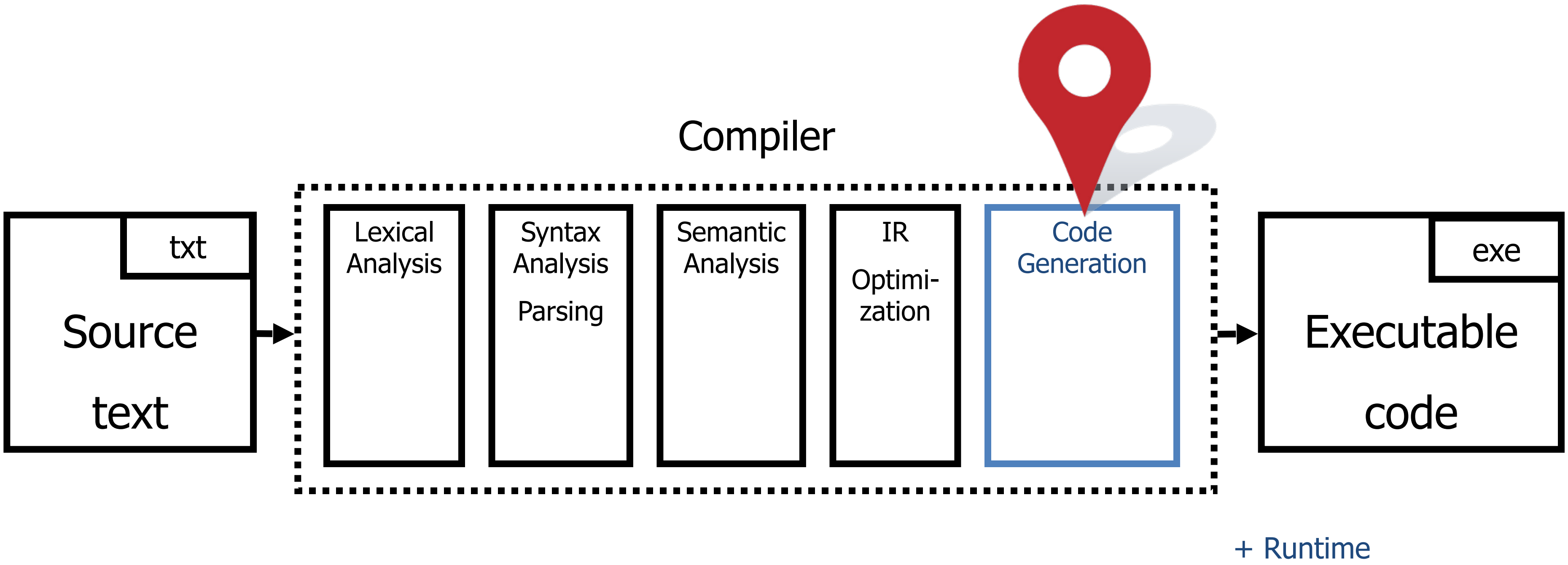
THEORY OF COMPILATION

LECTURE 09

ACTIVATION RECORDS

You are here


Compiler
txt
Source
text
Lexical
Analysis
Syntax
Analysis
Parsing
Semantic
Analysis
IR
Optimi-
zation
Code
Generation
exe
Executable
code
+ Runtime

Supporting Procedures

```
n = f(a[i]);
```

```
t1 := i * 4
t2 := a + t1
t3 := *t2
param t3
t4 := call f, 1
n := t4
```

code that evaluates the parameter `a[i]`

# of parameters

- How is that done?
- What do we need from the compiler?

Supporting Procedures

- Extending our computing environment
  - (at least) enough memory for local variables
- Passing information into the new environment
  - call parameters
- Transfer of control to/from procedure
- Handling return values

Design Decisions

- Scoping rules
  - Static scoping *vs*. dynamic scoping
- Memory layout
  - Allocating space for local variables
- Caller/callee conventions
  - Saving and restoring register values

Static (Lexical) Scoping

```
main ( )
{
     int a = 0;
     int b = 0;
     {
           int b = 1;
           {
                 int a = 2;
          B2     printf("%d %d\n", a, b);
           }
B0     B1  {
                 int b = 3;
          B3     printf("%d %d\n", a, b);
           }
           printf ("%d %d\n", a, b);
     }
     printf ("%d %d\n", a, b);
}
```

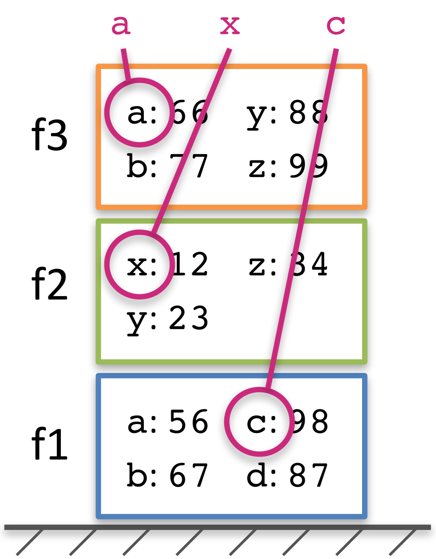
A name refers to its (closest) enclosing scope

known at compile time

Declaration	Scopes
a = 0	B_0, B_1, B_3
b = 0	B_0
b = 1	B_1, B_2
a = 2	B_2
b = 3	B_3

Dynamic Scoping

- Every function call creates new definitions for variables in its scope
  - definitions are maintained in a global stack
- When entering scope
  - push local variables on stack
- When exiting scope
  - pop local variables



- **Evaluating the identifier in any context binds to the current top of stack**
  - determined **at runtime**

Example

```
int x = 42;

int f() { return x; }
int g() { int x = 1; return f(); }
int main() { print g(); print x; }
```

- What values are output by `main`?
  - static scoping?
  - dynamic scoping?

Why do we care?

- **We need to generate code to access variables**

- Static scoping
  - identifier binding is known at **compile time**
  - address of the variable is known at **compile time**
  - assigning addresses to variables is part of code generation
  - no runtime errors of “access to undefined variable”
  - can check types of variables

Variable Addressing for Static Scoping

(first attempt)

```
int x = 42;

int f() { return x; }
int g() { int x = 1;
          return f(); }
int main() { print g(); }
```

identifier	address
x (global)	0x42
x (inside g)	0x73

Variable Addressing for Static Scoping

(first attempt)

```
int a[11];

void quicksort(int m, int n)
{
   int i;
   if (n > m) {
      i = partition(m, n);
      quicksort(m, i-1);
      quicksort(i+1, n);
   }
⋮

int main() {
   …
   quicksort(1, 9);
}
```

identifier	address
a (global)	0x42
`i` (inside quicksort)	…

what is the address of the variable **`i`** in the procedure `quicksort`?

