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.”

```c
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:**

<table>
<thead>
<tr>
<th>ptr1</th>
<th>ptr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

**After:**

<table>
<thead>
<tr>
<th>ptr1</th>
<th>ptr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

```c
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()`.

Before:

<table>
<thead>
<tr>
<th>ptr1</th>
<th>ptr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
</tr>
</tbody>
</table>

After:

<table>
<thead>
<tr>
<th>ptr1</th>
<th>ptr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
</tr>
</tbody>
</table>

```c
void mixup2(int *ptr1, int *ptr2) {
    swap((ptr1), (ptr2), 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.

```c
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:

```c
eliza[3] = ... * burr[0];
```

**Name and explain** the instruction(s) that implement this product, and explain why 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.

```c
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;
}
```
(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:

```c
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 must be written in decimal.

```c
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.

```c
int story(int raise, int *glass, char *freedom) {
    if (____________________________ == 'f') {
       ____________________________ = ____________________________;
    } else {
       ____________________________ = ____________________________;
    }
    int tonight = ____________________________;
    for (int i = ______________; i >= 0; i -= _____________) {
        tonight += ______________;
    }
    return _________________________ * _________________________;
}
```