

Project 2 - Search

CS 251, Fall 2024

Copyright Notice

© 2024 Ethan Ordentlich, University of Illinois Chicago

This assignment description is protected by [U.S. copyright law](#). Reproduction and distribution of this work, including posting or sharing through any medium, such as to websites like [chegg.com](#) is explicitly prohibited by law and also violates [UIC's Student Disciplinary Policy](#) (A2-c. Unauthorized Collaboration; and A2-e3. Participation in Academically Dishonest Activities: Material Distribution). Material posted on [any third party](#) sites in violation of this copyright and the website terms will be removed and your user information will be released.

Learning Objectives

In this project, we'll take a *client view* of C++ data structures. We'll become familiar with how to use two more of the fundamental data structures: sets and maps. By the end of this project, you will...

- Be able to assess your own code's correctness by writing a variety of test cases
- Have implemented a document search engine that supports complex queries
- Have experience working with built-in C++ data structures (`set` and `map`)

Remember, if you get stuck for more than 30 minutes on a bug, you should come to office hours. You should also come to office hours if you have questions about the guide or starter code, even if you haven't written any code yet.

Search Engines

For our search engine, each web page has a URL ("Uniform Resource Locator") that serves as its unique id and a string containing the body text of the page. We will preprocess the body text, and store it in an appropriate data structure for fast query and retrieval.

Some questions we'd like to be able to answer include:

- Which pages contain the word "pointer"?
- Which pages contain either "simple" or "cheap"?
- Which pages contain both "tasty" and "healthy"?
- Which pages contain "tasty", but do not contain "mushrooms"?
- Which pages contain "tasty", but not "mushrooms"; or contain "simple"; and do all of the before and also contain "cheap"?

Restrictions

Now that we're more comfortable with C++, we have a slightly different set of restrictions:

- **Feel free to include additional standard C++ libraries, but no external libraries.**
- Don't use structs, pointers, `new`, or `malloc`.
- Don't add global variables.
- Don't change the provided function signatures in `include/search.h` or `search.cpp`.
 - This will cause compilation errors in the autograder.

Logistics

- Due: 11:59 PM Tuesday, October 1
- Submit to Gradescope
 - `search.cpp`
 - `search_tests.cpp`
- Use grace tokens:
 - https://script.google.com/a/macros/uic.edu/s/AKfycbw_Pi4Pn-dxBnXAZrSPBY49m-w705uxyf0JIYGDXA0db52p86hby2Ki9vu4-b59vIQJcg/exec
 - This form requires your UIC Google Account. The permissions it requests are to know who you are, and for the script to edit our spreadsheet database in the back end.
 - The form will become active **once the project autograder closes**.
 - See [\[FA24\] CS 251 Course Info](#) for details.

FAQ

[\[FA24\] Search FAQ](#)

Testing

One of the learning outcomes of CS 251 is being able to evaluate the correctness of your own code, especially without an autograder. In real-world projects, you usually don't have an autograder available while you're writing code. As much as we practice reading and tracing C++, this isn't a great way to verify that our code is correct. As humans, we're slower than computers, and will make mistakes.

The most common way to evaluate some code's correctness is with **tests**, which run the code and check the results. Autograders are made up of many tests – but outside of your classes, you won't have an autograder. Instead, many programmers write their own tests, or an autograder to test their own code. To practice this, starting with this project, you will write tests for your own programs. This is likely the first time you are writing tests, so we'll give a lot of explicit guidance to start with!

We will evaluate your tests in 2 ways:

1. **Your tests will be graded.** When you submit, we will run your tests on a reference solution that tracks what sorts of “behaviors” your tests successfully test. You’ll see automated feedback on your tests, including whether your tests pass on a reference solution; and whether your tests cover the desired behaviors.
2. **The test suites that you will see are not comprehensive.** On Gradescope, we will have a small number of *hidden tests* for which you will not know your score until after the grace token due date. On this project, we will tell you what these hidden tests are checking for, so that you can practice writing your own test cases.

