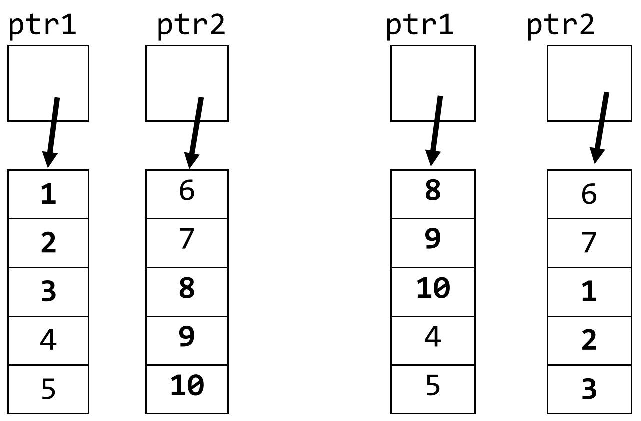
Pointers and Generics

Recall our generic `swap` function from class (reproduced below). It is used to make two values trade places in memory, and is commonly used in sorting arrays. There's a right way to call this swap function in normal circumstances, but we're asking you to use it a bit "creatively" to achieve particular results. Note: what matters for the correctness of these results is that if you were to print the contents of what ptr1 and ptr2 point to (see comment in code), it would match the "after."

```
void swap(void *a, void *b, size_t sz) {
    char tmp[sz];
    memcpy(tmp, a, sz);
    memcpy(a, b, sz);
    memcpy(b, tmp, sz);
}
```

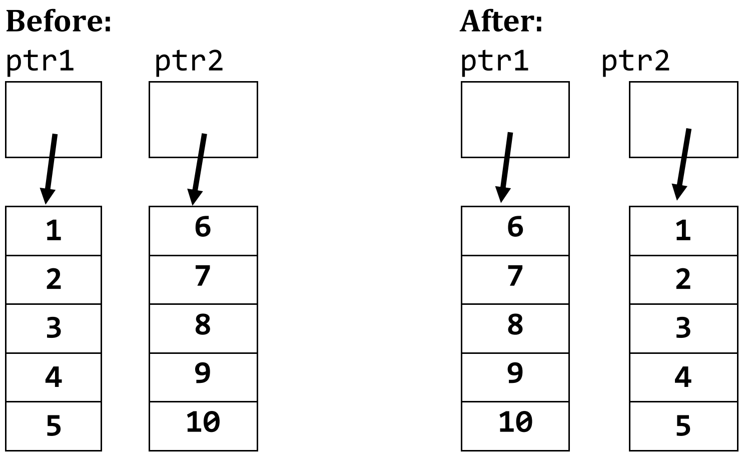
(a) Complete the `mixup1` function to create this before and after result. Your solution must consist of ONLY completing the arguments of the one call to `swap`, as shown.

Before: **After:**



```
void mixup1(int *ptr1, int *ptr2) {

        swap(                                    ,

                                                 ,

                                                  );

}
```

(b) Complete the `mixup2` function to create this before & after result. Your solution must consist of ONLY completing the arguments of the one call to `swap`, as shown. *In this case, the third argument should not be edited other than to specify a single argument (that should be a standard type) to* `sizeof()`.



```
void mixup2(int *ptr1, int *ptr2) {

        swap( [                              ],

              [                              ],

              sizeof(  [              ]  ));
}
```

Assembly

Consider the following x86-64 code output by gcc using the settings we use for this class (-Og):

```
<ham>:
  mov    (%rdi),%eax
  lea    (%rax,%rax,2),%esi
  add    %esi,%esi
  mov    $0x0,%ecx
  imul   $0x31,%esi
  jmp    L1
L3:
  lea    (%rcx,%rax,1),%edx
  movslq %edx,%rdx
  mov    %esi,(%rdi,%rdx,4)
  add    $0x2,%eax
  jmp    L2
L4:
  mov    %ecx,%eax
L2:
  cmp    $0x9,%eax
  jle    L3
  add    $0x3,%ecx
L1:
  cmp    $0x9,%ecx
  jle    L4
  mov    $0xa,%eax
  retq
```