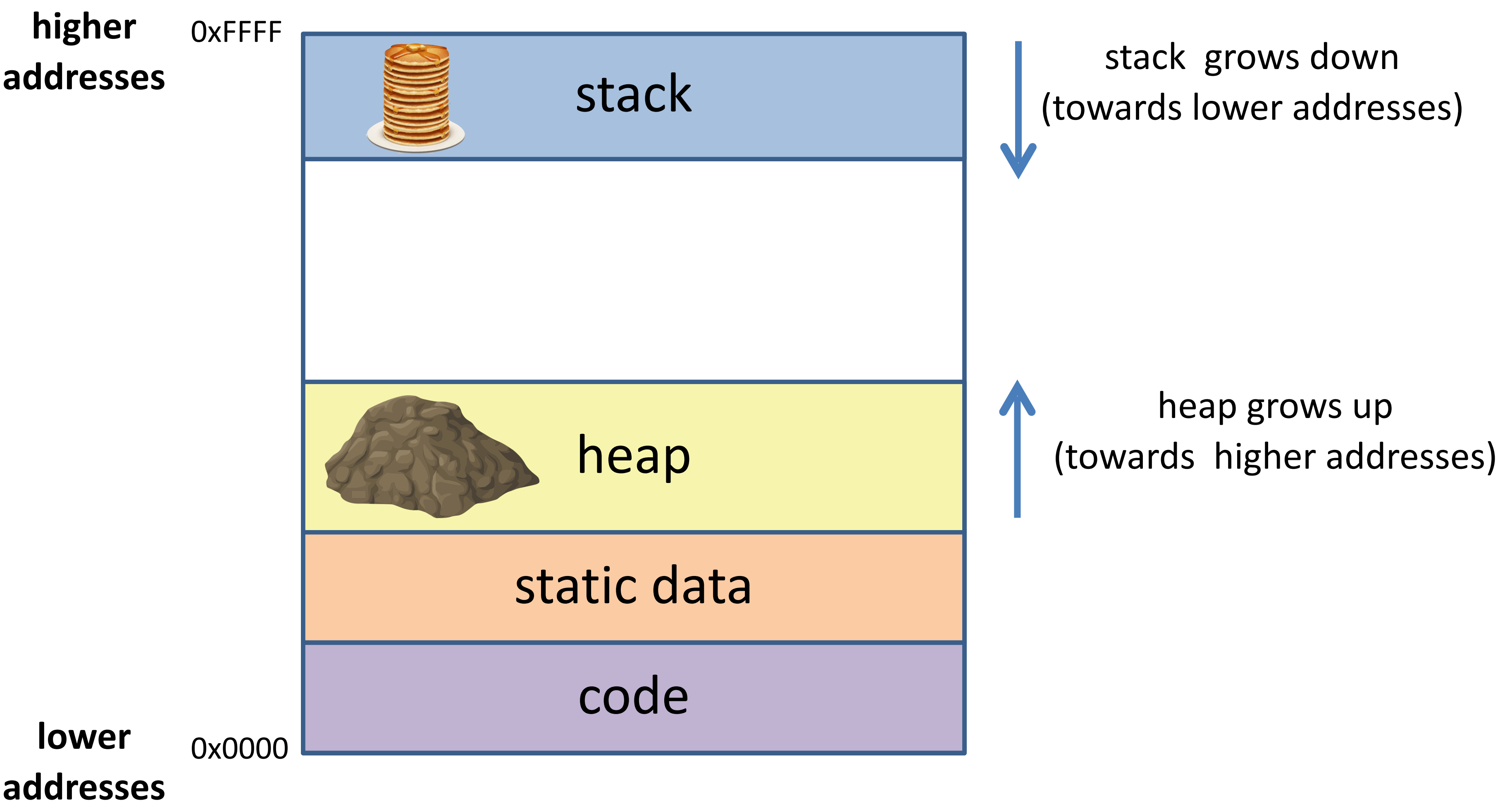
(or arguments **`m`**, **`n`**, for that matter?)

Activation Record (frame)

- Separate space for each procedure invocation
  - **Managed at runtime**
  - But **code** for managing it generated by the **compiler**

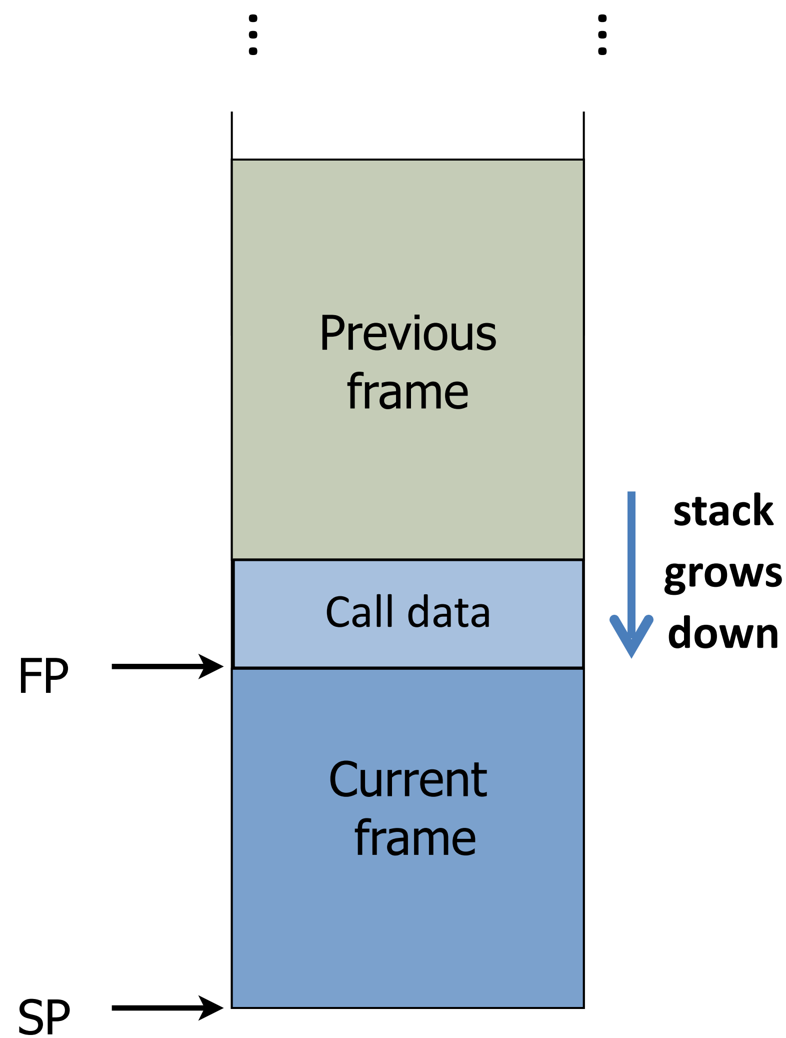
- Desired properties
  - Efficient allocation and deallocation
    - procedure calls are frequent
  - Flexible size
    - different procedures may require different memory sizes

Memory Layout



Runtime Stack

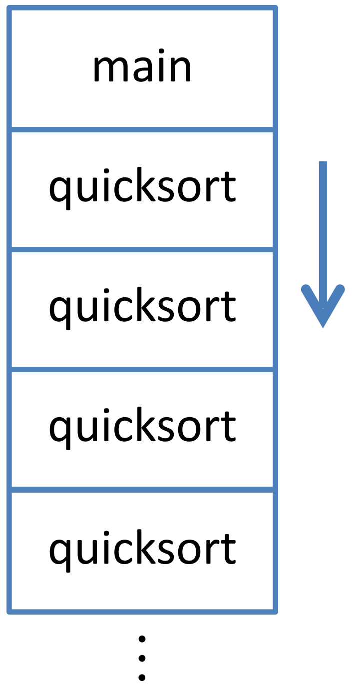
- FP – frame pointer
  - beginning of current frame + some offset
    - Sometimes called BP (base pointer)



