Checkin 26

In what direction does the stack grow?

Checkin 26

In what direction does the stack grow?

Administrivia

Announcements

University of Kansas | Drew Davidson



Activation Records

Previously

Memory Layout

Memory Layout

- Static allocation
- The heap and the stack



Architecture

The Memory Sections Concept

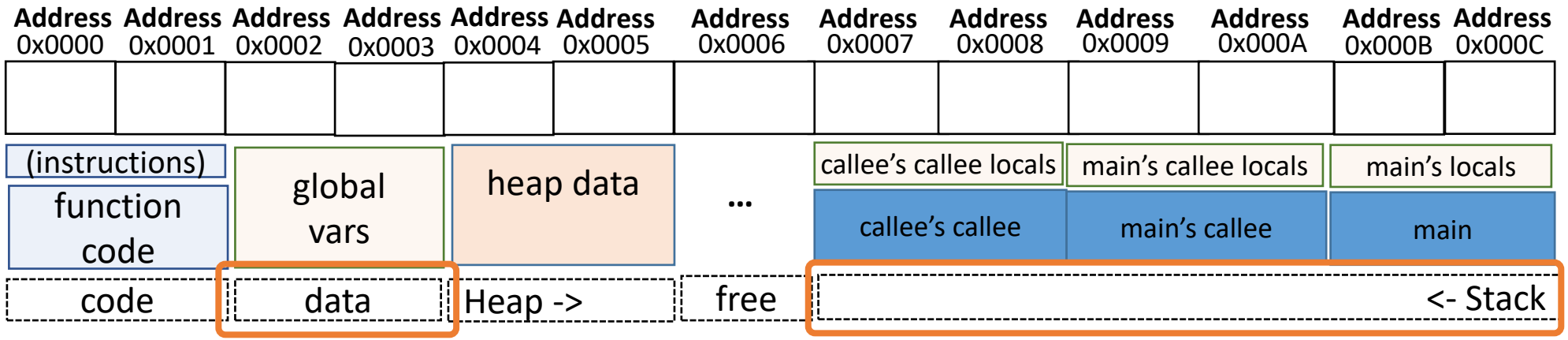
Review: Memory Layout

Let's do a quick recursion example

Memory Zones for different uses

- Text section (assembled opcodes of the program)
- Data section (global variables at hard-coded locations)
- Heap (runtime-allocated memory – malloc and new)
- Stack (function locals and bookkeeping)
- Free (unallocated space)

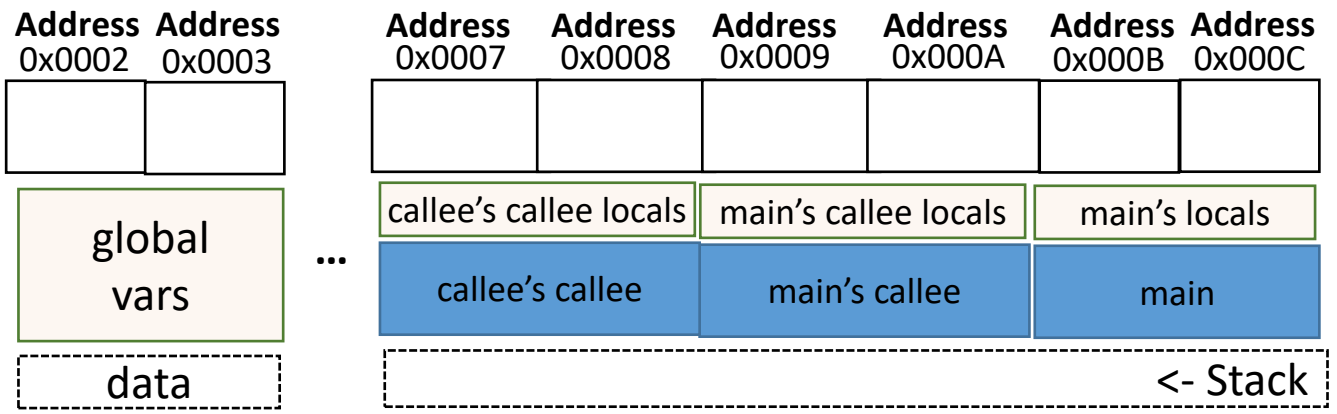
We'll visualize just these sections



The Memory Sections Concept

Review: Memory Layout

```
1.  int16 g = 2;
2.  void main(){
3.     int16 a;
4.     a = g;
5.     g = g - 1;
6.     if (0 < g){
7.       main();
8.     }
9.     cout << a;
10. }
```



The Memory Sections Concept

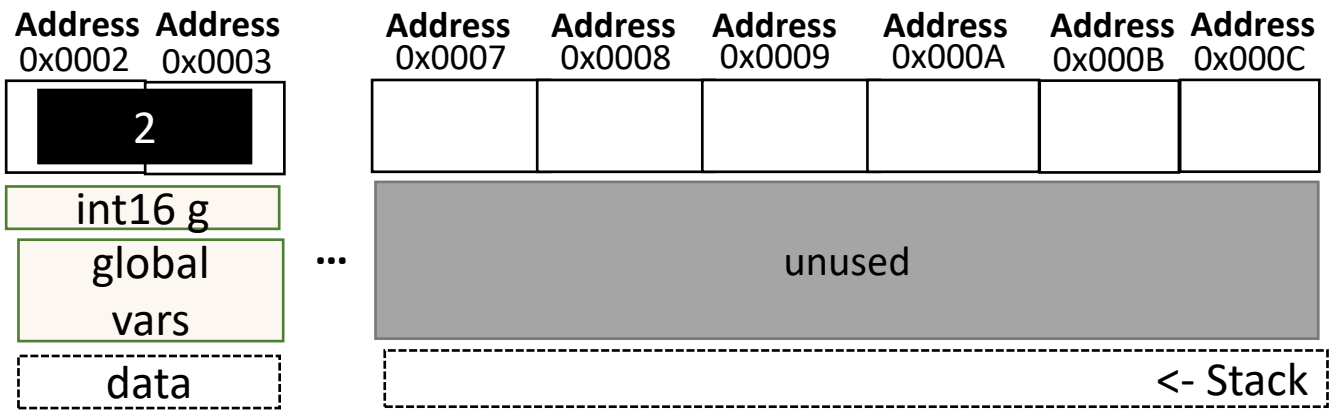
Review: Memory Layout

```
1.  int16 g = 2;
2.  void main(){
3.     int16 a;
4.     a = g;
5.     g = g - 1;
6.     if (0 < g){
7.       main();
8.     }
9.     cout << a;
10. }
```

Code Operations

On initialization
- globals set (g = 2)
- setup empty stack

Enter main
- setup call frame



The Memory Sections Concept

Review: Memory Layout

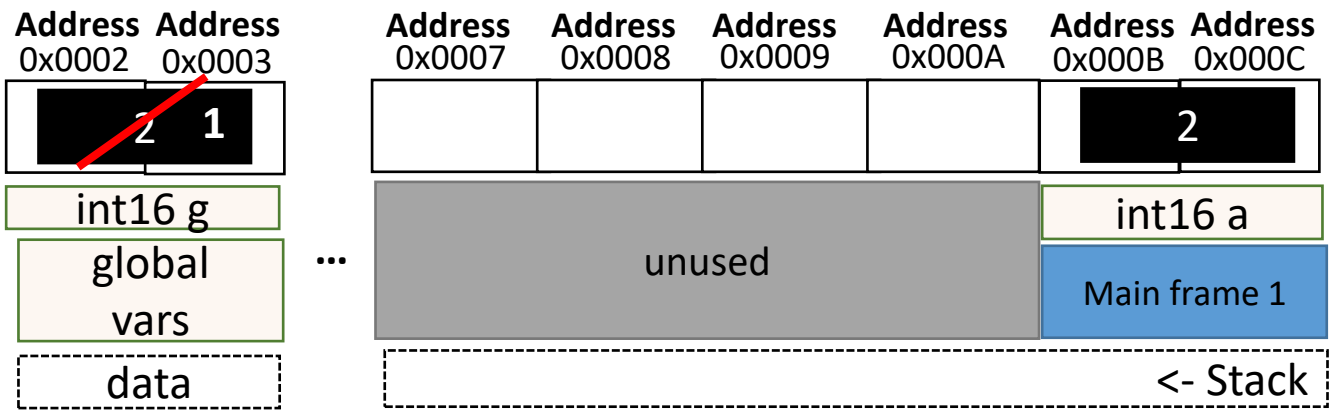
```
1.  int16 g = 2;
2.  void main(){
3.      int16 a;
4.      a = g;
5.      g = g - 1;
6.      if (0 < g){
7.        main();
8.      }
9.      cout << a;
10. }
```

Code Operations

On initialization
- globals set (g = 2)
- setup empty stack

Enter main
- setup call frame
- update local a
- update global g
- call main

Enter main
- setup call frame



The Memory Sections Concept

Review: Memory Layout

```
1.  int16 g = 2;
2.  void main(){
3.      int16 a;
4.      a = g;
5.      g = g - 1;
6.      if (0 < g){
7.        main();
8.      }
9.      cout << a;
10. }
```

Code Operations

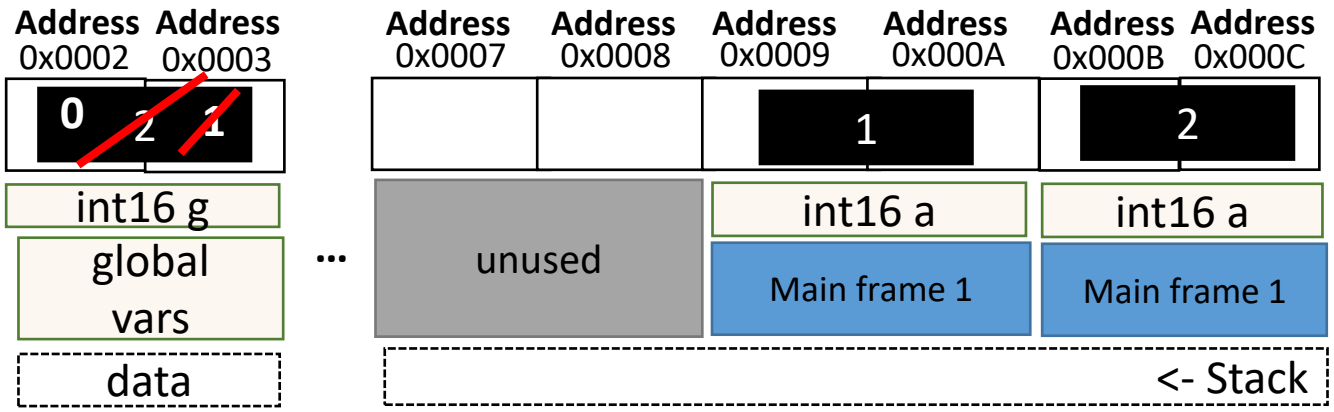
On initialization
- globals set (g = 2)
- setup empty stack

Enter main
- setup call frame
- update local a
- update global g
- call main

Enter main
- setup call frame
- update local a
- update global g
- print a
- return

Output

1 2



The Memory Sections Concept

Review: Memory Layout

```
1.  int16 g = 2;
2.  void main(){
3.     int16 a;
4.     a = g;
5.     g = g - 1;
6.     if (0 < g){
7.       main();
8.     }
9.     cout << a;
10. }
```

Code Operations

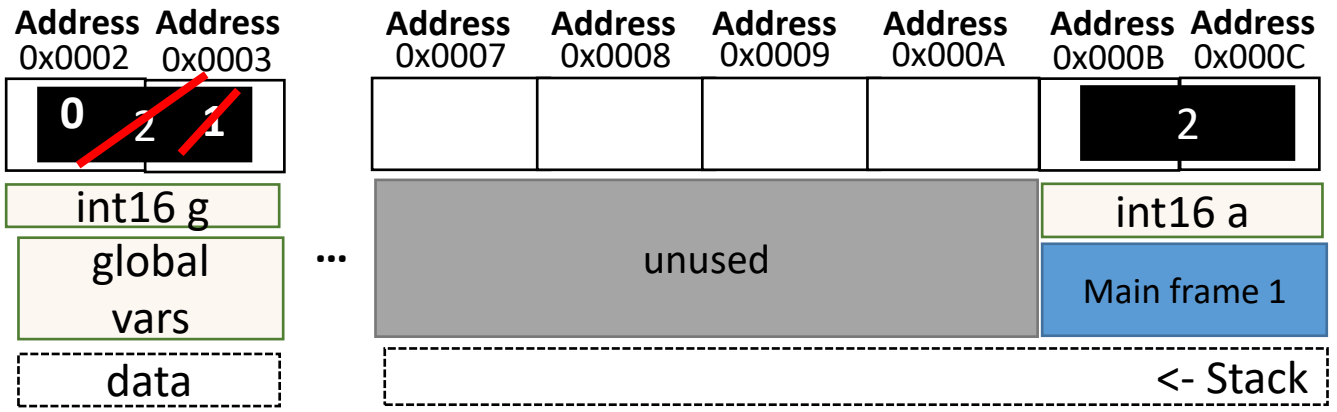
On initialization
- globals set (g = 2)
- setup empty stack

