

# Lab 6 - Compiling, compilation, compiler

Due on 3/4/2024 at 11:59PM

## Introduction:

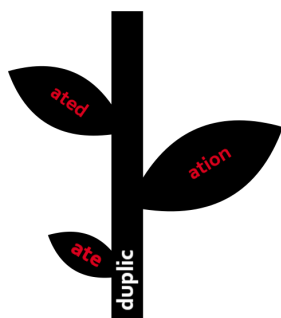
What's the red fruit that goes into ketchup? **You say "tomayto"** ➞


[https://en.wikipedia.org/wiki/Let%27s\\_Call\\_the\\_Whole\\_Thing\\_Off](https://en.wikipedia.org/wiki/Let%27s_Call_the_Whole_Thing_Off)\_, **I say "tomahtoe"** ➞

<https://bit.ly/3npJ3jS>\_. A professor sets their answer key to expect *execution* and you type *execute* -- you deserve the points, right? You google for *Practicing how to dribble* -- shouldn't a document titled *How to practice dribbling* appear in the search results?

**Information Retrieval** ➞ [https://en.wikipedia.org/wiki/Information\\_retrieval](https://en.wikipedia.org/wiki/Information_retrieval)\_(IR) is the proper name of the academic field of computer science devoted to the study of how to develop search engines. An IR system is basically a set of algorithms that turn a user's search query into a list of related materials. Although the researchers in the field of IR build systems to search many different types of material (documents, videos, songs, etc), their original goal was to develop ways to allow users to search for the few relevant documents in a large collection. They hypothesized a system that would simply list the documents in the collection in decreasing order of relevance to the user's query. Relevancy is a subjective, qualitative metric -- they needed some way to quantify it. The researchers started out using a simple method to calculate relevancy: count the number of times that each word in a user's query appears in each of the candidate documents and assign documents with higher counts greater relevancy.

This brilliantly simple technique worked great, until it didn't. Recalling the example above, shouldn't the system count *practice*, if found in some candidate document, as a match for *practicing* from the query? Of course it should. And the same is true for *dribble* and *dribbling*. There are many, many examples of cases like this in the English language where words with similar meaning have different *suffixes* (any letters "added at the end of a word to form a derivative") and the same *stem* ("the root or main part of a noun, adjective, or other word, to which inflections or formative elements are added."). This realization caused IR system designers to acknowledge that they could not do a simple, literal scan for query terms in a candidate document, count up the matches, and use that number to determine relevancy.



What did they do instead? Simple: first, they said to strip off the pesky suffixes of every single word in the query *and* the candidate document. After that? *Then* do the simple, literal search. But what's the best way to remove the suffix from a word and pare it to its stem? In **1979/1980**, 

(<https://tartarus.org/martin/PorterStemmer/>)\_scientist **Martin Porter**

 ([https://en.wikipedia.org/wiki/Martin\\_Porter](https://en.wikipedia.org/wiki/Martin_Porter))\_designed

a *stemming* algorithm that he thought was the ideal way to lop off extraneous suffixes and reduce words to their core. To understand better the meaning of stem and suffix, here are a few examples of words before and after they are run through the *Porter Stemming Algorithm*:

#### Before Porter Stemming:

Execution  
Translation  
Grandiose  
Ambiguous  
Ambiguity  
Speculation

#### After Porter Stemming:

Execut  
Translat  
Grandios  
Ambigu  
Ambigu  
Specul

Look closely at the stems of *ambiguous* and *ambiguity* -- they are the same! That's exactly what IR system designers hoped their algorithm would yield. Determining that *ambiguous* and *ambiguity* have the same meaning is now the product of a simple letter-by-letter comparison because the stems are the same.

A successful implementation of the Porter Stemming Algorithm requires the implementation of a significant number of helper functions. *Helper functions* are functions that perform operations in service of a larger, more complex function. Computer scientists "discover" helper functions when they are writing/implementing that overarching function. The discovery happens when they recognize that they are writing certain blocks of code over and over. The developer 1) names the operation that these blocks perform and 2) abstracts the code in those blocks into separate functions. Like any other function, a helper function improves the readability of code and reduces the likelihood of errors introduced by mistaken copy/paste.