- SP – stack pointer
  - end of current frame

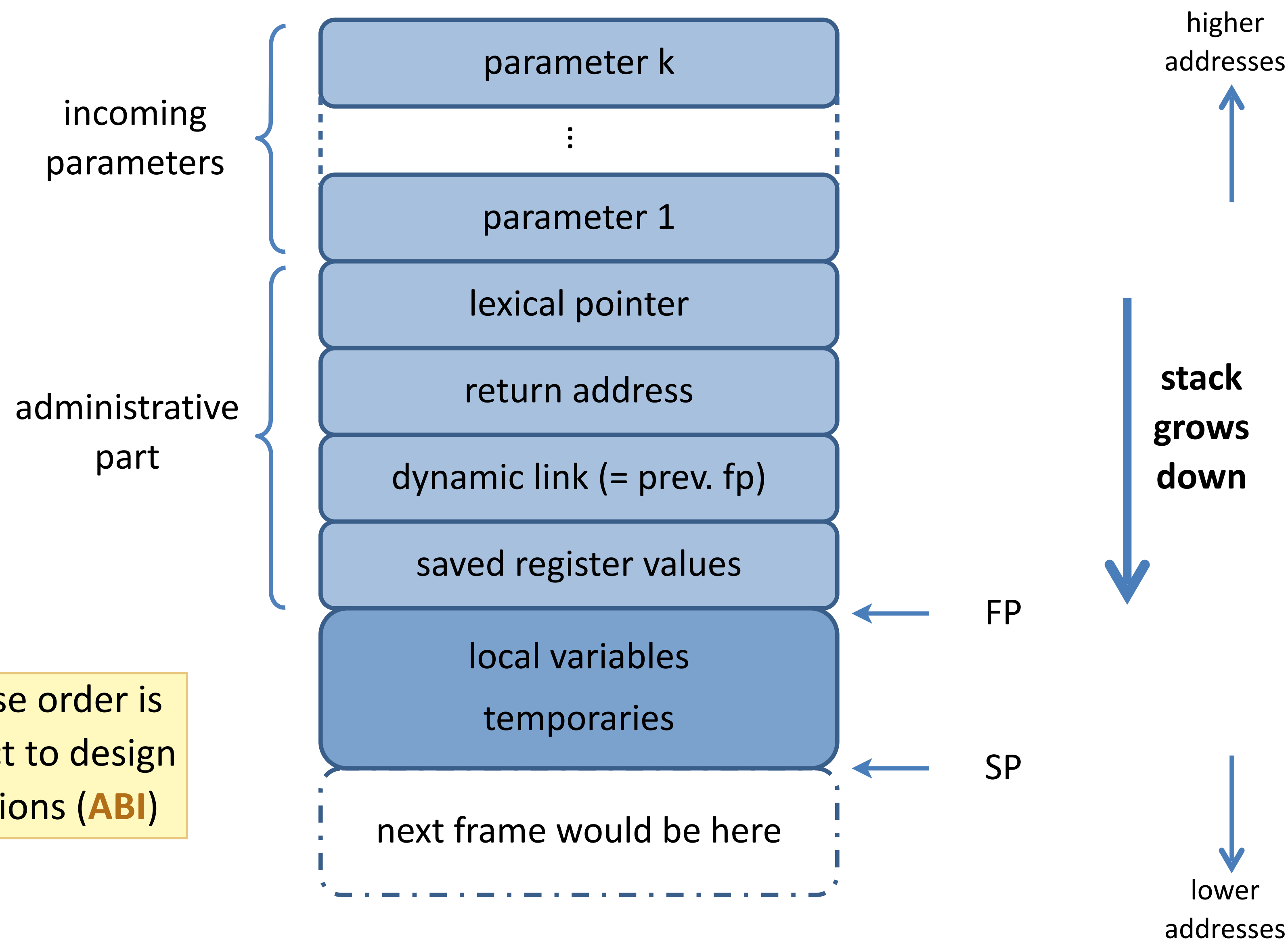
Runtime Stack

- Stack of activation records
  - Call = push new activation record
  - Return = pop activation record
- Only one “active” record at a time*
  - top of stack
- This allows to handle recursion



* per thread

Activation Record (frame)



Precise order is subject to design decisions (**ABI**)

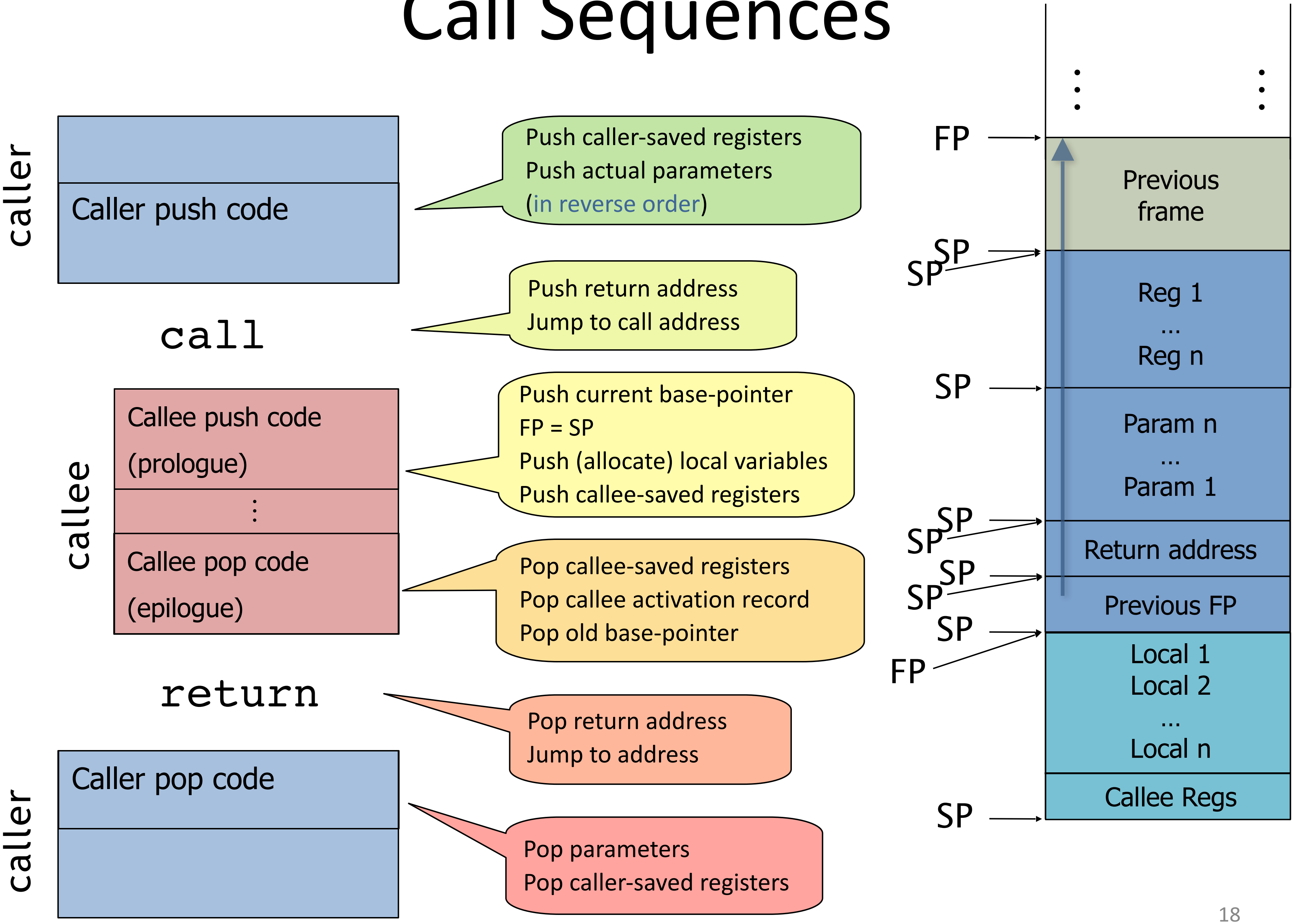
Call Sequences

- **The processor does not save the content of registers on "call" instruction**
  - (Why is this a problem?)

- So who will?

