

# Lab 8

## Writing functions

*20 March 2025*

In this lab, I'll again be walking you through process (similar to Lab 3) for program design, but this time focusing on individual functions rather than entire programs. Once you've established that you can do all the steps I'll let you take some shortcuts, but for now: Follow the process! (And remember to hand in frequently.)

Consider the problem of counting the number of abbreviations—i.e. “words” that end in a period<sup>1</sup>—in a given vector.

1. In a file named `morefunctions.h`, write a comment that summarises what the function will do (you can reuse what I wrote or paraphrase it to make it clearer);
2. then follow that comment with a function declaration appropriate to that description. Remember that function declarations are the header line of a function followed with a semicolon, and include the return type, the name of the function, and parameters (in that order).
3. In a file named `morefunctions.cpp`, include the corresponding `.h` and also create a *stub* for that function (same header as in the `.h` file, and return any valid literal, like 0 or -1 or 7).
4. At this point, you should be able to compile, but not yet run, the test cases by typing

```
compile -c morefunctions.cpp
```

If that does not succeed, edit the two files until the compiler is happy.

5. Write out at least two test cases *by hand* (in a notebook is fine);
6. then, in a file named `test_morefunctions.u`, enter the boilerplate for a `.u` file (you can refer to last week's file or see below in this handout), and add a test block corresponding to the test cases you wrote by hand.

---

<sup>1</sup>It's not your job here to decide what *really* is a word or if something *really* is an abbreviation. For purposes of this problem, it's an abbreviation if the string ends with a period, full stop.

7. At this point, you should be able to compile, but not yet run, the test cases by typing

```
compile -c test_morefunctions.u
```

If that does not succeed, edit the test case file (and/or the header file, but probably the test case file) until the compiler is happy.

8. *Now* you can fully compile and run the tests:

```
compile morefunctions.cpp test_morefunctions.u -o test_morefunctions
```

then

```
./test_morefunctions
```

The compiling should succeed but the tests should fail (because you haven't properly written the function yet!). (This is a good time to hand in if you haven't yet.)

9. Now would also be a good time to *add those two lines to your readme* so you don't misremember or mistype them later.
10. Write out a brief English or pseudocode description of how the function might accomplish its task, *by hand* (in a notebook is fine);
11. then, encode this algorithm into C++ in your function definition. At any reasonable time, try recompiling and running your tests: with the compiler and the automatic tester in place, it's very easy to do so and can sometimes give you valuable feedback.
12. Don't consider yourself done until you've tested!

For reference, boilerplate for .u files:

```
#include "name of corresponding .h file"
using namespace std;

test suite some_appropriate_identifier
{
    test another_identifier
    {
```

```
    check ( value-or-expression ) expect == value; // OR...
    check ( value-or-expression ) expect true;   // or false
}

//add additional test blocks here, as many as necessary/reasonable
}
```