Essentially, we won’t directly give you feedback on your work. Instead, we’ll give you feedback on how well you check your own work.

We’re putting this in a red box because it’s so important.

There are hidden tests on Gradescope for which you will not see your score until after we publish grades. Instead, we will tell you what these hidden tests check for, and you will write your own tests. The autograder will give you feedback on how well your tests cover the behavior checked by the hidden tests.

If:

- Your code passes the non-hidden tests,
- Your tests use `EXPECT_THAT` or `ASSERT_THAT` correctly,
- Your tests pass on your code and the solution,
- And the autograder says that your tests check what we ask for,

Then you are *likely* to pass the hidden tests. Even if you write the minimum test cases we ask for, there are **many** possible programs that satisfy these cases and have incorrect behavior on others. We cannot tell you whether your program is sufficiently tested.

You might notice that each “testing” section comes before its “implementing” section. This is intentional! We’re practicing something called “**test-driven development**,” where we write the tests before the code. This helps solidify our understanding of what we are trying to accomplish.

Finally, any test suite might be imperfect. Just because you think you’ve finished a part doesn’t mean that it’s bug-free! We see programs all the time that have bugs caused by an insufficient test suite in an earlier part.

To learn more about GoogleTest, which is our test framework of choice, read the [\[FA24\] GoogleTest Crash Course](#) document.

Throughout this project, write all of your tests in `search_tests.cpp`.

Running Code

As in Project 1, we will use a Makefile. Here's the main targets:

- `make tests`
 - This default target builds the `search_tests` executable. We can run all tests in this executable with `./search_tests`, or we can use another target to run more-specific tests.
 - `make test_all`
 - `make test_clean_token`
 - `make test_gather_tokens`
 - `make test_build_index`
 - `make test_find_query_matches`
 - `make test_search_engine`
 - Build and run the tests for a specific function, as determined by GoogleTest's suites.
 - `make search_main`
 - Builds the `search_main` executable. Won't work until you're done or nearly done with the project, see [Task: searchEngine](#).
 - `make run_search`
 - Build and run the `search_main` executable.
-

Tasks


Task: `cleanToken`

This project's definition of "cleaning" is different from Project 1's definition of "cleaning". A "token" is one word. To clean a token:

- Remove any leading or trailing punctuation, as determined by [ispunct](#).
 - Punctuation in the middle of the token is not modified.

- Some Unicode¹ characters like curly quotes (“ instead of ") are not considered punctuation by `ispunct`. Don't worry about this, just follow what `ispunct` says.
- Return the empty string if the token does not contain at least one letter, as determined by `isalpha`.
- Convert the token to lowercase.

The value returned from `cleanToken` is the trimmed, lowercase version (or empty string if the token is to be discarded because it contains no letters).

 [FA24] Search FAQ

Subtask: Testing

Here are some examples of test cases that we've provided for you, which you can see in the local test suite (`tests/clean_tokens.cpp`), and which will run when you run the tests locally.

- Tokens without any punctuation or special characters
- Tokens with punctuation at the beginning
- Tokens with punctuation at the end
- Tokens without any letters
- Tokens with uppercase letters, and possibly punctuation

The following cases will be tested in the autograder, but you will **not** see your score until we publish grades. As mentioned earlier, you will write your own tests to verify that your code performs correctly in these situations. Your tests will be autograded, and you'll receive feedback on what behaviors they test, and what behaviors they miss.

- Tokens with punctuation at both the beginning and end
- Tokens with punctuation in the middle, as well as possibly at the ends

Here are some scenarios that we don't give feedback on whether you've tested or not, but *have caused bugs for students in previous semesters*. This list is not, and cannot, be comprehensive.