  There are several options:
  - **Caller** saves registers before call, restores after call
  - **Callee** saves when called, restores before return
  - But can also have **both** save/restore some registers

Call Sequences



x86 Runtime Stack

Register	Usage
%esp	Stack pointer
%ebp	Frame pointer

Stack registers

Instruction	Usage
push, pusha, …	push on runtime stack
pop, popa, …	pop from runtime stack
call	transfer control to called routine
ret	transfer control back to caller

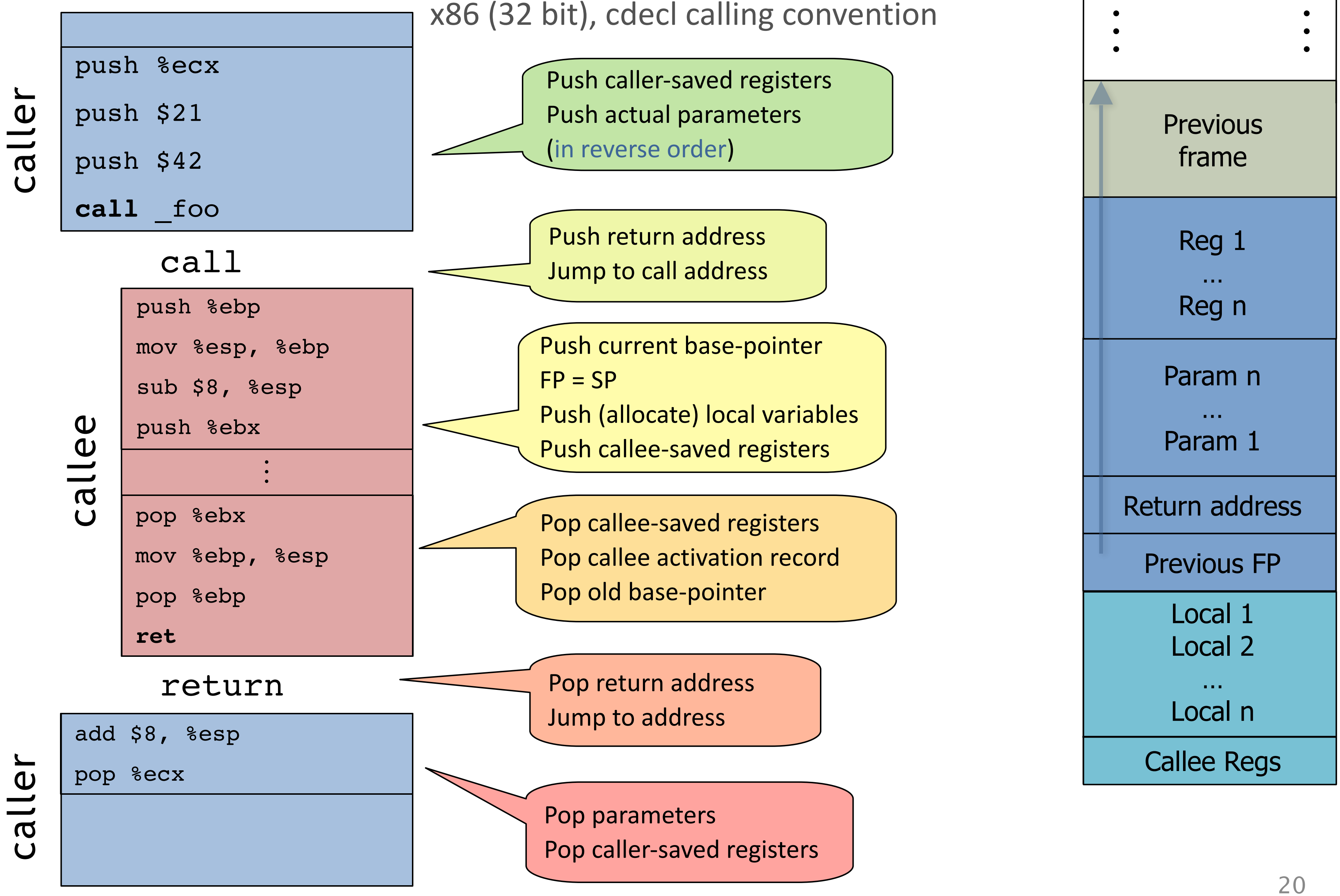
Stack and subroutine instructions

%eip	Instruction pointer

Call Sequences

`foo(42, 21)`

x86 (32 bit), cdecl calling convention



"To Callee-save or to Caller-save?"

- That is indeed the question
  - **Callee-saved** registers need only be saved when callee modifies their value
  - **Caller-saved** registers need only be saved if the caller needs their value after the call returns
- Some conventions and heuristics are followed
  - x86 cdecl: %eax, %ecx, %edx are **caller-saved**; the rest are **callee-saved**.

These calling conventions are part of the architecture's **ABI** (Application Binary Interface)

“To Callee-save or to Caller-save?”

x86 cdecl: %eax, %ecx, %edx are **caller-saved**; the rest are **callee-saved**.

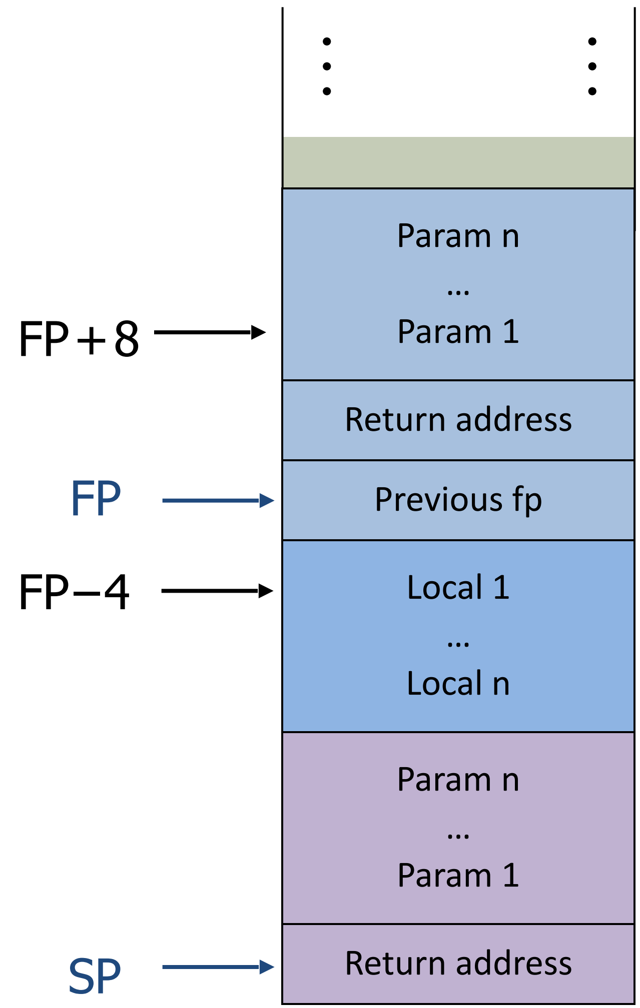
```
bar:
   push %ebp
   mov %esp, %ebp
                        push %ebx
   mov %ebx, 6
   mov (@garray,%ebx,4), %eax
   mov %ebx, 7
   mov (@garray,%ebx,4), %ecx
   add %ecx, %eax
                        pop %ebx
   pop %ebp
   ret
```

```
main:                          printf("%d", bar());
   ⋮
   mov %edx, (@str)  ; "%d"
                        push %edx
   call bar
                        pop %edx
   push %eax
   push %edx
   call printf

   ⋮
```

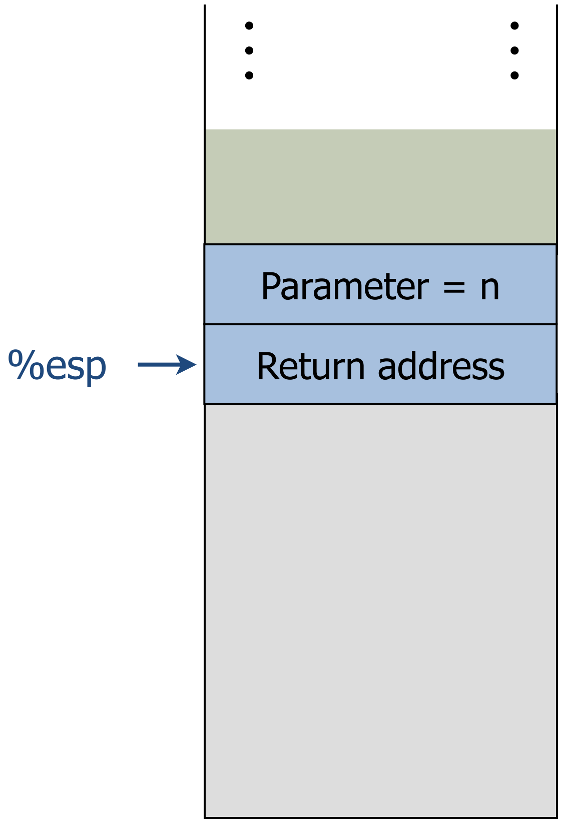
Accessing Stack Variables

- Use offset from FP (%ebp)
- Remember –
  stack grows downwards
- Above FP = parameters
- Below FP = locals
- Examples
  - FP + 4 = return address
  - FP + 8 = first parameter
  - FP – 4 = first local



