Exam 2 topics

Lectures

Programming with Memory

x86 Basics

x86 Control Flow

x86 Procedures, Call Stack

Representing Data Structures

Buffer Overflows

Processes Model

Shells

Labs

Pointers in C

x86 Assembly

x86 Stack

Data structures in memory

Buffer overflows

Processes

Topics

C programming: pointers, dereferencing, arrays, structs, cursor-style programming, using malloc

x86: instruction set architecture, machine code, assembly language, reading/writing x86, basic program translation

Procedures and the call stack, data layout, security implications

Processes, shell, fork, wait

Assignments

Pointers

x86

Buffer

Concurrency

Exam 2: ISA + Process/Shell
December 5
(Thursday after break)

Practice problems

For Exam 2: ISA

Fun attendance question: what is something you ate or did over break that brought you joy?

♡ 0

Nobody has responded yet.

Hang tight! Responses are coming in.

Struct practice problem (similar to CSAPP 3.45)

```
struct s {
    char *a;
    short b;
    int *c;
    char d;
    int e;
    char f;
};
```
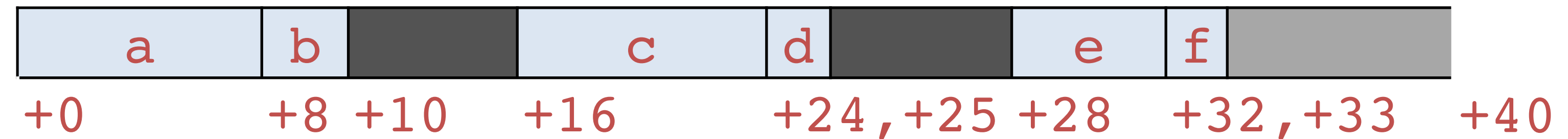
1. Draw a picture of how this struct is laid out in memory, labeling the byte offset of each field (starting with `a` at offset `+0`);

2. Modify your picture to show how much space a single element of this struct would take if used as an element of an array (e.g., the total size).

Recall: a short is 2 bytes in C

3. Rearrange the fields of the struct to minimize wasted space. Draw the new offsets and the total size.

Struct practice problem (similar to CSAPP 3.45)

```
struct s {
   char *a;
   short b;
   int *c;
   char d;
   int e;
   char f;
};
```

1. Draw a picture of how this struct is laid out in memory, labeling the byte offset of each field (starting with `a` at offset $+0$);

a	b		c	d		e	f	

+0   +8 +10   +16   +24,+25 +28   +32,+33   +40

2. Modify your picture to show how much space a single element of this struct would take if used as an element of an array (e.g., the total size).
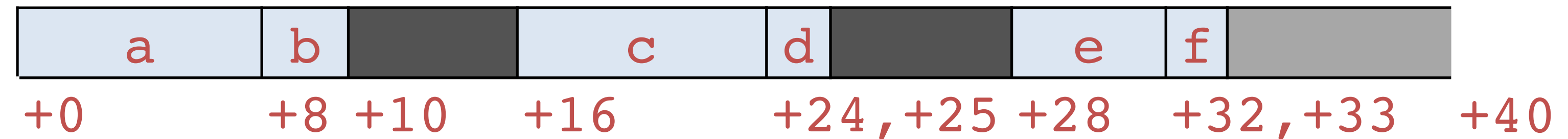
Recall: a short is
2 bytes in C

5

Struct practice problem (similar to CSAPP 3.45)

```
struct s {
    char *a;
    short b;
    int *c;
    char d;
    int e;
    char f;
};
```
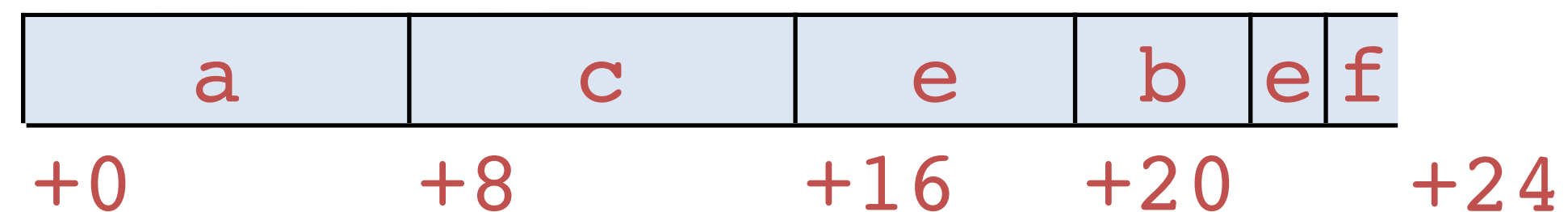
1. Draw a picture of how this struct is laid out in memory, labeling the byte offset of each field (starting with `a` at offset `+0`);

a	b		c	d		e	f	

+0   +8 +10   +16   +24,+25 +28   +32,+33 +40

2. Modify your picture to show how much space a single element of this struct would take if used as an element of an array (e.g., the total size).