There are 9 helper functions needed for a clean, readable implementation of the Porter Stemming Algorithm:

1. `is_vowel`
2. `is_consonant`
3. `ends_with_double_consonant`
4. `ends_with_cvc`
5. `contains_vowel`
6. `count_consonants_at_front`
7. `count_vowels_at_back`

8. `ends_with`
9. `new_ending`



Your task in this lab is to implement the specified helper functions according to the specifications given below. In doing so, you will contribute to a robust implementation of the Porter Stemming Algorithm that will serve as the core of the *BearING* search engine.

Good luck!

## Program Design Task:

"I don't need pseudocode because my code pseudo works." If you want your code to *really* work, then I recommend writing the pseudocode or drawing a flow chart for your implementation of the 9 helper functions required to complete the implementation of the *BearING* search engine *before* you get started coding.

## Program Design Requirements:

Your pseudocode or flow chart must describe the entirety of the process you plan to use to meet the requirements for each helper function. You may choose to write the flow chart or the pseudocode at any level of detail but remember that this is a tool for you! Your pseudocode or flow chart must include a separate, labeled description for each of the 9 helper functions given above. For places where you must handle corner cases, you must describe in your pseudocode how you will handle those.



## Programming Task:

Your programming task is to implement *and document* the 9 helper functions required to complete the implementation of the Porter Stemming Algorithm and the *BearING* application. In this lab you will not prompt the user for input. Your implementation will be validated by a series of unit tests.

Unit tests are *not* like academic tests and are used commonly in professional software development to check whether a unit of code performs correctly. The unit of code usually exercised by unit tests is the function. Unit tests for a function will call that function with a variety of arguments in order to check whether it operates correctly under all possible conditions. Knowing how/when to use Unit Tests is something that you will want to learn as a professional software engineer.

Implement the 9 helper functions according to the following specifications:

Name	Parameters	Return Type	Semantics
<code>is_vowel</code>	a <code>char</code>	<code>bool</code>	<code>is_vowel</code> returns <code>true</code> when its argument vowel and <code>false</code> otherwise.

<code>is_consonant</code>	a <code>char</code>	<code>bool</code>	<code>is_consonant</code> returns <code>true</code> when its argument is a consonant and <code>false</code> otherwise.
<code>ends_with_double_consonant</code>	a <code>std::string</code>	<code>bool</code>	<code>ends_with_double_consonant</code> returns <code>true</code> if the last two characters of the argument are A. both consonants, and B. equal to one another. It returns <code>false</code> otherwise.
<code>ends_with_cvc</code>	a <code>std::string</code>	<code>bool</code>	<code>ends_with_cvc</code> returns <code>true</code> if the last three characters of the argument are a consonant, a vowel and then a consonant (hence the name cvc). It returns <code>false</code> otherwise.
<code>contains_vowel</code>	a <code>std::string</code>	<code>bool</code>	<code>contains_vowel</code> returns <code>true</code> if there is a vowel anywhere in the argument. It returns <code>false</code> otherwise. For instance, <code>contains_vowel(std::string{"free"})</code> is <code>true</code> and <code>contains_vowel(std::string{"brrr"})</code> is <code>false</code> .
<code>count_consonants_at_front</code>	a <code>std::string</code>	<code>int</code>	<code>count_consonants_at_front</code> returns the number of consecutive consonants at the beginning of the argument. For example, <code>count_consonants_at_front(std::string{"the"})</code> is 3.
<code>count_vowels_at_back</code>	a <code>std::string</code>	<code>int</code>	<code>count_vowels_at_back</code> returns the number of consecutive vowels at the end of the string passed as an argument. For example, <code>count_vowels_at_back(std::string{"free"})</code> is 2.
<code>ends_with</code>	two <code>std::string</code> s, <i>candidate</i> and <i>suffix</i> (in that order).	<code>bool</code>	If <i>candidate</i> is an empty string and <i>suffix</i> is an empty string, <code>ends_with</code> returns <code>true</code> . If <i>candidate</i> is an empty string and <i>suffix</i> is a non-empty string, <code>ends_with</code> returns <code>false</code> . If <i>candidate</i> ends with <i>suffix</i> , <code>ends_with</code> returns <code>true</code> . Otherwise, it returns <code>false</code> . For instance, <code>ends_with(std::string{"testing"}, std::string{"ing"})</code> is <code>true</code> . <code>ends_with(std::string{"tester"}, std::string{"art"})</code> is <code>false</code> .
<code>new_ending</code>	a <code>std::string</code> , a number and a <code>std::string</code>	<code>std::string</code>	<code>new_ending</code> creates a new string from <i>candidate</i> by removing its last <i>suffix length</i> characters and replacing them with <i>replacement</i> . <code>new_ending</code>