Enter main
- setup call frame
- update local a
- update global g
- call main
- print a
- return

Enter main
- setup call frame
- update local a
- update global g
- print a
- return

Output

1 2



The Memory Sections Concept

Review: Memory Layout

```
1.  int16 g = 2;
2.  void main(){
3.     int16 a;
4.     a = g;
5.     g = g - 1;
6.     if (0 < g){
7.       main();
8.     }
9.     cout << a;
10. }
```

Code Operations

On initialization
- globals set (g = 2)
- setup empty stack

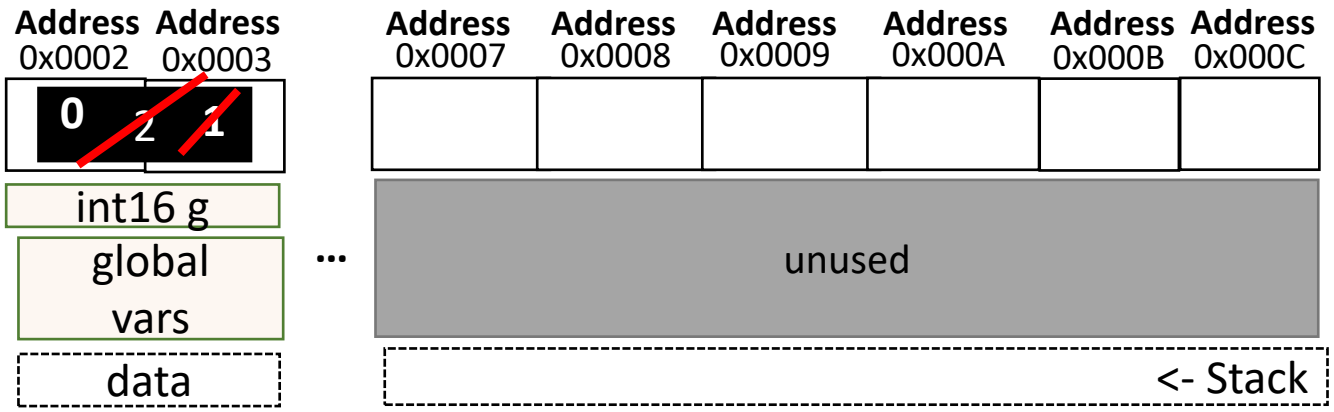
Enter main
- setup call frame
- update local a
- update global g
- call main
- print a
- return

Enter main
- setup call frame
- update local a
- update global g
- print a
- return

Output

1 2

The compiler needs to write code to do all of this!



Today's Outline

Activation Records

Managing the Stack

- Managing data
- Managing control

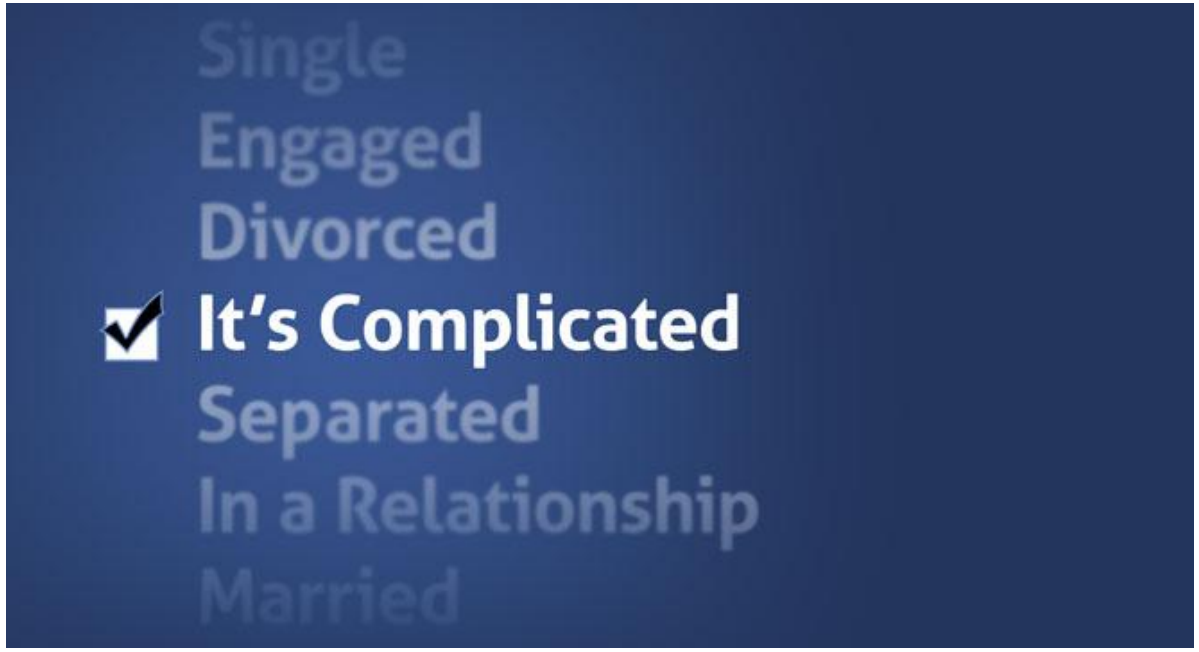


Architecture

Stack Frames Complicate Code

Beyond Static Allocation

Program must manage usage of stack memory



Stack Pointer as Register

Beyond Static Allocation

Program must manage usage of stack memory

Keep a pointer that tracks the base of the entire stack size

%rsp = “stack pointer”: track the edge of the stack

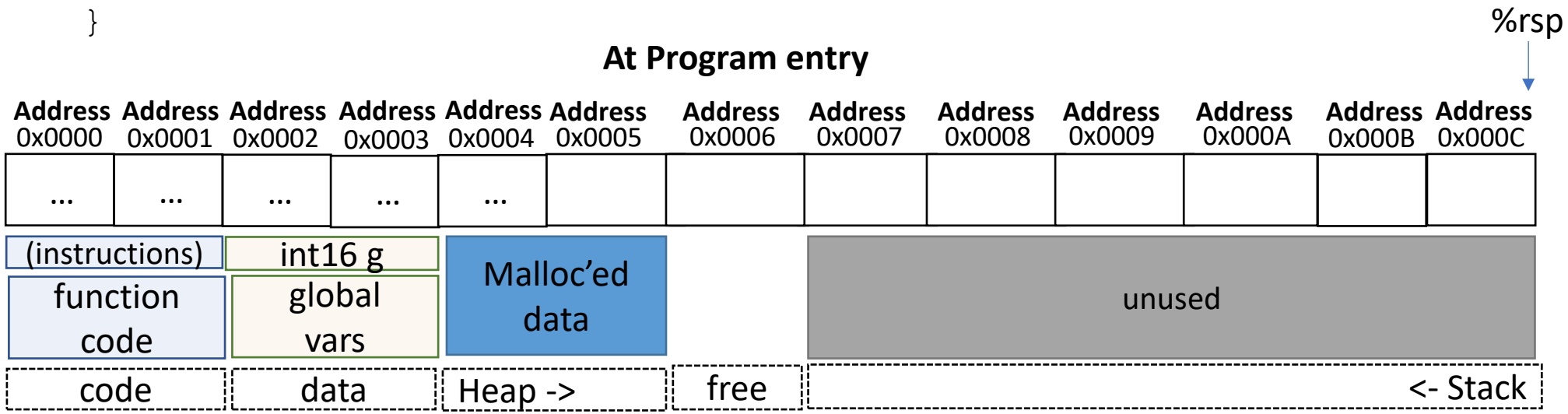
Review: The Stack

Beyond Static Allocation

```
int16 g = 2;
void main(){
    int16 a;
    a = g;
    g = g - 1;
    if (0 < g){
        main();
    }
    TOCONSOLE a;
}
```

Keep a pointer that tracks the base of the entire stack size

%rsp = “stack pointer”: track the edge of the stack



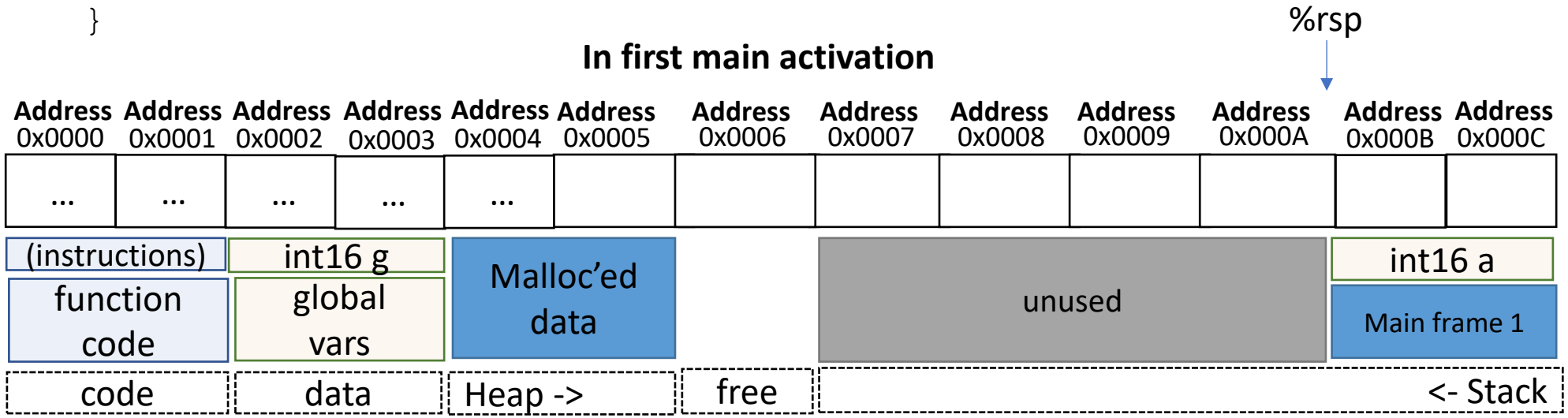
Review: The Stack

Beyond Static Allocation

```
int16 g = 2;
void main(){
    int16 a;
    a = g;
    g = g - 1;
    if (0 < g){
        main();
    }
    TOCONSOLE a;
}
```

Keep a pointer that tracks the base of the entire stack size

%rsp = “stack pointer”: track the edge of the stack



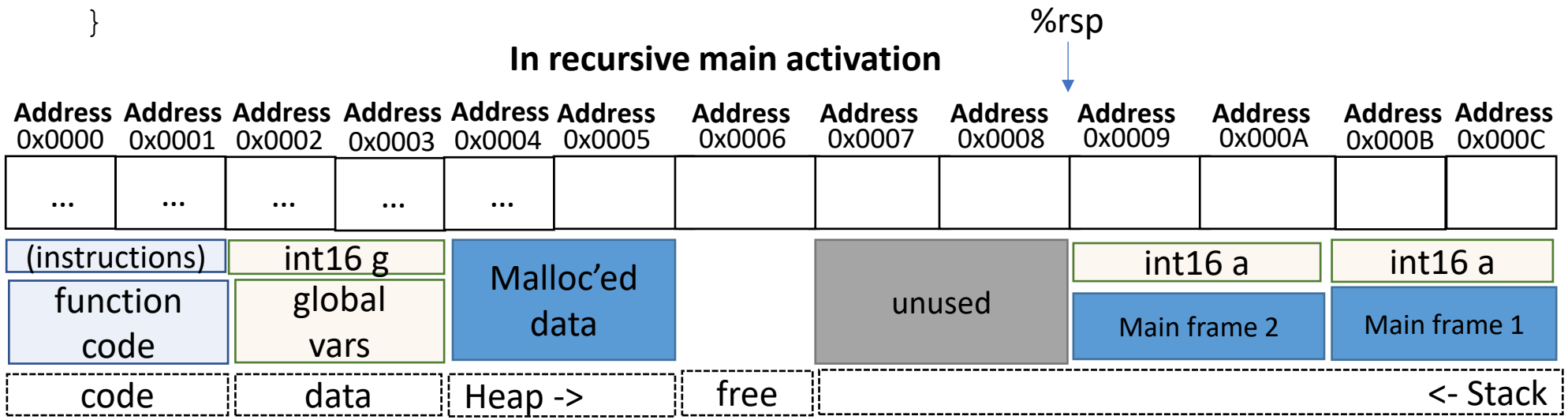
Review: The Stack

Beyond Static Allocation

```
int16 g = 2;
void main(){
    int16 a;
    a = g;
    g = g - 1;
    if (0 < g){
        main();
    }
    TOCONSOLE a;
}
```