Recall: a short is 2 bytes in C

3. Rearrange the fields of the struct to minimize wasted space. Draw the new offsets and the total size.

a	c	e	b	e	f

+0   +8   +16   +20   +24

Fun attendance question: what snacks/drinks would you like for the last CS240 class next Monday?

♥ 0

Nobody has responded yet.

Hang tight! Responses are coming in.

2-D array practice problem

```
long a[2][3];
```

1. Draw a picture of how this array is laid out in memory, labeling the indices and byte offset of each element (starting with `a[0][0]` at offset $+0$);

Recall: `index = C*r + c`

scale by element size

```
long get_elem_1_2(long a[2][3]){
    return a[1][2];
}
```
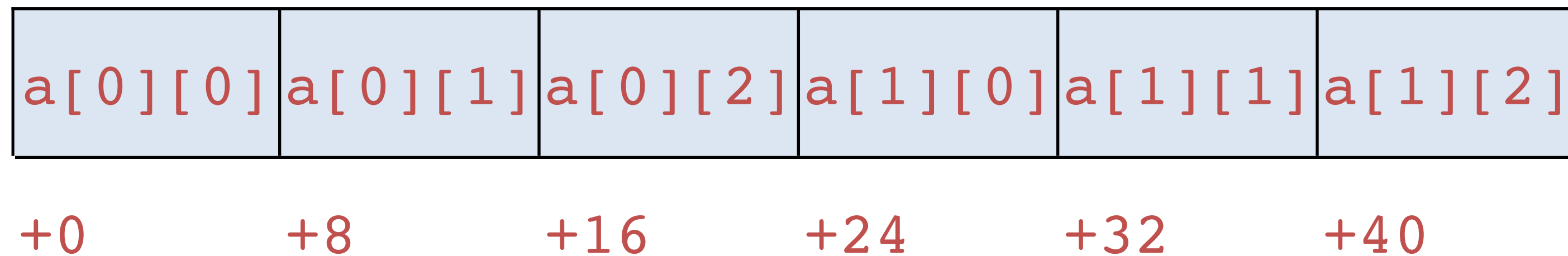
2. Write x86 assembly code to implement this function.

2-D array practice problem: solution

```
long a[2][3];
```

1. Draw a picture of how this array is laid out in memory, labeling the indices and byte offset of each element (starting with `a[0][0]` at offset $+0$);

Recall: $index = C*r + c$

scale by element size

a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]	a[1][2]

+0   +8   +16   +24   +32   +40

```
long get_elem_1_2(long a[2][3]){
  return a[1][2];
}
```

2. Write x86 assembly code to implement this function.

Since we know the size, we can calculate
C*r+c = 3*1+2 = 5, 5*sizeof(long) = 5*8 = 40

```
movq 40(%rdi),%rax
retq
```

x86 struct/LinkedList practice problem

```
nodeFunc2:
        pushq    %rbp
        pushq    %rbx
        subq     $8, %rsp
        movl     %esi, %ebx
        movslq   %esi, %rax
        testq    %rdi, %rdi
        je       .L1
        movq     %rdi, %rbp
        movl     8(%rdi), %esi
        cmpl     %esi, %ebx
        jb       .L5
.L3:    movq     0(%rbp), %rdi
        movl     %ebx, %esi
        call     nodeFunc2
.L1:    addq     $8, %rsp
        popq     %rbx
        popq     %rbp
        ret
.L5:    movl     %esi, %ebx
        jmp      .L3
```

```
typedef struct Node {
  struct Node* next;
  unsigned int value;
} Node;

long nodeFunc2(Node* node, unsigned int x) {
    // ???
}


long nodeFunc1(Node* node) {
    nodeFunc2(node, 0);
}
```

Consider the above function that calculates something useful about a linked list of unsigned integers using a helper function.

