Binding and Scoping

Read: Scott, Chapter 3.1, 3.2 and 3.3.1, 3.3.2 and 3.3.6

Lecture Outline

- Notion of binding time
- Object lifetime and storage management

- An aside: Stack Smashing 101

- Scoping
  - Static scoping
  - Dynamic scoping

Notion of Binding Time

- Binding time (Scott): the time an answer becomes associated to an open question

Notion of Binding Time

- Static
  - Before program execution

- Dynamic
  - During program executes

Examples of Binding Time Decisions

- Binding time (Scott): the time an answer becomes associated to an open question

- Binding a variable name to a memory location
  - Static or dynamic
  - Determined by scoping rules
- Binding a variable/expression to a type
  - Static or dynamic
- Binding a call to a target subroutine
  - Static (as it is in C, mostly) or dynamic (virtual calls in Java, C++)

Example: Binding Variables to Locations

- Map a variable to a location
  - Map variable at use to a location
  - Map subroutine at use to target subroutine
- Determined by scoping rules
  - Static scoping
    - Binding before execution
  - Dynamic scoping
    - Binding during execution
- More on scoping later…

```
int x,y;
void foo(int x)
{
   y = x;
   int y = 0;
   if (y) {
      int y;
      y = 1;
   }
}
```

General View of Dynamic Binding

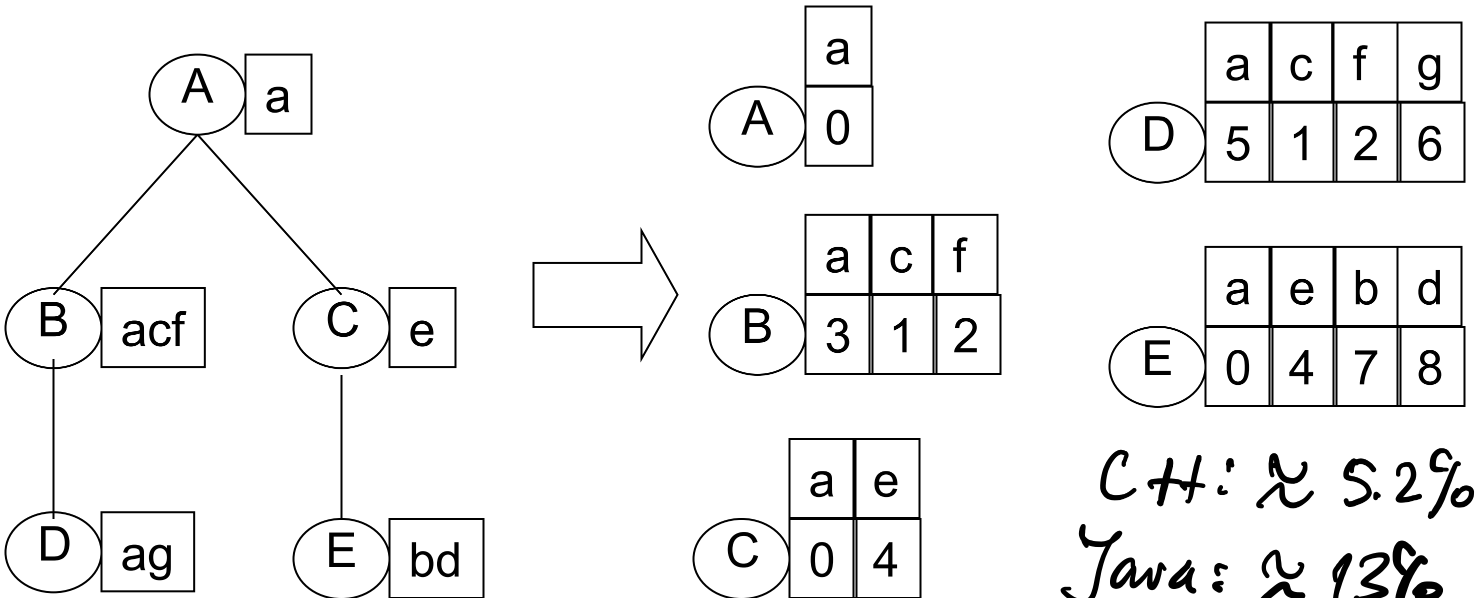
- Dynamic binding → FLEXIBILITY
  - What are the advantages of dynamic binding?
  - Disadvantages?

PERFORMANCE HIT

- An example: Cost of dynamic binding of call to target method in OO languages

Example: Cost of Dynamic Dispatch in C++

- Source: Driesen and Hölzle, OOPSLA’ 96



C++: ≈ 5.2% in dispatch
Java: ≈ 13%

Virtual function tables (VFTs)
Capital characters denote classes,
lowercase characters message selectors,
and numbers method addresses

```
load [object_reg+#VFTOffset],table_reg
load [table_reg+#selectorOffset],method_reg
call method_reg
```

Extra instructions: cost extra

Other Choices Related to Binding Time

- Pointers: introduce "heap variables"
  - Good for flexibility – allows dynamic structures
  - Bad for efficiency – direct cost: accessed indirectly; indirect cost: compiler unable to perform optimizations
- Most PLs support pointers
  - Issues of management of heap memory
    - Explicit allocation and deallocation
    - Implicit deallocation (garbage collection)
- PL design choices – many subtle variations
  - No pointers (FORTRAN 77)
  - Explicit pointers (C++ and C)   A *p;
  - Implicit pointers (Java)   A p;

Lecture Outline

- Notion of binding time
- Object lifetime and storage management
- An aside: Stack Smashing 101
- Scoping
  - Static scoping
  - Dynamic scoping

Storage Allocation Mechanisms

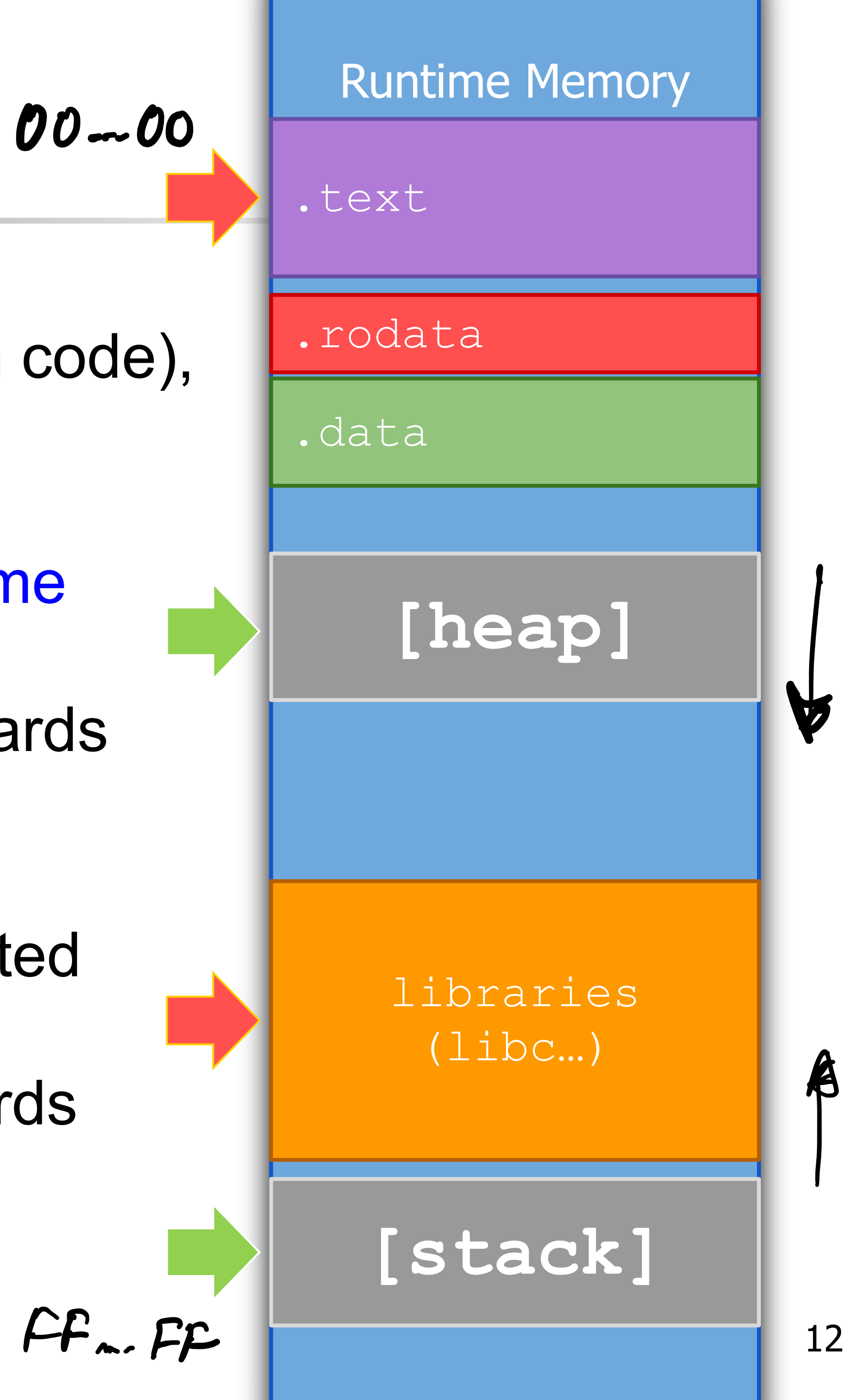
- Static storage – an object is given absolute address, which is the same throughout execution
  - What is an example of static data?
- Stack storage – stack objects are allocated on a run-time stack at subroutine call and deallocated at return
  - Needs a stack management algorithm
  - What is an example of stack data?
- Heap storage - long-lived objects are allocated and deallocated at arbitrary times during execution
  - Needs the most complex storage management algorithm

Combined View

- Static storage: .text (program code), .rodata, .data, etc.

- Stack contains one stack frame per executing subroutine
- Stack grows from higher towards lower memory addresses

- Heap contains objects allocated and not yet de-allocated
- Heap grows from lower towards higher memory addresses



Memory graph courtesy of RPISEC/MBE class.

Examples of Static Data

- Program code
- Global variables
- Tables of type data (e.g., inheritance structure)
- Dispatch tables (VFTs) and other tables
- Other