Keep a pointer that tracks the base of the entire stack size

%rsp = “stack pointer”: track the edge of the stack

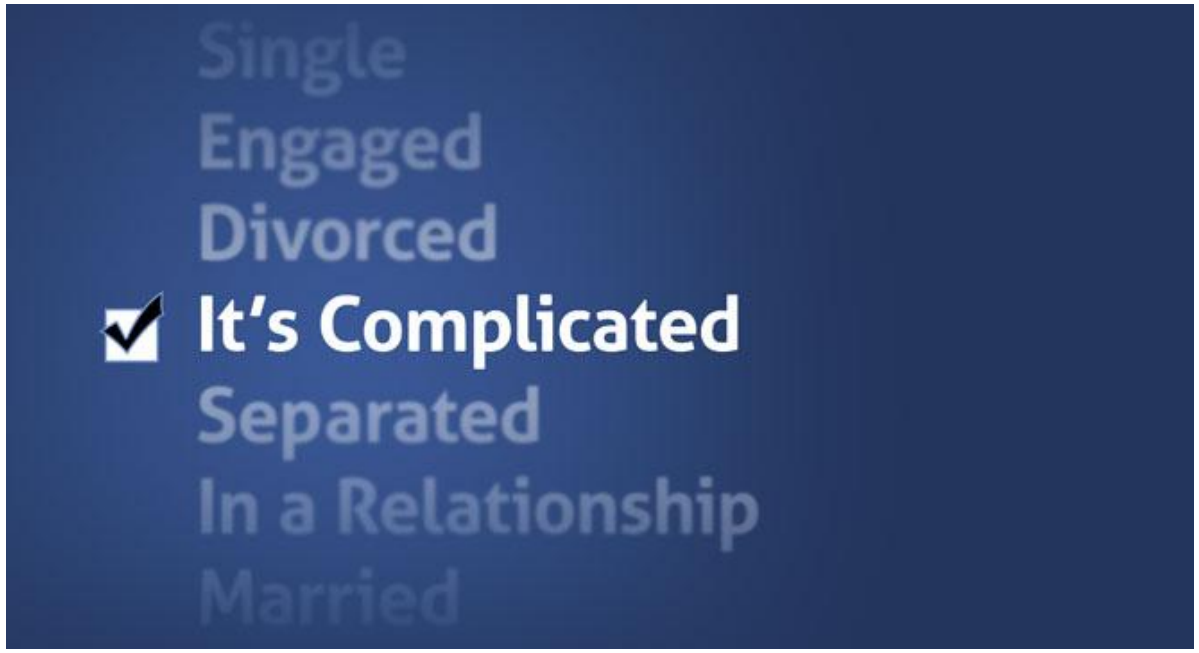


Stack Frames Complicate Code

Beyond Static Allocation

Program must manage usage of stack memory

Variable's address is no longer static: can't be hard-coded



Frame-Relative Variable Addressing

Beyond Static Allocation

Variable's address is no longer static: can't be hard-coded

```
int16 g = 2;
void main(){
    int16 a;
    a = g;
    g = g - 1;
    if (0 < g){
        main();
    }
    TOCONSOLE a;
}
```

Keep a pointer that tracks the base of the current stack frame

%rsp = "stack pointer": track the edge of the stack

%rbp = "base pointer": track the base (beginning of the **current** frame)

Frame-Relative Variable Addressing

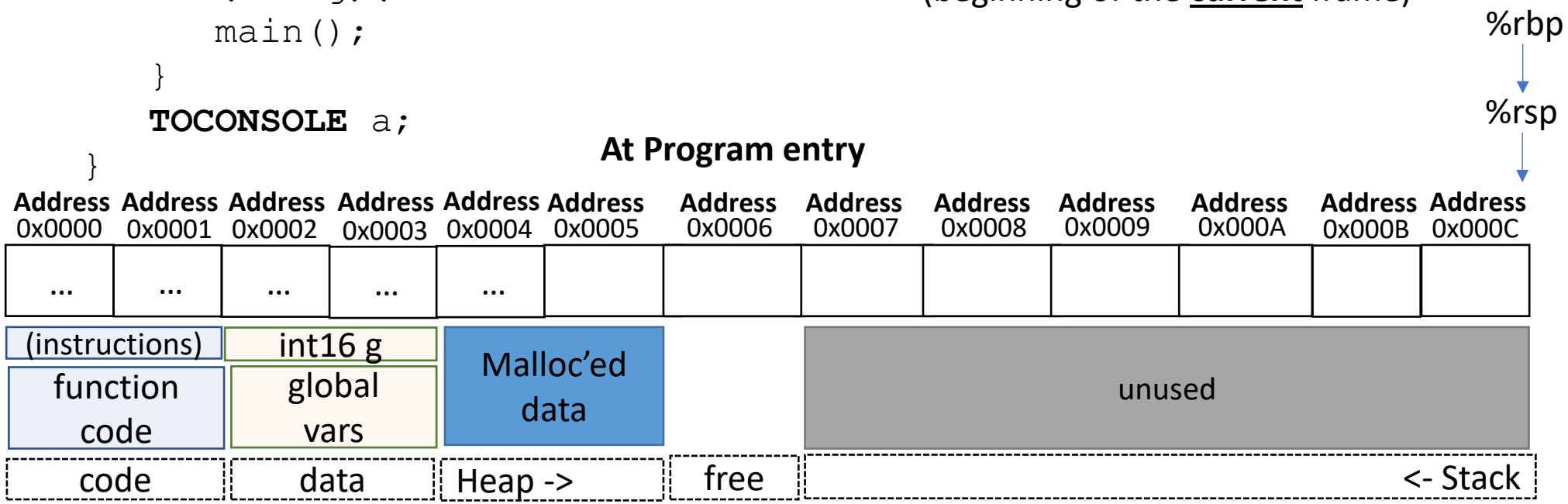
Beyond Static Allocation

```
int16 g = 2;
void main(){
    int16 a;
    a = g;
    g = g - 1;
    if (0 < g){
        main();
    }
    TOCONSOLE a;
}
```

Keep a pointer that tracks the base of the current stack frame

%rsp = "stack pointer": track the edge of the stack

%rbp = "base pointer": track the base (beginning of the **current** frame)



Frame-Relative Variable Addressing

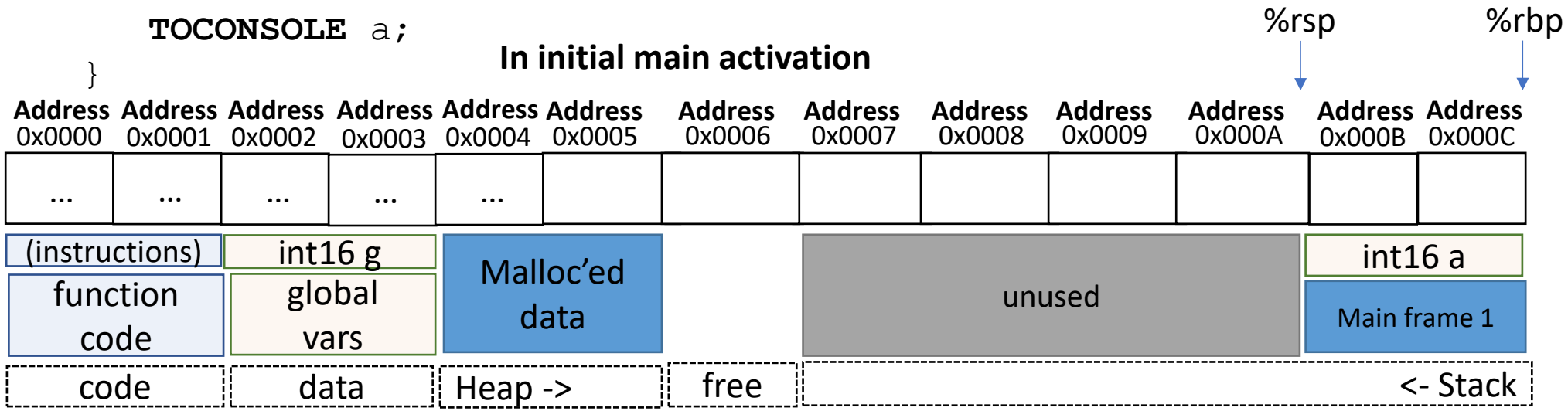
Beyond Static Allocation

```
int16 g = 2;
void main(){
    int16 a;
    a = g;
    g = g - 1;
    if (0 < g){
        main();
    }
    TOCONSOLE a;
}
```

Keep a pointer that tracks the base of the current stack frame

%rsp = "stack pointer": track the edge of the stack

%rbp = "base pointer": track the base (beginning of the **current** frame)



Frame-Relative Variable Addressing

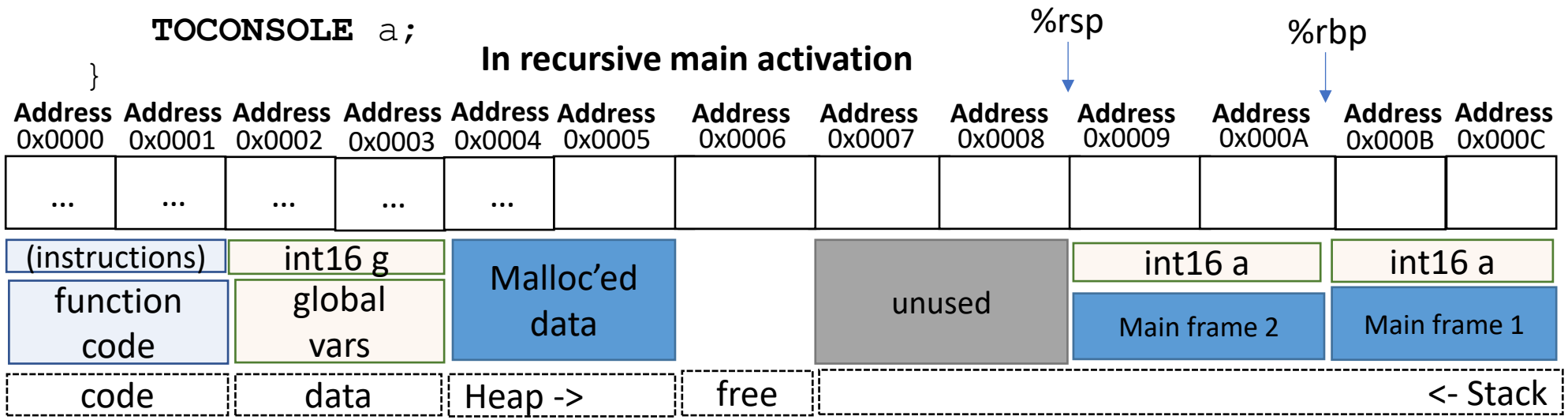
Beyond Static Allocation

```
int16 g = 2;
void main(){
    int16 a;
    a = g;
    g = g - 1;
    if (0 < g){
        main();
    }
    TOCONSOLE a;
}
```

Keep a pointer that tracks the base of the current stack frame

%rsp = "stack pointer": track the edge of the stack

%rbp = "base pointer": track the base (beginning of the **current** frame)



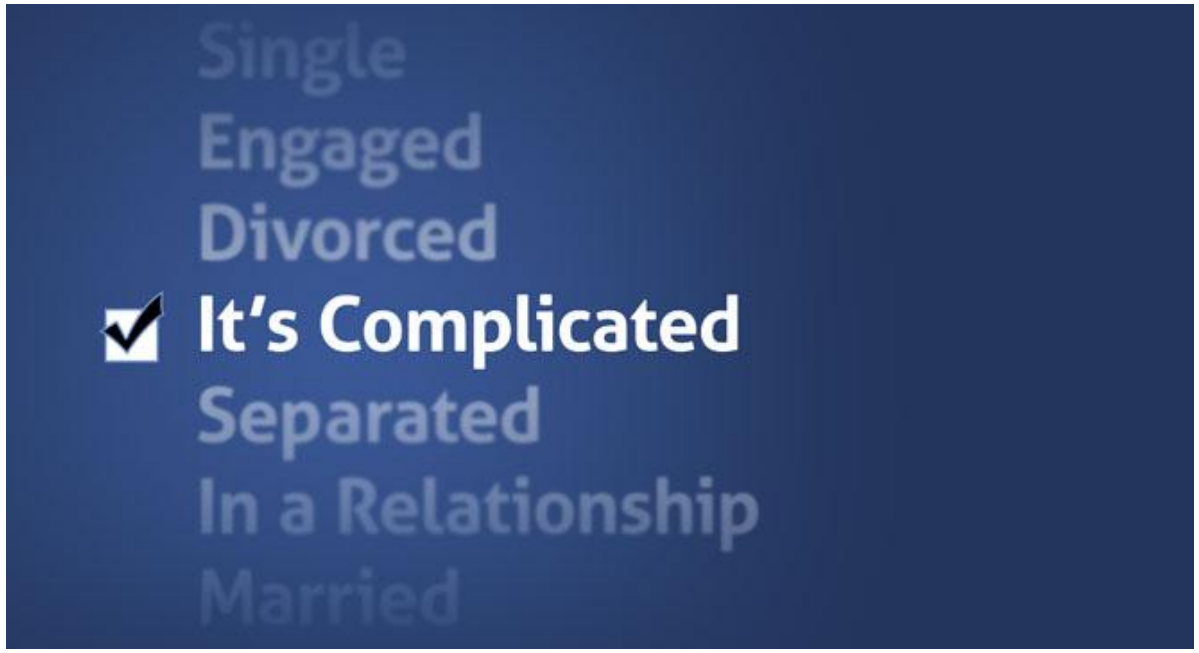
Complicating Code

Beyond Static Allocation

Program must manage usage of stack memory

Variable's address is no longer static: can't be hard-coded

Return address is not static



Returning Control to Caller

x64 Stack Frames

We need a way to transfer control into a callee

- We need a way to transfer control *back* to the caller
- Could we just use jmp for this?

```
int g = 5;
void bar(){
  g = g + 5;
}

void main(){
  bar();
  bar();
}
```

```
.globl _start
.data
gbl_g: .quad 5
.text
fn_bar:
    movq (glb_g), %rax
    addq $5, %rax
    movq %rax, (glb_g)
       jmp pt1

_start:
    jmp fn_bar
  pt1: nop
    jmp fn_bar
  pt2: nop
```



Callq/Retq

Parameter Passing

callq: Do a jump, save the address of the instruction to which to return

retq: Jump back to the saved address

Restoring the Caller's Frame

Beyond Static Allocation

When we're done with a call, we need to restore the **OLD** frame

- Return the instruction pointer to the call site
- Restore the base pointer to the base of the caller's frame
- Restore the stack pointer to the edge of the caller's frame

We need to keep track of some of this information!