	a <code>std::string</code> named <i>candidate</i> , <i>suffix length</i> and <i>replacement</i> , respectively (and in that order)	returns that new string. For example, <code>new_ending(std::string{"testing"}, 3,  std::string{"ed"})</code> is "tested". You may as that <i>suffix length</i> is always less than or eq the length of <i>candidate</i> .
--	---	--

You will know whether your helper functions are written correctly when all the unit tests pass. If all the unit tests pass, you will receive full points for the programming part of this lab!

Documenting functions is a very important part of a professional programmer's job. Every programming project has their own preferred format for writing comments that describe functions and their inputs/outputs. For the *BearING* project, we ask that your comments conform with the following format:

```

/*
 * <function name>
 *
 * <short description of what the function does>
 *
 * input: <short description of all input parameters>
 * output: <short description of all output parameters
 *         and the return value>
 */

```

Comment your code immediately before the function's definition or declaration/prototype (if you wrote one), whichever comes first. Conforming to the comment standard is as easy as replacing all the text between (***and including!***) the `<` and `>`s with a response to the prompt.

## Programming Requirements:

If you are a Windows-based developer, start with this **skeleton**

(<https://uc.instructure.com/courses/1657742/files/177683139?wrap=1>) 

([https://uc.instructure.com/courses/1657742/files/177683139/download?download\\_frd=1](https://uc.instructure.com/courses/1657742/files/177683139/download?download_frd=1)) . If you are a macOS-based developer, start with this **skeleton**

(<https://uc.instructure.com/courses/1657742/files/177683655?wrap=1>) 

([https://uc.instructure.com/courses/1657742/files/177683655/download?download\\_frd=1](https://uc.instructure.com/courses/1657742/files/177683655/download?download_frd=1)) . If you are a Linux-based C++ developer, start with this **skeleton**

(<https://uc.instructure.com/courses/1657742/files/177687505?wrap=1>) 

([https://uc.instructure.com/courses/1657742/files/177687505/download?download\\_frd=1](https://uc.instructure.com/courses/1657742/files/177687505/download?download_frd=1)) . These skeletons provide the starting point for a successful implementation of the *BearING* search engine. If you do not use this skeleton code you will not be able to complete this lab. The first time that you run the skeleton code you will receive compiler errors (they are *actually* linker errors).

**This is expected.** The code will not compile until you provide at least a basic definition of the 9 helper functions specified above.

All of your work will be done in the `helpers.cpp` file. You should not edit code in any of the other files.

As you work on the *BearING* search engine, you may make the following assumptions:

1. The vowels are a, e, i, o, and u.
2. The consonants are the letters that are not vowels.
3. The *suffix length* parameter of the `new_ending` helper function will never be longer than the length of *candidate*.
4. All input will be lower case -- no upper-case letters will be given.