Factorial – `fact(int n)`

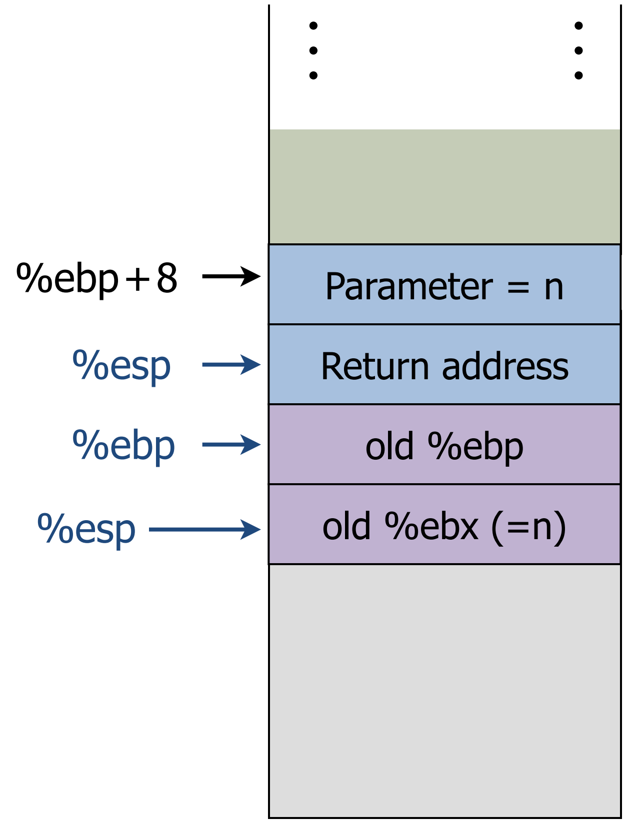
```
int fact(int n) {
    if (n <= 1) {
        return 1;
    }
    else {
        return fact(n - 1) * n;
    }
}
```



(stack immediately after call)

Factorial — `fact(int n)`

```
fact:
 pushl %ebp               # save ebp
 movl %esp,%ebp           # ebp = esp
 pushl %ebx               # save ebx
 movl 8(%ebp),%ebx        # ebx = n
 cmpl $1,%ebx             # n <= 1 ?
 jle .lresult             # then done
 leal -1(%ebx),%eax       # eax = n-1
 pushl %eax               #
 call fact                # eax = fact(n-1)
 addl $4,%esp             #
 imull %ebx,%eax          # eax = eax * n
 jmp .lreturn             #
lresult:
 movl $1,%eax             # return 1
lreturn:
 movl -4(%ebp),%ebx       # restore ebx
 movl %ebp,%esp           # restore esp
 popl %ebp                # restore ebp
 ret
```



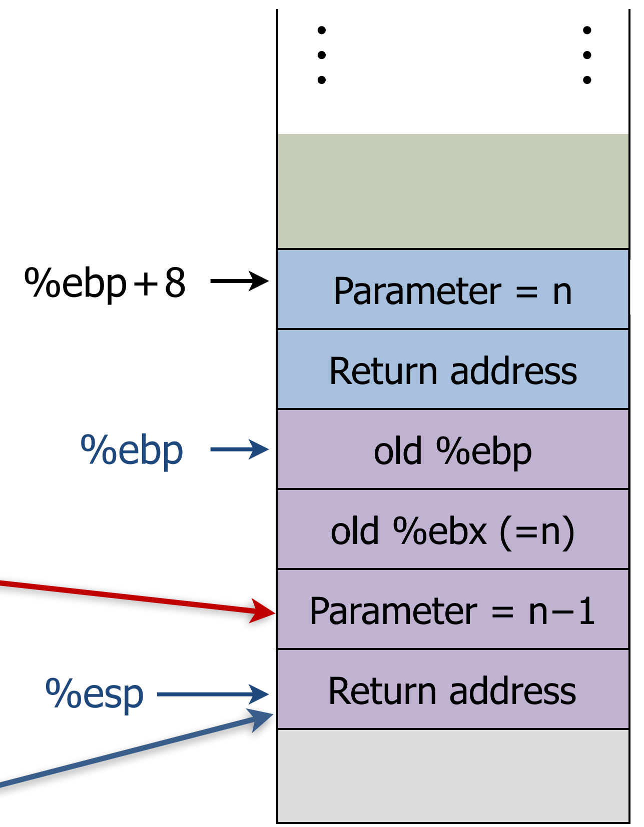
(stack after prologue)

Factorial — `fact(int n)`

```
fact:
 pushl %ebp               # save ebp
 movl %esp,%ebp           # ebp = esp
 pushl %ebx               # save ebx
 movl 8(%ebp),%ebx        # ebx = n
 cmpl $1,%ebx             # n <= 1 ?
 jle .lresult             # then done
 leal -1(%ebx),%eax       # eax = n-1
 pushl %eax               #
 call fact                # eax = fact(n-1)
 addl $4,%esp             #
 imull %ebx,%eax          # eax =
 jmp .lreturn             #
lresult:
 movl $1,%eax             # return 1
lreturn:
 movl -4(%ebp),%ebx       # rest
 movl %ebp,%esp           # restore esp
 popl %ebp                # restore ebp
 ret
```

Caller pops parameters

Callee pops return address



(stack immediately after second call)

Nested Procedures

Exist *e.g.* in Pascal, JavaScript

- Idea: A routine can have sub-routines
- Any sub-routine can access anything that is defined in its own scope or any surrounding scope
  - “non-local” variables

- A sub-routine is allowed to call itself, its own sub-routines, any of its ancestors, and also its *siblings*
  - (but not “nephews” or “grandchildren”)

Example: Nested Procedures

```
program p;
    var x: Integer;
    procedure a
        var y: Integer;
        procedure b
            begin … … end;
        function c
            var z: Integer;
            procedure d begin … y … end;
        begin … … end;
    begin … … end;
begin … … end.
```

possible call sequence:

p $\rightsquigarrow$ a $\rightsquigarrow$ a $\rightsquigarrow$ c $\rightsquigarrow$ b $\rightsquigarrow$ c $\rightsquigarrow$ d

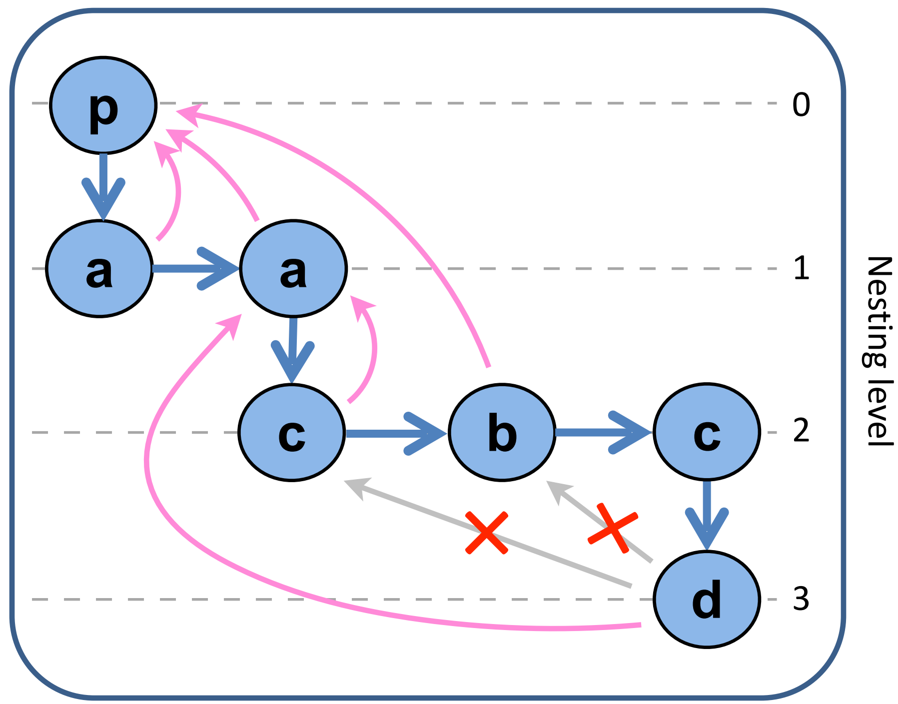
what is the address of variable “y” in procedure d?