- Tokens with numbers at an end
- Tokens that are 1 letter long, like `x`
- Tokens that have a lot of punctuation, like `.....a` or `a.....` or `.....a.....`
- Tokens that start or end with something that is not a punctuation, letter, or digit (e.g. “)

¹ Unicode is fun! It covers things like glyphs in other languages and emoji. The problem is that a single glyph can be made up of *multiple chars*. So, the length of 🍌 is actually 4 `char` long, despite being 1 picture. We don't even have to go to emoji to get weird, since “ (the curly quote) is actually 3 `char` long. Trying to put human culture into computer boxes tends to have an absurd amount of edge cases.

```
string cleanToken(string s)
```

Read the provided tests for `cleanToken`. It's hard to fully specify program behavior in English. **The tests that we give you are part of the project description**, and we've deliberately written the tests for this project in a more approachable style for you to use as examples.

Write tests for the `cleanToken` function in `search_tests.cpp`. These tests must be in the `CleanToken` suite, i.e. `TEST(CleanToken, YourTestNameHere)`.

At this point, submit in Gradescope to see the autograder output.

You should see something like:

Passed Tests

Compilation (0/0)

Hidden Test Reminder (0/0)

Your test coverage: `cleanToken` (2/2)

followed by the failed `cleanToken` public tests, since you haven't implemented it yet. In the "Your test coverage: `cleanToken`" output, we run your tests on *our course staff solution*, and check the output. You'll see something at the bottom that reads:

Tested: punctuation at both ends

Tested: punctuation in middle, and not at either end

Tested: punctuation in middle, and at start (but not end)

Tested: punctuation in middle, and at end (but not start)

Lines starting with "Tested" are scenarios that your tests cover. On the other hand, lines starting with "Missing" are scenarios that your tests do not currently cover.

The coverage test will pass if you test all listed scenarios and if your tests pass on the course staff's solution. If your tests fail on the course staff solution, you're checking incorrect behavior!

Coverage tests are **only** on Gradescope because they need to run on our solution. The total score in Gradescope will be hidden, because there's hidden tests.

Subtask: Implementing

Now that you have a complete test suite (our tests, combined with yours) that you can run locally, you can check your code yourself.

Implement the `cleanToken` function according to its documentation.

Run the tests with `make test_clean_token`, and correct any bugs in your code or test suite before moving on. Many later functions depend on this one working correctly.

Make sure that each task is fully complete and correct before moving on. Every function you're implementing relies on earlier functions working correctly. If you wait until you think you're done to test for the first time, you will probably have an unpleasant surprise.

Additionally, **passing the tests does not mean your function is completely correct.** Each function in this project depends on all of the earlier functions working correctly. A failing test for a later function might actually be due to a hidden bug in an earlier function. You may (will) need to go back to fix code you've already passed tests for!

Task: `gatherTokens`

This function extracts unique tokens ("words"), separated by spaces, from a given piece of text, returning a `set<string>`.

Subtask: Testing

Testing `gatherTokens` is interesting, because it relies on `cleanToken` working correctly. We'll rely on the `cleanToken` test suite for testing how `gatherTokens` responds to `cleanToken` edge cases, and focus our `gatherTokens` tests on that function's behavior.

As before, we've provided a few tests for you to run locally, which can be found in `tests/gather_tokens.cpp`. The hidden test cases that you should test yourself are:

- Text with leading spaces
- Text with trailing spaces
- Text with multiple spaces between words

```
set<string> gatherTokens(string text);
```

Write tests for the `gatherTokens` function in `search_tests.cpp`.

These tests must be in the `GatherTokens` suite, i.e. `TEST(GatherTokens, YourTestNameHere)`.

[FA24] Search FAQ

Subtask: Implementing

This function begins by *tokenizing* the text: dividing it into individual tokens (“words”), which are strings separated by whitespace. Tokens should be cleaned before being stored.

Implement the `gatherTokens` function according to its documentation.

Use the tests we’ve provided and the tests you’ve written (`make test_gather_tokens`) to convince yourself that you’ve implemented it correctly before moving on.

Task: `buildIndex`

At this point, we’re going to start working with data.

Data Format

The data files come in pairs of lines:

- The first line of a pair is a page URL.
- The second line of a pair is the body text of that page, with all newlines removed (basically the text of the page in a single string).