### Tips/Tricks

I recommend you start your work by defining each of the 9 helper functions using *stubs*. The stub for a function has all the same parameters and return types as the actual function implementation would but has a body that performs no meaningful action. A stub implementation for a function named `func` that returns a `bool` and takes a single parameter -- a `std::string` -- might look like this:

```
bool func(std::string p) {  
    return true;  
}
```

A stub implementation for a function named `func` that returns a `std::string` and takes a single parameter -- a `std::string` -- might look like this:

```
std::string func(std::string p) {  
    return std::string{""};  
}
```

Let's try that translation on one of the functions that you are going to write for this lab. We know that `is_vowel` takes a single parameter whose type is a `char` and that it returns a `bool`-typed result. Okay, so, let's take a breath and work through that sentence slowly. First, the name of the function is `is_vowel`:

```
is_vowel
```

Great. Now, what type of value does the function return? A `bool`:

```
bool is_vowel
```

We're rolling. Does the function take any parameters? Yes, one, a `char`-typed variable:

```
bool is_vowel(char p)
```

To finish the stub, we just need to put it a simple-minded body that returns a value of the proper type. Because `is_vowel` returns a `bool`, let's just have our stub return `false`:

```
bool is_vowel(char p) {
    return false;
}
```

You did it! You wrote the stub for `is_vowel`!

Beginning this lab by implementing stubs for each of the 9 helper functions will give you the confidence that your code compiles and that your development environment is configured correctly. If it were me, I would not begin doing an actual implementation of any of the 9 required functions until my code compiled completely without warnings or errors.

The unit tests associated with your code will test each of the functions in the order that they are listed above. In other words, I would transform the stub implementations to actual implementations in the order that the helper functions are described above. The code that executes the unit tests are designed to direct your eyes to the failing test cases. As you pass the tests cases for each of the helper functions that you implement, you can move on to implementing the next one. If you patiently repeat this process, you will be done with this lab before you can say "complet" (get it? That's a stem!)

To implement several of the helper functions, you will find it helpful to *iterate* -- go element by element -- over the individual characters in a string. Think of a `std::string` as a list of `char` actors (which is, in fact, how a `std::string` is usually represented in memory). Then you can treat each of the `char` actors in a `std::string` (accessible by their index!) as if they were an individual variable whose type is `char`. The first `char` actor of a `std::string` is "at" index `0`. The second `char` actor of a `std::string` is at index `1`. The final `char` actor of the `std::string` is at the index `n`, where `n` is one less than the length of the string. (Remember that this is C++ and that all counting/indexing begins at `0`).

You can use the `at` method (methods are something that we will discuss in more depth later in this course; you can think of them as a way to ask a particular "object" to either 1) do an action that changes itself or 2) report information about itself) on a `std::string` object (there's that word again!) to access the `char` actor at a particular index. For instance,

```
#include <iostream>

int main() {
    std::string str{"teasing"};
    std::cout << "str.at(0): " << str.at(0) << "\n";
    std::cout << "str.at(str.length() - 1): " << str.at(str.length() - 1) << "\n";
    return 0;
}
```

will print

```
str.at(0): t
str.at(str.length() - 1): g
```

([Godbolt link](https://godbolt.org/z/ocYe36cbh) → <https://godbolt.org/z/ocYe36cbh>)

But the `at` method is not the only way to access/update individual elements in a `std::string` object. You can use the `[ ]` operators, too!



For instance,

```
#include <iostream>
#include <string>

int main() {
    std::string name{"Alice"};
    std::cout << "name[0]: " << name[0] << "\n";
    std::cout << "last character of name: " << name[4] << "\n";
    name[0] = 'E';
    name[3] = 's';
    std::cout << "name: " << name << "\n";
    return 0;
}
```

will print

```
name[0]: A
last character of name: e
name: Elise
```

([Godbolt link](https://godbolt.org/z/q776ojE1h) → <https://godbolt.org/z/q776ojE1h>.)