Nested Procedures

- **A routine can call a nestling, an ancestor, or a sibling**
  - When “c” uses (non-local) variables from “a”, which instance of “a” is it?
  - How do you find the right activation record at runtime?

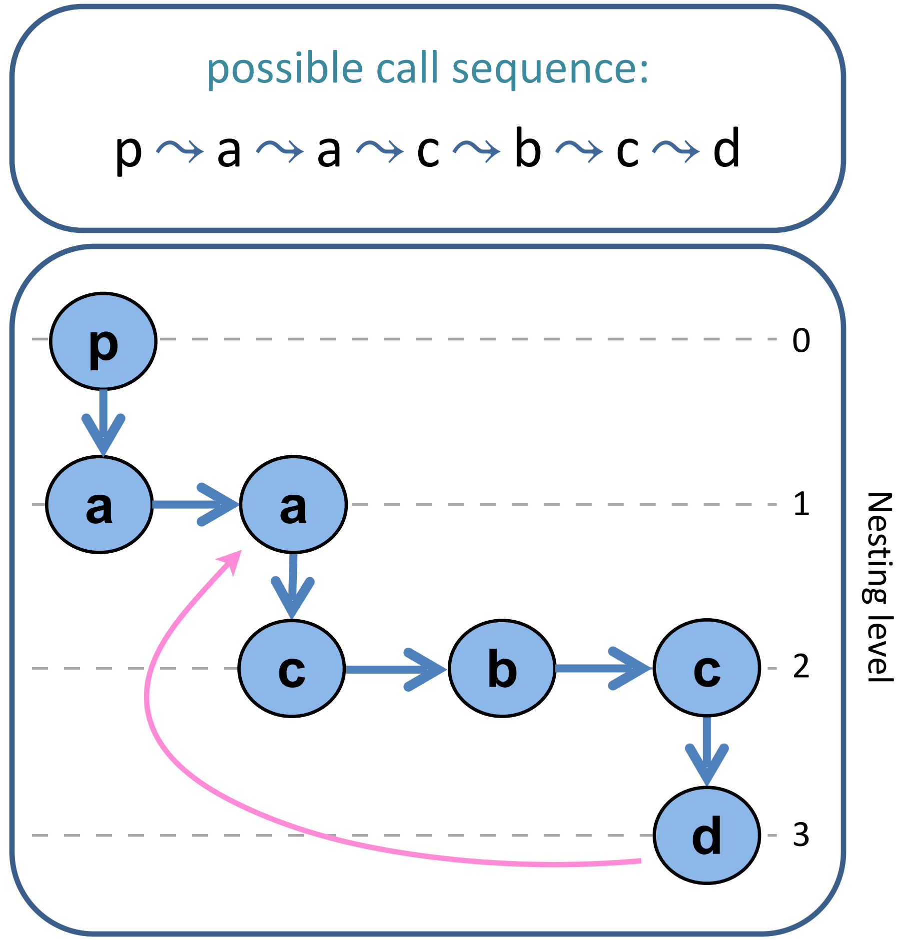
possible call sequence:

p ⇝ a ⇝ a ⇝ c ⇝ b ⇝ c ⇝ d



Nested Procedures

- <u>Goal</u>: **find the closest frame of some routine in the stack from a given nesting level**
- A routine may occur more than once in a sequence of calls
  - if a routine at level k uses variables of the same nesting level $\Rightarrow$ it refers to the current activation record
  - if it uses variables of nesting level $j < k$ $\Rightarrow$ it must be the **last** routine called at level j
- If some routine is last at level j on the stack, then it **must be an ancestor** of the current routine



Nested Procedures

- Goal: find the closest frame of some routine in the stack from a given nesting level
- Solution: **store a *lexical pointer* (a.k.a. *access link*) in the activation record**
  - Lexical pointer points to the **last** activation record of the nesting level **above** it
    - According to what we just said, this is enough to reach any variable being referenced
- Lexical pointers are created **at runtime**
- Number of links to be traversed is always **known at compile time**

Activation Record (frame)

program p;
  procedure a; begin … end
begin a(21, 42); end

caller

```
push %ecx
push $21
push $42
push %ebp
call _foo
```

Push caller-saved registers
Push actual parameters
(in reverse order)
Push lexical pointer

