CS 15 Code Style Guide

How to write clear, concise, and modular C++ for CS 15

"A computer language . . . is a novel formal medium for expressing ideas about methodology, not just a way to get a computer to perform operations. Programs are written for people to read, and only incidentally for machines to execute."

-- Structure and Interpretation of Computer Programs Harold Abelson, Gerald J. Sussman, Julie Sussman

A program is a written description of your approach to solving a problem. It is a description that a computer can execute, but human readers are the target audience for whom you write. The compiler doesn't care about your variable names, code formatting, or comments-- it chews on what's underneath. People, on the other hand, rely on all those things, and you want them to understand and have confidence in your solution! Therefore, endeavor to write clear, concise, modular code.

All code you submit will be graded for structure and organization (including readability) as well as for functionality. Formatting matters. When you submit your work, it should be beautiful, and there should be no question of your dedication and correctness. Prepare it as if you were writing up an example for a good textbook. You wouldn't use a book that had no chapters, figures, captions, section headings, paragraph breaks, etc. Similarly, no one wants to read unorganized, undocumented code with huge functions and 500-character lines.

Just as with a paper for a literature course, the first draft of a program that works is not usually the version to submit. We allocate points of your homework/project grades for code that meets the criteria for readability. For example, we expect that you will clearly and consistently indent your code and adhere to an 80 column limit. We will also grade your internal documentation.

This document contains requirements (and also hints) for how to write programs that others, including the graders, will find easy to read. No document can be complete, and if you ever have questions about what is readable and what is not, please come see us!

Furthermore, although some of these style guidelines are specific to our course and others would not be applicable to all programming languages, writing readable, concise code is an extremely valuable skill that is important to practice early in your CS career; it will benefit you forever. No matter for whom you're writing your code, they will appreciate being able to read it without difficulty.

Finally, this style guide is written for CS 15 students, and refers to key CS 15 concepts. Therefore, it's important to continually refer back to this style guide as you progress through the course and gain a better understanding of the ideas that will contextualize these guidelines. Check it before each homework, and you'll see your code- writing skills (and grade) soar.

Internal Documentation (Comments)

Internal documentation comes mainly in the form of comments, and is crucial to having easy-to-understand code. Ideally, it would allow a reader with little to no prior knowledge of C++ to understand the gist of what is happening in your code, and why. For this reason, much of your

internal documentation will focus on the whys of your program — how does a given section of code contribute to the overall functionality of your program? This kind of documentation is important to include at many levels, from the README that describes your project as a whole, to the files that make up your program, to the individual functions you write, and even down to short but potentially confusing snippets of code.

There are two options for when to comment your code:

Before writing your code. Use comments (potentially combined with pseudo-code) to plan what you're going to write. As you write your code, update the comments to reflect changes in your plan.
As you write your code. Describe what you're doing in whatever piece of code you're currently writing. As your code becomes more complete, you can edit and potentially remove comments.

Writing your comments after finishing your code is not an option! If you show code to a member of the course staff, it is acceptable for them to say "I can't read this code without documentation. I'll help someone else while you work on that. Let me know when the documentation is complete."
The most important part of commenting is that the comments and code are in-sync with each other. Regardless of when you choose to write your comments, give them a once-over before submitting them to make sure that they still accurately represent your implementation. We have pondered for hours why code seemed to be different from what the comments said, only to conclude eventually that the code was unintentionally different from what the comments said. Then we had to find out which (if either!) was correct for the problem. Confidence in the code goes down after this.
File Header Comments
File header comments go at the top of each file. They must include some straightforward but important information — your name, the date, and the assignment title. When you edit a provided file that already has an author's name in it, you should add "Edited by" and then your name underneath the original author's name.
Another thing we look for in file headers is the purpose of the file. This is discussed in greater detail below.
Header comments should also include any known bugs or "to do" items. For example, "Currently only supports sophomores. TODO: support students of other years."
When you write a file's purpose in the header comments, the important thing to keep in mind is that you are writing for any potential users or clients of the file. A client will primarily be interested in what the file does at a high-level and how they can use it. A client typically does not care about low-level implementation details. With this in mind, there are typically three different types of files we must write header comments for:
When a user opens your driver file (the file that contains main()), they are probably wondering: What does this program do as a whole? What purpose does it serve? Is it a good fit for what I need to accomplish? Therefore, when you write the file header for a driver file, you should describe the whole program at a high-level, leaving out any low-level implementation details.