How painful is it to see that `4` used as a literal? It's just asking to turn in to a bug sooner or later (for instance if (when!) we update the initial value of `name` and its length changes)! What we really want is to know the length of the `std::string` `name` and use *that* to determine how to access the final `char`acter of the value in the variable named `name`. Good thing that there is a (member) function that will give us just that information.

```
#include <iostream>
#include <string>

int main() {
    std::string name{"Alice"};
    int name_length = name.length();
    std::cout << "name[0]: " << name[0] << "\n";
    std::cout << "last character of name: " << name[name_length - 1] << "\n";
    name[0] = 'E';
    name[3] = 's';
    std::cout << "name: " << name << "\n";
    return 0;
}
```

([Godbolt link](https://godbolt.org/z/h5KzdW4sd) → <https://godbolt.org/z/h5KzdW4sd>.)

will print

```
name[0]: A
last character of name: e
name: Elise
```

and the code is so much more robust to changes in the contents of `name`!

What if we want to get chunks out of a `std::string`? Accessing one `char`acter at a time from a `std::string` would be a really tedious way to get a *substring*. The C++ designers agreed and helpfully gave us a neat function to create a substring from another `std::string`. The function is, amazingly, called `substr`. The first parameter to `substr` is the place (don't forget that we count



from 0, of course) in the larger `std::string` to start the substring. The second parameter is the length of the substring to extract. For instance,

```
#include <iostream>
#include <string>

int main() {
    std::string outer{"trample"};
    std::string tram{outer.substr(0, 4)};
    std::string amp{outer.substr(2, 3)};
    std::cout << "outer: " << outer << "\n";
    std::cout << "tram: " << tram << "\n";
    std::cout << "amp: " << amp << "\n";
    return 0;
}
```

([Godbolt link](https://godbolt.org/z/3nf51YY5G)  <https://godbolt.org/z/3nf51YY5G>.)


will print

```
outer: trample
tram: tram
amp: amp
```

## Success

A successful submission for this lab will include source code for 9 helper functions in `helpers.cpp` that a) pass all the unit tests included in the skeleton code *and* b) are properly documented (using comment blocks formatted according to the specifications set out above).

## Critical Thinking Task:

**Time is money**  [https://en.wikipedia.org/wiki/Time\\_is\\_money\\_\(aphorism\)](https://en.wikipedia.org/wiki/Time_is_money_(aphorism)), as they say. But, perhaps it is really better to think about money and time as two sides of the same coin (sorry for the pun!). Think of a scenario where you are out with your friends on a weeknight having dinner. You have a test the next day so you are in some hurry to get home to study after you finish eating. As you debate whether to walk home, take transit or call a Lyft, you consider the following:

1. Walking home will take the longest, but will cost the least.
2. Even though you may have to wait for the next bus/shuttle, using transit will be faster than walking but will be (relatively speaking) more expensive than walking.
3. Using Lyft will get you home fast but will cost the most.

In whatever decision you make there will be a tradeoff between time and money.

Tradeoffs occur all the time in computer science and software development. One of the most common tradeoffs faced by software developers is time against space -- a software developer is faced with the choice of whether to use an algorithm that runs fast (time) but uses lots of memory (space) or one that runs slowly but uses very little memory. In making a decision about which version of the algorithm to employ, the developer must consider their available resources (Is their software running on a giant server or the chip on a watch?), the impact on the user (Will their software fail to do the job correctly if it is too slow? For example, what would happen if a lane-

departure system didn't activate corrective steering action immediately after detecting the car drifting into the other lane?), the cost of implementation (How hard it is to understand the algorithm, debug the code in the future, make improvements in the next version?), etc.

Your job in this critical thinking task is to describe another fundamental tradeoff that you think is common in computer science and provide three (3) metrics that a software developer can use to make a decision when faced with such a dilemma.

## Critical Thinking Requirement:

In no more than 500 words, describe another fundamental tradeoff that you think is common in computer science and describe three (3) metrics that a software developer can use to make a decision when faced with such a dilemma. You may, but are not required to, use external references to help you build your response. If you decide to use material from outside the class, it is your responsibility to properly document your references. Failure to do so is a violation of this course's policy on academic honesty and will result in severe consequences. If you need help generating the proper documentation in such cases, please contact us! You can also refer to UC's incredible [Academic Writing Center](https://www.uc.edu/campus-life/learning-commons/programs/writing-center.html) or the [OWL](https://owl.purdue.edu/) at Purdue.