1. Identify which pieces of x86 refer to `next` and `value`.
2. Identify the base case of the recursive function `nodeFunc2`. What is returned in this case?
3. Identify the recursive case of `nodeFunc2`. What is the argument passed to the recursive call?
4. What is `nodeFunc1` calculating with helper `nodeFunc2`?

x86 struct/LinkedList practice problem

```
nodeFunc2:
        pushq    %rbp
        pushq    %rbx
        subq     $8, %rsp
        movl     %esi, %ebx
        movslq   %esi, %rax
        testq    %rdi, %rdi
        je       .L1
        movq     %rdi, %rbp
        movl     8(%rdi), %esi
        cmpl     %esi, %ebx
        jb       .L5
.L3:    movq     0(%rbp), %rdi
        movl     %ebx, %esi
        call     nodeFunc2
.L1:    addq     $8, %rsp
        popq     %rbx
        popq     %rbp
        ret
.L5:    movl     %esi, %ebx
        jmp      .L3
```

At call, `%rsp` must be a multiple of 16

```c
typedef struct Node {
  struct Node* next;
  unsigned int value;
} Node;                    2nd argument renamed for clarity
long nodeFunc2(Node* node, unsigned int max) {
  if (node == 0) {  node = %rdi
    return max;    base case
  }
  if (node->value > max) {  note order of x86 comparison
    max = node->value;
  }
  nodeFunc2(node->next, max);
}  recursive case
long nodeFunc1(Node* node) {
    nodeFunc2(node, 0);
}
```

`8(%rdi)` accesses `node->value`, `(%rdi)` accesses `node->next`, `%rdi` accesses `node`, the pointer itself

if (`node->value > x`), jump to `.L5`, sets `%ebx` to `node->value`

`%ebx` calculates the max of `node->value` and `x`

in the base case, returns second arg, `x` (the maximum value found so far)

`nodeFunc1` uses its helper to **find the maximum value** within a linked list.

11

x86 recursive procedure practice problem
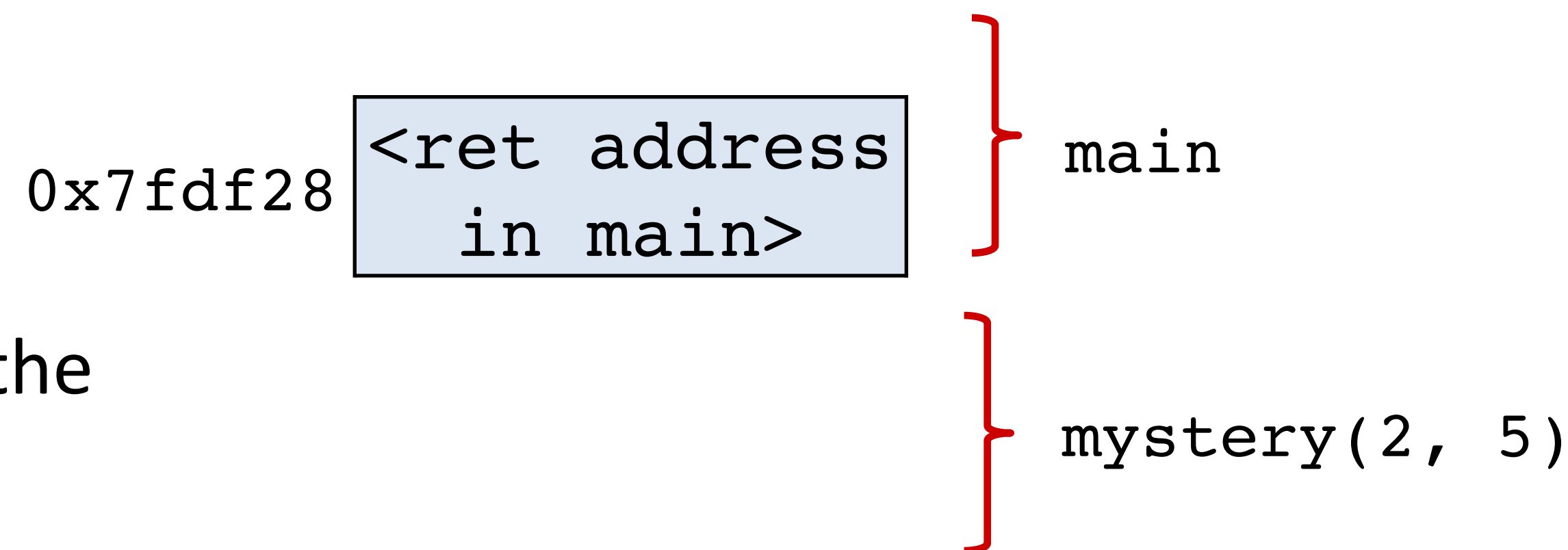
```
       mystery:
401106 mov      $0x0,%eax
40110b test     %edi,%edi
40110d jne      401110 <mystery+0xa>
40110f ret
401110 push     %rbx
401111 mov      %esi,%ebx
401113 sub      $0x1,%edi
401116 call     401106 <mystery>
40111b movslq   %ebx,%rsi
40111e add      %rsi,%rax
401121 pop      %rbx
401122 ret
```