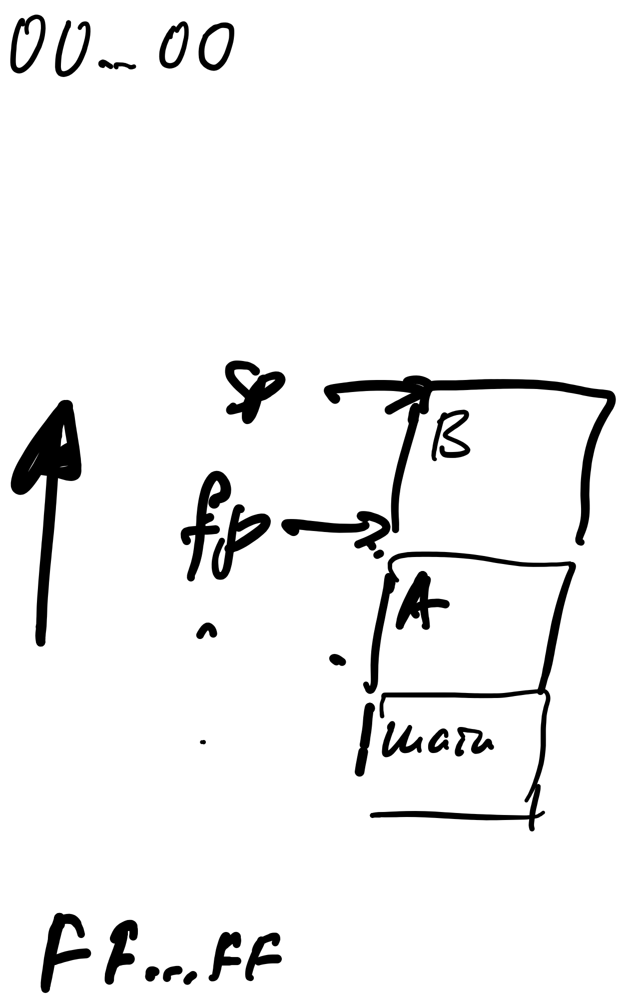
Examples of Stack Data

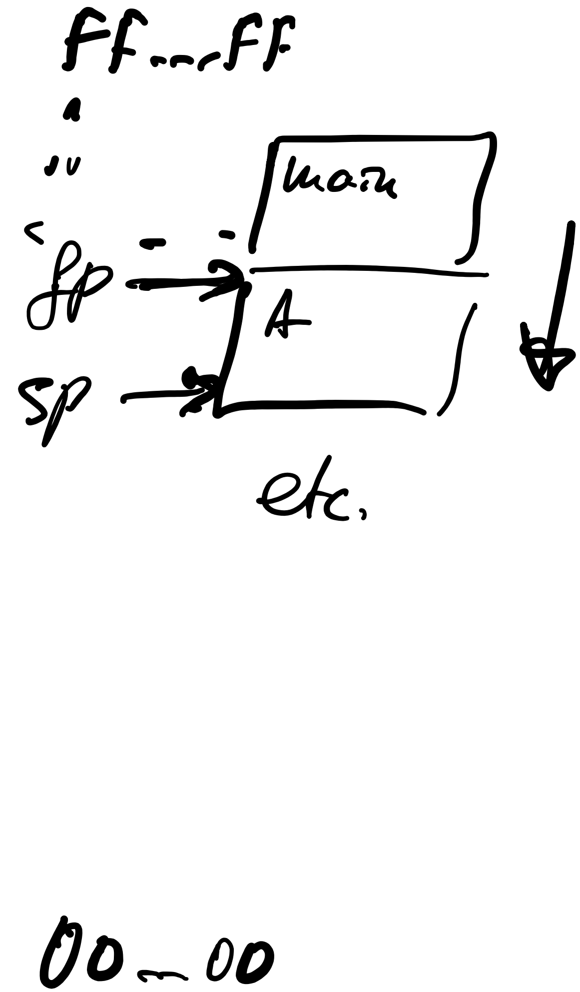
- What data is stored on the stack?
- Local variables, including parameters
- Compiler-generated temporaries (i.e., for expression evaluation)
- Bookkeeping (stack management) information
- Return address

Run-time Stack

- Stack contains frames of all subroutines that have been entered and not yet exited from
- Frame contains all information necessary to update stack when subroutine is exited
- Stack management uses two pointers: fp (**f**rame **p**ointer) and sp (**s**tack **p**ointer)
  - fp points to a location at the start of current frame
    - In higher memory (but lower on picture)
  - sp points to the next available location on stack (or the last used location on some machines)
    - In lower memory (but higher up on picture)
  - fp and sp define the beginning and the end of the frame

Run-time Stack





Run-time Stack Management

- Addresses for local variables are encoded as sp + offset
  - But may also have fp - offset
- Idea:
  - When subroutine is entered, its frame is placed on the stack. sp and fp are updated accordingly
  - All local variable accesses refer to this frame
  - When subroutine is exited, its frame is removed from the stack and sp and fp are updated accordingly

Frame Details

- Arguments to called routines

- Local variables, including parameters
- Temporaries

- Miscellaneous bookkeeping information
  - Saved address of start of caller's frame (old fp)
  - Saved state (register values of caller), other

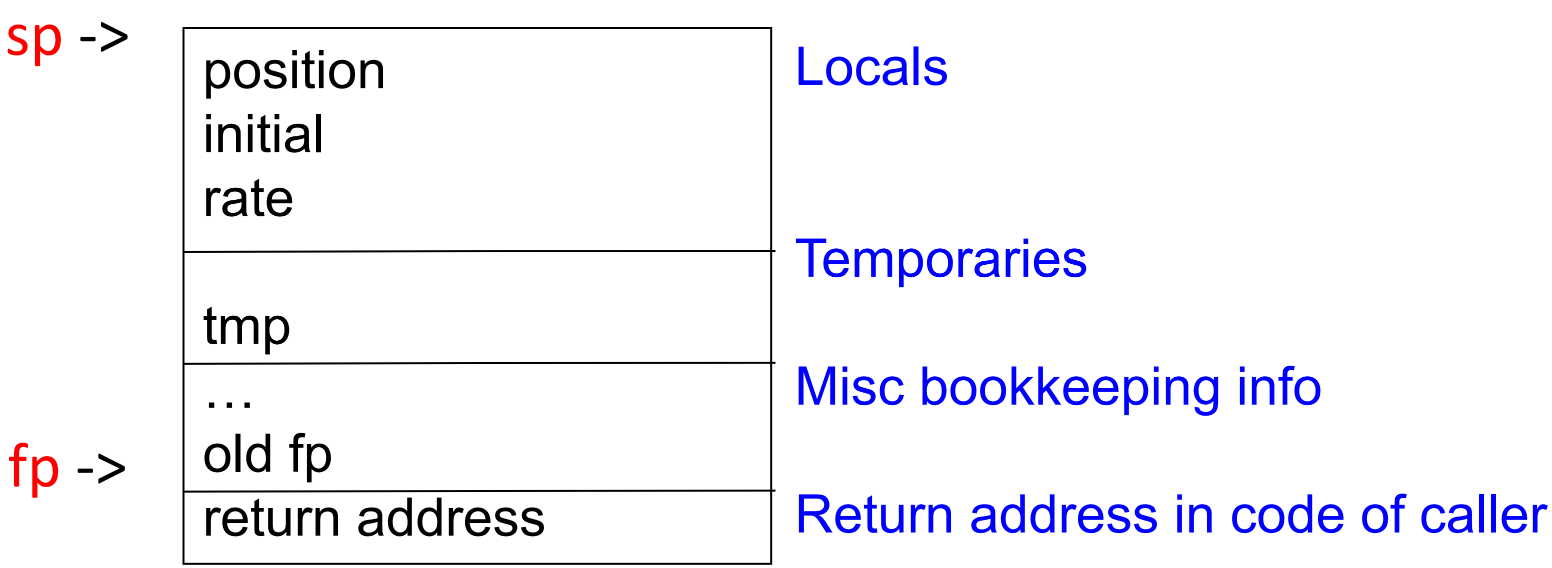
- Return address

Frame Example

```
void foo(double rate, double initial) {
  double position; …
  position = initial + rate*60.0; …
  return;
}
```

Assume **bar** calls **foo**.

Frame for **foo**:



Lecture Outline

- Notion of binding time
- Object lifetime and storage management
- An aside: Stack Smashing 101
  - Slides courtesy of RPISEC/MBE
- Scoping
  - Static scoping
  - Dynamic scoping

Stack Frames

- In x86-64 RBP is fp and RSP is sp. Define the stack frame for the currently executing function
  - local variables
  - pointer to previous frame
  - return address

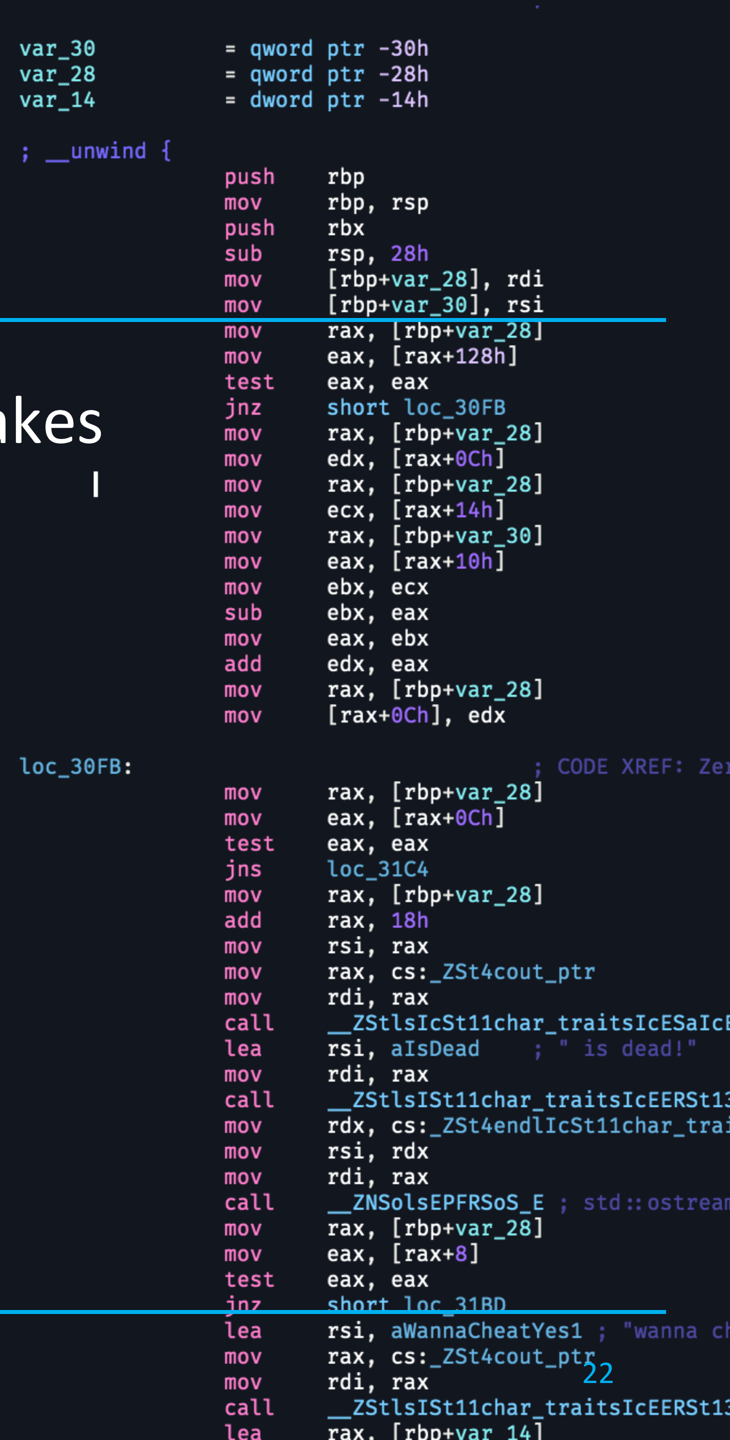
note: for 64bit, each 'slot' is 8 bytes

```
void foo() {
  long long x = 0x1337;
  char str[16];
  strcpy(str, "ABCDEFGH0123456");
}
```

What is corruption?

- What happens if a programmer makes a simple mistake:

```
char foo[64];
int money = 0;
gets(foo);
```



gets()?

```
var_30          = qword ptr -30h
var_28          = qword ptr -28h
var_14          = dword ptr -14h

; __unwind {
                push    rbp
                mov     rbp, rsp
                push    rbx
                sub     rsp, 28h
                mov     [rbp+var_28], rdi
```

```
NAME
       gets - get a string from standard input (DEPRECATED)

SYNOPSIS
       #include <stdio.h>

       char *gets(char *s);

DESCRIPTION
       Never use this function.

       gets() reads a line from stdin into the buffer pointed to by s until either a terminating new-
       line or EOF, which it replaces with a null byte ('\0').  No check for buffer overrun  is  per-
       formed (see BUGS below).
```

```
                test    eax, eax
                jns     loc_31C4
                mov     rax, [rbp+var_28]
                add     rax, 18h
                mov     rsi, rax
                mov     rax, cs:_ZSt4cout_ptr
                mov     rdi, rax
                call    __ZStlsIcSt11char_traitsIcESaIcE
                lea     rsi, aIsDead    ; " is dead!"
                mov     rdi, rax
                call    __ZStlsISt11char_traitsIcEERSt13
                mov     rdx, cs:_ZSt4endlIcSt11char_trai
                mov     rsi, rdx
                mov     rdi, rax
                call    __ZNSolsEPFRSoS_E ; std::ostream
                mov     rax, [rbp+var_28]
                mov     eax, [rax+8]
                test    eax, eax
                jnz     short loc_31BD
                lea     rsi, aWannaCheatYes1 ; "wanna ch
                mov     rax, cs:_ZSt4cout_ptr
                mov     rdi, rax
                call    __ZStlsISt11char_traitsIcEERSt13
                lea     rax, [rbp+var_14]
```