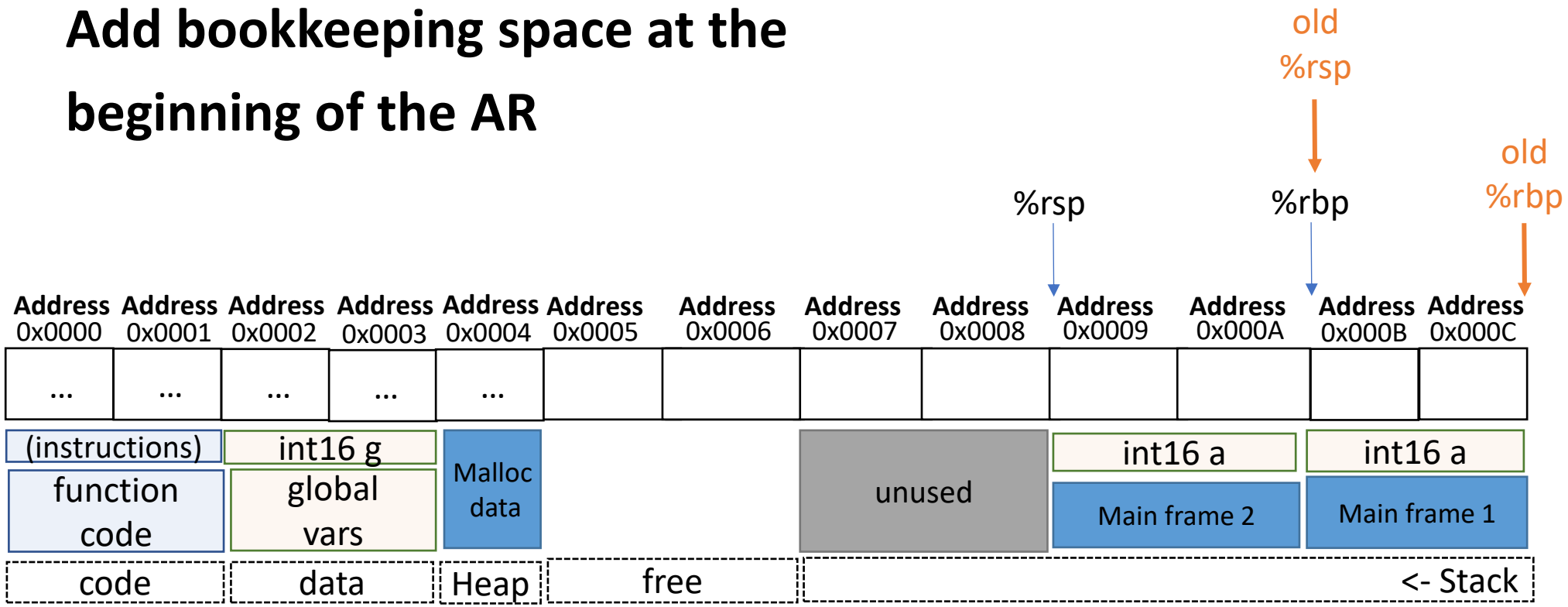
Call Stack Bookkeeping

Beyond Static Allocation

We need to store (on the stack):

- The call site to resume execution after call
- The base pointer to restore the old stack frame after call

Add bookkeeping space at the beginning of the AR



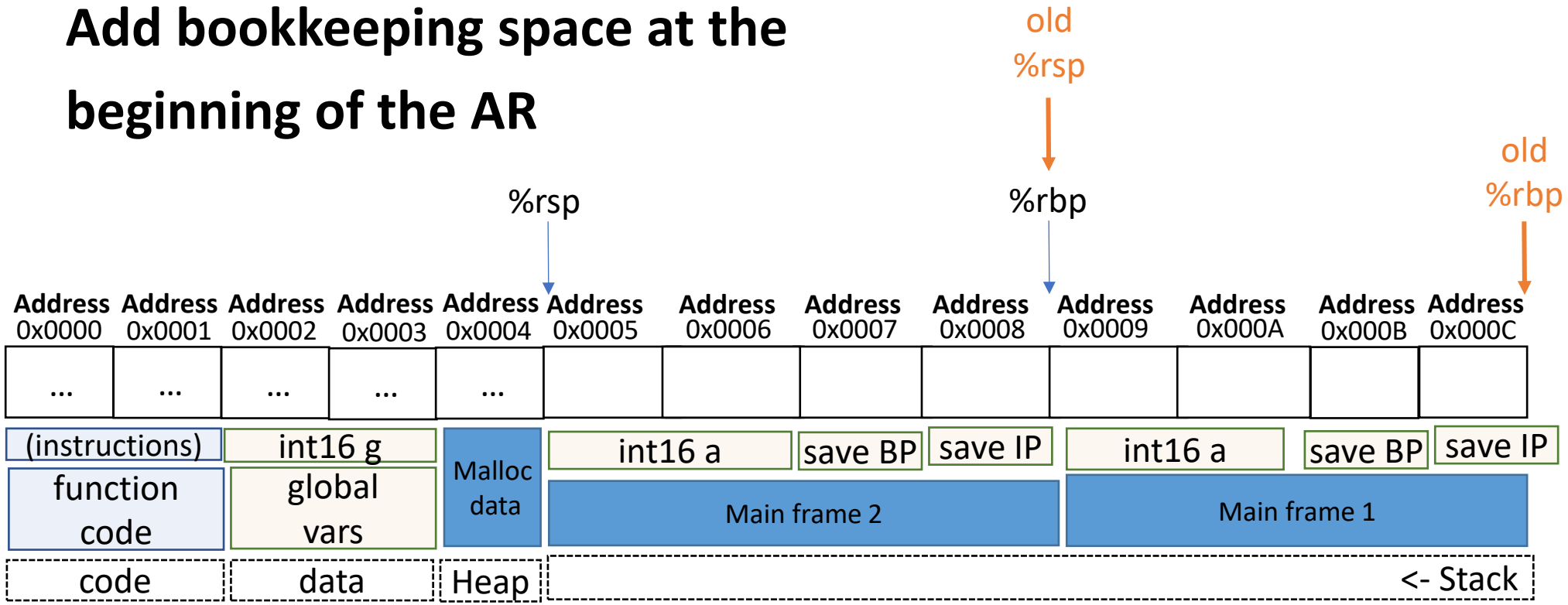
Call Stack Bookkeeping

Beyond Static Allocation

We need to store (on the stack):

- The call site to resume execution after call
- The base pointer to restore the old stack frame after call

Add bookkeeping space at the beginning of the AR



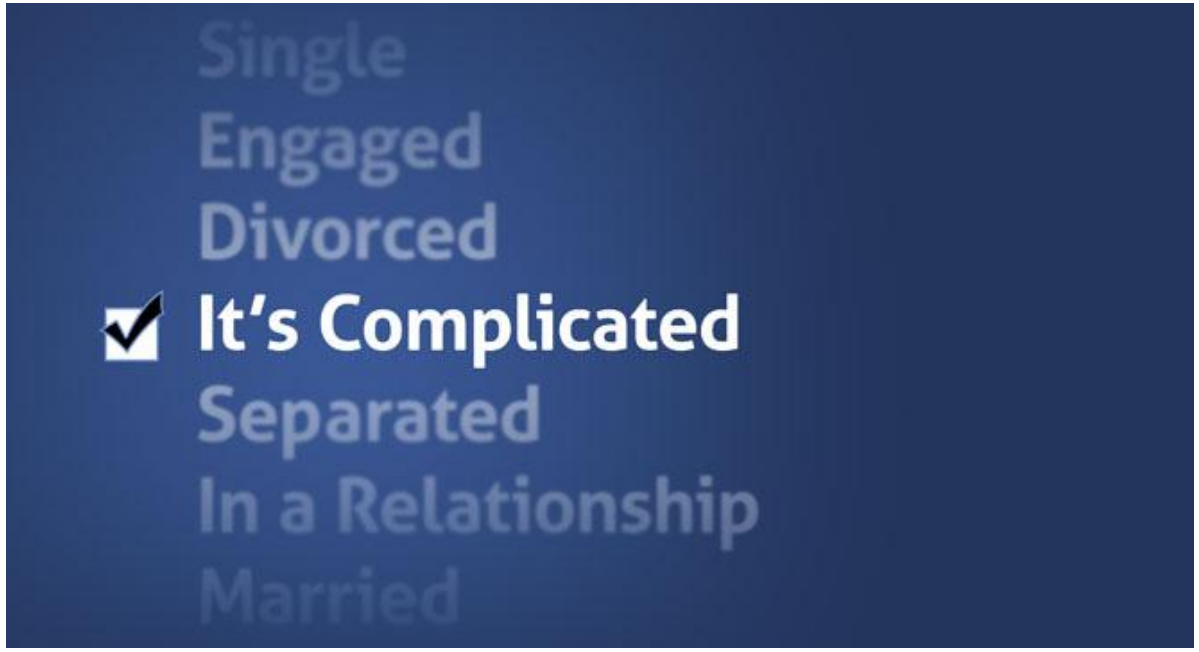
Solving Stack Frame Complications

Beyond Static Allocation

Program must manage usage of stack memory

Variable's address is no longer static: can't be hard-coded

Return address is not static



Solving Stack Frame Complications

Beyond Static Allocation

Program must manage usage of stack memory

%rsp tracks the stack edge

Variable's address is no longer static: can't be hard-coded

%rbp tracks the current frame's base

Caller's rbp saved on the stack when we enter a callee

allows restore of data context

Return address is not static

Call instruction saves the return point on top of the stack
Ret instruction jumps to the address on top of the stack

allows restore of control context

Putting it all together

Beyond Static Allocation

Let's show how stack frames are maintained in code

- Save caller context as bookkeeping values
- Set up callee frame
- Address locals relative to the frame
- Break down the callee frame
- Restore caller context from bookkeeping values

Putting it all together

Beyond Static Allocation

Two new instructions will be convenient to add

- Pushq <op1> - decrements %rsp by 8, and stores op1 at (%rsp)

Putting it all together

Beyond Static Allocation

Two new instructions will be convenient to add

- Pushq <op1> - decrements %rsp by 8, and stores op1 at (%rsp)
- Popq <op1> - retrieves the value at %rsp, increments %rsp by 8

Activation Records

Where we left off…

```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```



Activation Records

Where we left off…