1. What register is being saved to the stack? Why?

2. What instruction address gets saved to the stack? Why?

3. What is this function computing?

0x7fdf28 <ret address in main> } main

} mystery(2, 5)

4. Fill in the top of this stack after the function returns to main for

mystery(2, 5).

What is each value returned, in order?

12

x86 recursive procedure practice problem
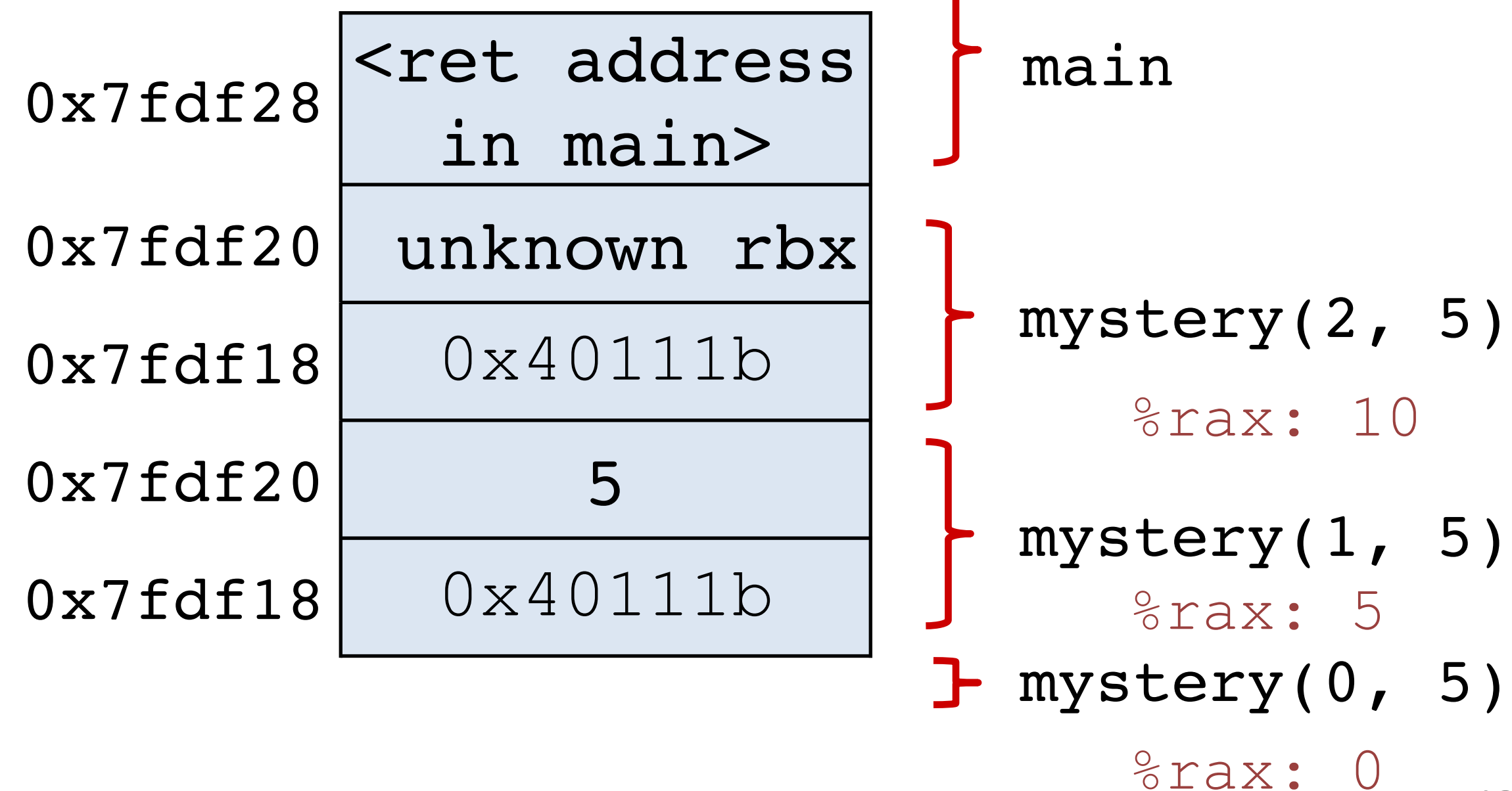
```
        mystery:
401106 mov     $0x0,%eax
40110b test    %edi,%edi
40110d jne     401110 <mystery+0xa>
40110f ret
401110 push    %rbx
401111 mov     %esi,%ebx
401113 sub     $0x1,%edi
401116 call    401106 <mystery>
40111b movslq  %ebx,%rsi
40111e add     %rsi,%rax
401121 pop     %rbx
401122 ret
```

```
int mult(int x, int y) {
    if (x == 0)  return 0;
    return y + mult(x – 1, y);
}

%rax: 0
```