When a user opens a header (.h) file, they probably have noticed that it is a class header and are wondering: What is this class? What role does it play in the overall program? How does it interact with the other classes? What needs are best met by this class? They may be deciding whether to repurpose the class you wrote for a completely different program or, conversely, whether to use the rest of your program but replace that class with something else.

The file purpose for a header file should describe what instances of the class represent and what the abstract state of an instance is. For example: "StudentList is a class that represents an ordered list of Student instances. Every new StudentList begins empty, and clients can then add and remove Students from the list." You may go on to describe other behavior (i.e., public functions) that the class provides. If there are limitations, document them: "This class only handles lists of length 100 or less."

There should be very few, if any, implementation details in the header file purpose. The only implementation details included should be high-level ones that may be useful for a client to know. Here's an example of a helpful implementation detail: "The list is implemented using an array, and therefore provides quick access to elements." Here's an example of unnecessary implementation details: "The list is implemented using a pointer to a heap-allocated array, and an integer variable maintaining the capacity of that array. When the array fills up, its contents are copied to a new, larger heap-allocated array and the integer capacity is updated." The former focuses on efficiency, which is relevant to a client's decision process. The latter gets into low-level details that a client does not need to know about.

It is never the purpose of a class to "contain nodes." That is an implementation detail that clients don't care about. Implementation-specific information should go in the private section of the class and/or in the implementation (.cpp) file.

When a user opens a .cpp file for a class, it is probably because they have decided to use your class, either in your program or a program of their own, and want to know in greater detail how it works. They are probably wondering: How does this class actually work under the hood? Is there some bug still in this class I should know about? Is there some weird quirk or subtlety in the way this class is meant to be used?

Many of these low-level implementation details will be contained within the function definitions within the .cpp file, as well as the contracts for those functions (see below). Therefore, in the file purpose for an implementation file, it sufficies to simply state that the file implements the corresponding class, e.g.: "This file contains an implementation of the StudentList class." The only other details you may wish to include here are information about latent bugs/limitations of the class, or other quirks to the file that you believe may be helpful for a user to know.

Function Contracts

Function contracts appear just before the function header and are meant to explain how the function fits into the larger program. All functions should have function contracts, including int main in main.cpp. They must include the function's purpose, any parameters, any return values, and any other information that might affect the rest of the program. A function contract should not include implementation details, nor should it include information that can be found in the function prototype.

Purpose should say what the function does, not how the function does it. Often this takes the form of answering how does this function fit into the larger program. It should not include any implementation details. E. g., a function's purpose might be "To update the table so it maps the given key to the given value. The previous value associated with that key, if any, is lost."
Parameters should include information about what they represent in the function and larger program. This section should also include any restrictions or expectations on what will be passed as a parameter.
This section should not include information that can be gleaned from the function prototype: just saying "two ints and a string" is unhelpful.
Return value (if the function is non-void). "Return the number of distinct items in the table" or "Return the first item in the list greater than or equal to the given value."
Effects: This section should include side effects caused by the function. For example, in removeFromBack(), a list would change size (decrease in size by one). Input and output would be considered side effects, too.
Other information: This would include any memory management obligations imposed on the client. For example, the client should be told if they will need to recycle storage returned to them. This is not likely to happen in CS 15, but arises frequently enough in industry and other courses that it's worth mentioning. One should definitely include any exit or failure conditions that would cause the program to halt or the function to throw an exception. Also include identified or suspected bugs within the function ("this function does not work for __ reason" or "I think this function is causing my program to __").