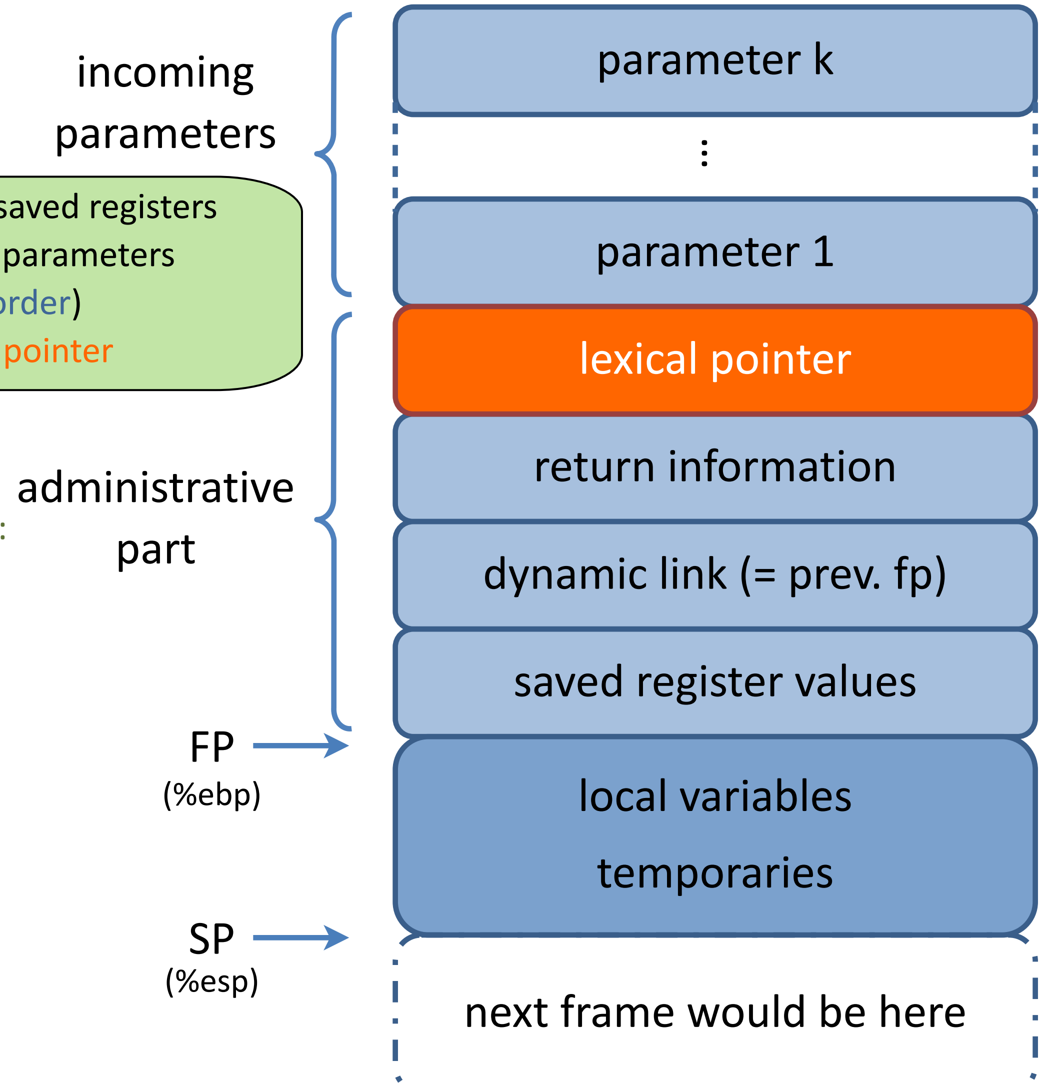
when calling a child:
push own FP

procedure a;
  procedure b;
  begin c(63); end
  procedure c; begin … end

caller

```
push %ecx
push $63
mov  8(%ebp), %esi
push %esi
call _foo
```

when calling a sibling (or self):
push a copy of the current lexical pointer

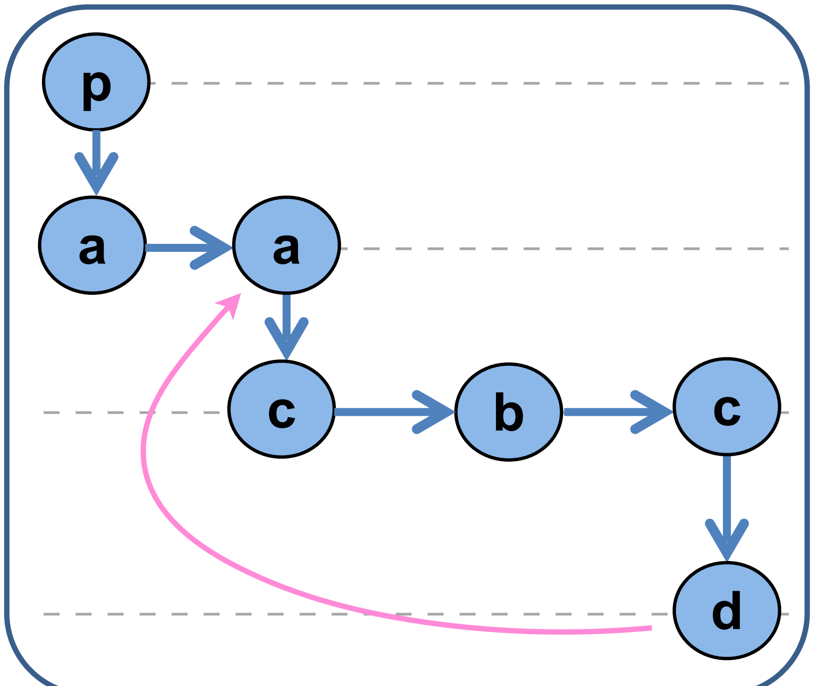


Lexical Pointers

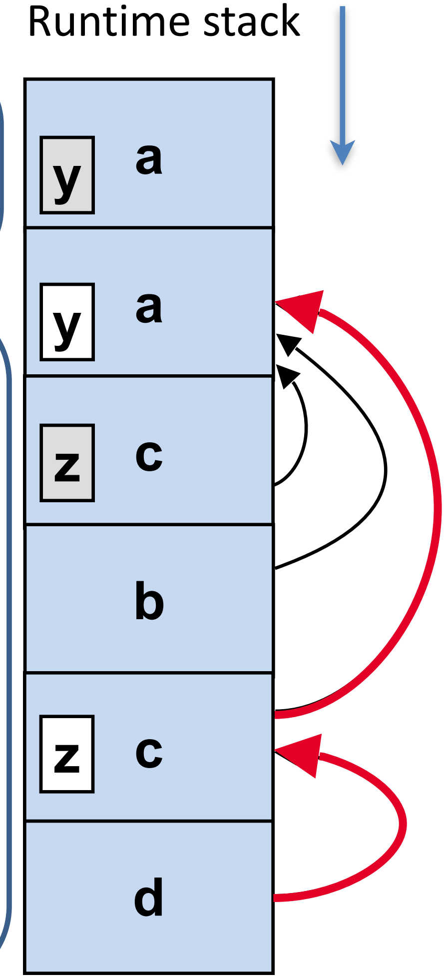
```
program p;
    var x: Integer;
    procedure a
        var y: Integer;
        procedure b
        begin… … end;
        function c
            var z: Integer;
            procedure d
            begin … y … end;
        begin … … end;
    begin … … end;
begin … … end.
```

possible call sequence:

p ⇝ a ⇝ a ⇝ c ⇝ b ⇝ c ⇝ d



Runtime stack

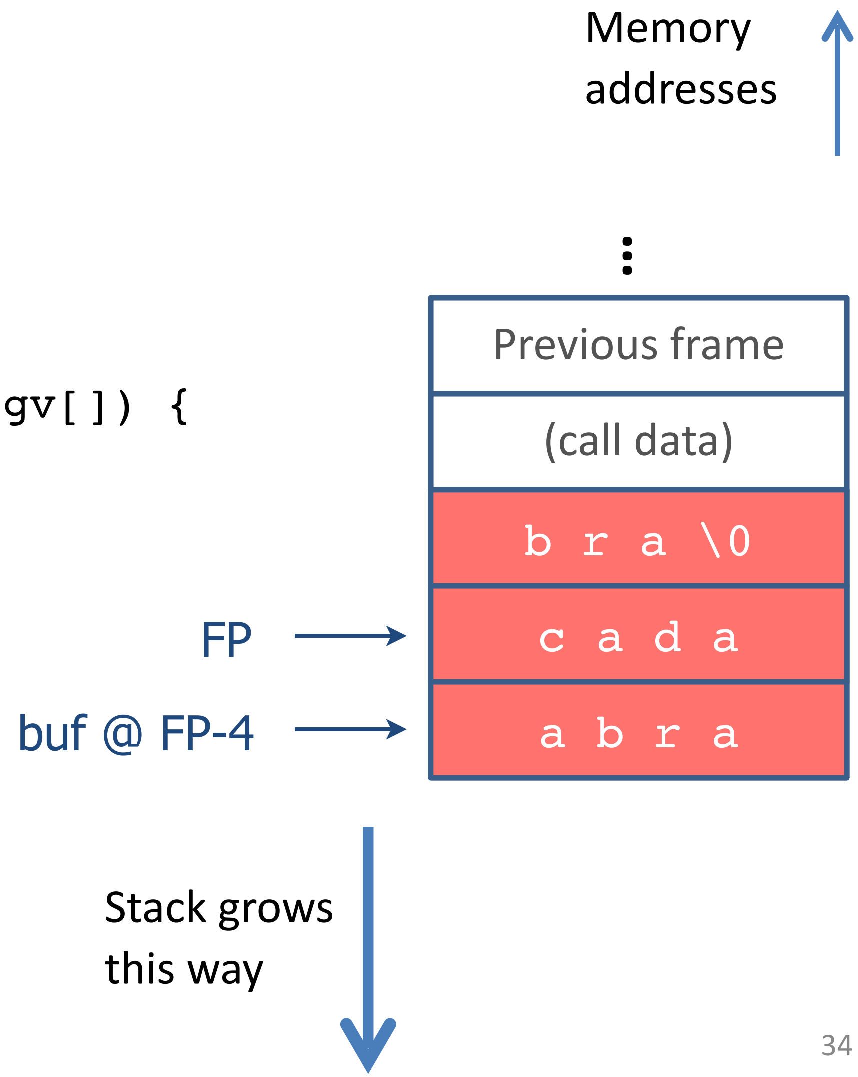


Security Exploit(s)

Buffer Overflow

```
void foo (char *x) {
  char buf[4];
  strcpy(buf, x);
}
int main (int argc, char *argv[]) {
  foo(argv[1]);
}
```

```
% ./a.out abracadabra
Segmentation fault
```