1. What register is being saved to the stack? Why?
   %rbx, so that it is not overwritten in the recursive call
2. What instruction address gets saved to the stack? Why?
   0x40111b, return address after recursive call
3. What is this function computing?

   Multiplies its two arguments

Address	Value	
0x7fdf28	<ret address in main>	main
0x7fdf20	unknown rbx	mystery(2, 5)
0x7fdf18	0x40111b	
		%rax: 10
0x7fdf20	5	mystery(1, 5)
0x7fdf18	0x40111b	%rax: 5
		mystery(0, 5)
		%rax: 0

13

x86 short answer practice problems

ex

1. Which x86 instructions implicitly change the stack pointer? How do they change it?

2. What are some things defined by the *word size*? What is the word size we have been using for x86 in class?

3. Describe the general idea of a buffer overflow exploit in C code compiled to x86.

4. Describe how a child process's memory is related to the memory of the parent process.

x86 short answer practice problems

ex

1. Which x86 instructions implicitly change the stack pointer? How do they change it?

pushq	popq	call	ret
`%rsp -= 8`	`%rsp += 8`	`%rsp -= 8`	`%rsp += 8`

2. What are some things defined by the *word size*? What is the word size we have been using for x86 in class? Register size, address size, pointer size

NOT instruction size (variable-width instruction size)

3. Describe the general idea of a buffer overflow exploit in C code compiled to x86.

Buffer overflow occurs when code lacks bounds checking in writing untrusted input to a destination region of memory that is too small. Buffer overflow attacks can overwrite the return addresses on the stack to point to further exploit code.

4. Describe how a child process's memory is related to the memory of the parent process.

The child process starts with a copy of the state of the parent's memory. It is a private copy: the child and the parent do not share memory once the child is created.

x86 arithmetic practice problem

```
long funmath0(long x, long y) {
    return x + 4*y + 21;
}
```

```
long funmath1(long x, long y) {
    return 2*x + 4*y + 21;
}
```

```
long funmath2(long x, long y) {
    return 6*x + 5*y + 21;
}
```

Implement the above functions in x86 *without* `addq` or `mulq`.
You can use `leaq` and any other x86 instruction.

Recall: addressing modes can only multiply by 1, 2, 4, or 8.

x86 arithmetic practice problem

ex

```
long funmath0(long x, long y) {
    return x + 4*y + 21;
}
```

```
funmath0:
    leaq    21(%rdi,%rsi,4), %rax
    ret
```

```
long funmath1(long x, long y) {
    return 2*x + 4*y + 21;
}
```

```
funmath1:
    leaq    (%rdi,%rsi,2), %rax
    leaq    21(%rax,%rax), %rax
    ret
```

```
funmath2:
    leaq    (%rdi,%rdi,2), %rdx
    leaq    (%rsi,%rsi,4), %rax
    leaq    21(%rax,%rdx,2), %rax
    ret
```

```
long funmath2(long x, long y) {
    return 6*x + 5*y + 21;
}
```

Implement the above functions in x86 *without* `addq` or `mulq`.

You can use `leaq` and any other x86 instruction.

Recall: addressing modes can only multiply by 1, 2, 4, or 8.