Here is an example of a good function contract:

```
/*
 * name:      getElement
 * purpose:   get the StringArrayList element at a given index
 * arguments: an integer idx representing a StringArrayList index
 * returns:   the string element at index idx
 * effects:   none
 * other:     throws an exception if idx is out-of-bounds
 */
string StringArrayList::getElement(int idx) {
        ...
}
```

In-line Comments
Include:

Bugs. If you have identified a bug to a specific line of your program, make a note. Graders are more understanding of documented bugs versus bugs they have to suss out. At least, it shows you know it's there.

Confusion. If there's an unclear or complicated section of your code, something that would not be obvious to someone with experience coding, explain it.
Cases. If you are writing the code to deal with a specific case that's relevant to key invariants of the data structure, you should identify the relevance of the invariant.

Don't Include (in your submission):

Questions: In-line comments asking questions about the code or what should be done ("Do I need to increment this here?", "I don't know if this works") are not helpful and should not be in submitted code. Those kinds of questions can and should be addressed in office hours.
Outdated TODOs: Comments that indicate things that need to be written and/or fixed are useful while you are working, but the things you TO-DID should not still say TODO. If a part of your implementation is missing, a TODO comment is helpful as an acknowledgement.
Comments documenting low-level details that are obvious from the code itself. For instance, here is an example of code with excessive comments:

```
// increments i by 1
i++
// assigns x to the sum of y and z
x = y + z
// returns the size of s
return s.size()
```

All of the comments above are self-evident from the code they document.

Commented-out code should not be submitted! It's great to test out multiple potential solutions to a problem, but by the time you submit, you should have chosen the code that works the best and deleted anything you decided not to use. The occasional (but notable) exception to this rule is unit testing code that is meant to crash your program.
The README
A README is a file used to describe a program. Every open source project has a README file that describes what is going on in a program, what is necessary for it to work properly etc. In this course, it additionally will answer questions related to the topics/data structures covered in the assignment. The first thing someone opens when they look at a project is the README. When we are grading, that is the first file we open and the place we will go if we have questions or confusion about the code. It is crucial to fill it out thoroughly and in detail.

README files contain some straightforward, but important information:

Your name, the date, the assignment title, and the course (CS 15)
How to compile and run your program
Acknowledgements of any help you received — TAs you sought help from in office hours, peers you vented about your coding struggles with (without showing them your code, of course!), snippets of code from lecture or the course website you used as inspiration, online resources you found helpful (including our class forum), etc. Even if an assignment was particularly easy

for you and you didn't receive help from any of these sources, this section should not be left blank. You could instead list one obvious source of information, such as the professor's lecture on a relevant topic.
Any bugs still in your program that you were unable to fix, and where they can be found in the code
Data structures section for you to describe the data structures and/or ADTs used in the assignemnt
Sometimes the spec will include questions that should be answered in your README. It's worth noting here that those tend to be a significant portion of your grade.
Time spent section for you to note how many hours you spent on a particular assignemnt

There are also parts of the README that are a little less straightforward:

You should describe the purpose of your program. This can and should be done in three sentences or less, but it's important to think hard about what information would be useful to a potential user of your program. Namely, the implementation details of a program are almost always irrelevant to its purpose. When I was deciding between Atom, VSCode, Vim, Emacs, and Sublime, I considered the features of each editor and how effectively each one suited my needs. I did not wonder about the underlying implementation details of each editor at all. The only acceptable implementation details in a program purpose should be high-level and relevant to the user (see the description of .h header file comments for an example).
You should list and briefly describe every file you submitted. A list like the following would suffice:

StudentList.cpp: Implementation of StudentList class.
StudentList.h: Interface of the StudentList class.
main.cpp: Driver file which interacts with the user and the StudentList class.
unit_tests.h: Unit tests for the StudentList class.
README: This file.
test_data1.txt: Test input data.
test_data2.txt: Test input data.