The first two lines in a file form the first URL-content pair. The third and fourth lines form another pair, the fifth and sixth another, and so on. To view an example data file, open any of the `.txt` files, such as `data/tiny.txt` or `data/cplusplus.txt` in the starter code.

Inverted Index

We’re searching a *lot* of data – to do this efficiently, we must choose the right data structure to store our data.

Consider the index in the back of a book, such as a textbook. A book's index lists words and the pages on which each word appears. For example, we might look for the word "Internet", and find that it lists pages 18 and 821. This means that the word "internet" occurs on page 18, and again on page 821. A book's index is an example of an [inverted index](#), which we can use to go from a *word* to the *locations* the word appears.

Our search engine will use an inverted index to allow for fast queries of web pages, given search terms. Therefore, we need to build one!

Subtask: Conceptual Understanding

Before writing code for this section, write or draw the inverted index for `data/tiny.txt`. This will not be graded, but you'll use this diagram in the next section when you write the test.

There's an example inverted index for `data/tinier.txt` in the `tests/build_index.cpp` file.

Subtask: Testing

Testing this function is difficult, because it reads from a file, and because it depends on `cleanToken` and `gatherTokens`. Additionally, the expected inverted index may be large and difficult to type manually. We'll take advantage of the fact that many of the behaviors that we want to test are covered by the tests we wrote earlier for `cleanToken` and `gatherTokens`, and not write as many tests for `buildIndex`.

The index is returned by reference, and is a map from tokens (keys) to a set of URLs (values) where that key can be found. The function returns an `int` that is the number of webpages that were processed and added to the index. (If the file is not found, return 0.)

We've provided a single example test that uses the file `data/tinier.txt` in `tests/build_index.h`. The hidden tests will check:

- That you run against `data/tiny.txt`
- When the file is not found.


Make sure that you check both return values here: the `int` return, and the index return-by-reference. The other files are too large to test without writing significantly more code.

```
int buildIndex(string filename, map<string, set<string>>& index)
```

Use the inverted index you made for `data/tiny.txt` to write an additional test for this function based on the given example test. Also, test when the file is not found.

This test must be in the `BuildIndex` suite, i.e. `TEST(BuildIndex, YourTestNameHere)`.

Run the tests with `make test_build_index`.

 [FA24] Search FAQ

Subtask: Implementing

This function takes in a filename as an argument, reads the content, and populates the provided map with an inverted index. Note that this function returns an `int`; the inverted index is returned by reference. Don't clear the existing data in the index.

The data format is given above, and you can assume that files follow this format. Use `gatherTokens` to get the set of unique tokens from the text on each page. Then, for each token in the set, update the inverted index to indicate that this token is present on the page's URL.

```
int buildIndex(string filename, map<string, set<string>>& index)
```

Implement the `buildIndex` function according to the above and its documentation. Ensure that our tests and your tests pass before moving on.

Task: `findQueryMatches`

This function uses the inverted index to find the URLs of pages that match a search query.

The sentence string argument can either be a single search term or a compound sequence of multiple terms. A search term is a single word, and a sequence of search terms is multiple consecutive words, each of which (besides the first one) may or may not be prefixed with a modifier like `+` or `-`.

To find the matches for a query, we process the search terms from left to right:

- For a single search term, matches are the pages that contain the specified term.

- A sequence of terms is a compound query, as follows:
 - A term with no modifier has its matches unioned with the results so far.
 - A term with the **+** modifier has its matches intersected with the results so far.
 - A term with the **-** modifier uses set difference. That is, matches for this term are removed from the results so far.
- Each term should be cleaned before searching.

If you would like to review set operations from CS 151, [this resource](#) might be helpful.

You may assume that the query sentence is well-formed, which means:

- The query sentence contains at least one search term.
- The first search term in the query sentence will never have a modifier.
- Every search term has at least one letter.
- There are no leading or trailing spaces.