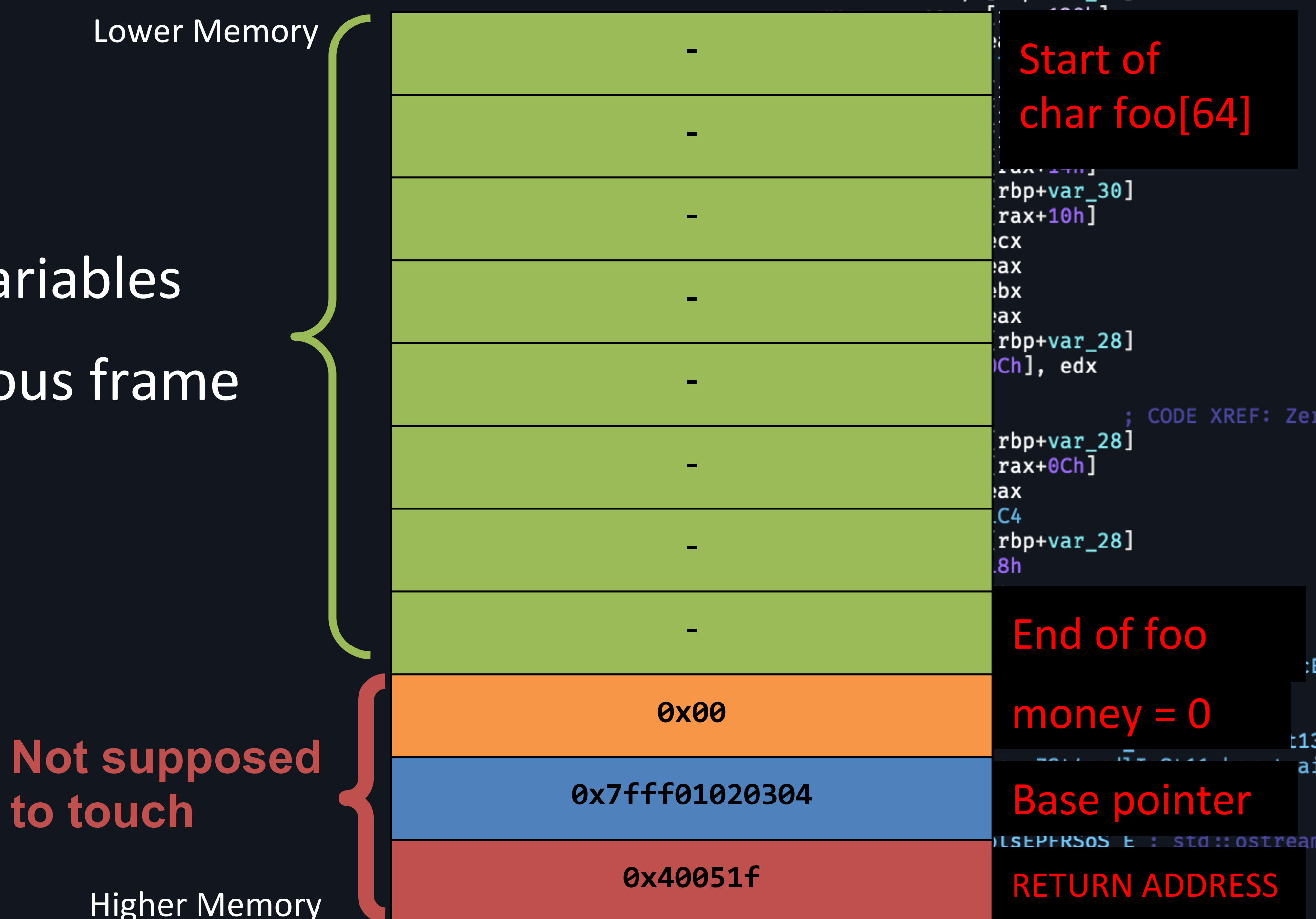
Stack Smashing 101

```
var_30          = qword ptr -30h
var_28          = qword ptr -28h
var_14          = dword ptr -14h

; __unwind {
                push    rbp
                mov     rbp, rsp
                push    rbx
                sub     rsp, 28h
                mov     [rbp+var_28], rdi
                mov     [rbp+var_30], rsi
                mov     rax, [rbp+var_28]
```

main() has a

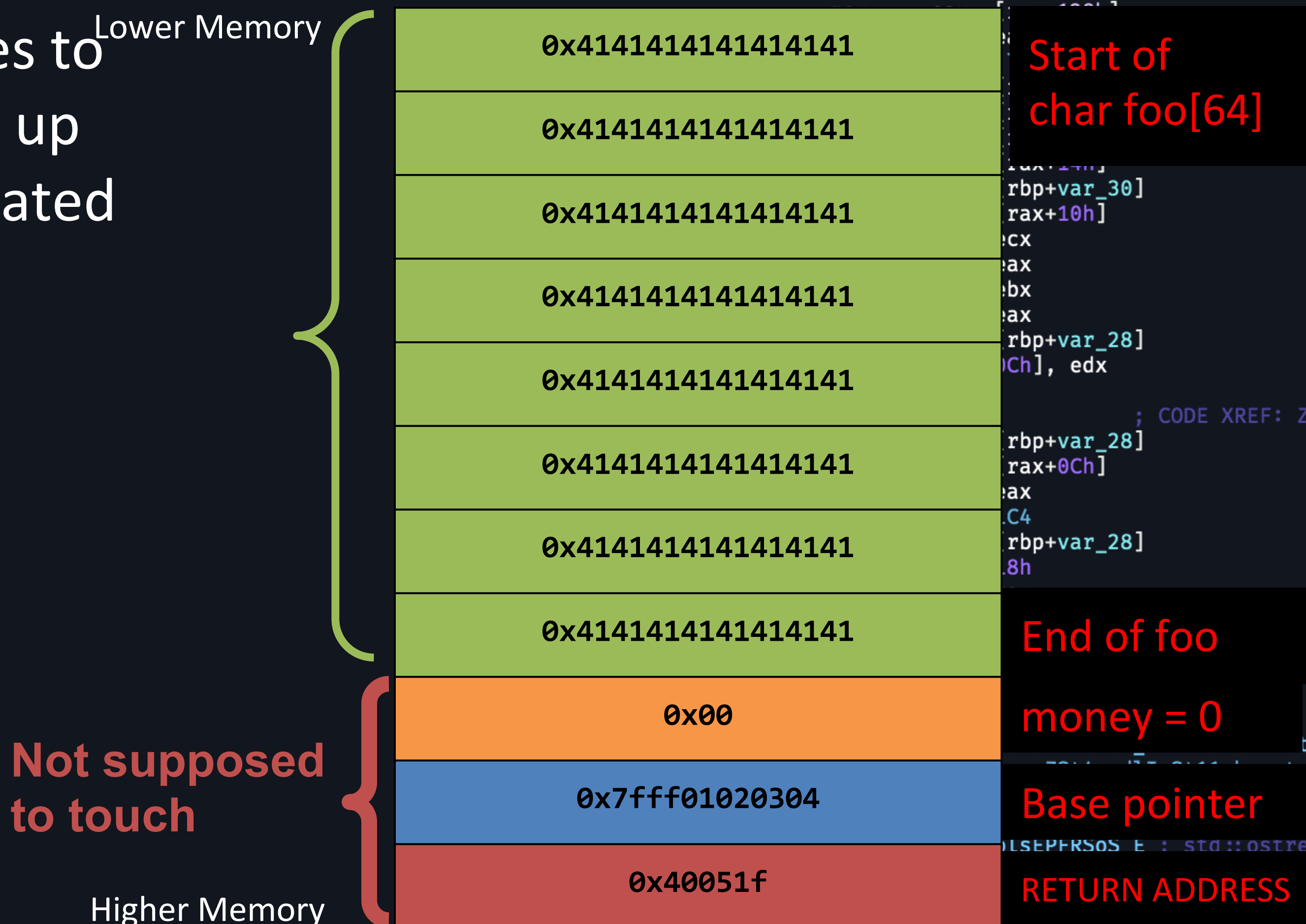
stack frame

- Contains local variables
- Pointer to previous frame
- Return address



Stack Smashing 101

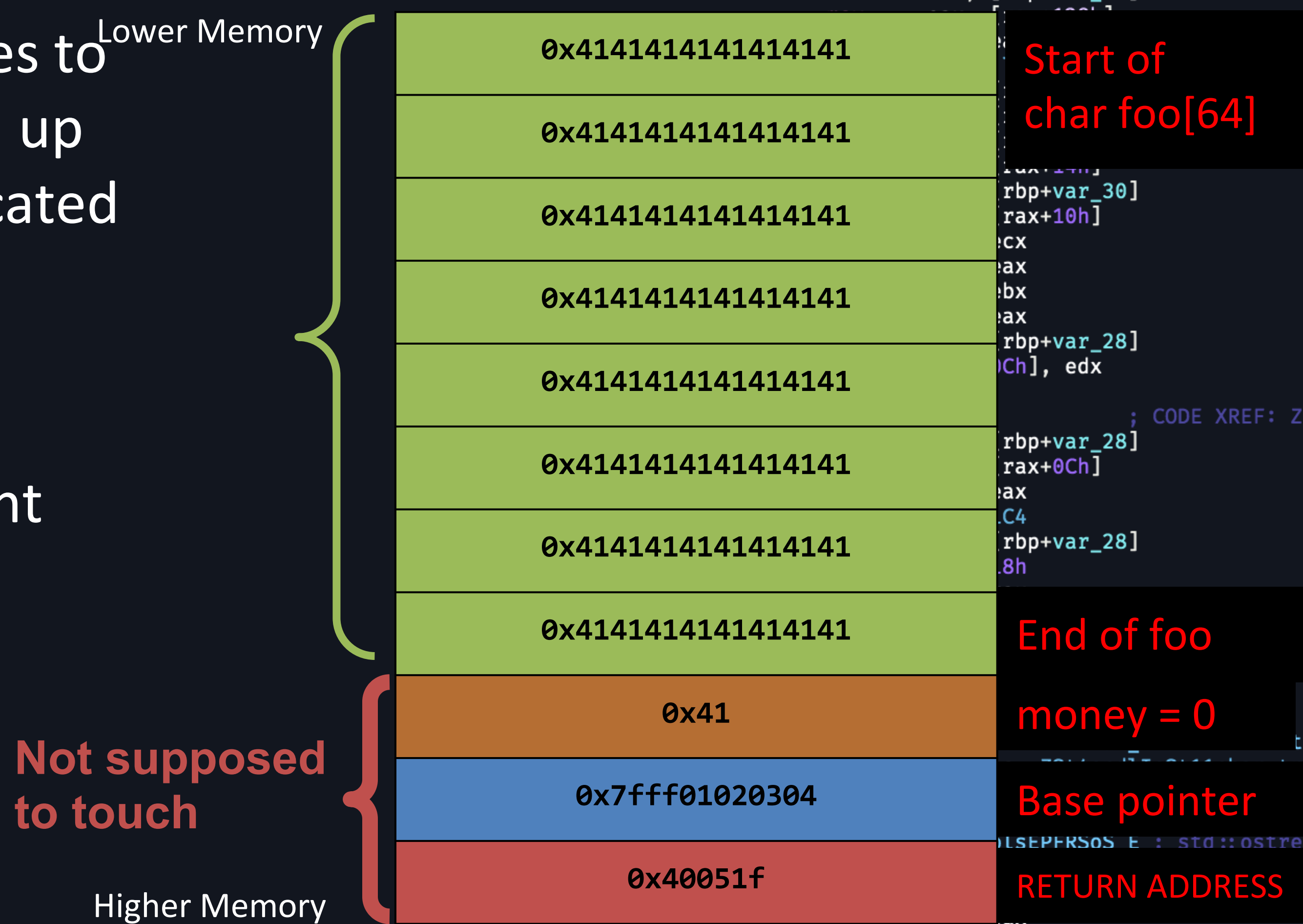
As gets() continues to read input, we fill up the 64 bytes allocated for buffer foo