(a) Fill in the C code below so that it is consistent with the above x86-64 code. Your C code should fit the blanks as shown, so do not try to squeeze in additional lines or otherwise circumvent this (this may mean slightly adjusting the syntax or style of your initial decoding guess to an equivalent version that fits). Your C code should not include any casting. Note that with the compiler set to `-Og`, some optimization has been performed. One thing you'll notice right away is that gcc chose not to create an actual `eliza` array, but instead kept track of its values in other ways. We will ask about optimizations in more detail in later parts of this question.

```
int ham(int *burr) {
    int eliza[4];
    eliza[0] = 7;
    eliza[1] = 7;
    eliza[2] = 1;

    eliza[3] = _____________ * burr[0];    // part (b)


    for (int i = 0; i < ___________; i+=___________) {


        for (int j = ___________; j < ___________; j+=___________) {


            burr[__________] = eliza[0]*eliza[1]*eliza[2]*eliza[3]; //(c)


        }

    }

    if (eliza[0] > eliza[1]) {                   // part (d)

        return 8;

    }

    if (burr[0] < burr[1] && burr[0] > burr[1]) { // part (d)

        return 9;

    }

    return ___________;

}
```

(b) Refer back to the C code, on the line marked for part (b). It reads:

```
        eliza[3] = … * burr[0];
```

Name and explain the instruction(s) that implement this product, and explain <u>why</u> gcc would choose to do it that way.

Assembly

For the following parts, to the following x86-64 code output by gcc using the settings we use for this class (-Og):

```
<ham>:

      40052d:       shl    $0x4,%edi
      400530:       mov    %edi,%r9d
      400533:       mov    %edi,%r10d
      400536:       mov    $0x0,%eax
      40053b:       lea    0x2(%rdi),%edi
      40053e:       jmp    400562 <ham+0x35>
      400540:       movslq %edx,%rcx
      400543:       add    (%rsi,%r8,8),%rcx
      400547:       movb   $0x58,(%rcx)
      40054a:       add    %r9d,%eax
      40054d:       add    $0x3,%edx
      400550:       jmp    40055a <ham+0x2d>
      400552:       mov    $0x0,%edx
      400557:       movslq %r10d,%r8
      40055a:       cmp    %edx,%edi
      40055c:       jg     400540 <ham+0x13>
      40055e:       sub    $0x1,%r10d
      400562:       test   %r10d,%r10d
      400565:       jg     400552 <ham+0x25>
      400567:       repz retq
```

(a) Fill in the C code below so that it is consistent with the above x86-64 code. Your C code should fit the blanks as shown, so do not try to squeeze in additional lines or otherwise circumvent this (this may mean slightly adjusting the syntax or style of your initial decoding guess to an equivalent version that fits). Your C code should not include any casting. Note that with the compiler set to `-Og`, some optimization has been performed. We will ask about optimizations in more detail in later parts of this question. There is an ASCII table on the following page.

```
int ham(int aaron, char **alex)
{

    int burr = _______________________________;

    for (int i = ______________ * ___________; /* see part (b) */

                      i > _________________; ________________) {

        for (int j = __________; j < ___________________;

                                     ___________________) {

            alex[i][j] = 'X';

            _____________________ += _________________________;

        }

    }

    return burr;

}
```

ASCII Character Codes (Decimal)

