Programming for Engineers Portfolio : Quarter 2, Set 3

The deadline for this portfolio has been changed to Friday 20th at 22:00, with the sanity check now on Mon 16th at 9:00. Thanks to your year rep for bringing this up and collecting information on behalf of the cohort.

In the following it is fine to put supporting files within the directory (for example, source files to drive and test headers), and any scripts will be able to access supporting files (such as test input data). Prefixing them with the exercise number is probably a good idea, but not required nor checked for.

A number of the exercises are about using functions and types as APIs, which provide boundaries between parts of a program. Just like a program should be defined in terms of its IO, a function or API should be defined in terms of its prototypes and semantics, and there should be no assumption about how it is implemented.

An implicit requirement for these exercises it that they should not leak memory, cause undefined behaviour (e.g. through dangling pointers or dereferencing null), or otherwise crash. However, you can assume that any tests will not choose inputs that cause stack overflow in recursive functions.

Because of the order we've decided to take through C++ features, we haven't been able to discuss namespaces yet. That results in a necessary evil, in that you may end up with `using namespace std` in header files - none of the exercises require it, but you may still end up doing it, given we haven't told you what it means yet. So in the grand scheme of things that is bad, but for this portfolio it is acceptable to do so. Later on we'll see *why* that's a bad thing, but for now you can just treat header files like source files.

Errata

Changes since the original version:

- e1-e4 : Two of definitions in `slist.hpp` did not match the declarations. The intent is that the definitions always follow the declarations, so those who provided alternative definitions have done the right now. Fixed in the current version.
- e1-e4 : After some in-person queries the comments on the public API of slist have been elaborated a bit.
- e5 : The example interaction was wrong, as it talked about an older version of the exercise that included summation.

- e7 : There was a typo, asking for a source file called `.hpp`; it should have been `.cpp`.

The file `slist.hpp` declares a list type and a function for printing lists. The list has the same intepretation as the list in the lecture, so:

- An empty list is represented by `nullptr`
- A list points at the front of the list, and the back is at the end of the list.

While these exercises all work on one type, they are designed to be assessable (i.e. testable) independently, so if there is a mistake in one it shouldn't affect the others.

1. Create a header file `e1_slist_print.hpp` which defines a function with the following prototype:

   ```
   void e1_slist_print(slist *p);
   ```

   When called, the function should print the list to `cout`, one element per line, with the front of the list first.

   The function can be recursive or iterative.

   The function should *only* use the list functions in the public API (i.e. it should not manipulate the list nodes directly, and if the members of the `slist` type are modified your function should still work).

2. Create a header file `e2_slist_reverse.hpp` which defines a function with the following prototype:

   ```
   slist *e2_slist_reverse(slist *p);
   ```

   This function should return a *new* list with the same items as p, but with the elements in reverse. The original list p should remain unchanged, and the two lists should be distinct (no shared nodes).

   The function can be recursive or iterative. You may find an iterative method easier to think about, unless you're quite happy with recursive functions.

   You can create helper functions if necessary with the same prefix; for example, you could create a function `e2_slist_reverse_helper` that does some of the work. This is not required though, it depends how you choose to solve the problem.

   The function should *only* use the list functions in the public API.

   Hint: There is nothing wrong with searching for "How do I reverse a list?" to get some ideas about how to do it. It's just the code you shouldn't copy. Also, you'll find most descriptions want you to modify the list directly, but here we can't do that - you have to use the functions. Other methods will only work on doubly-linked lists, but here we only have a single link.

Personally I find the easiest method is to draw diagrams:

```
Original list: --->A--->B--->C

Return list:       A<---B<---C---
```

If you visit each node in the original list in sequence, how do you build the list in the return list at the same time?

3. The header file `slist.hpp` has a declaration for `slist_append`, but the current definition immediately aborts via `assert`. Modify the implementation of `slist_append` so that it performs the correct function (as given by the comment on the function prototype).

   The function can be recursive or iterative.

   This function is part of the internal implementation of the type and its API, so it is allowed to directly access members of the struct as necessary.

   Note: look carefully at the documentation. The function is supposed to join together two existing lists into one, re-using the nodes from those lists; it shouldn't allocate a new list.

4. Create a program called `e4_test_slist_append.cpp` (i.e. it should contain a `main`) which uses the public API of the list type to try to test the `slist_append` function. The program should return a success exit code if it detects no errors, and any error code if the `slist_append` function fails.

   Note: `assert` will automatically terminate the program with an error code if the assertion fails, so using `assert` to check for errors works well here.

---

5. Create a program `e5_product.cpp` which takes a sequence of integers as command-line parameters (via `argc` + `argv`), and prints their product. Empty sequences should be handled by using the multiplicative identity.

   For example, we could call the program as follows:

   ```
   $ ./e5_product 2 3
   6
   ```

   or

   ```
   $ ./e5_product 5 6 7 8
   1680
   ```

   Note: Originally the example erroneously mentioned sum.

---

6. The header file `e6_integrate.hpp` declares a function `f`, and then defines a function `integrate` which immediately aborts. Modify the definition

of `integrate` to implement numerical integration using the rectangle (or Riemann) method. The method used should be roughly:

$\int_a^b f(x)dx \approx \frac{1}{n}\sum_{i=0}^{n-1} f\left(a + \frac{i}{n}(b-a)\right)$

The assessment method is quite tolerant, and allows a number of possible interpretations including left, right, or mid-point rule, as long as the approximation gets better as n increases.

7. Create a source file `e7_integrate_f.cpp` which uses `e6_integrate.hpp` to numerically integrate the function:

   $f(x) = \sin(x) * \exp(-\cos(x^2))$

   The integration range $[a, b]$ and number of integration points n should be passed as parameters to the program, in the order a, b, n.

   For example, we might integrate over the range $[0, 1]$ using 100 rectangles:

   ```
   $ ./e7_integrate_f 0 1 100
   0.198758
   ```

   or over $[1, 2]$ using 1000 rectangles:

   ```
   $ ./e7_integrate_f 1 2 1000
   1.68745
   ```

   These numbers were calculated manually using Excel and so may not exactly match the output you get, but that doesn't necessarily mean your program is wrong.

   The assessment will be not be looking for a specific exact value, rather it will be looking for convergence as $n \to \infty$.

   Note: originally the file name was specified as `e7_integrate_f.hpp` rather than `e7_integrate_f.cpp`.

---

8. Create a file called `e8_file`
9. Create a directory called `e9_directory`
10. Create a program `e10_success.cpp` that always completes successfully.