Exercise 2: Const-ness

Clearly explain why `&s` sometimes matches the corresponding `&ncs`/`&cs` values and other times it does not.

Because the output of the '&' depends on how the string is passed to the function. If the string is passed by reference, the output will match the corresponding &ncs/&cs values, but if the string is passed by value, the output will be a different memory address altogether.

Clearly explain why `ncs` before/after only sometimes match, and why `cs` before/after always match.

The 'ncs' variable can be modified by functions that take it as a non-const reference, but not by functions that take it as a value. The 'cs; variable cannot be modified by any function because it is const, so its value before and after calling a function will always be the same.

Clearly explain why the two calls to `overloaded_func()` have different outputs.

Because the parameter specifies whether it will take a const or non-const argument. Because 'ncs' is non-const, the program will run cout << "overloaded_func() with non-const ref called on " << s << endl;. Since 'cs' is cost, the program will run cout << "overloaded_func() with const ref called on " << s << endl;.

Try uncommenting `LINE 1` and/or `LINE 2`. Clearly explain why the compiler always allows `s.size()` to be called but sometimes rejects calls to `s.append()`. Make sure these lines are re-commented when you're done.

The compiler allows s.size() to be called because size will simply return the length of the string. It won't edit the string size, making it accessible to both const and non-const variables. s.append() will modify the data in a given string, which is not allowed in const variables.

Uncomment `LINE 3`. Clearly explain why the compiler complains here but not at the next call to `non_const_by_val()`.

Because 'cs' is const and unmodifiable. Yet when we try to use the function 'non_const_by_ref', there is an error as that function asks to append (modify) the data directly. For non_const_by_val(), this simply takes a copy of 'cs', and that copy is allowed to be modified, which is why there is no error there.

Exercise 3: Type vs Class

Clearly explain what `LINE 1`, `LINE 2`, and `LINE 3` are doing.

- Line 1: Initializes and declares a string named 'x' with the value "hello" inside of it.
- Line 2: Initializes a reference string named 'rx' and sets it to the string x from line 1, this means that the location of 'rx' is the same as string 'x', and any changes to 'rx' or 'x' will make the same change to the other string as well.
- Line 3: Initializes a reference string named 'crx' and sets it to the string x from line 1, but is made const. This means that the value inside 'crx' cannot be changed forever. We are allowed to view the current 'x' string, but we cannot change 'x' through 'crx' like we could in string 'rx'.

Clearly explain the output values for `&x`, `&rx`, and `&crx`. What does this say about the number of string objects in this program?

- The output values are all the same, which is the location of string ‘x’. This is because for ‘rx’ and ‘crx’, we created a reference to string ‘x’ using ‘&’.
- Technically, although we have 3 strings in the form of ‘x’, ‘rx’, and ‘crx’, they all point to the same location and therefore to the memory, there is only one string object.

Try uncommenting each of `LINE 4`, `LINE 5`, and `LINE 6`. Why does `LINE 6` cause an error even though the string object `x` is non-const?

- Because even though object ‘x’ is non-const, we aren’t appending ‘x’, we are using the data from ‘x’, via ‘crx’ (which is its reference), to edit x. However, since ‘crx’ is const, we can never be able to change its value inside, meaning we cannot change ‘x’ through ‘crx’: ‘crx’ is almost like a read-only version of ‘x’.

Exercise 4: Basic Polymorphism and Virtual Functions

Clearly explain the output.

In this program, we have made 2 different objects: ‘c’ from the class ‘Cow;, and ‘fc’ from the class ‘FullCow’. Cow takes the properties from class ‘Animal’, and FullCow takes the properties from ‘Cow’. We then use the function ‘poke’ which calls the ‘make_sound’ function within both ‘Cow’ and ‘FullCow’, giving us the result of:

- moo ← from Cow
- Ooof - I ate too much ← from FullCow

Remove the `virtual` keyword on the `make_sound()` method in `Cow` and `FullCow`, and re-run the program. Why was `FullCow` still able to override `Cow`’s `make_sound()` method?

Because we declared make_sound() as virtual within the base ‘Animal’ class. Therefore, ‘Cow’ and ‘FullCow’ don’t need to explicitly have the keyword ‘virtual’ since they would inherit that property from their parent class.

Add a trailing final keyword to `Cow’s make_sound()` method. Why does the compiler complain?

Because ‘FinalCow’ has its own make_sound() method that is it trying to use, but since it is the child class of ‘Cow’, it has to use ‘override’ to use its own make_sound() method within its own class. Since ‘Cow’ made its make_sound() method final, the compiler complains because ‘FinalCow’ is trying to use ‘override’ on a function defined as ‘final’. In short, you can’t override a final function.

Exercise 5: Slicing

Compile and run my code. Clearly explain the output.

- Cow c was made. It holds its own make_sound() method. Thus, the output it provides is “moo”.
- Animal & cr was made and set to (Cow) c. By using ‘&’, it has made cr into a reference object to ‘c’. This means that it takes all the properties of (Cow) c, including its make_sound() method, without having to specifically be created as one. Thus, the output it provides is “moo”.

- Animal cs was made and set to (Cow) c. Since it is not a reference, it is not tied to the method properties of c. Therefore, it uses the Animal make_sound() method instead of the one provided in Cow. Thus, the output it provides is “I don't know how to make a sound!”

Exercise 6: lvalues and rvalues

Compile and run my code. Clearly explain each output line.

- func(int && x) rvalue ref - because 5 is a temporary expression and an rvalue
- func(int && x) rvalue ref - because 2+3 is a temporary expression and an rvalue
- func(int && x) rvalue ref - because int{5} is a temporary variable
- func(int & x): lvalue ref - because it refers to a defined int variable, x, and therefore is an lvalue
- func(int && x) rvalue ref - because even though x is involved, x/2 (=5) is not a defined variable, and therefore is a temporary expression, and thus an rvalue
- func(int && x) rvalue ref - this is a temporary expression because move(x) would return a rvalue
- func(int const & x): const lvalue ref - because it refers to a defined const int variable, y, and therefore is an const lvalue
- func(int && x) rvalue ref - because even though y is involved, int y/2 (=2) is not a defined variable, and therefore is a temporary expression, and thus an rvalue
- func(int const && x) const rvalue ref - this is a temporary expression because move(y) would return a rvalue

Disable the second `#if` pragma, and rerun the code. Explain the difference(s).

The only difference is that the second and eighth outputs used the func(int & x) overload instead of the func(int const & x)

Also disable the first `#if` pragma. Explain why the compiler complains.

Because now there are 2 int functions that take an &x as a parameter.