Fall 2019 CS 31

Project 6 Solutions

1.

a. First, because of the operators' order of precedence, the expression `*ptr + 1 = 20` means `(*ptr) + 1 = 20`. The expression `(*ptr) + 1` evaluates to the int 31, not an int variable that can be assigned to. When corrected to `*(ptr+1) = 20`, the expression means `*(&arr[1]) = 20`, which means `arr[1] = 20`.

   Second, the while loop doesn't access `arr[2]`, and tries to access `arr[-1]`. One possible fix is

```
for (ptr = arr; ptr < arr + 3; ptr++)
        cout << *ptr << endl;
```

b. `findMax` puts the correct value in `pToMax`, but `pToMax` is a *copy* of the caller's variable `ptr`, so `findMax`has no effect on `ptr`. The parameter `pToMax` must be passed by reference, not by value:

```
void findMax(int arr[], int n, int*& pToMax)
```

c. The declaration `int* ptr;` declares `ptr` to be a pointer to int, but leaves it uninitialized — it does not point to any particular int. That uninitialized pointer is copied into the parameter `ncubed`. In the expression `*ncubed = n*n*n`, the attempt to dereference the uninitialized `ncubed` pointer leads to undefined behavior. A fix would be to make sure `computeCube` is passed a valid pointer; one possibility is, in the main routine, to say:

```
int k;
int* ptr = &k;
```

d. The test `str1 != 0` is asking if the `str1` pointer itself has a value different from the null pointer. (The integer constant `0` used in a context where a pointer is required means the null pointer.) The test we want, though, is to see if the character *pointed to* by `str1` is different from the zero byte that marks the end of a C string. (The same applies to `str2`.)

   Similarly, the test `str1 != str2` is asking whether those two pointers have different values (i.e., they point to different places). But what should be tested is whether the characters they *point to* have different values. (The same applies to `str1 == str2`.)

   The corrected function body is thus

```
while (*str1 != '\0'  &&  *str2 != '\0')  // 0 instead of '\0' is also OK
{
    if (*str1 != *str2)
        return false;
    str1++;
    str2++;
}
return *str1 == *str2;
```

e. The storage for the local variable `anArray` goes away when the function `getPtrToArray` returns. But that function returns a pointer to that storage. Attempting to follow that pointer in the main routine (implied by `ptr[i]`) yields undefined behavior.

2.

a. `double* cat;`

b. `double mouse[5];`
c. `cat = &mouse[4];` or `cat = mouse + 4;`
d. `*cat = 25;`
e. `*(mouse + 3) = 54;`
f. `cat -= 3;`
g. `cat[1] = 27;`
h. `cat[0] = 42;`
i. `bool b = *cat == *(cat+1);`
j. `bool d = cat == mouse;` or `bool d = cat == &mouse[0];`

3.

a.
```
double mean(const double* scores, int numScores)
{
    int k = 0;
    double tot = 0;
    while (k != numScores)
    {
        tot += *(scores + k);
        k++;
    }
    return tot/numScores;
}
```

b.
```
const char* findTheChar(const char* str, char chr)
{
    for (int k = 0; *(str+k) != 0; k++)
        if (*(str+k) == chr)
            return str + k;

    return nullptr;
}
```

c.
```
const char* findTheChar(const char* str, char chr)
{
    for ( ; *str != 0; str++)
        if (*str == chr)
            return str;

    return nullptr;
}
```

4.
```
3       (see note 4 below)
4       (see notes 1, 5, and 6 below)
79      (see notes 3 and 5 below)
-1      (see note 6 below)
9       (see note 2 below)
22
19
```

1. `maxwell` is called with pointers to `array[0]` and `array[2]`. It returns a pointer to whichever of the ints pointed to has a larger value. Since `array[0]` has the larger value, the function returns `&array[0]`. The expression `*ptr = -1` sets `array[0]` to -1.
2. `ptr[i] = 9;` sets `array[3]` to 9.
3. `*(array+1) = 79;` sets `array[1]` to 79.
4. `&array[5] - &array[2]` is 3.
5. The `swap1` function swaps its *copies* of the pointers passed in to it, with no effect on the ints pointed to.
6. The `swap2` function swaps the ints pointed to.

5.
```
void removeS(char* source)
{
    char* destination = source;
    for ( ; *source != '\0'; source++)
    {
        if (*source != 's'  &&  *source != 'S')
        {
            *destination = *source;
            destination++;
        }
    }
    *destination = '\0';  // Don't forget the zero byte at the end
}
```

or

```
void removeS(char* source)
{
    char* destination = source;
    while (*source != '\0')
    {
        if (*source != 's'  &&  *source != 'S')
        {
            *destination = *source;
            destination++;
        }
        source++;
    }
    *destination = '\0';  // Don't forget the zero byte at the end
}
```