Solutions Q1 -- Q5

Q1

The output is `4294967271`. This is because `25u` defined an unsigned integer through the suffix `u`. However, an unsigned integer can not represent negative values, thus causing underflow. Due to the size of underflow `25`, the value returned equals `2^32 - 25`, as an unsigned integer takes 32 bits.

Q2

The output is

```
i: 7
j: 5
k: 7
```

This is because `i++` and `++i` both increase the value of `i` by 1, although the former increases the value after assignment and the latter before assignment. That's why `j = 5`, as `i` increases after `j` is initialized, and `k = 7`, as `i` increases and the assigned to `k`.

Q3

The result is

```
main: a: 3, b: 5
swap1: a: 5, b: 3
main: a: 3, b: 5
swap2: *a: 5, *b: 3
main: a: 5, b: 3
swap3: a: 3, b: 5
main: a: 3, b: 5
```

The first line shows the initial values of `a` and `b`.

The first function `swap1` did not change the value of `a` and `b` because by inputting `a` and `b` the programme creates a copy in local scope and does not affect the `a` and `b` in the `main()` function scope. The output of `swap1` gives swapped `a` and `b` as those outputs are the local `a` and `b`, which are swapped.

`swap2` inputs the pointers of the main scope `a` and `b`. Although the two pointers are passed to `swap2` as two copies, they still point to the `a` and `b` of the main scope. Hence swapping the values pointed by the two pointers actually swaps the main scope `a` and `b`. This results in the `*a` and `*b` in `swap2` being `5` and `3` and the same for scope `a` and `b`.

The last function `swap3` takes the references as inputs. By the nature of references, changing them directly changes the values referred to by them. Note that before this function, the values of `a` and `b` have already been changed to `5` and `3` by `swap2`. In the function, `int temp = a` initializes `temp` as a integer `5`. `a = b` sets the main scope `a` at 3, with function scope `a`'s value also set at 3. `b = temp` changing both the main scope and function scope `b` to 5.

Q4

The output is:

```
constructor called, x = 3
constructor called, x = 5
copy constructor called, x = 3
assignment operator called, x was 3 and became 5
assignment operator called, x was 3 and became 5
destructor called, x = 5
destructor called, x = 5
destructor called, x = 5
```

First `MyClass a{ 3 }; MyClass b{ 5 };` constructs `a` and `b` with the standard constructor using an `int` input. `MyClass c{ a };` takes a `MyClass` input and hence invokes the copy constructor, resulting in the underlying `int` to be `3`.

`c = b` invokes the assignment operator in `c`, setting the underlying `int` of `c` at `5`. Similarly for `a = b`, now the underlying `int` of `a`, `b`, and `c` are all `5`.

As we reach the end of the main function, the destructor has been automatically called, although the there is no `delete` in the destructor. The destructor simply returns the underlying `int` of `a`, `b`, and `c`, resulting in the last three lines of the output.

Q5

The output is:

```
I am const, x = 1
I am not const, x = 2
I am not const, x = 3
I am const, x = 4
I am const, x = 1
I am not const, x = 2
```

First the initialization sets `a` and `b` as `MyClass` object with underlying `int` 1 and 2, `c` and `d` as a pointer of `MyClass` objects with underlying `int` 3 and 4. `a` and `d` are `const` variables and `c` and `d` use `new` to manage their memory allocation. Afterwards, `e` and `f` are initialized as references to `a` and `b`, instead of as `MyClass` objects with underlying `a` and `b`. This makes them `MyClass` objects with underlying `int` 1 and 2.

The two `MyClass::get()` functions are called for non-`const` and `const` inputs due to the `const` following `get()` in the first function. That's why we see the programme determines whether our variables are `const` or not at the output.

Finally the memory for the pointers `c` and `d` are released using `delete`. We don't have to do this for the other variables as they are on stack memory and will be automatically destroyed at the end of the main scope.