It's important to list every file you submitted, even if the description is obvious! For example, if one of your files is mistakenly left out of your submission, having it listed in the README will help the TA understand what happened and remedy that.
You should describe, chronologically and in detail, how you tested and debugged your program as you were working on it. In this course, you will be required to write unit tests for every program you write. Unit testing is necessarily something you do as you write your code, so the full process is not always perfectly visible from the final files you submit. This is your chance to convince the grader that you thoughtfully and thoroughly tested your program as you were writing it. Include, for example, information about how you isolated and tested particularly tricky functions, and describe bugs you found while unit testing and how you fixed those, or edge cases that came up through your testing. If you tested as you wrote code and documented your ideas throughout the assignment, this step should be trivial.

Last but not least, seeing as this is a Data Structures course, you should describe the data structures and/or ADTs you used in your program! This is your chance to explain the implementation details of your program. What data structures or ADTs did you practice writing this week? How do they work? What are their advantages and disadvantages? Some examples of data structures you could include, if applicable, are array lists and stacks. Similarly, if you learned and practiced implementing a new algorithm, briefly explain how it works. Write as if this section is meant to be read by somebody who has only taken an introductory computer science course, but is really interested in learning about the concepts from our course.

The Code Itself

Organization into Classes and Structs (Data Structures!)

If a struct is meant to be used in a class, it should be defined inside of that class. If it is used internally to the class, it should be private. For example, a Node struct used inside a linked list should be private. If a struct needs to be declared outside of any other class for whatever reason, it should be declared in its own header file.

Non-interface functions in a class should be declared private. The only functions that should be public are the ones required for the client to be able to use the class.

All data members should be declared private.

Recursion

Recursive functions should only be concerned with the current element. Avoid dereferencing "next" or children pointers in the function body; rather, just recurse to the next or child node.

Variables

No global variables: no variables should be declared or defined outside of functions with the exception of global constants.

Use of literals should be avoided; use global (or static) constants instead. If your program calls for an 8x8 two-dimensional array, for example, create a global constant ARRY_SIZE = 8, then declare your array with int[ARRY_SIZE][ARRY_SIZE] rather than int[8][8].

Variable names should be descriptive, with the exceptions of variables with very limited scope: i and j for loops (triple-nested loops should not occur); curr_<type> (e.g. curr_node) as a pointer; temp or aux for very temporary variables (accessed by 3 lines of code or fewer). One appropriate use for temp would be swapping two elements in an array, for example. Foo, var, and x are not descriptive variable names and should never be used.

Brevity

Use boolean expressions and values. For example, there is no reason to write

```
if (isBig == true)
```

when you can write

```
if (isBig)
```

Don't write functions longer than 30 lines between the opening and closing brace. This will affect your modularity grade.

Do not rewrite code that is given to you, create functions with nearly identical uses, or write in-place code when there is a function you can call to do the same work.

Use helper functions to simplify code, especially if they can be called in many places. But a helper function can also be useful if it provides a name for a computation that makes other code clearer. For example, rather than test

```
if (front == nullptr)
```

