# Practice problems

For Exam 2: ISA

# 2-D array practice problem

ex

```
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.

# x86 arithmetic practice problem

ex

```
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 struct/LinkedList practice problem

ex

```
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`?

# Struct practice problem (similar to CSAPP 3.45)

ex

```
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.

# x86 recursive procedure practice problem

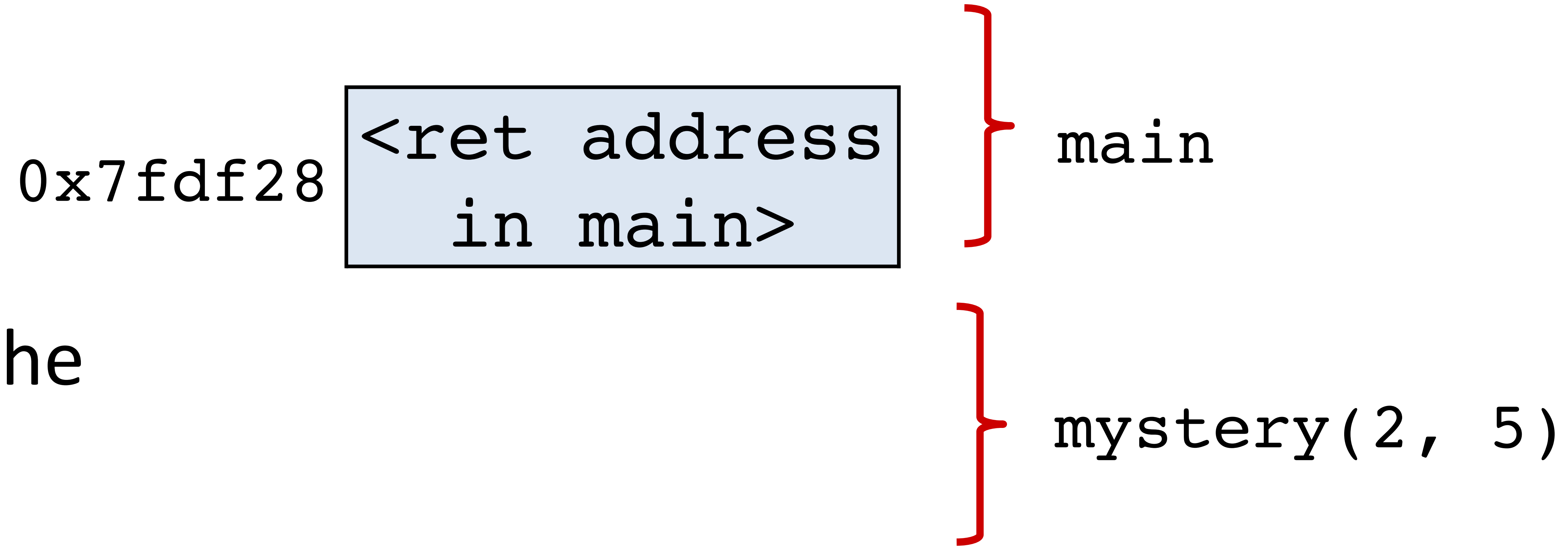
ex

```
        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 registers is being saved to the stack? Why?

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

3. What is this function computing?



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?

# x86 short answer practice problems

ex

1. Which x86 instructions implicitly modify the stack? In what ways does each change the stack pointer?

2. What are some things defined by the word size in x86? 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.