```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

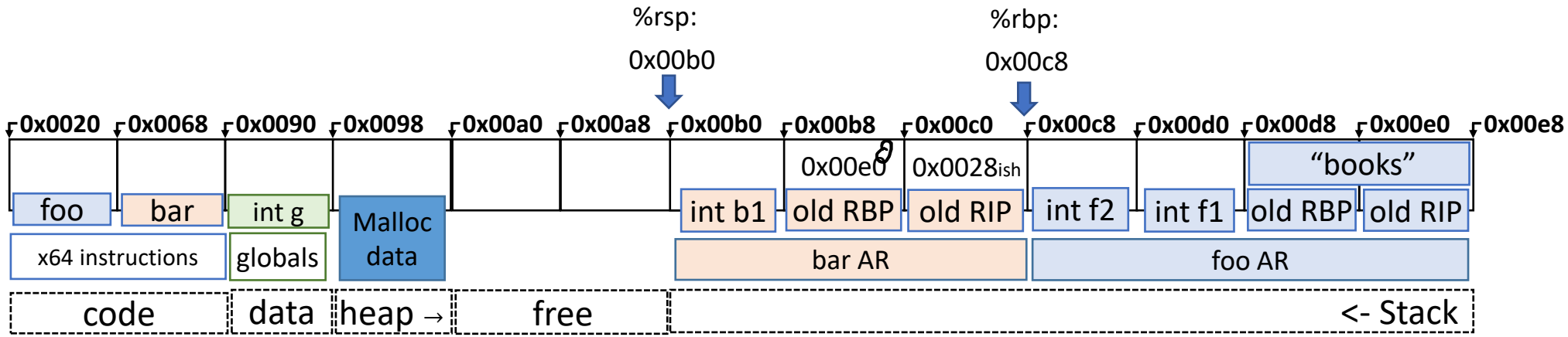


Activation Records

Where we Left off…

```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

Let's zoom in on just the stack + free space
(since that's all that changes for AR setup/teardown

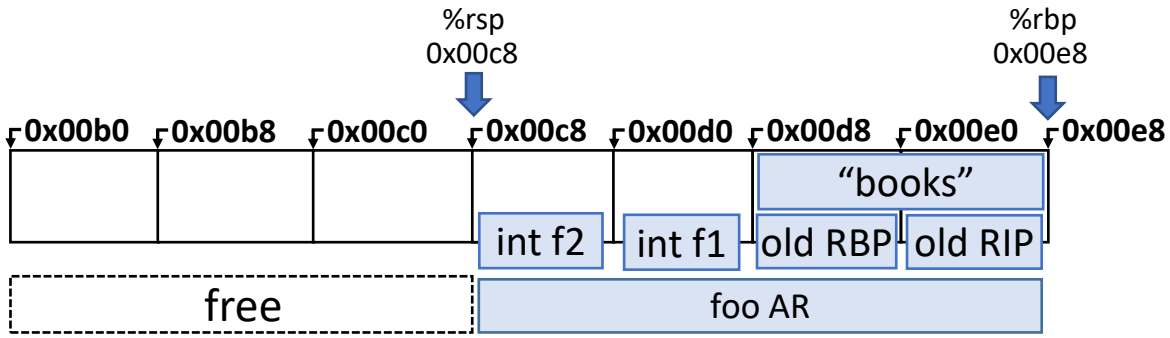


Activation Record Setup

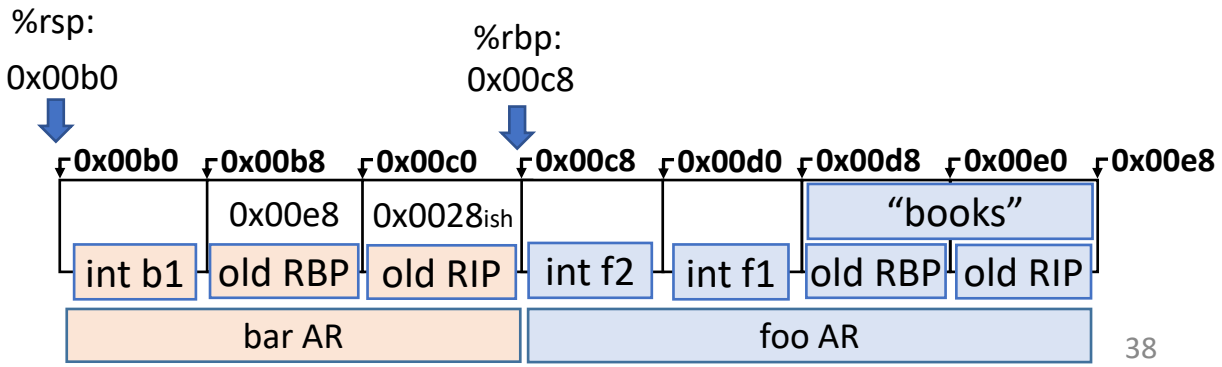
Finishing off ARs

```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

```
enter bar
leave bar
enter foo
call bar
leave foo
```



How do we get from the above ARs to the below ARs?



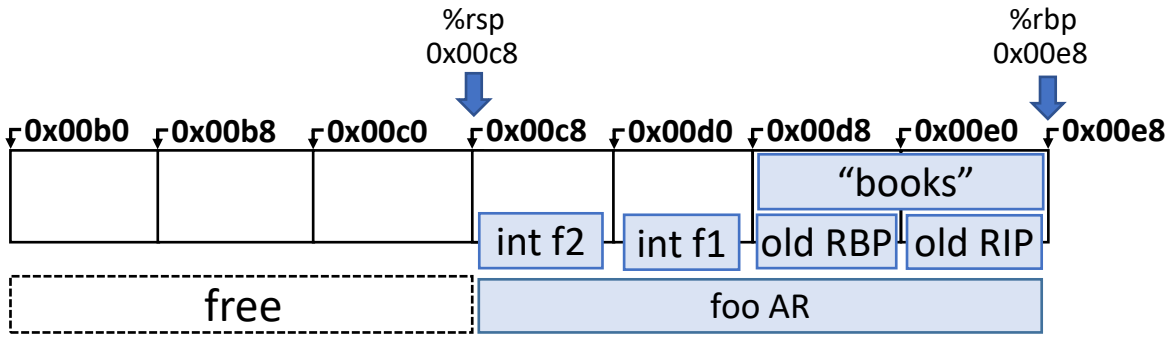
Activation Record Setup

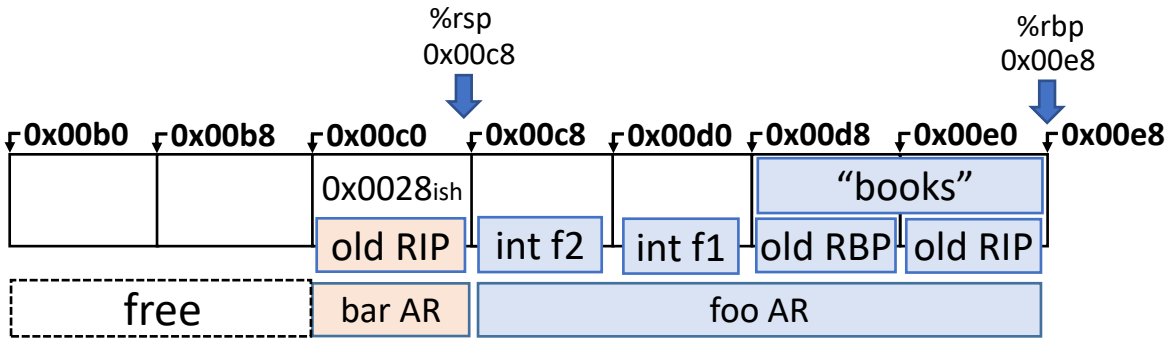
Finishing off ARs

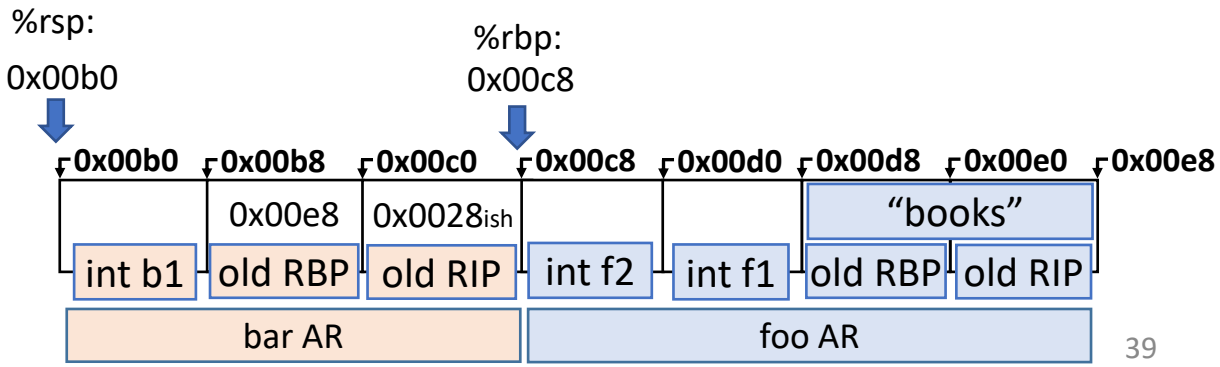
```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

callq bar

```
enter bar
leave bar
enter foo
call bar
leave foo
```





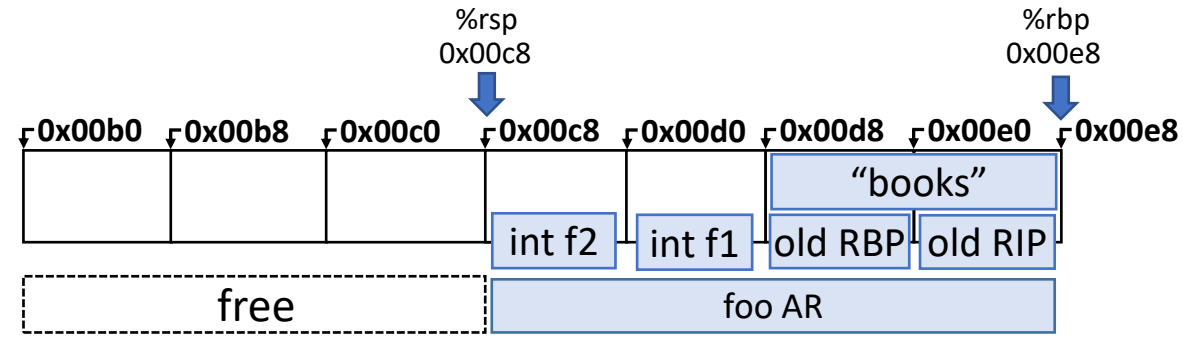


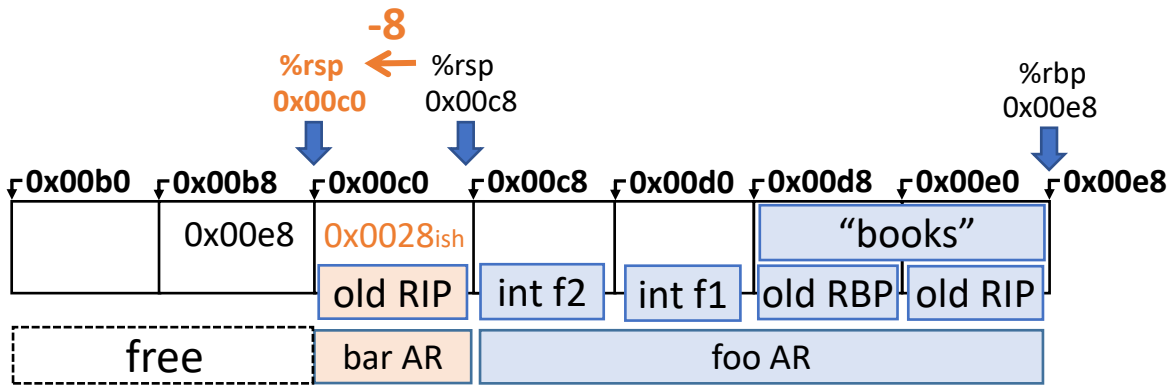
Activation Record Setup

Finishing off ARs

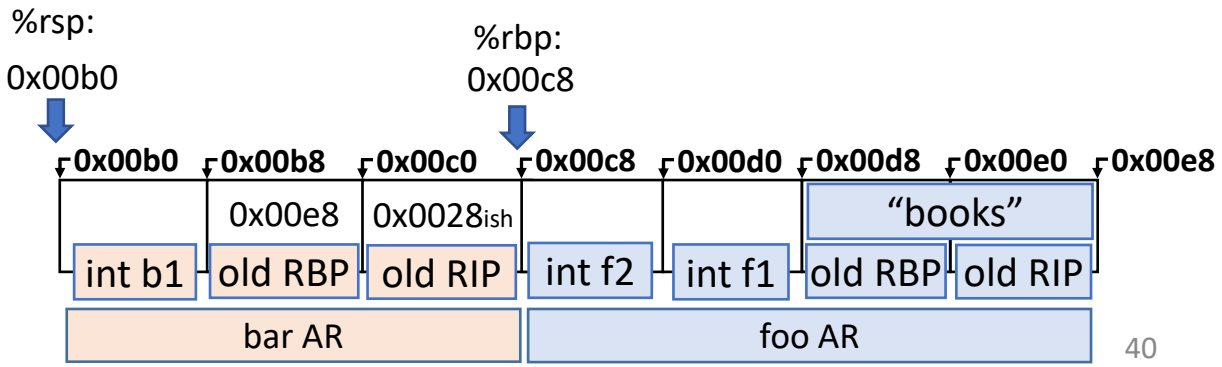
```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