Stack Smashing 101

As gets() continues to read input, we fill up the 64 bytes allocated for foo

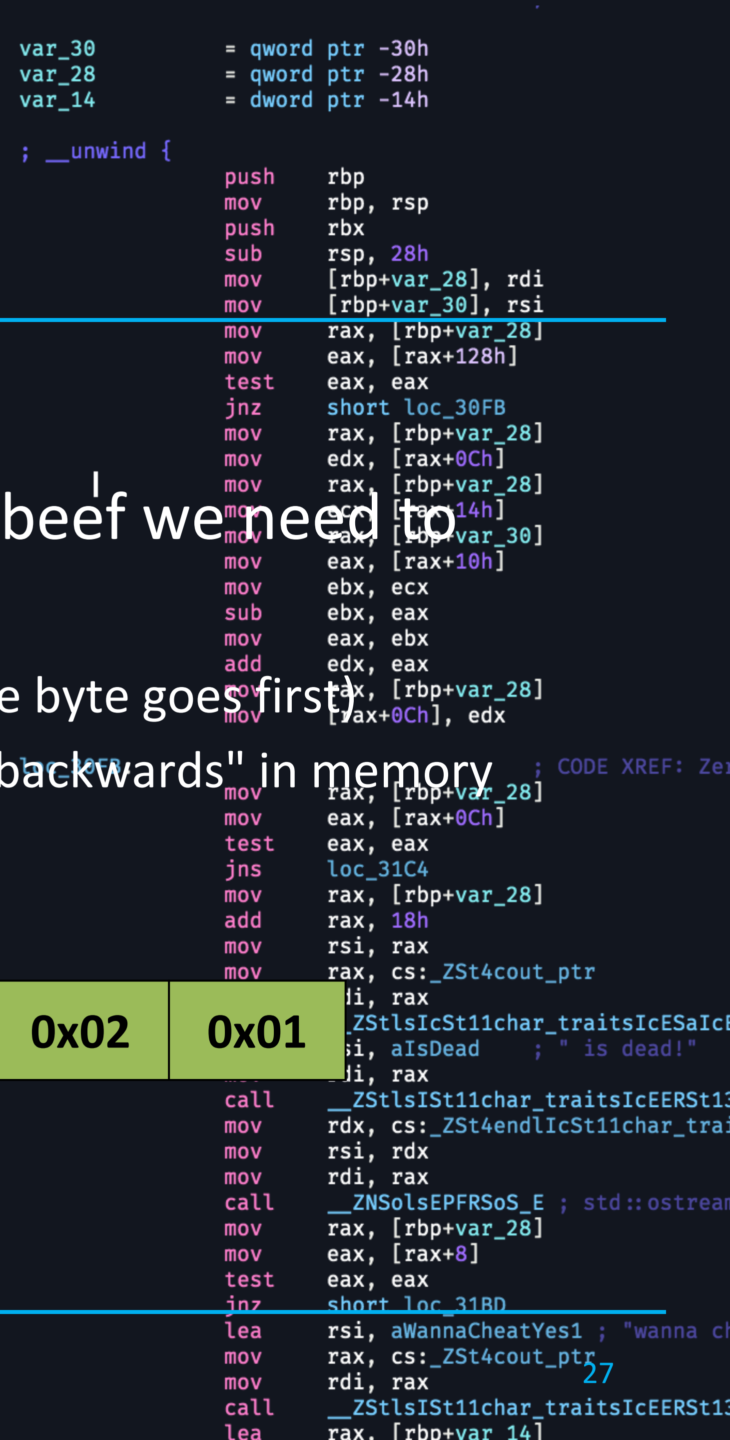
Go far enough, it corrupts important data!



Stack Smashing 101

- We can give ourselves money!
- If we want to set money to 0x1337beef we need to know:
  - Most x86 machines are little endian (little byte goes first)
  - Meaning the byte order for numbers is "backwards" in memory
  - 0x01020304 would be

0x04	0x03	0x02	0x01



Stack Smashing 201

- What else can we corrupt?
- What happens if you corrupt further? When does it segfault?
  - What was that about a return address?

```
var_30          = qword ptr -30h
var_28          = qword ptr -28h
var_14          = dword ptr -14h

; __unwind {
                push    rbp
                mov     rbp, rsp
                push    rbx
                sub     rsp, 28h
                mov     [rbp+var_28], rdi
                mov     [rbp+var_30], rsi
                mov     rax, [rbp+var_28]
                mov     eax, [rax+128h]
                test    eax, eax
                jnz     short loc_30FB
                mov     rax, [rbp+var_28]
                mov     edx, [rax+0Ch]
                mov     rax, [rbp+var_28]
                mov     ecx, [rax+14h]
                mov     rax, [rbp+var_30]
                mov     eax, [rax+10h]
                mov     ebx, ecx
                sub     ebx, eax
                mov     eax, ebx
                add     edx, eax
                mov     rax, [rbp+var_28]
                mov     [rax+0Ch], edx

loc_30FB:                               ; CODE XREF: Ze
                mov     rax, [rbp+var_28]
                mov     eax, [rax+0Ch]
                test    eax, eax
                jns     loc_31C4
                mov     rax, [rbp+var_28]
                add     rax, 18h
                mov     rsi, rax
                mov     rax, cs:_ZSt4cout_ptr
                mov     rdi, rax
                call    __ZStlsIcSt11char_traitsIcESaIcE
                lea     rsi, aIsDead    ; " is dead!"
                mov     rdi, rax
                call    __ZStlsISt11char_traitsIcEERSt13
                mov     rdx, cs:_ZSt4endlIcSt11char_trai
                mov     rsi, rdx
                mov     rdi, rax
                call    __ZNSolsEPFRSoS_E ; std::ostrean
                mov     rax, [rbp+var_28]
                mov     eax, [rax+8]
                test    eax, eax
                jnz     short loc_31BD
                lea     rsi, aWannaCheatYes1 ; "wanna ch
                mov     rax, cs:_ZSt4cout_ptr
                mov     rdi, rax
                call    __ZStlsISt11char_traitsIcEERSt13
                lea     rax, [rbp+var_14]
```

Stack Smashing 201

```c
int func() {
        puts("Hello World");
        return 17;
}
int main() {
        int res = func();
        return 0;
}
```

When **func()** is called, runtime stores the **return address** on the stack (i.e., the address of the instruction that immediately follows **call func** in **main**)

```
var_30          = qword ptr -30h
var_28          = qword ptr -28h
var_14          = dword ptr -14h

; __unwind {
                push    rbp
                mov     rbp, rsp
                push    rbx
                sub     rsp, 28h
                mov     [rbp+var_28], rdi
                mov     [rbp+var_30], rsi
                mov     rax, [rbp+var_28]
                mov     eax, [rax+128h]
                test    eax, eax
                jnz     short loc_30FB
                mov     rax, [rbp+var_28]
                mov     edx, [rax+0Ch]
                mov     rax, [rbp+var_28]
                mov     ecx, [rax+14h]
                mov     rax, [rbp+var_30]
                mov     eax, [rax+10h]
                mov     ebx, ecx
                sub     ebx, eax
                mov     eax, ebx
                add     edx, eax
                mov     rax, [rbp+var_28]
                mov     [rax+0Ch], edx

loc_30FB:                               ; CODE XREF: Ze
                mov     rax, [rbp+var_28]
                mov     eax, [rax+0Ch]
                test    eax, eax
                jns     loc_31C4
                mov     rax, [rbp+var_28]
                add     rax, 18h
                mov     rsi, rax
                mov     rax, cs:_ZSt4cout_ptr
                mov     rdi, rax
                call    __ZStlsIcSt11char_traitsIcESaIcE
                lea     rsi, aIsDead    ; " is dead!"
                mov     rdi, rax
                call    __ZStlsISt11char_traitsIcEERSt13
                mov     rdx, cs:_ZSt4endlIcSt11char_trai
                mov     rsi, rdx
                mov     rdi, rax
                call    __ZNSolsEPFRSoS_E ; std::ostream
                mov     rax, [rbp+var_28]
                mov     eax, [rax+8]
                test    eax, eax
                jnz     short loc_31BD
                lea     rsi, aWannaCheatYes1 ; "wanna ch
                mov     rax, cs:_ZSt4cout_ptr
                mov     rdi, rax
                call    __ZStlsISt11char_traitsIcEERSt13
                lea     rax, [rbp+var_14]
```

Stack Smashing 201

Before the call:

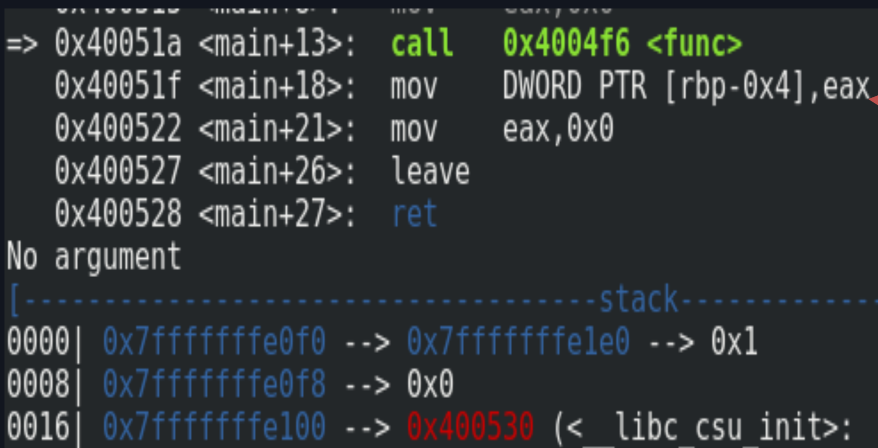
RIP main

```
=> 0x40051a <main+13>:  call   0x4004f6 <func>
   0x40051f <main+18>:  mov    DWORD PTR [rbp-0x4],eax
   0x400522 <main+21>:  mov    eax,0x0
   0x400527 <main+26>:  leave
   0x400528 <main+27>:  ret
No argument
[------------------------------------stack-------------
0000| 0x7fffffffe0f0 --> 0x7fffffffe1e0 --> 0x1
0008| 0x7fffffffe0f8 --> 0x0
0016| 0x7fffffffe100 --> 0x400530 (<__libc_csu_init>:
```

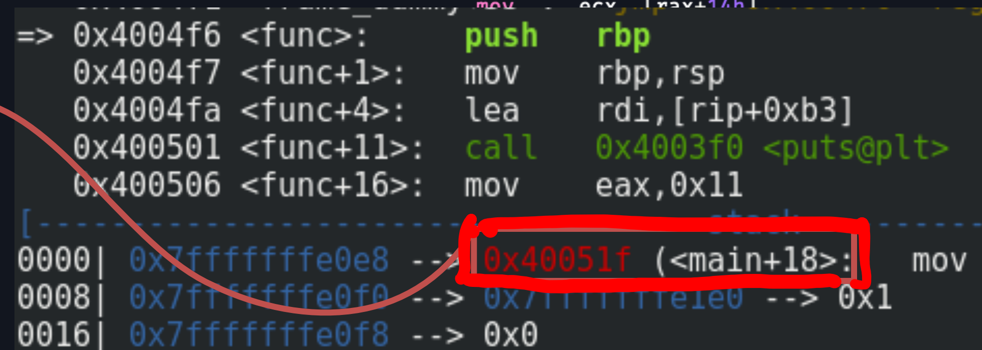
```
var_30          = qword ptr -30h
var_28          = qword ptr -28h
var_14          = dword ptr -14h

; __unwind {
push    rbp
mov     rbp, rsp
push    rbx
sub     rsp, 28h
mov     [rbp+var_28], rdi
mov     [rbp+var_30], rsi
mov     rax, [rbp+var_28]
mov     eax, [rax+128h]
test    eax, eax
jnz     short loc_30FB
mov     rax, [rbp+var_28]
mov     edx, [rax+0Ch]
mov     rax, [rbp+var_28]
mov     ecx, [rax+14h]
mov     rax, [rbp+var_30]
mov     eax, [rax+10h]
mov     ebx, ecx
sub     ebx, eax
mov     eax, ebx
add     edx, eax
mov     rax, [rbp+var_28]
mov     [rax+0Ch], edx

; CODE XREF: Ze
mov     rax, [rbp+var_28]
mov     eax, [rax+0Ch]
test    eax, eax
jns     loc_31C4
mov     rax, [rbp+var_28]
add     rax, 18h
mov     rsi, rax
mov     rax, cs:_ZSt4cout_ptr
mov     rdi, rax
call    __ZStlsIcSt11char_traitsIcESaIcE
lea     rsi, aIsDead    ; " is dead!"
mov     rdi, rax
call    __ZStlsISt11char_traitsIcEERSt13
mov     rdx, cs:_ZSt4endlIcSt11char_trai
mov     rsi, rdx
mov     rdi, rax
call    __ZNSolsEPFRSoS_E ; std::ostrean
mov     rax, [rbp+var_28]
mov     eax, [rax+8]
test    eax, eax
jnz     short loc_31BD
lea     rsi, aWannaCheatYes1 ; "wanna ch
mov     rax, cs:_ZSt4cout_ptr
mov     rdi, rax
call    __ZStlsISt11char_traitsIcEERSt13
lea     rax, [rbp+var_14]
```