0	Ctrl-@	32	Space	64	@	96	`
1	Ctrl-A	33	!	65	A	97	a
2	Ctrl-B	34	"	66	B	98	b
3	Ctrl-C	35	#	67	C	99	c
4	Ctrl-D	36	$	68	D	100	d
5	Ctrl-E	37	%	69	E	101	e
6	Ctrl-F	38	&	70	F	102	f
7	Ctrl-G	39	'	71	G	103	g
8	Backspace	40	(	72	H	104	h
9	Tab	41	)	73	I	105	i
10	Ctrl-J	42	*	74	J	106	j
11	Ctrl-K	43	+	75	K	107	k
12	Ctrl-L	44	,	76	L	108	l
13	Return	45	-	77	M	109	m
14	Ctrl-N	46	.	78	N	110	n
15	Ctrl-O	47	/	79	O	111	o
16	Ctrl-P	48	0	80	P	112	p
17	Ctrl-Q	49	1	81	Q	113	q
18	Ctrl-R	50	2	82	R	114	r
19	Ctrl-S	51	3	83	S	115	s
20	Ctrl-T	52	4	84	T	116	t
21	Ctrl-U	53	5	85	U	117	u
22	Ctrl-V	54	6	86	V	118	v
23	Ctrl-W	55	7	87	W	119	w
24	Ctrl-X	56	8	88	X	120	x
25	Ctrl-Y	57	9	89	Y	121	y
26	Ctrl-Z	58	:	90	Z	122	z
27	Escape	59	;	91	[	123	{
28	Ctrl-\	60	<	92	\	124	|
29	Ctrl-]	61	=	93	]	125	}
30	Ctrl-^	62	>	94	^	126	~
31	Ctrl-_	63	?	95	_	127	Delete

(b) Refer back to the C code for ham, on the line marked for part (b) (a multiply operator between two blanks). **Name and explain** the instruction(s) that calculate this multiplication, and how/why gcc optimized here:

(c) Refer to the following C and x86-64 code:

```
int eliza(char *peggy)
{
    int len = strlen(peggy);
    if (len == 8) return 8;
    else return len;
}

<eliza>: // optimized (-O2)
  400569:   sub     $0x8,%rsp
  40056d:   callq   400410 <strlen@plt>
  400572:   add     $0x8,%rsp
  400576:   retq
```

You'll notice for eliza that although the C code includes an if statement, there are no conditional jumps in the assembly code. **Explain** how/why gcc optimized here.

Assembly

Consider the following x86-64 code output by gcc using the settings we use for this class. This function calls another function, story, and you will be asked to reverse-engineer both of them.

```
0000000000400511 <schuyler>:
  400511:       push   %rbx
  400512:       sub    $0x10,%rsp
  400516:       mov    %edi,%ebx
  400518:       mov    $0x4005c4,%edx
  40051d:       lea    0xc(%rsp),%rsi
  400522:       callq  4004ed <story>
  400527:       lea    (%rbx,%rax,2),%eax
  40052a:       add    $0x10,%rsp
  40052e:       pop    %rbx
  40052f:       retq
```

(a) Fill in the C code below so that it is consistent with the x86-64 code above for schuyler. Your C code should fit the blanks as shown, so do not try to squeeze in additional lines or otherwise circumvent this. This may mean adjusting the syntax, style, or expression of your initial decoding guess to an equivalent version that fits the structure of the provided C code. All int literals in your C code <u>must be written in decimal</u>.

```
int schuyler(int peggy)
{
    int angelica;

    int eliza = story(______________________________,

                      ______________________________, "helpless");

    ______________________________ *= 2;

    return ___________________________ + ___________________________;
}
```

(b) Now fill in the story function. Note that you aren't expected to have memorized the precise ASCII value of the letter 'f' that appears in the C code, but you should be able to infer its hexadecimal value in the x86-64 code, and thus be able to complete the line of code. All int literals in your C code must be written in decimal.

```
00000000004004ed <story>:
  4004ed:       cmpb   $0x66,(%rdx)
  4004f0:       jne    4004f6 <story+0x9>
  4004f2:       mov    %edi,(%rsi)
  4004f4:       jmp    4004fc <story+0xf>
  4004f6:       movl   $0x18,(%rsi)
  4004fc:       mov    $0x0,%eax
  400501:       jmp    400509 <story+0x1c>
  400503:       add    $0x4c,%eax
  400506:       sub    $0x2,%edi
  400509:       test   %edi,%edi
  40050b:       jns    400503 <story+0x16>
  40050d:       lea    (%rax,%rax,2),%eax
  400510:       retq
```

```
int story(int raise, int *glass, char *freedom)
{
    if (_____________________________ == 'f') {

        _____________________________ = _____________________________;

    } else {

        _____________________________ = _____________________________;

    }

    int tonight = _____________________________;

    for (int i = ____________________; i >= 0; i -= _____________) {

        tonight += ______________;

    }

    return __________________________ * __________________________;
}
```