callq bar






```
enter bar
leave bar
enter foo
call bar
leave foo
```



Activation Record Setup

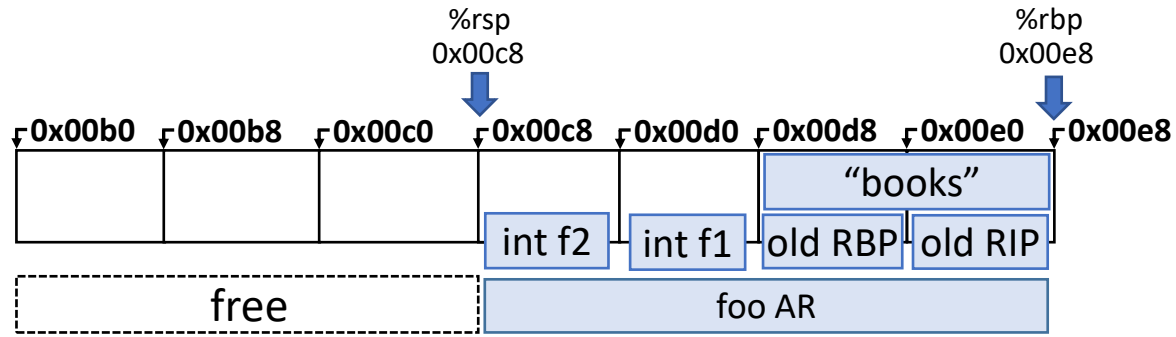
Finishing off ARs

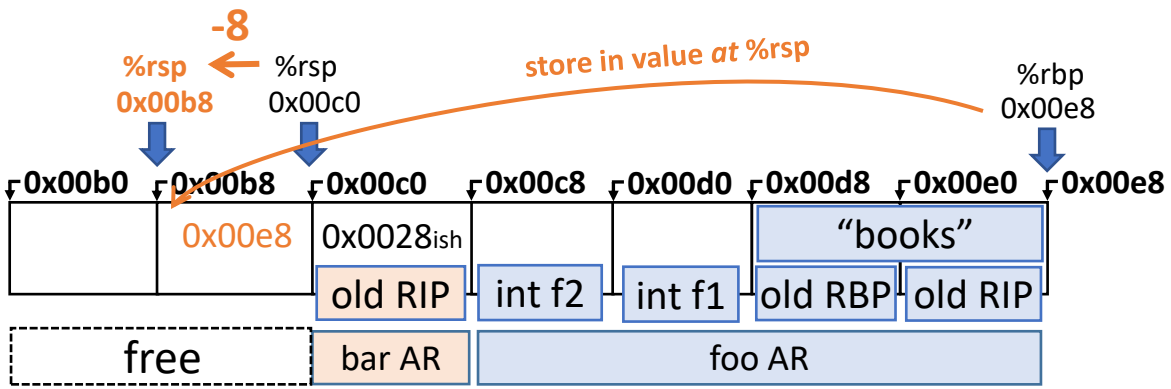
```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

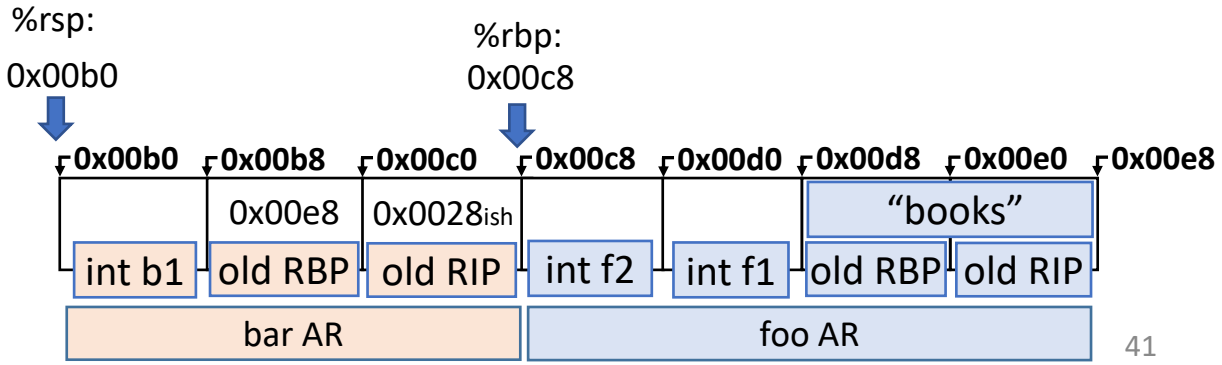
callq bar

pushq %rbp

```
enter bar
leave bar
enter foo
call bar
leave foo
```







Activation Record Setup

Finishing off ARs

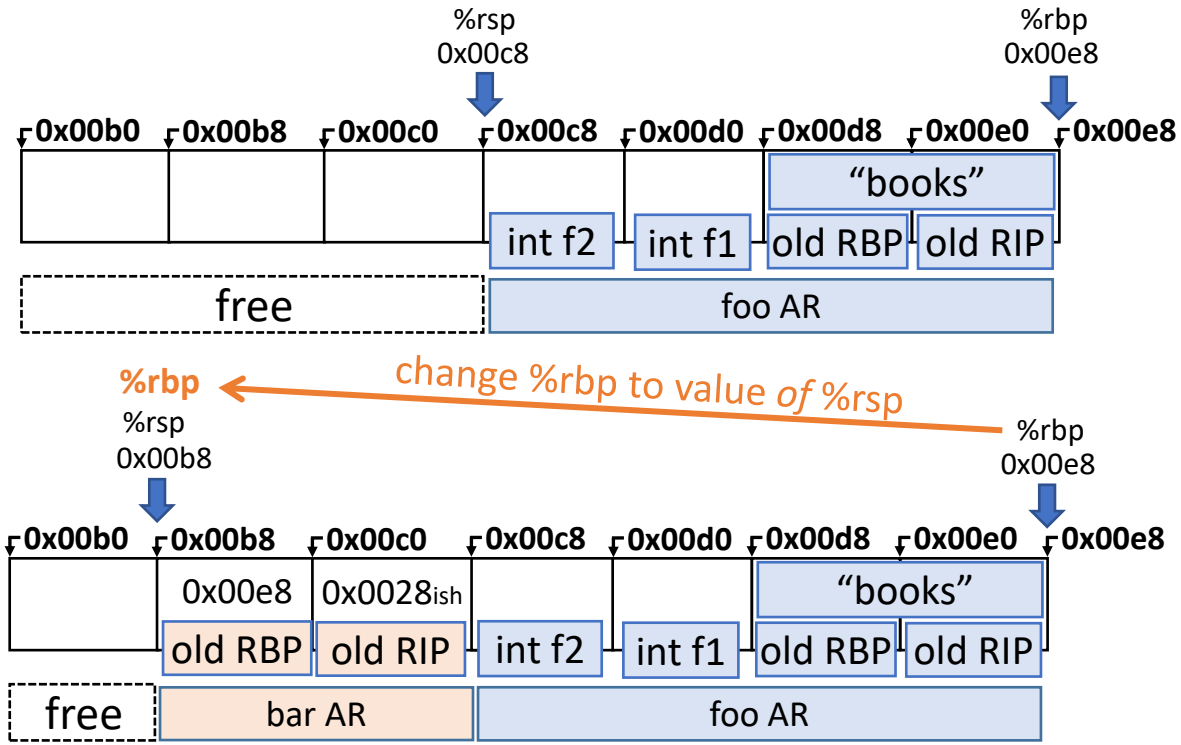
```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

callq bar

pushq %rbp

movq %rsp, %rbp

```
enter bar
leave bar
enter foo
call bar
leave foo
```



Activation Record Setup

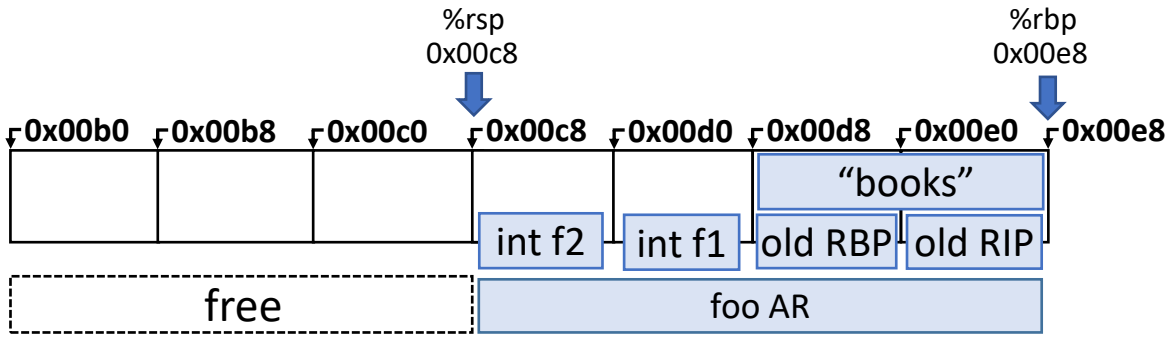
Finishing off ARs

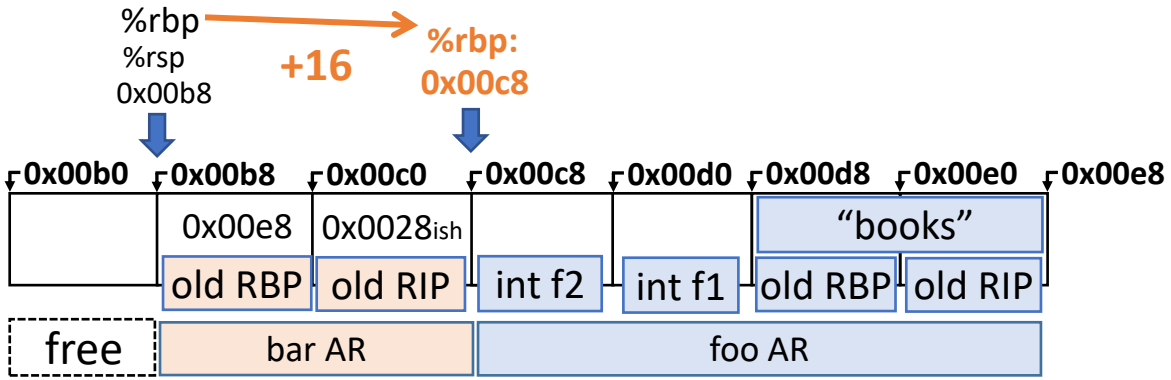
```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

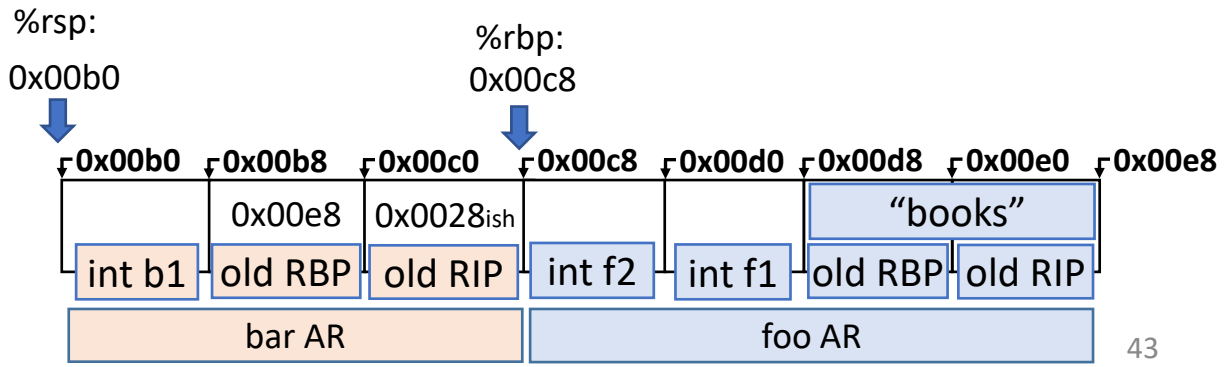
callq bar

pushq %rbp
movq %rsp, %rbp
addq $16, %rbp

```
enter bar
leave bar
enter foo
call bar
leave foo
```







Activation Record Setup

Finishing off ARs

```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

```
enter bar
leave bar
enter foo
call bar
leave foo
```