Stack Smashing 201

Before the call:



After the call:

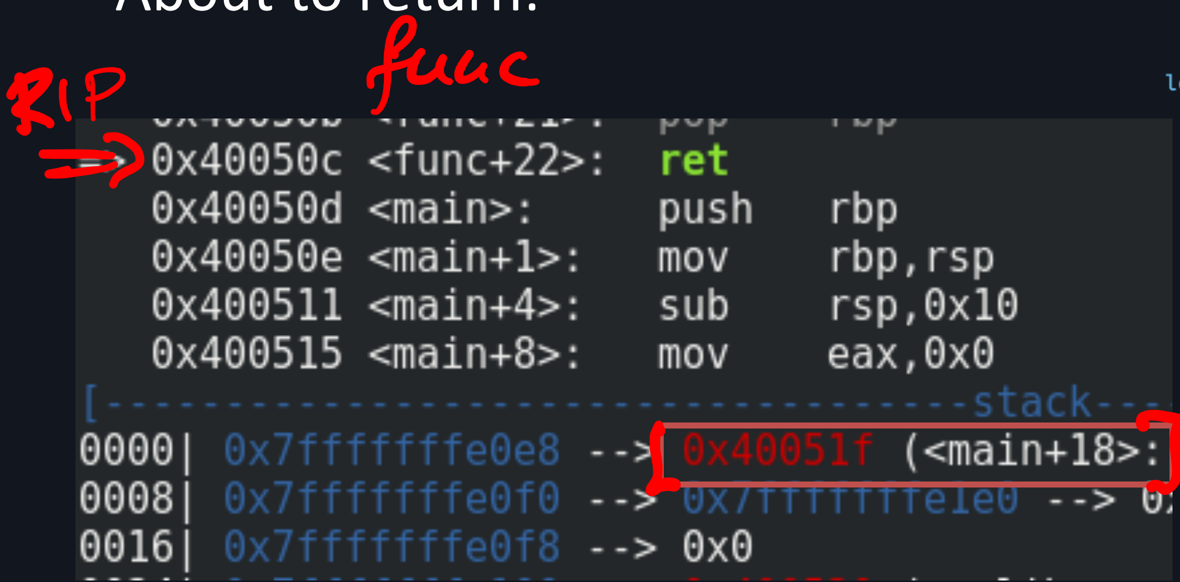


Return address points back to where it left off in main

Stack Smashing 201

Returning just takes whatever is on the top of the stack, and jumps there, equivalently: `pop rip`

About to return:



Stack Smashing 201

Returning just takes whatever is on the top of the stack, and jumps there, equivalently: `pop rip`

About to return:

```
=> 0x40050c <func+22>:   ret
   0x40050d <main>:      push   rbp
   0x40050e <main+1>:    mov    rbp,rsp
   0x400511 <main+4>:    sub    rsp,0x10
   0x400515 <main+8>:    mov    eax,0x0
[--------------------------------stack---
0000| 0x7fffffffe0e8 --> 0x40051f (<main+18>:
0008| 0x7fffffffe0f0 --> 0x7fffffffe1e0 --> 0x
0016| 0x7fffffffe0f8 --> 0x0
```

RIP

Returned back to main:

main:

```
=> 0x40051f <main+18>:   mov    DWORD PTR [rbp-0x4],eax
   0x400522 <main+21>:   mov    eax,0x0
   0x400527 <main+26>:   leave
   0x400528 <main+27>:   ret
   0x400529:    nop    DWORD PTR [rax+0x0]
[---------------------------------stack-----------
0000| 0x7fffffffe0f0 --> 0x7fffffffe1e0 --> 0x1
0008| 0x7fffffffe0f8 --> 0x0
0016| 0x7fffffffe100 --> 0x400530 (<__libc_csu_init>:
```

Stack Smashing 201

Returning just takes whatever is on the top of the stack, and jumps there, equivalently: `pop rip`

About to return:

What if we change this???

```
=> 0x40050c <func+22>:   ret
   0x40050d <main>:      push   rbp
   0x40050e <main+1>:    mov    rbp,rsp
   0x400511 <main+4>:    sub    rsp,0x10
   0x400515 <main+8>:    mov    eax,0x0
[-----------------------------stack-----
0000| 0x7fffffffe0e8 --> 0x40051f (<main+18>:
0008| 0x7fffffffe0f0 --> 0x7fffffffe1e0 --> 0x
0016| 0x7fffffffe0f8 --> 0x0
```

Returned back to main:

```
=> 0x40051f <main+18>:   mov    DWORD PTR [rbp-0x4],eax
   0x400522 <main+21>:   mov    eax,0x0
   0x400527 <main+26>:   leave
   0x400528 <main+27>:   ret
   0x400529:     nop    DWORD PTR [rax+0x0]
[-----------------------------stack-----------------
0000| 0x7fffffffe0f0 --> 0x7fffffffe1e0 --> 0x1
0008| 0x7fffffffe0f8 --> 0x0
0016| 0x7fffffffe100 --> 0x400530 (<__libc_csu_init>:
```

```
var_30          = qword ptr -30h
var_28          = qword ptr -28h
var_14          = dword ptr -14h

; __unwind {
                push    rbp
                mov     rbp, rsp
                push    rbx
                sub     rsp, 28h
                mov     [rbp+var_28], rdi
                mov     [rbp+var_30], rsi
                mov     rax, [rbp+var_28]
                mov     eax, [rax+128h]
                test    eax, eax
                jz      short loc_30FB
                mov     rax, [rbp+var_28]
                mov     edx, [rax+0Ch]
                mov     rax, [rbp+var_28]
                mov     ecx, [rax+14h]
                mov     rax, [rbp+var_30]
                mov     eax, [rax+10h]
                mov     ebx, ecx
                sub     ebx, eax
                mov     eax, ebx
                add     edx, eax
                mov     rax, [rbp+var_28]
                mov     [rax+0Ch], edx

loc_30FB:                               ; CODE XREF: Ze
                mov     rax, [rbp+var_28]
                mov     eax, [rax+0Ch]
                call    __ZStlsISt11char_traitsIcEERSt13
                mov     rdx, cs:_ZSt4endlIcSt11char_trai
                mov     rsi, rdx
                mov     rdi, rax
                call    __ZNSolsEPFRSoS_E ; std::ostream
                mov     rax, [rbp+var_28]
                mov     eax, [rax+8]
                test    eax, eax
                jnz     short loc_31BB
                lea     rsi, aWannaCheatYes1 ; "wanna ch
                mov     rax, cs:_ZSt4cout_ptr
                mov     rdi, rax
                call    __ZStlsISt11char_traitsIcEERSt13
                lea     rax, [rbp+var_14]
```

?!?!?!?

```
var_30          = qword ptr -30h
var_28          = qword ptr -28h
var_14          = dword ptr -14h

; __unwind {
                push    rbp
                mov     rbp, rsp
                push    rbx
                sub     rsp, 28h
                mov     [rbp+var_28], rdi
                mov     [rbp+var_30], rsi
                mov     rax, [rbp+var_28]
```

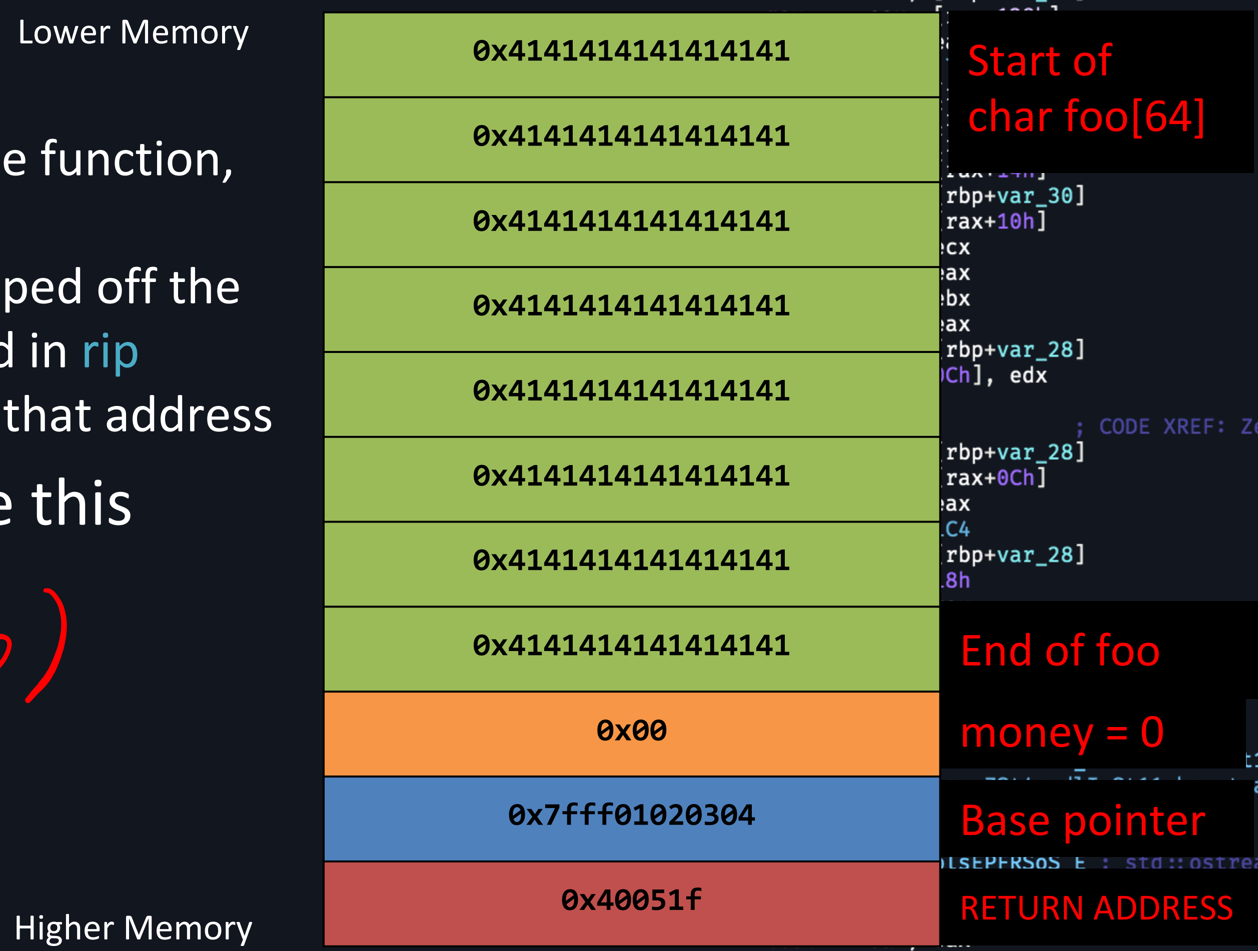
Stack Smashing 201

Without corruption:

- At the end of the function, it returns
- 0x40051f is popped off the stack and stored in rip
- Control goes to that address

We want to change this

gets(foo)




```
                lea     rsi, aWannaCheatYes1 ; "wanna ch
                mov     rax, cs:_ZSt4cout_ptr
                mov     rdi, rax
                call    __ZStlsISt11char_traitsIcEERSt13
```

```
var_30          = qword ptr -30h
var_28          = qword ptr -28h
var_14          = dword ptr -14h

; __unwind {
                push    rbp
                mov     rbp, rsp
                push    rbx
                sub     rsp, 28h
                mov     [rbp+var_28], rdi
                mov     [rbp+var_30], rsi
                mov     rax, [rbp+var_28]
```

Stack Smashing 201

Lower Memory

Higher Memory

Corrupted:

- At the end of the function, it returns
- 0x4141414141414141 is popped off the stack and stored in rip
- Control goes to that address
- but it's invalid memory...