Here is a model answer to help you formulate your submission:

Software developers often face the fundamental tradeoff between time and space -- it is usually the case that the faster an algorithm runs, the more memory it will require (and vice versa). Deciding between two algorithms that accomplish the same task -- one that is fast but uses lots of memory and one that is slow but conserves space -- is not easy. A developer can consider these factors when faced with this dilemma:

Use case: The developer can consider whether it is important to the end user for the algorithm to compute its results quickly. For instance, when the software will be controlling systems that interact with the physical world (autopilot, self-driving car, robot, etc), timely reaction to physical stimuli is more important than the amount of space required to generate that reaction.

Market: The developer can consider where this application will be used. If the software is for deployment on a cell phone, favoring the algorithm that uses less memory is the right choice. But, if the software is for deployment on a supercomputer, time is of the essence.

Organizational cost: If the algorithm that is faster but sips memory is more complicated than the algorithm that is slower but guzzles RAM, debugging and maintaining the former will be more expensive and time consuming than the latter.

## Question Task:

For this lab you exercised your skills writing functions. Maybe this was the first time that you wrote a function that you could reuse yourself. Wasn't it awesome to realize that you didn't have to write the same thing twice? How cool is it that we can write unit tests that will help us debug our code?

Do you think that you will use that pattern in the future? And, it's pretty awesome that we can use strings in a way that mirrors the way that we use vectors -- did that blow your mind?

## Question Requirement:

Submit a file named `question.pdf` which contains your question. Take particular notice of the fact that the question does not have to be subtle, or deep or philosophical for you to receive full points. The goal of the question is for me to be able to get a sense whether there are common questions about the material which would indicate where I failed to teach the material appropriately.

## Deliverables:

1. The pseudocode describing the algorithms you used to implement the 9 helper functions required to complete the *Bear/ING* search engine in PDF format (named `design.pdf`).
2. The C++ source code for the helper functions required for successful operation of the *Bear/ING* application (named `helpers.cpp`).
3. The response to the Critical Thinking prompts in PDF format (named `tradeoff.pdf`).
4. A written question about the material covered so far in this class in PDF format (name the file `question.pdf`).

## Rubric:

Points	Task	Criteria
15	Design	Pseudocode uses algorithmic thinking to express a high-level description of how to implement each of the 9 helper functions required for a success <i>Bear/ING</i> implementation.
15	Critical Thinking	In fewer than 500 words, Critical Thinking Task response describes a fundamental tradeoff that is common in computer science and describes three (3) metrics that a software developer can use to make a decision when faced with such a dilemma.
10	Programming	<code>is_vowel</code> , <code>is_consonant</code> and <code>contains_vowel</code> functions pass all unit tests.
10	Programming	<code>count_constants_at_front</code> and <code>count_vowels_at_back</code> functions pass all unit tests.
10	Programming	<code>ends_with</code> , <code>ends_with_double_consonant</code> and <code>ends_with_cvc</code> pass all unit tests.
20	Programming	<code>new_ending</code> passes all unit tests.

10	Programming	Each of the 9 helper functions is properly documented with comments that match the format given in the Programming Requirements section.
5	Programming	Code for each of the 9 helper functions is written to match the pseudocode and all variables are given appropriate/meaningful names.
5	Question	Your question shows an engagement with the material.

## Related Learning Objectives:

1. Writing boolean expressions using relational and logical operators
2. Using if-statements to implement selective program execution
3. Using algorithmic thinking to design programs
4. Write syntactically correct for/while loops
5. Identify the components of for/while loops
6. Use each of the three types of loops in the appropriate situation
7. Use methods on objects

## Credits:

stem by Marek Polakovic from the Noun Project