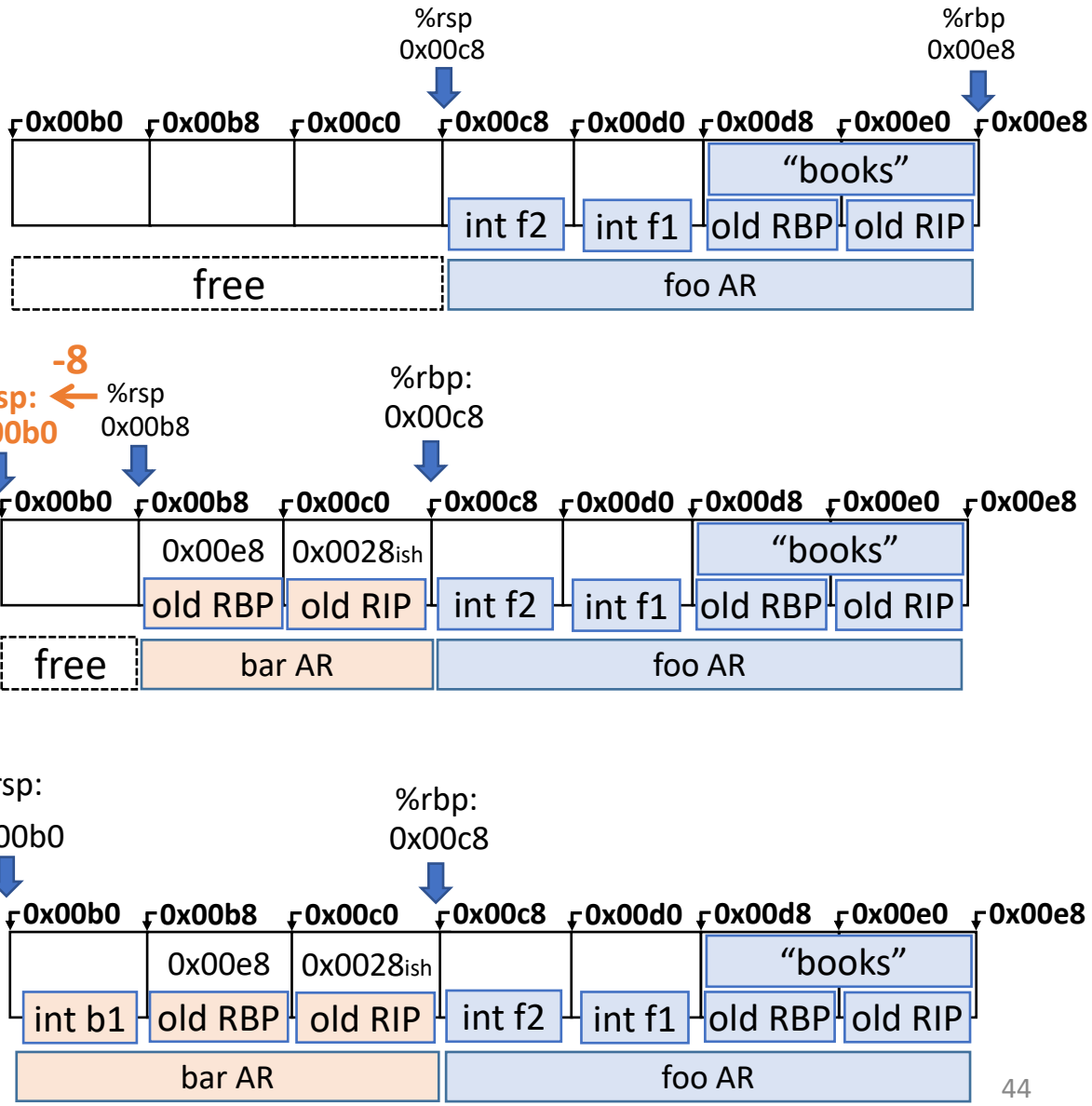
callq bar

pushq %rbp

movq %rsp, %rbp

addq $16, %rbp

subq $8, %rsp



Activation Record Setup

Finishing off ARs

```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

callq bar

pushq %rbp

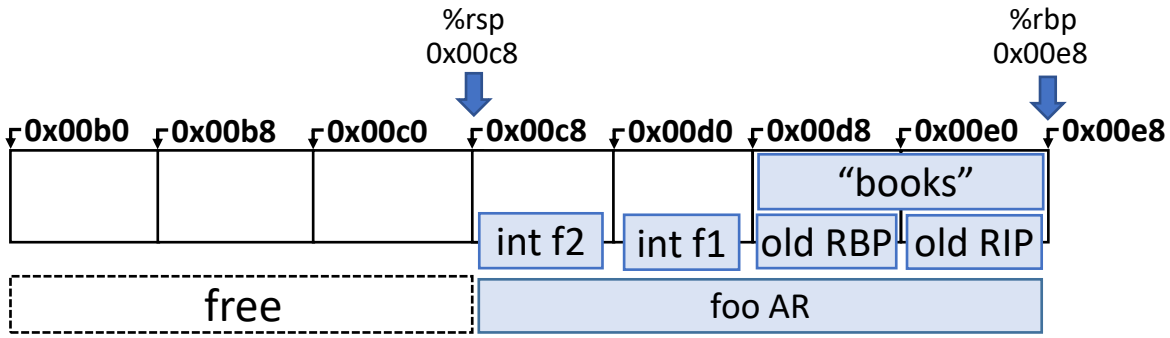
movq %rsp, %rbp

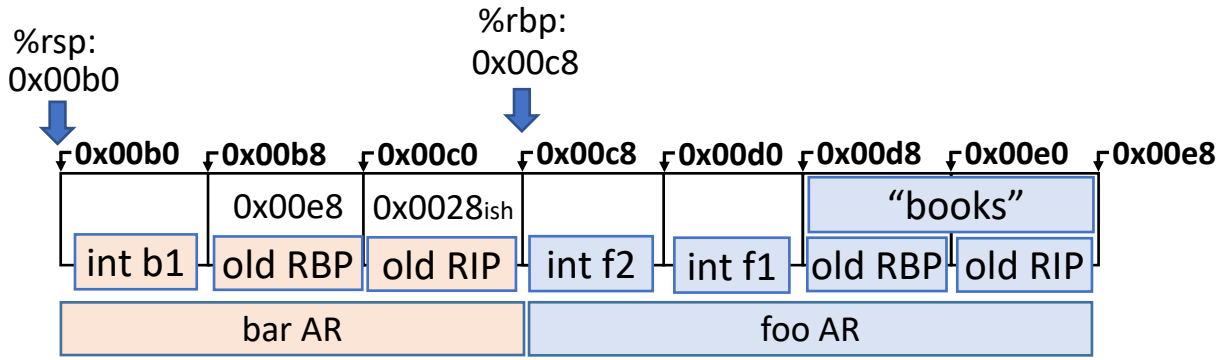
addq $16, %rbp

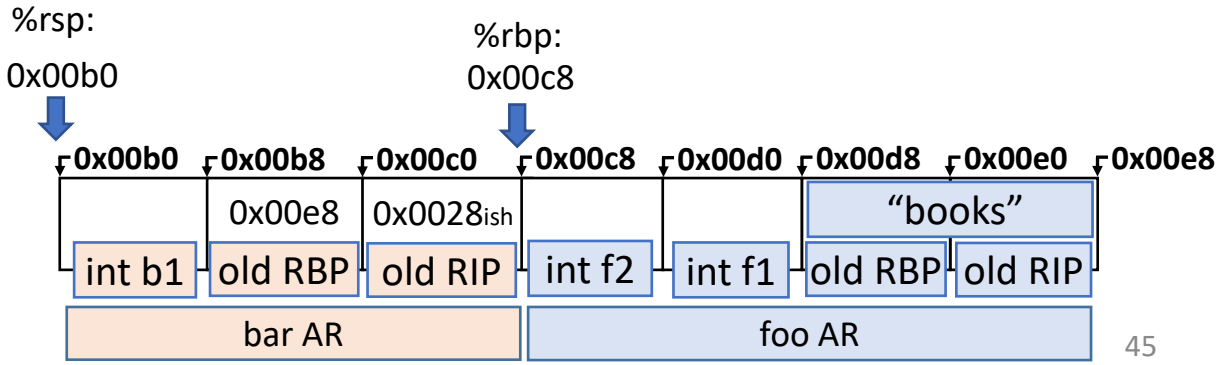
subq $8, %rsp

```
enter bar
leave bar
enter foo
call bar
leave foo
```

Done!







Activation Record Setup

Finishing off ARs

Function prologue

- The sequence of instructions at the beginning of a function to set up the AR
- Basically the same for every function

```
pushq %rbp
movq %rsp, %rbp
addq $16, %rbp
subq $8, %rsp
```

Function epilogue

- The sequence of instructions at the end of a function to tear down the AR

Activation Record Setup

```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

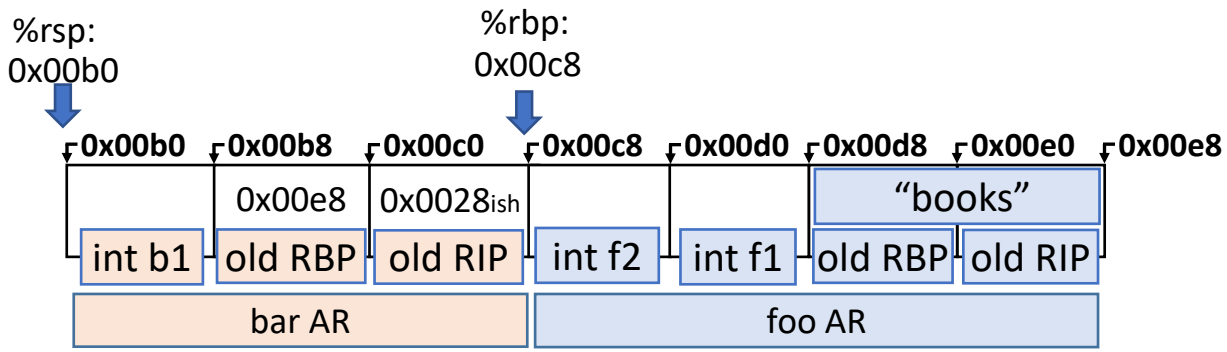
```
enter bar
leave bar
enter foo
call bar
leave foo
```