Segmentation fault

MALICIOUS ADDRESS

```
jnz     short loc_31BD
lea     rsi, aWannaCheatYes1 ; "wanna ch
mov     rax, cs:_ZSt4cout_ptr
mov     rdi, rax
call    __ZStlsISt11char_traitsIcEERSt13
lea     rax, [rbp+var_14]
```

Lecture Outline

- Notion of binding time
- Object lifetime and storage management

- An aside: Stack Smashing 101

- Scoping
  - Static scoping
  - Dynamic scoping

Scoping

- In most languages the same variable name can be used to denote different memory locations
- Scoping rules: map variable to declaration (location)
- Scope: region of program text where a declaration is visible
- Most languages use static scoping
  - Mapping from variable to declaration (or location) is made at compile time
- Block-structured programming languages
  - Nested subroutines (Pascal, ML, Scheme, etc.)
  - Nested blocks (C, C++ { … })

Static Scoping in Block Structured Programming Languages

- Also known as lexical scoping
- Block structure and nesting of blocks gives rise to the closest nested scope rule
  - There are local variable declaration within a block
  - A block inherits variable declarations from enclosing blocks
  - Local declarations take precedence over inherited ones
    - Hole in scope of inherited declaration
    - In other words, inherited declaration is hidden
- Lookup for non-local variables proceeds from inner to outer enclosing blocks

Example - Block Structured PL

```
main
    a, b, c: integer
    procedure P
        c: integer          main.a, main.b, P.c     main.P, P.S, main.R
        procedure S
            c, d: integer       main.a,main.b,S.c,S.d   main.P, P.S, S.R
            procedure R
            …                   S.R,P.S,main.P
            end R
            R()
        end S
        R()
        S()
    end P
    procedure R
        a: integer          R.a, main.b, main.c     main.R, main.P
        … = a, b, c
    end R
    P()
end main
```

Nested block structure allows locally defined variables and subroutines

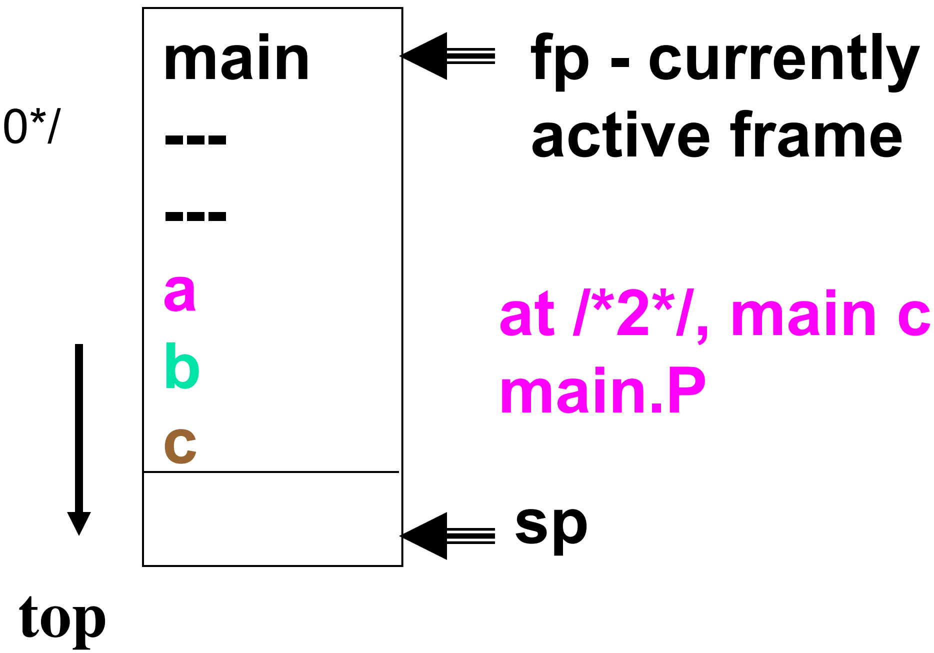
```
main
    a, b, c: integer
    procedure P
        c: integer
        procedure S
            c, d: integer
            procedure R
                …
            end R
            R()
        end S
        R()
        S()
    end P
    procedure R
        a: integer
        … = a, b, c
    end R
    P()
end main
```

Rule: a variable is visible if it is declared in its own block or in a textually surrounding block **and** it is not 'hidden' by a binding in a closer block (i.e., there is no hole in scope)

Example with Frames

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
        c: integer
        procedure S /*8*/
            c, d: integer
            procedure R /*10*/
            …
            end R /*11*/
            R()  /*9*/
        end S /*12*/
        R() /*4*/
        S() /*7*/
    end P /*13*/
    procedure R /*5*/
        a: integer
        … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