Examples

Here are some example queries and how they are interpreted:

- **"pointer"**
 - matches all pages containing the term **"pointer"**
- **"simple cheap"**
 - means **simple OR cheap**
 - matches pages that contain either **"simple"** or **"cheap"** or both
- **"tasty +healthy"**
 - means **tasty AND healthy**
 - matches pages that contain both **"tasty"** and **"healthy"**
- **"tasty -mushrooms"**
 - means **tasty WITHOUT mushrooms**
 - matches pages that contain **"tasty"** but do not contain **"mushrooms"**
- **"tasty -mushrooms simple +cheap"**
 - means **tasty WITHOUT mushrooms OR simple AND cheap**
 - matches pages that match
 $((("tasty" \text{ without } "mushrooms") \text{ or } "simple") \text{ and } "cheap")$
 - In order...
 - Start with all the pages that contain **"tasty"**
 - Take out all pages (set difference) that contain **"mushrooms"** (-)
 - Add all pages (set union) that contain **"simple"**
 - Keep only pages (set intersection) that contain **"cheap"** (+)

Subtask: Testing

This function is, surprisingly, much more straightforward to test, because we don't have to call `buildIndex` to create an index. Instead, we can create an index ourselves! See `tests/find_query_matches.h` for our tests, which you can use as examples.

The hidden test cases that you should write tests for are:

- The first query term does not appear in the index
- A later query term, possibly with a modifier, does not appear in the index

These are not special cases. You should handle them as above, where the pages that contain the query term are the empty set.

```
set<string> findQueryMatches(const map<string, set<string>>& index,
const string& sentence)
```

Write tests for the `findQueryMatches` function in `search_tests.cpp`.

These tests must be in the `FindQueryMatches` suite, i.e. `TEST(FindQueryMatches, YourTestNameHere)`.

[FA24] Search FAQ

Subtask: Implementing

The C++ `<algorithm>` library provides [set union](#), [set intersection](#), and [set difference](#) functions that can be applied to sorted containers. These functions can be used for many more things than `sets`, so they're tricky to use correctly. Here's an example of how to use these functions.

We use `set_union` as an example. The input sets are A and B, and we need to have an existing result set `result`. We call the function as follows:

```
set_union(A.begin(), A.end(),
          B.begin(), B.end(),
          inserter(result, result.begin()));
```

If $A = \{1, 2, 3\}$ and $B = \{2, 4\}$, and `result = {6}`, then after the function, `A` and `B` will be the same; and `result = {1, 2, 3, 4, 6}`.

Note that the end `result` contains 6, which is in neither `A` nor `B`. If we wanted `result` to only contain the union, we would need to start with `result` being the empty set.

Warning: `result` cannot be one of the inputs. This has *undefined behavior*.²

```
set_union(result.begin(), result.end(),
          B.begin(), B.end(),
          inserter(result, result.begin()));
```

It probably won't crash, but it will cause *very* confusing bugs. Don't do it. Please.

You're also free to write your own functions to perform set operations.

```
set<string> findQueryMatches(const map<string, set<string>>& index,
                             const string& sentence)
```

Implement the `findQueryMatches` function according to its documentation.

Since `index` is `const`, you can't use `[]` – see the [\[FA24\] Search FAQ](#) for advice.

Run the tests with `make test_find_query_matches`.

Task: `searchEngine`

Finally, put the pieces together! In `search_main.cpp`, we take a filename from input, and pass it to `searchEngine`. From here, `searchEngine` should:

- Print how many web pages were processed to build the index and how many distinct words were found across all pages
- Enter the command loop:
 - Prompt the user to enter a query

² We talk a little bit about “undefined behavior” (UB) in lecture, but only as a No Good Very Bad thing to avoid. But what does it mean? Specifically, undefined behavior means that the program can do literally anything, including deleting your hard drive (in practice, segfaults are much more likely). This isn't totally bad – compilers can do some fancy optimizations by assuming that UB doesn't happen... but then weird things happen when it does. Like a variable being [both true and false](#), or [time travel](#). Programming is *weird*.

- Find the matching pages, and print the URLs (one on each line)
- If the query is the empty string, exit the command loop
- Print a closing message

The exact prompts and messages are in the sample execution below.