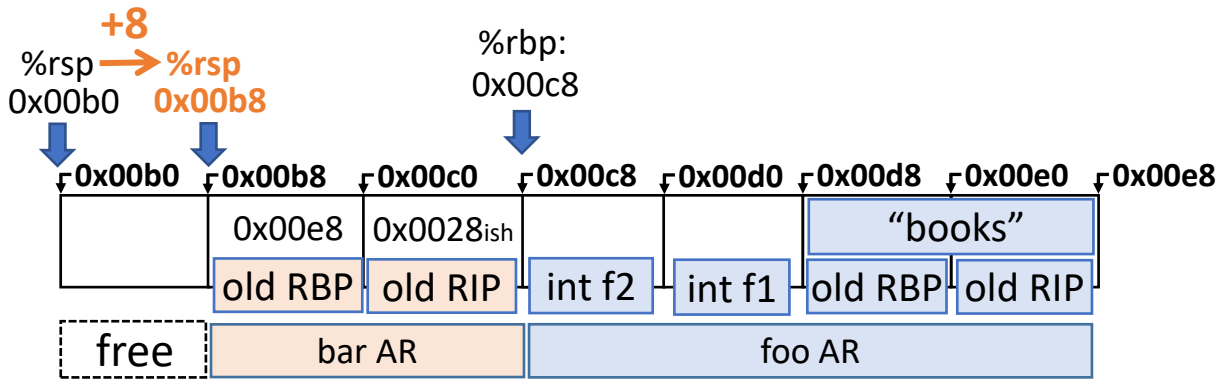
addq $8, %rsp

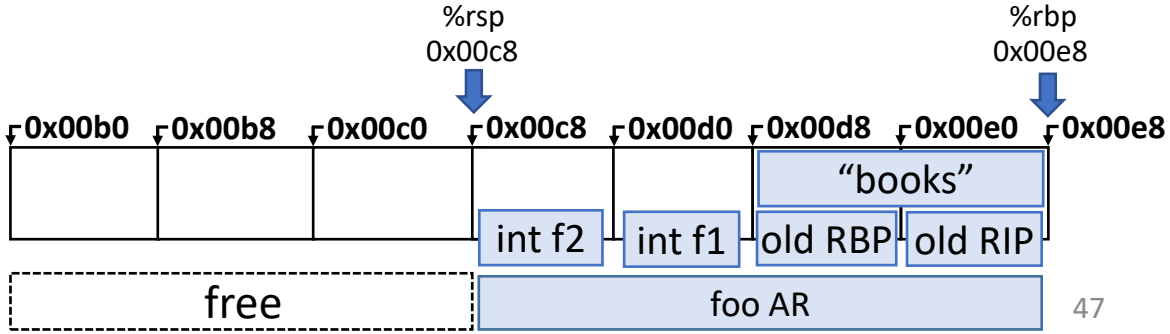
popq %rbp

retq

Finishing off ARs







Activation Record Setup

```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

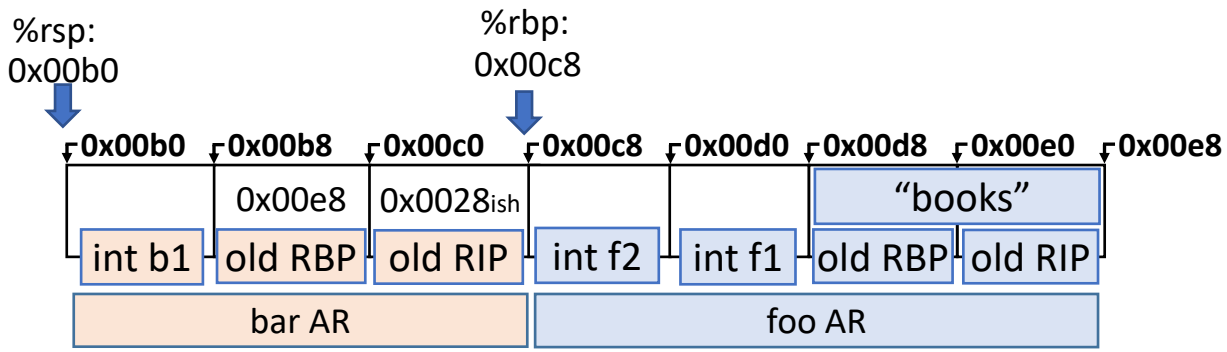
```
enter bar
leave bar
enter foo
call bar
leave foo
```

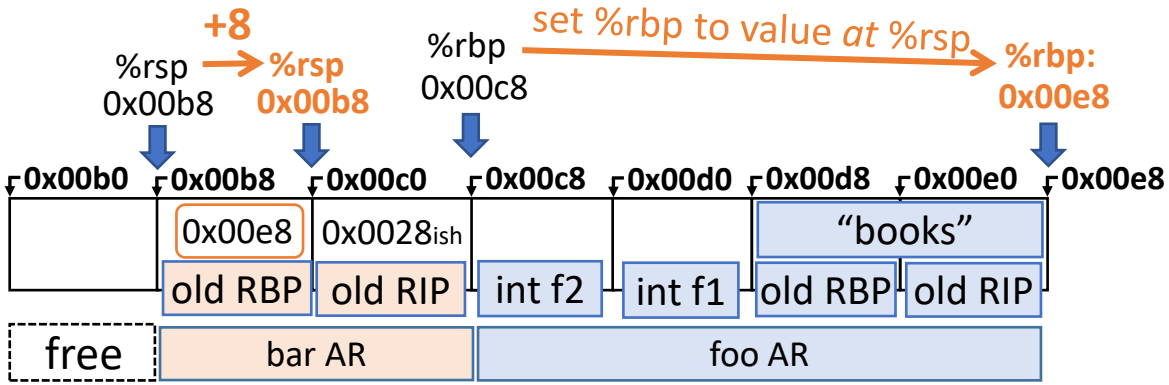
addq $8, %rsp

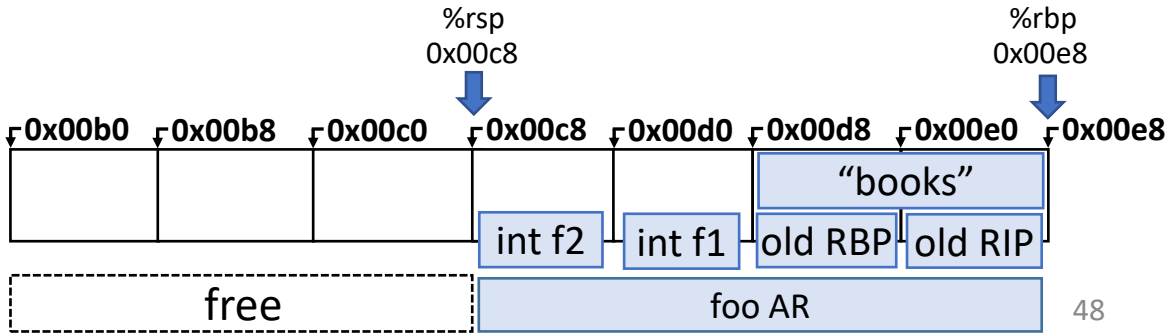
popq %rbp

retq

Finishing off ARs







Activation Record Setup

```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

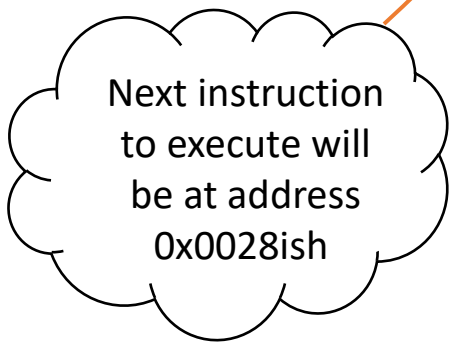
```
enter bar
leave bar
enter foo
call bar
leave foo
```

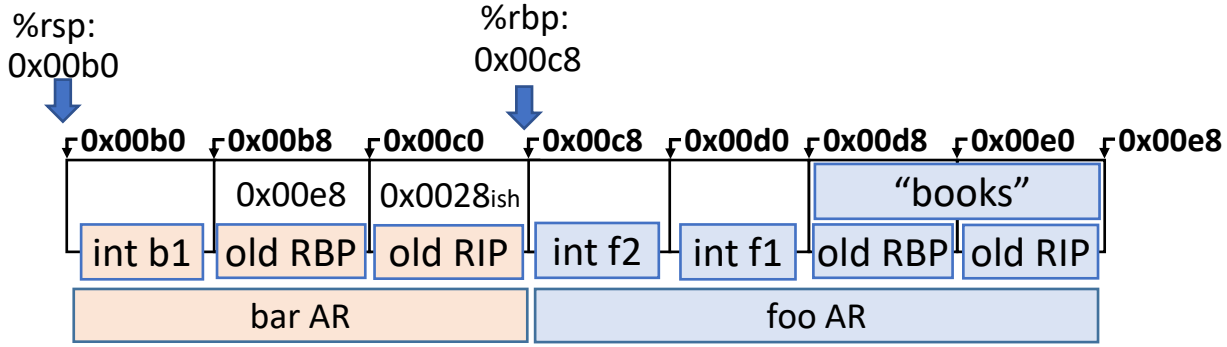
Finishing off ARs

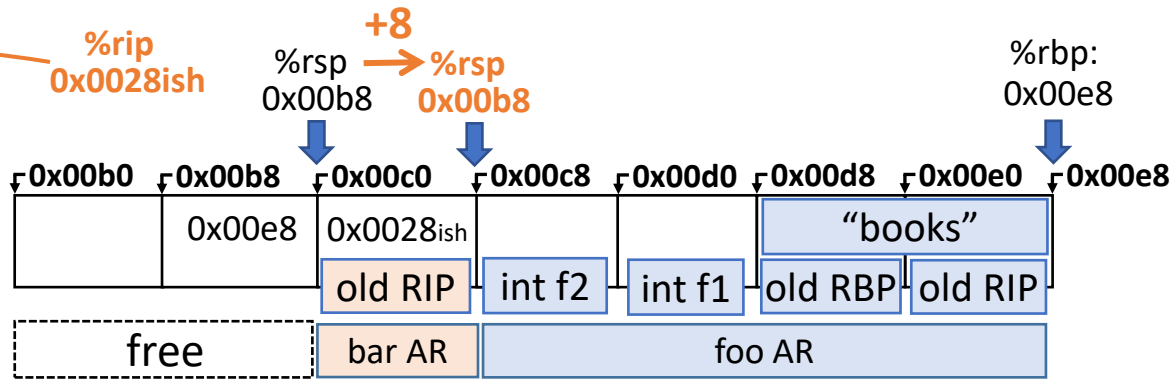
addq $8, %rsp

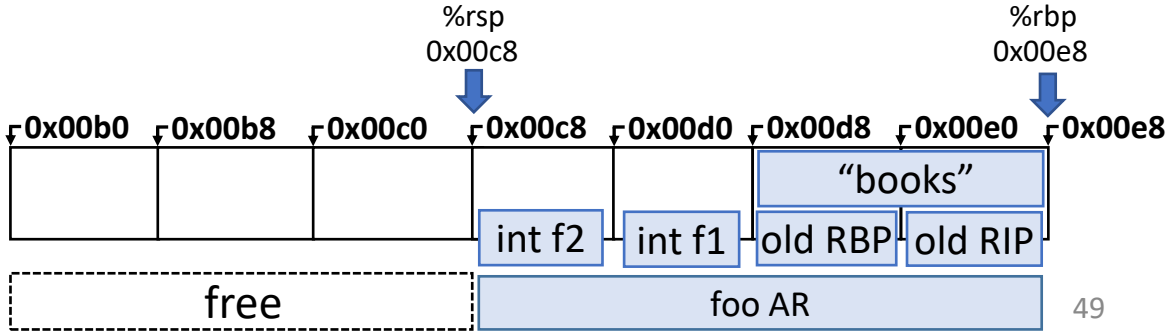
popq %rbp

retq









Activation Record Setup

```
int g;
void bar(){
  int b1;
}
void foo(){
  int f1;
  int f2;
  bar();
}
```

```
enter bar
leave bar
enter foo
call bar
leave foo
```

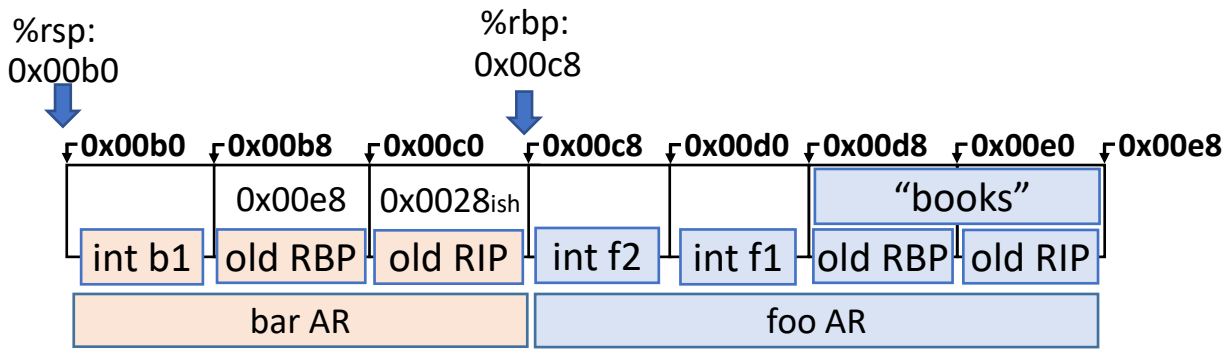
addq $8, %rsp

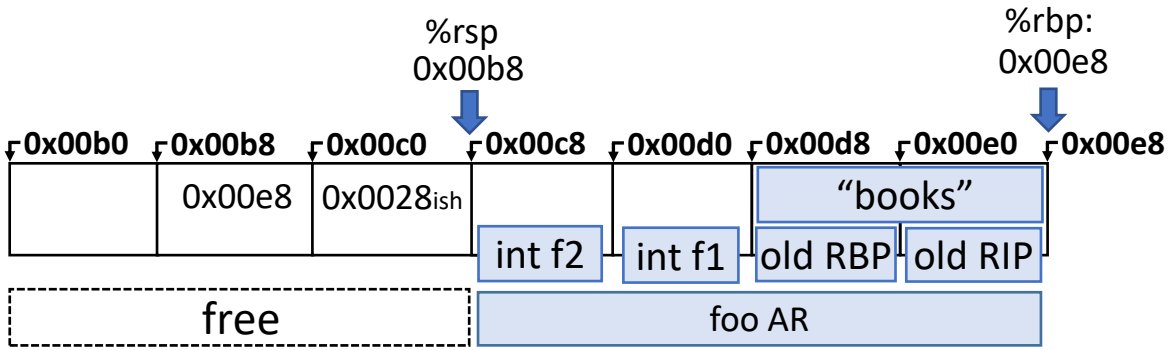
popq %rbp

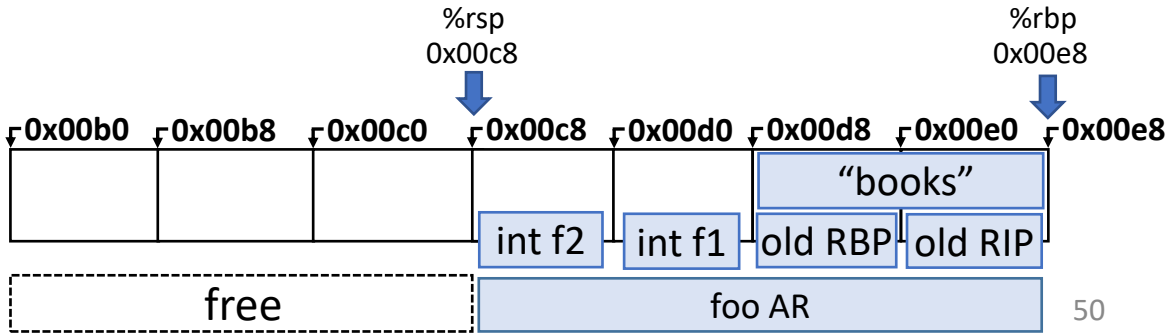
retq

Done!

Finishing off ARs







Activation Record

Finishing off ARs

Purpose: Save the locals in dynamic memory

Solution: Use the stack

Relative addresses for locals

Solution: Adjust %rbp for current function

Complication: Each function needs to re-use %rbp

Solution: Save %rbp in stack frame

Activation Record

Finishing off ARs