all the time, you can define a function called
if (isEmpty())
Try to avoid doing unnecessary work. For example, rather than recomputing something several times, compute it once and save it in a well-named variable.
Whitespace
Indentations should be made up of either 4 or 8 spaces, not tab characters (with the important exception of makefiles. If you don't understand why a Makefile must contain tabs, ask a TA). Tabs display differently based on a computer's settings, but spaces always look the same. Most text editors have a setting to output a specified number of spaces instead of a tab character whenever you press the "tab" key.
Furthermore, indentation should be consistent! Indentation is a crucial part of code's readability, as it helps readers understand how code fits together. Make sure the levels of indentation accurately reflect which function or code block(s) any given line is part of.
In our course, the width of your indentation should be either 4 or 8 spaces. Indentation of only two spaces makes your code more difficult for us to read. If this causes you to struggle to adhere to the 80 column rule, then your code has too many levels of nesting, and needs to be more concise, more modular, or both.
Binary operators (operators that take two pieces of input, such as '+', '*', '=', and '==') should have spaces around them. There are two exceptions to this rule. The first is the dot operator (.) for accessing struct or class members. The second is the arrow operator (->) for dereferencing a struct or class pointer and then accessing a member. There should not be spaces around either of these operators. To clarify, you should do object.member and object_ptr->member.
Unary operators (operators that take only one piece of input, such as '++') should not have spaces around them.
When declaring pointers, the asterisk should be attached to the variable name, not the type (Node *curr rather than Node* curr).
Rationale: The compiler interprets the * as a decoration of the variable name on its right, not part of the type: Node* np1, np2; declares one pointer variable and one variable that contains a Node, which is confusing as written. Putting the * next to the variable it modifies makes it clear to the reader which variables, if any, hold pointer values: Node *np1, np2;. (np2 is poorly named, but its status in the declaration is clear.)
In lists declaring and initializing variables, there should be a space after every comma.
There should be a single space between a loop or conditional keyword (such as for, if, and while) and its corresponding opening parenthesis, but no space between the name of a function and its corresponding opening parenthesis.
This should go without saying, but do not violate any of these guidelines in order to make your code adhere to the 30 line or 80 column rules. This will make your code difficult for the grader to read, and therefore will not help your score.
Other Guidelines
No line should be longer than 80 characters. In other words, no files that you wrote should have more than 80 columns. Typing "wc -L *" into the terminal will display the number of columns in each file in your current folder.

There are some style rules that pertain to conditional statements and loops, as well. These determine the control flow of your program, so having easy-to-read conditional statements and loops is immensely important to a reader's ability to understand what is happening in your code.
Use the keywords "and," "or," and "not" as opposed to the older, equivalent operators "&&," "||," and "!" so your code is both easier to read and less prone to difficult-to-spot bugs caused by typos. (Note: You should use "!="" when checking for inequality)
Don't treat non-boolean variables as booleans — we know that when a number is 0 or pointer is null, it will evaluate to false. For readability, however, you should explicitly compare the variable to zero. For example, write while (n > 0) rather than while (n) (Sorry, Python programmers.)
The break keyword should only be used when you have a switch statement. In addition to making loops more difficult for a reader to understand, it undermines one objective of our course, which is to learn to write code (including loop conditions) thoughtfully.
Returning from inside of a loop, however, is often useful and encouraged. The continue keyword may be used occasionally, but should be avoided when possible.
Curly braces. We are somewhat flexible on this, but be consistent, but here are some notes:
If an open curly brace ({) is placed at the end of a line of code, there should be a space before it to separate it from what came before. It may also be placed on the next line lined up under the beginning of the statement it's part of.
Close curly braces (}) should be on a line by themselves unless the statement they are part of continues. For example, you may write "} else {" all on one line (or you can put each item on its own line).
We do not require the use of curly braces for blocks of a single statement. That is, you may write an if statement or a for loop without any curly braces if the body is one line long. However, as a rule, do not put an entire if statement on one line.
The keyword auto should only be used when declaring variables of the iterator type. This is a Data Structures course — you're expected to develop a very strong grasp of which class, struct, or primitive data type you are working with at any given time.
When working with pointers, the keyword nullptr is preferable to the old-fashioned, C-style constant NULL.
[EDIT: Added Jan - 27 - 2022] Finally, we gently recommend against using this-> to refer to a class' members from inside of one of its functions. Points will not be taken off for doing so, but students should know that it is typically unnecessary.

Finally, our parting word of advice is to always give your code a quick once-over immediately before submitting. Maybe there are function contracts or file headers that are no longer relevant; maybe you forgot to document something as you were writing; or maybe there is a particularly complicated function that you will realize needs better in-line commenting. Documentation and Style points are, in many ways, the easiest points to earn on any given CS 15 assignment — whether or not you have time to get your code working perfectly (or at all) before the due date, you can always get full points for this section. Don't let yourself miss out on what should be very reliable credit just because something slipped your mind!