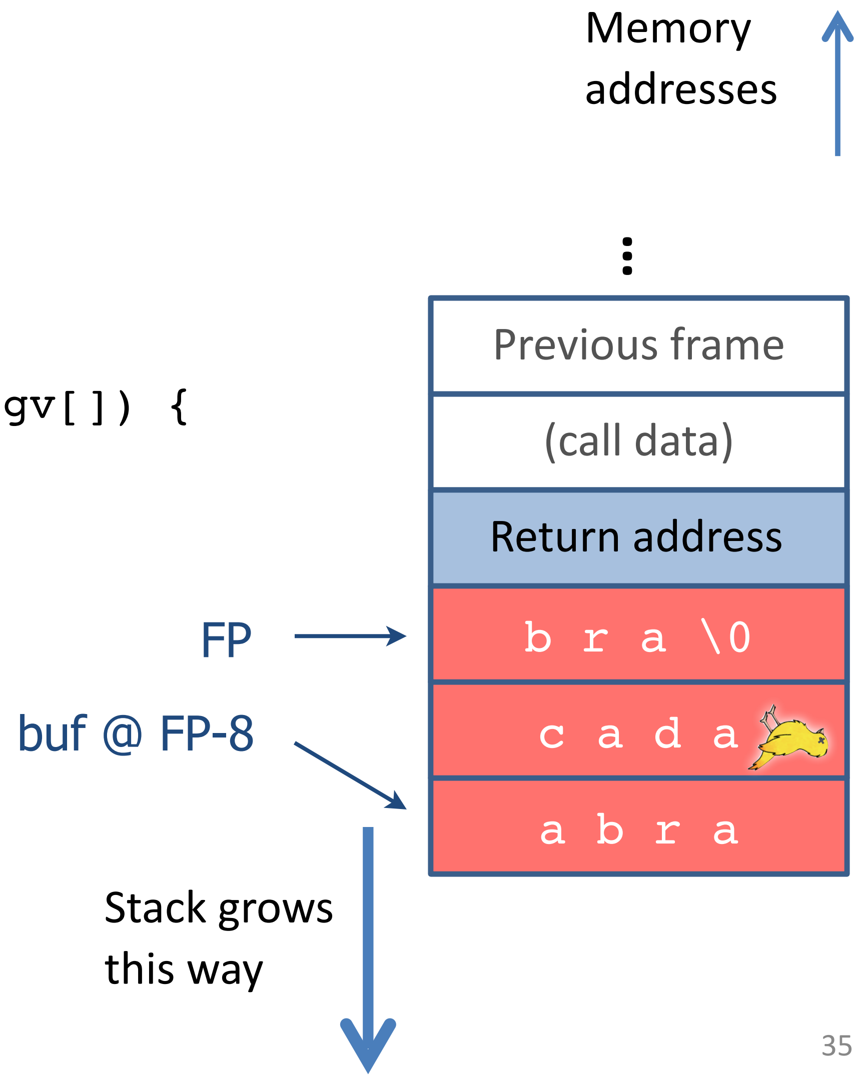
(YMMV)

Security Exploit(s)

Buffer Overflow

```c
void foo (char *x) {
   char buf[4];
   strcpy(buf, x);
}
int main (int argc, char *argv[]) {
   foo(argv[1]);
}
```

```
% ./a.out abracadabra
<busted>
```



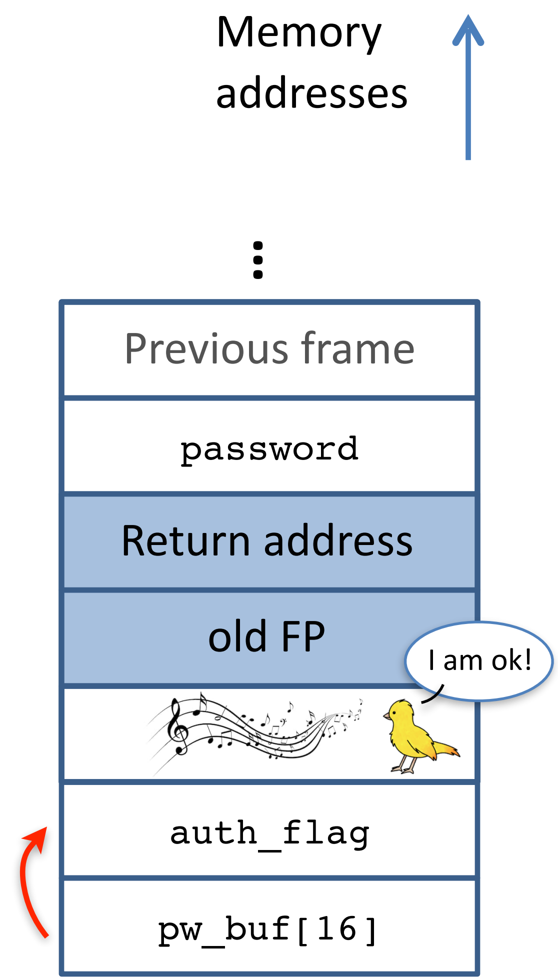
(YMMV)

Buffer overflow

```
int check_authentication(char *password) {
    int auth_flag = 0;
    char pw_buf[16];

    strcpy(pw_buf, password);
    if (strcmp(pw_buf, "brillig") == 0)
        auth_flag = 1;
    if (strcmp(pw_buf, "outgrabe") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if (check_authentication(argv[1])) {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf(“-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n”);
    }
    else
        printf("\nAccess Denied.\n");
}
```



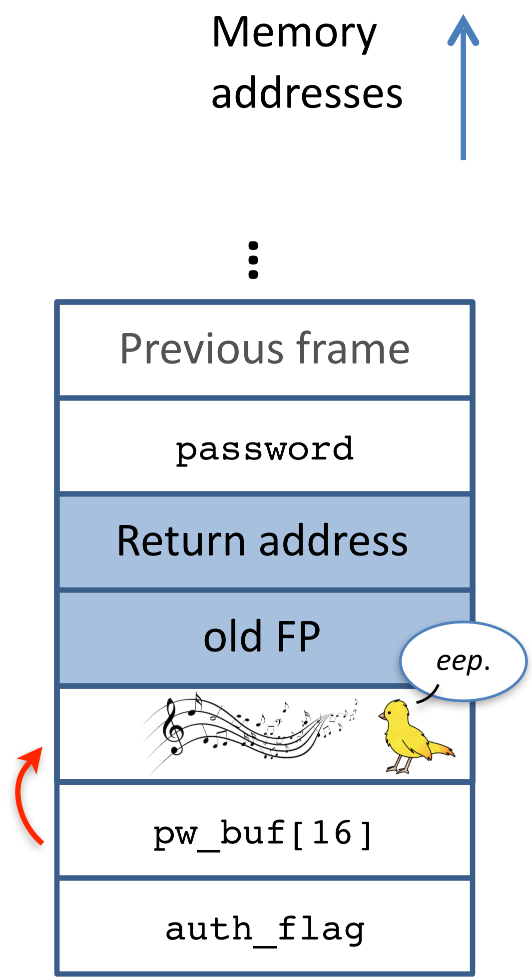
(source: “hacking – the art of exploitation, 2nd Ed”)

Buffer overflow

```
int check_authentication(char *password) {
    char pw_buf[16];
    int auth_flag = 0;

    strcpy(pw_buf, password);
    if (strcmp(pw_buf, "brillig") == 0)
        auth_flag = 1;
    if (strcmp(pw_buf, "outgrabe") == 0)
        auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: %s <password>\n", argv[0]);
        exit(0);
    }
    if (check_authentication(argv[1])) {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf("      Access Granted.\n");
        printf(“-=-=-=-=-=-=-=-=-=-=-=-=-=-\n”);
    }
    else
        printf("\nAccess Denied.\n");
}
```



(source: “hacking – the art of exploitation, 2nd Ed”)

Activation Records: Summary

✓ Compile time memory management for procedure data
  - Used to pass parameters, store local variables, and restore registers

✓ Works well for data with well-scoped lifetime
  - deallocation when procedure returns
  - no need to call free() for stack data

Coming Up