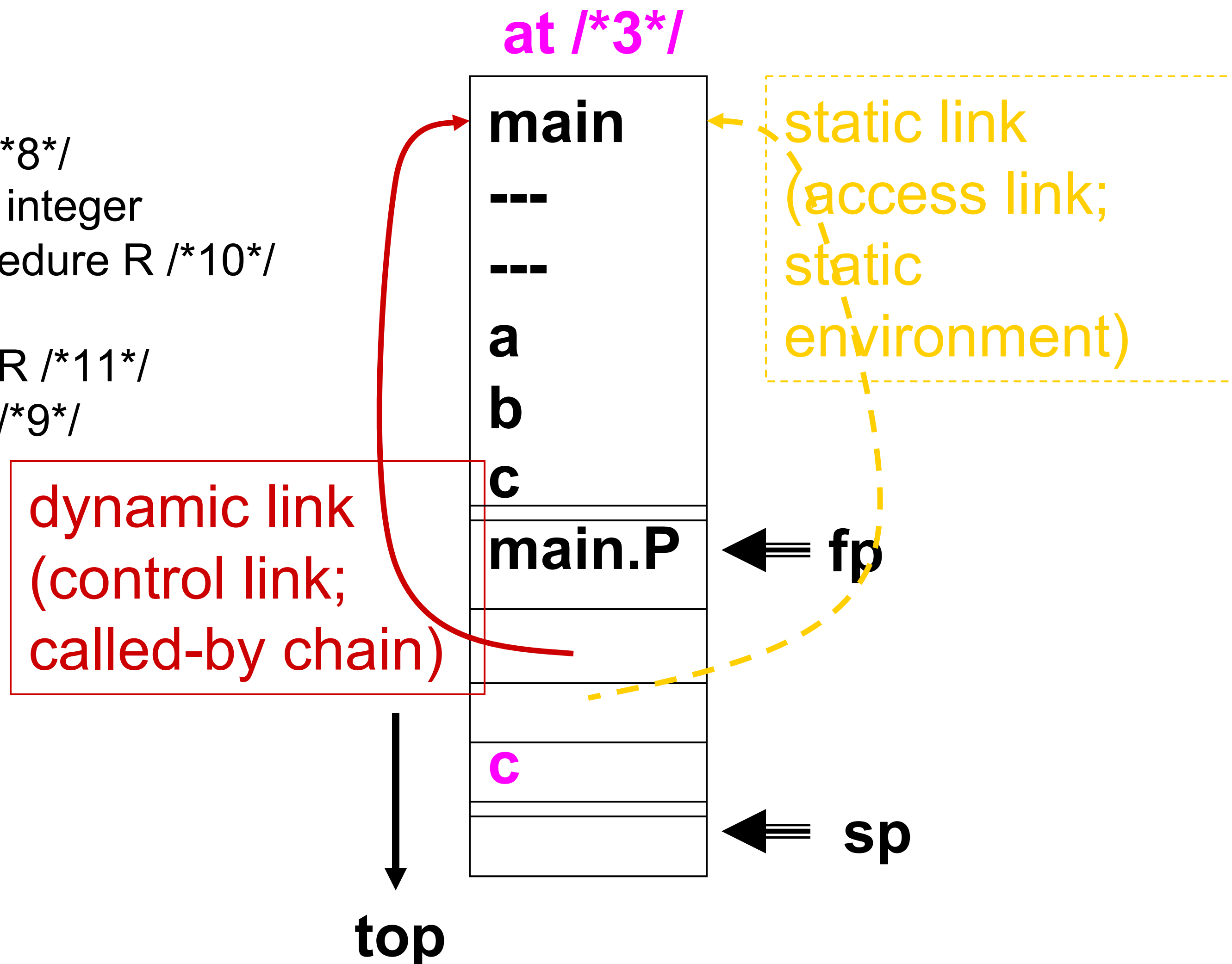
at /*1*/



at /*2*/, main calls main.P

Example

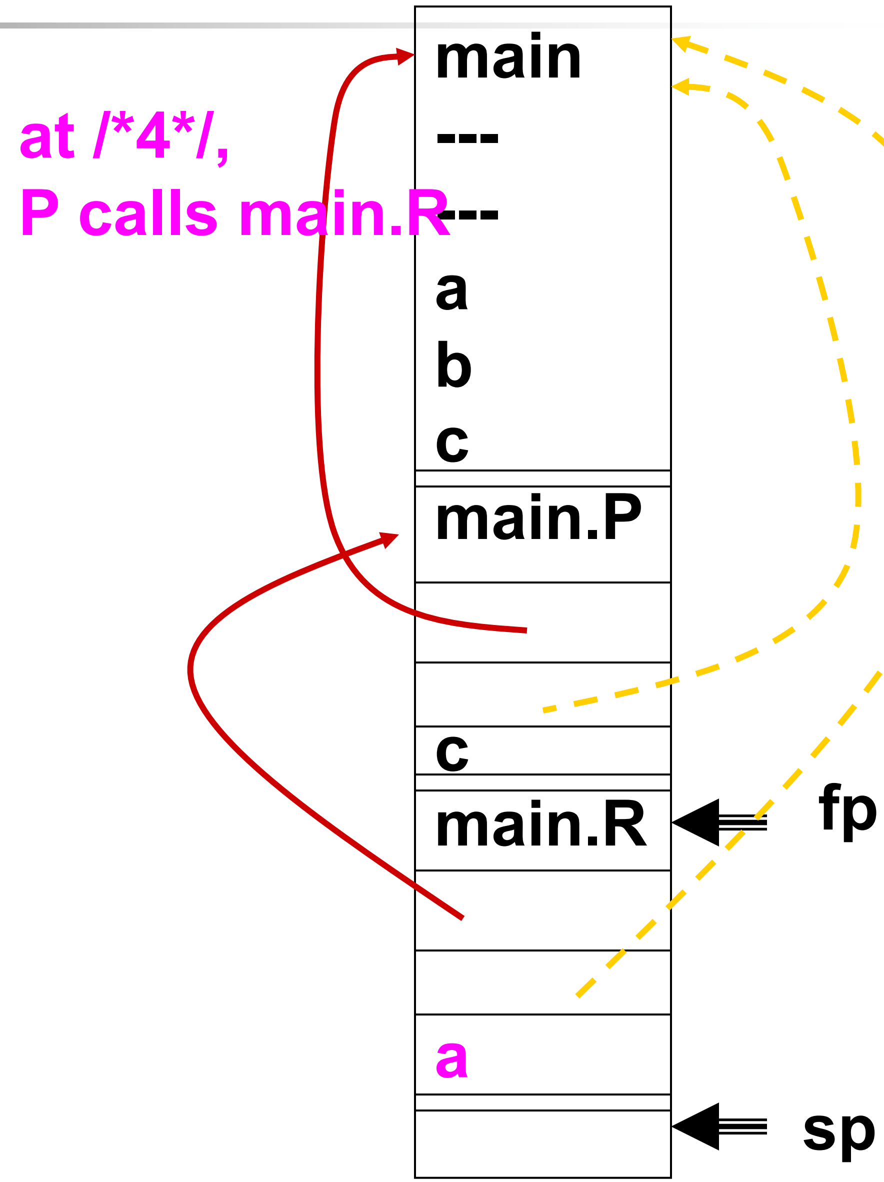
```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
        c: integer
        procedure S /*8*/
            c, d: integer
            procedure R /*10*/
            …
            end R /*11*/
            R()  /*9*/
        end S /*12*/
        R() /*4*/
        S() /*7*/
    end P /*13*/
    procedure R /*5*/
        a: integer
        … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```



Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
        c: integer
        procedure S /*8*/
            c, d: integer
            procedure R /*10*/
            …
            end R /*11*/
            R()  /*9*/
        end S /*12*/
        R() /*4*/
        S() /*7*/
    end P /*13*/
    procedure R /*5*/
        a: integer
        … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

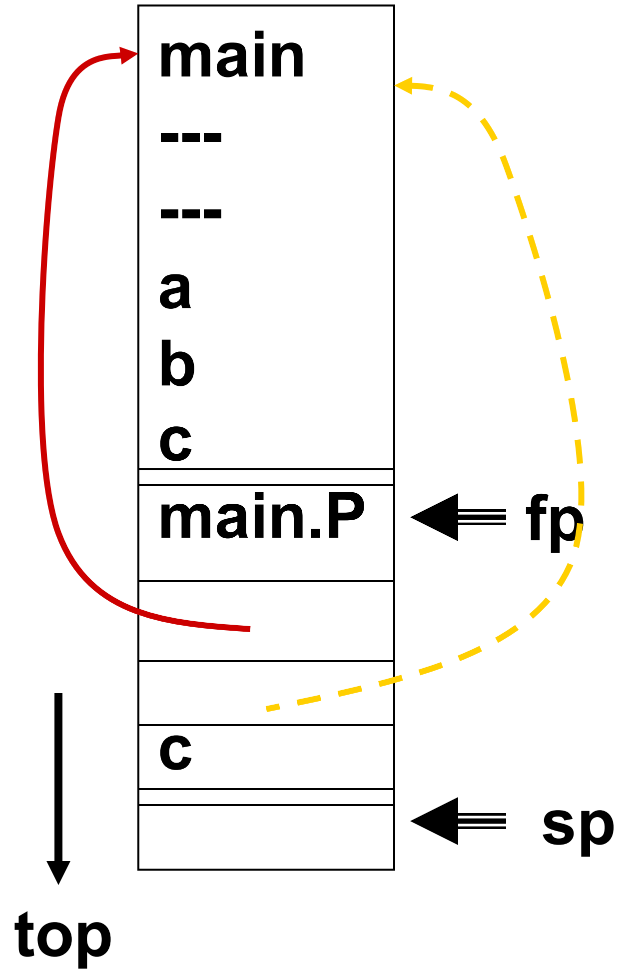
at /*5*/



Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
        c: integer
        procedure S /*8*/
            c, d: integer
            procedure R /*10*/
            …
            end R /*11*/
            R()  /*9*/
        end S /*12*/
        R() /*4*/
        S() /*7*/
    end P /*13*/
    procedure R /*5*/
        a: integer
        … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

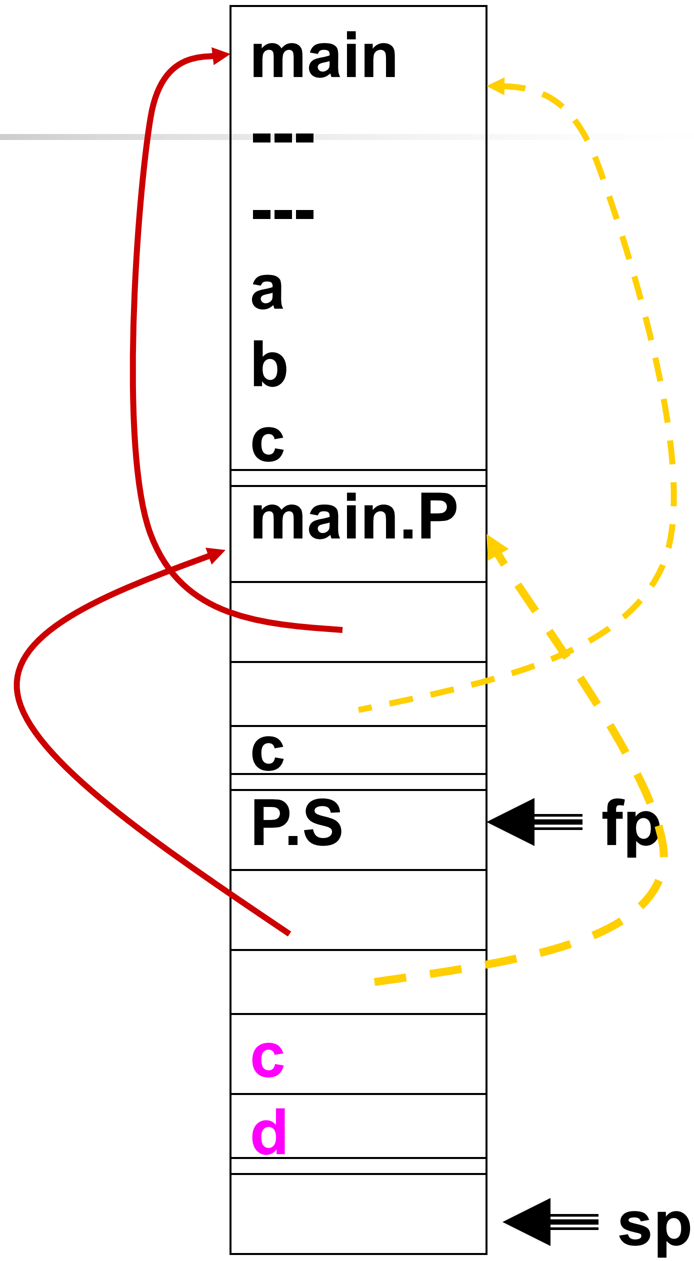
at /*6*/ main.R exits
sp ← fp
fp ← old fp
in main.R's frame



Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
        c: integer
        procedure S /*8*/
            c, d: integer
            procedure R /*10*/
            …
            end R /*11*/
            R()  /*9*/
        end S /*12*/
        R() /*4*/
        S() /*7*/
    end P /*13*/
    procedure R /*5*/
        a: integer
        … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

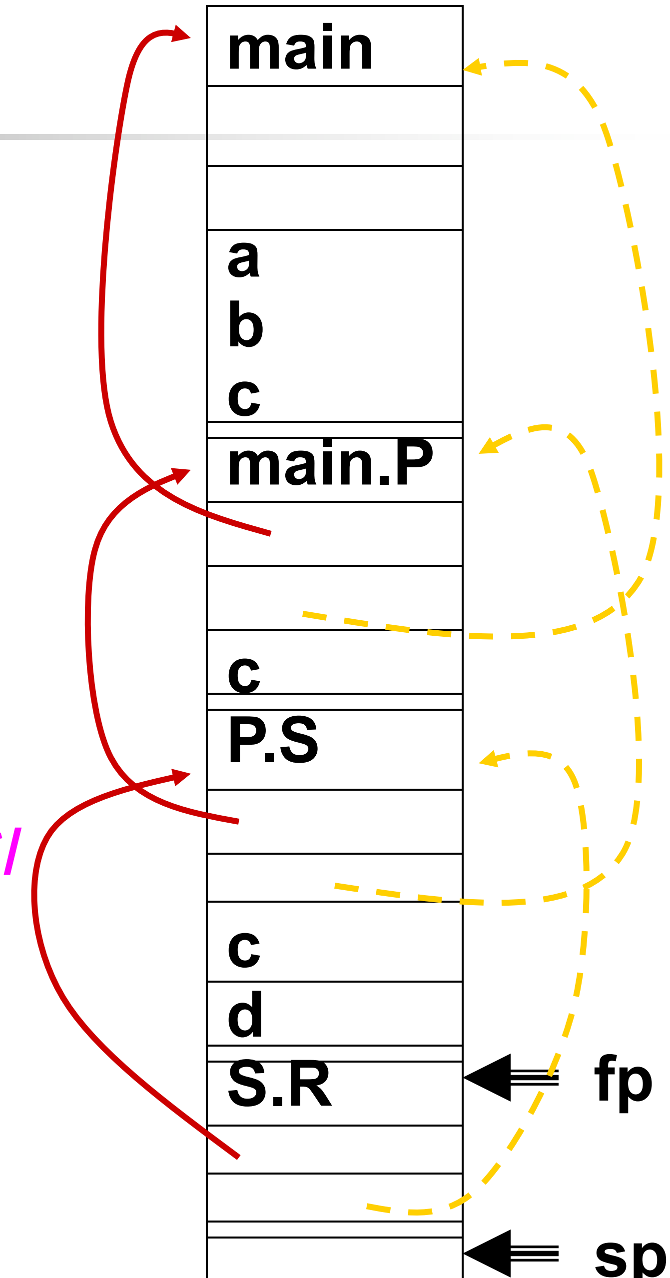
at /*7*/,
P calls P.S;
at /*8*/:



Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
        c: integer
        procedure S /*8*/
            c, d: integer
            procedure R /*10*/
            …
            end R /*11*/
            R()  /*9*/
        end S /*12*/
        R() /*4*/
        S() /*7*/
    end P /*13*/
    procedure R /*5*/
        a: integer
        … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

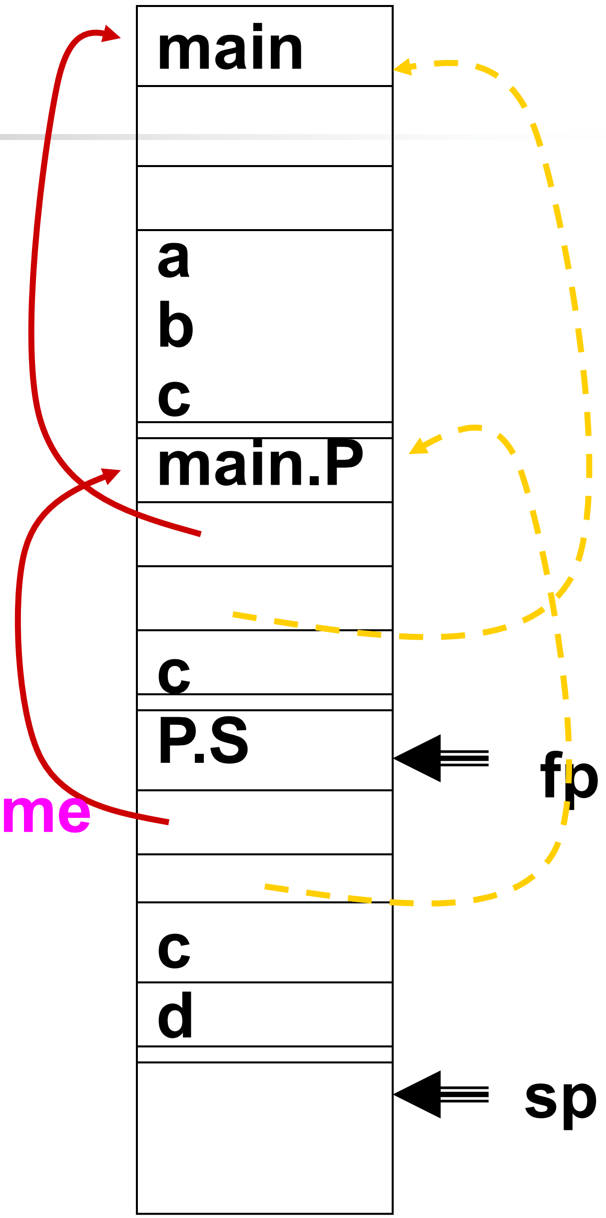
at /*9*/ S calls in S.R; at /*10*/



Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
        c: integer
        procedure S /*8*/
            c, d: integer
            procedure R /*10*/
            …
            end R /*11*/
            R()  /*9*/
        end S /*12*/
        R() /*4*/
        S() /*7*/
    end P /*13*/
    procedure R /*5*/
        a: integer
        … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

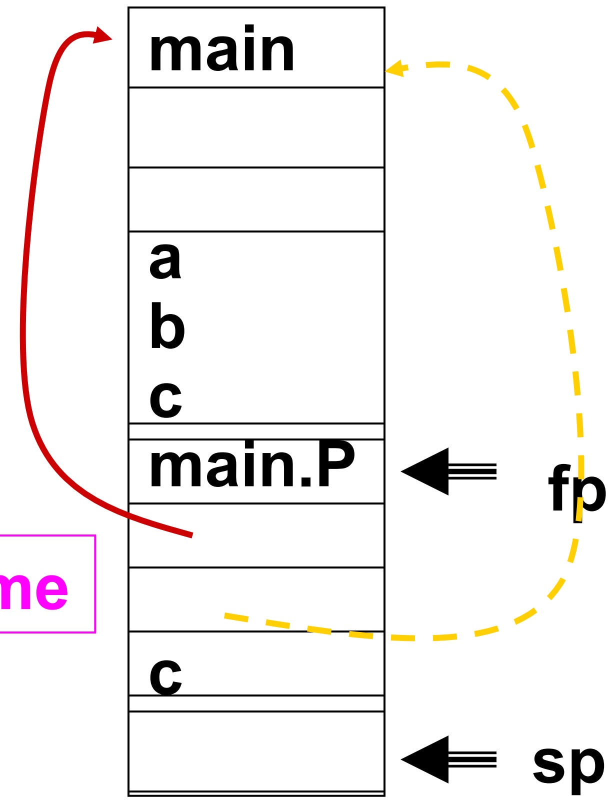
/*11*/ pop S.R' s frame



Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
        c: integer
        procedure S /*8*/
            c, d: integer
            procedure R /*10*/
            …
            end R /*11*/
            R()  /*9*/
        end S /*12*/
        R() /*4*/
        S() /*7*/
    end P /*13*/
    procedure R /*5*/
        a: integer
        … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

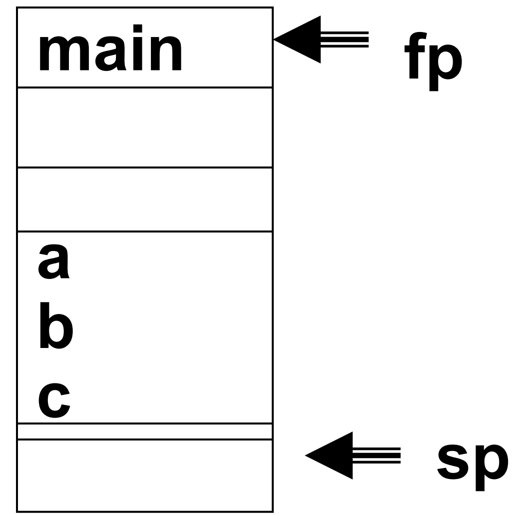
/*12*/pop S’s frame



Example

