ECS 122B: Programming Assignment #1

Instructor: Aaron Kaloti

Summer Session #2 2020

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.
- v.2: Stated explicitly that when you submit to Gradescope, `prog1.cpp` must not contain `main()`.
- v.3: Updated due date to reflect it was pushed back.

2 General Submission Details

Partnering on this assignment is prohibited. If you have not already, you should read the section on academic misconduct in the syllabus.

This assignment is due the night of Sunday, August 23. Gradescope will say 12:30 AM on Monday, August 24, due to the "grace period" (as described in the syllabus). *Rely on the grace period for extra time at your own risk.*

Some students tend to email me very close to the deadline. This is also a bad idea. There is no guarantee that I will check my email right before the deadline.

3 Grading Breakdown

Here is an approximate breakdown of the weight of this assignment.

- Manual review, including unit testing: 15%
- Correctness of part #1, as determined by autograder: 15%
- Correctness of part #2, as determined by autograder: 70%

4 Autograder Details

Once the autograder is released, I will update this document to include important details here. Note that you should be able to verify the correctness of your functions without depending on the autograder, especially since you must write unit tests (see below).

The autograder will use `g++` on a Linux command line to compile your code. (Refer to the example Linux command line interaction for part #2 to see the flags used.) On your end, you can likely get away with not using `g++` or a Linux command line. However, just make sure that whatever you use is pretty good about telling you compiler error messages *and* warnings. As you can see in the example Linux command line interaction for part #2, the `-Wall` and `-Werror` flags are used; expect the autograder to use these.

Below is a list of files that you will submit to Gradescope. Do not compress or archive the files.

- `prog1.cpp`
- `prog1.hpp`
- File(s) containing your unit tests (see below).

As explained below, you will not submit a makefile.

`prog1.cpp`, at least when you submit it to Gradescope, must not contain a `main()`. You should not have commented out code in your final submission.

4.1 Reminder about Gradescope Active Submission

Once the deadline occurs, whatever submission is your **active submission** will be the one that dictates your final score. By default, your active submission will be your latest submission. However, you can change which submission is your active submission; I will talk about how to do so during one of the lectures, after the autograder is up.

5 Tasks

On Canvas, you are provided an initial version of `prog1.hpp`. You must create `prog1.cpp`.

You should be using C++11. If you are compiling on a Linux command line with `g++`, then make sure to use the `-std=c++11` flag.

5.1 Makefile is *Not* Required

I hinted at this during one of the lectures. Although I would really like to, I will not require you to submit a makefile. I recommend that you make sure that you are confident about how to use a makefile, and if you have any questions about using makefiles and `make`, I am more than happy to answer them. However, given that many students may not even have access to the CSIF – the Linux environment that every student would have access to under *normal* circumstances and that is normally used as a reference environment – I think it would be too generally inconvenient to require everyone to create a makefile, particularly for Windows users who would have to do a bit of setup to get their machine to support Linux command line utilities.

5.2 Manual Review

The quality of your code will be evaluated by me or one of the TAs. You are expected to follow a reasonable and consistent style and not pick bad variable names. You are also expected to use helper functions and other means of properly keeping code relatively easy to read. You should make reasonable, but not excessive, use of comments. Do not write useless comments; I do not need a comment telling me what `i += 1` does. You should also obey good C++-specific practices such as properly using `const` and `static`. More specific tips are given in the directions for part #2.

5.2.1 Unit Testing

For both parts #1 and #2, you must write unit tests and submit them to Gradescope. The autograder will not check these tests; they will be checked during manual review. I put all my tests into a file called `test_prog1.cpp`, but you can put your tests into whatever file or file(s) that you wish; I would not recommend using more than 1-2 files, just to keep thing relatively easy on the manual reviewer. Needless to say, your tests should not be in `prog1.cpp`, the file that will be evaluated by the autograder.

I am only saying this because I am sure that some students will ask: I am not sure *how many* tests you will need to write in order to get full credit. The reason I am not sure is that the *quality* of and *variety* in the tests you write matters too. For example, writing 10 tests for `findMatches()` that each pass one string of around length 10 and a target string of length 1 to the function is not as good as, say, 5-7 tests that have a decent amount of variety (e.g. in the length of the target string).