Since this function reads from and writes to the console, it's a lot trickier to test. It's not impossible – we've already seen tests that did similar things in Project 1. However, it's pretty tricky, so **you do not need to write your own tests for this function.**


For this function, we strongly recommend using `getline` instead of `>>` to read from `cin`.

```
void searchEngine(string filename)
```

Implement the `searchEngine` function according to the above and its documentation.

Test this function with `make test_search_engine`. As with the other public tests, these tests are available locally. This function does not have hidden tests.

At this point, you can now `make run_search` and run the entire application!

 [FA24] Search FAQ

Sample Execution

User inputs are highlighted in **red**.

```
$ make run_search
```

```
Enter a filename: data/cplusplus.txt
```

```
Stand by while building index...
```

```
Indexed 86 pages containing 1498 unique terms
```

```
Enter query sentence (press enter to quit): vector
```

```
Found 8 matching pages
```

```
https://www.cplusplus.com/reference/array/array/
```

```
https://www.cplusplus.com/reference/bitset/bitset/
```

```
https://www.cplusplus.com/reference/forward_list/forward_list/
```

```
https://www.cplusplus.com/reference/list/list/
```

```
https://www.cplusplus.com/reference/queue/priority_queue/
```

```
https://www.cplusplus.com/reference/stack/stack/
```

<https://www.cplusplus.com/reference/vector/vector-bool/>
<https://www.cplusplus.com/reference/vector/vector/>

Enter query sentence (press enter to quit): **vector +container**

Found 7 matching pages

<https://www.cplusplus.com/reference/array/array/>
https://www.cplusplus.com/reference/forward_list/forward_list/
<https://www.cplusplus.com/reference/list/list/>
https://www.cplusplus.com/reference/queue/priority_queue/
<https://www.cplusplus.com/reference/stack/stack/>
<https://www.cplusplus.com/reference/vector/vector-bool/>
<https://www.cplusplus.com/reference/vector/vector/>

Enter query sentence (press enter to quit): **vector +container -pointer**

Found 6 matching pages

<https://www.cplusplus.com/reference/array/array/>
https://www.cplusplus.com/reference/forward_list/forward_list/
<https://www.cplusplus.com/reference/list/list/>
https://www.cplusplus.com/reference/queue/priority_queue/
<https://www.cplusplus.com/reference/stack/stack/>
<https://www.cplusplus.com/reference/vector/vector/>

Enter query sentence (press enter to quit):

Thank you for searching!

(Not an input) In the last menu, I just hit enter without typing anything.

Grading Breakdown

Project grading is again based on **milestones**. As in Project 1, to complete a milestone, you'll need to both:

- Complete **every** milestone before it, and
- Pass **all** of its tests.
 - For test-writing milestones, this means checking all the scenarios we've indicated.

Milestone	Total Points
Writing tests: <code>cleanToken</code>	2

Implement: <code>cleanToken</code>	9
Writing tests: <code>gatherTokens</code>	13
Implement: <code>gatherTokens</code>	27
Writing tests: <code>buildIndex</code>	33
Implement: <code>buildIndex</code>	54
Writing tests: <code>findQueryMatches</code>	60
Implement: <code>findQueryMatches</code>	81
Implement: <code>searchEngine</code>	90

We also have hidden tests worth a total of **10 points**. You can earn credit for hidden tests after meeting the corresponding milestone above, but you won't know whether you've passed these tests until after we release grades after the grace token period. You'll need a completely correct implementation to receive credit for each hidden test. The tests that *you* write can help make it more likely that your code is completely correct.

Function	Hidden Test Points
<code>cleanTokens</code>	2
<code>gatherTokens</code>	2
<code>buildIndex</code>	3
<code>findQueryMatches</code>	3

Acknowledgements

Julie Zelenski - Stanford University; Joe Hummel, PhD - Northwestern University; Shannon Reckinger, PhD - University of Illinois Chicago; Adam Koehler, PhD - University of Illinois Chicago. Modifications by Ethan Ordentlich.