```
main
    a, b, c: integer /*1*/
    procedure P    /*3*/
        c: integer
        procedure S /*8*/
            c, d: integer
            procedure R /*10*/
            …
            end R /*11*/
            R()  /*9*/
        end S /*12*/
        R() /*4*/
        S() /*7*/
    end P /*13*/
    procedure R /*5*/
        a: integer
        … = a, b, c
    end R /*6*/
    P() /*2*/ …
end main /*14*/
```

at /*13*/



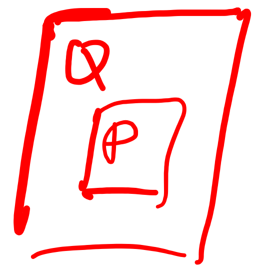
/*13*/ pop P's frame
/*14*/ pop main's frame
so that sp $\leftarrow$ fp

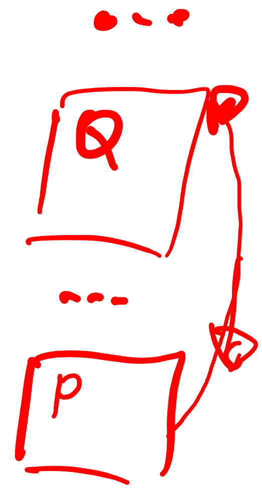
Static Link vs. Dynamic Link

- Static link for a frame of subroutine P points to the most recent frame of P's lexically enclosing subroutine
  - Bookkeeping required to maintain the static link
  - If subroutine P is enclosed *k*-levels deep from main, then the length of the static chain that begins at a frame for P, is *k*
  - To find non-local variables, follow static chain
- Dynamic link points to the caller frame, this is essentially old fp stored on frame

Observations

- Static link of a subroutine P points to the frame of the most recent invocation of subroutine Q, where Q is the lexically enclosing subroutine of P





- Dynamic link may point to a different subroutine’s frame, depending on where the subroutine is called from

An Important Note!

- For now, we assume languages that do not allow subroutines to be passed as arguments or returned from other subroutines, i.e., subroutines (functions) are third-class values
  - When subroutines (functions) are third-class values, it is guaranteed the static reference environment is on the stack
  - I.e., a subroutine cannot outlive its reference environment

An Important Note!

- Static scoping rules become more involved in languages that allow subroutines to be passed as arguments and returned from other subroutines, i.e., subroutines (functions) are first class values

- We will return to scoping later during our discussion of functional programming languages

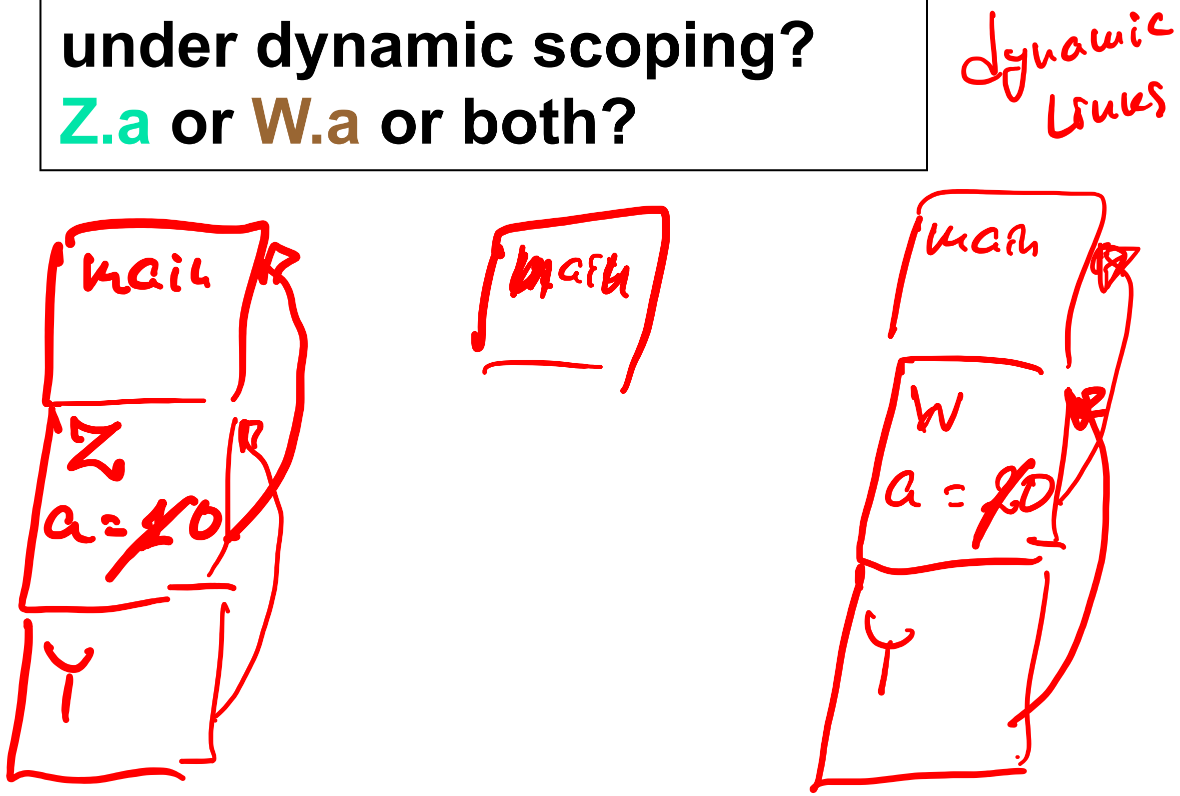
Dynamic Scoping

- Allows for local variable declaration
- Inherits non-local variables from subroutines that are live when current subroutine is invoked
  - Use of variable is resolved to the declaration of that variable in the most recently invoked and not yet terminated frame. I.e., lookup proceeds from closest predecessor on stack to furthest
  - (old) Lisp, APL, Snobol, Perl

Example

```
main
    procedure Z
        a: integer
        a := 1
        Y()
        output a
    end Z
    procedure W
        a: integer
        a := 2
        Y()
        output a
    end W
    procedure Y
        a := 0 /*1*/
    end Y
    Z()
    W()
end main
```

Which a is modified at /*1*/ under dynamic scoping? Z.a or W.a or both?



Example

```
main
    procedure Z
        a: integer
        a := 1
        Y()
        output a
    end Z
    procedure W
        a: integer
        a := 2
        Y()
        output a
    end W
    procedure Y
        a := 0; /*1*/
    end Y
    Z()
    W()
end main
```

main calls Z,
Z calls Y,
Y sets Z.a to 0.

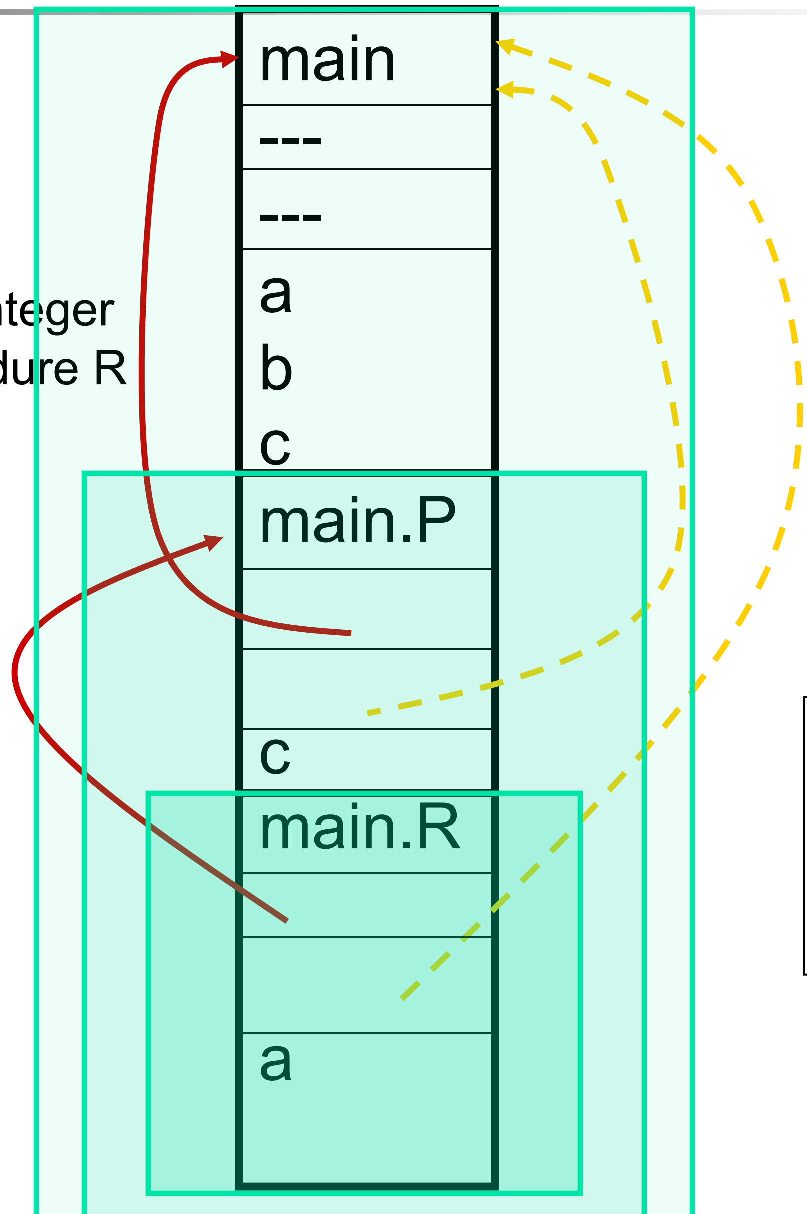
Example

```
main
    procedure Z
        a: integer
        a := 1
        Y()
        output a
    end Z
    procedure W
        a: integer
        a := 2
        Y()
        output a
    end W
    procedure Y
        a := 0; /*1*/
    end Y
    Z()
    W()
end main
```

main calls W,
W calls Y,
Y sets W.a to 0.

Static vs. Dynamic Scoping

```
main
    a, b, c: integer
    procedure P
        c: integer
        procedure S
            c, d: integer
            procedure R
            …
            end R
            R()
        end S
        R()
        S()
    end P
    procedure R
        a: integer
        … = a, b, c
    end R
    P()
end main
```



Static Scoping:
- a bound to R.a,
- b to main.b,
- c to main.c

Dynamic Scoping:
- a bound to R.a,
- b to main.b,
- c to P.c

Dynamic Scoping

- Dynamic scoping is considered a bad idea. Why?

- More on static and dynamic scoping to come!

The End