You can use any C++ unit testing framework (e.g. Catch, Google Test, Boost Test) that you wish, so long as you do in fact use a unit testing framework. If you use `assert()`-based tests and do not use a unit testing framework, then do not expect to receive full credit on the manual review. You do not need to submit any unit testing framework's files (e.g. `catch.hpp`) because, again, we will evaluate your tests by hand.

5.3 Coding Problems

5.3.1 Part #1: Naive String Matching

The function `findMatches` is declared in `prog1.hpp`. For your convenience, the declaration is repeated below.

```
std::vector<unsigned> findMatches(const std::string& s,
                                  const std::string& target);
```

Implement this function in `prog1.cpp`. `findMatches()` should return a vector of integers containing all nonnegative indices (in ascending order) at which the second argument occurs in the first argument. Put informally, `findMatches()` determines all of the locations at which the second string appears in the first string.

Below are some examples:

- If the first string is `"abcdefghijk"` and the second string is `"f"`, then the returned vector should contain 5.
- If the first string is `"aabbcdeebbcbcdbcde"` and the second string is `"bcd"`, then the returned vector should contain 3, 11, and 14.

5.3.2 Part #2: Weighted Interval Scheduling

For this part, you will implement a solver for the weighted interval scheduling problem in `prog1.cpp`. This means implementing the function `solveWeightedIntervalScheduling` that is declared in `prog1.hpp`. Your solver will only find the weight/value of the optimal schedule; the solution will not be reconstructed.

Below is an example Linux command line session showing the expected behavior of `solveWeightedIntervalScheduling()`. As stated above, you do not need to create a makefile, and you do not need the "driver program" (`run_wis.cpp`) either[1]. The particular example used is based on the one shown on slides 53-59 of the dynamic programming slides.

```
$ cat run_wis.cpp
#include "prog1.hpp"

#include <iostream>

int main(int argc, char *argv[])
{
    int opt_weight = solveWeightedIntervalScheduling(argv[1]);
    std::cout << "Optimal weight: " << opt_weight << '\n';
}
$ cat input1.txt
10 14 1
1 5 2
9 13 2
3 11 7
6 8 4
2 7 4
$ make
g++ -Wall -Werror -std=c++11  -c prog1.cpp -o prog1.o
g++ -Wall -Werror -std=c++11  test_prog1.cpp prog1.o -o test_prog1
g++ -Wall -Werror -std=c++11  run_wis.cpp prog1.o -o run_wis
$ ./run_wis input1.txt
Optimal weight: 8
$
```

The input file will always have the same format and will always be properly formatted. Each line corresponds to an interval/request and always contains three integers: the start time of the interval, the finish time[2], and the weight/value. Do not assume the number of intervals will always be the same.

You must create and meaningfully use at least one `struct`/`class`.

[1]Some of you may copy the below driver program for your own use. That is fine, but do be careful about copying things from a PDF. Sometimes, certain characters, such as the single quotation marks around the newline character, can be distorted, leading to unexpected compiler messages.

[2]The finish time is included in the duration of the interval. Specifically, if I say that an interval starts at time 4 and ends at time 9, then that interval lasts for 6 units of time, not 5 units.

In your quest to write good quality code, you should make reasonable use of helper functions, and you should write tests for these functions. (If you don't do these things, then do not expect to get full points on the manual review.) Here are examples of tasks you could devote helper functions to (and perhaps write tests for):

- Computing the values of $p(j)$, as described in the lecture slides.
- Determining the weight/value of the optimal schedule, given the weights and values of $p(j)$.
- Loading the intervals' data from the given file.

Any helper function for which you write unit tests should be declared in `prog1.hpp` (or else you will not be able to compile your tests). This is why you must submit `prog1.hpp` to the autograder; everyone's will differ. Conversely, any helper function defined in `prog1.cpp` that need not be exposed (i.e. that need not be declared in `prog1.hpp`) should be marked with the `static` keyword, for good form. Your helper functions should use `const` when appropriate.

You may find that you have to use a sorting algorithm at some point. Do not reinvent the wheel; do not create your own implementation of quicksort or mergesort or the like. Use functions that are already defined in the `<algorithm>` header.

UCDAVIS
COMPUTER SCIENCE