Mid-Quarter Assessment Solutions

1. Sticky Search

```
// blank 1
strcmp(searchIn, searchFor) == 0
or
strstr(searchIn, searchFor) != NULL && (strlen(searchIn) == strlen(searchFor))

// blank 2
strncmp(searchIn, searchFor, strlen(searchFor)) == 0
or
strstr(searchIn, searchFor) == searchIn

// blank 3
if (strlen(searchIn) < strlen(searchFor)) {
    return NULL;
}
char *endStr = searchIn + strlen(searchIn) - strlen(searchFor);
if (strcmp(endStr, searchFor) == 0) {
    return endStr;
}

// blank 4
strstr(searchIn, searchFor);
```

2. Spring Cleaning

Part 1: `springCleaning`

Sample Solution 1

```
void springCleaning(int *arr[], size_t nelems) {
    for (int i = 0; i < nelems; i += 2) {
        *(arr[i]) += *(arr[i + 1]);
        free(arr[i + 1]);
        arr[i + 1] = arr[i];
    }
}
```

Sample Solution 2

```
void springCleaning(int *arr[], size_t nelems) {
    for (int i = 0; i < nelems; i += 2) {
        *(arr[i + 1]) += *(arr[i]);
        free(arr[i]);
        arr[i] = arr[i + 1];
    }
}
```

Part 2: Memory Errors

The first memory error is that the size passed to **malloc** is incorrect. It uses the number of elements instead of the number of bytes. The **malloc** size should multiply the specified value by **sizeof(int *).**

The second memory error is that the heap-allocated integers are freed twice in the loop at the end, since **springCleaning** modifies the array such that each pair of pointers points to the same heap-allocated integer. To correctly free the heap-allocated integers, we would only want to free pointers at *odd-numbered* indexes, since that's the last time we will use the heap-allocated integer it points to.

3. A Roll Of The Dice

Sample Solution

```
int *getDiceRolls(int *lengthPtr) {
    int capacity = 2;
    int *rolls = malloc(capacity * sizeof(int));
    assert(rolls != NULL);
    int used = 0;

    int roll = rollDice();
    while (roll != 1) {
        if (used == capacity) {
            capacity += 2;
            rolls = realloc(rolls, capacity * sizeof(int));
            assert(rolls != NULL);
        }

        rolls[used] = roll;
        used++;
        roll = rollDice();
    }

    *lengthPtr = used;
    return rolls;
}
```

4. containsLargerThan

Part 1: containsLargerThan

```
bool containsLargerThan(void *base, size_t nelems,
                        size_t elem_size_bytes, void *elem,
                        int (*cmp_fn)(const void *, const void *)) {
    for (int i = 0; i < nelems; i++) {
        void *ith = (char *)base + i * elem_size_bytes;
        if (cmp_fn(ith, elem) > 0) {
            return true;
        }

        // or if (cmp_fn(elem, ith) < 0) return true;
    }

    return false;
}
```

Part 2: cmpStringsDesc

```
int cmpStringsDesc(const void *a, const void *b) {
    const char *strA = *(const char **)a;
    const char *strB = *(const char **)b;

    return strlen(strB) - strlen(strA);
}
```

5. uchars

Part 1: filluchars

```
unsigned int filluchars(uchar a, uchar b, uchar c, uchar d) {
    return (a << 24) | (b << 16) | (c << 8) | d;
}
```

Part 2: pulluchar

```
uchar pulluchar(unsigned int value, int ucharLocation) {
    return (value >> (ucharLocation << 3)) & 0xff;
}
```