AP Computer Science Principles: C++ Style Guide

Bear Creek High School, 2025-26

> "Programming is best regarded as the process of creating works of literature, which are meant to be read."
>
> - Donald Knuth

1 Introduction

Style is just as important as substance when writing software. Style helps to ensure that your code can be read and maintained by others, and it conveys to the reader that you are serious and care about the quality of your software.

C++ is a powerful, flexible language with an enormous amount of features. It also has relatively few built-in rules about syntax and style, unlike a language like Python, which is more heavy-handed about how its programs should look. This breadth of features and lack of enforced rules makes the adoption of a C++ style guide imperative. Without one, there are an uncountably large number of ways that a valid (that is, functional) C++ program might look, and it's unrealistic to expect that people reading your code will automatically understand its structure if you don't follow predictable standards.

As a result, the code that you write for this course will be expected to follow not only the syntactic rules that C++ enforces automatically, but also a handful of stylistic *conventions* that are widely agreed upon as good practice by real-world C++ programmers. These conventions (mostly adopted from the style guides created by Google and by the developers of C++ itself), while technically optional, really aren't. They allow your code to be read by other C++ programmers, they project a certain aesthetic integrity and predictability in the way your code looks, and they help prevent your programs from getting ugly, which can happen all too easily in the world of C++.

Adhering to this style guide is important. It is imperative that people other than you be able to read and understand your code, and that will be nearly impossible without a consistent format. You will see throughout this course just how frustrating it is to read another person's code when they don't follow stylistic rules.

Throughout AP Computer Science, your code will be graded not just on its function but also on its adherence to this style guide.

2 Naming

Naming is crucial in programming, as it allows others to quickly get an idea of whether something is a file, function, or variable, and it helps them infer what that thing might do. For this reason, Google considers naming the single most important rule in code style (and I agree), and you will be required to practice consistent naming in your code.

The following rules apply to naming when writing C++ programs for this course:

- When naming anything, prioritize making it simple and clear. People with no programming experience should be able to see something you've named and tell you what it does or represents.
- C++ file names should be in lowercase, with additional words separated by underscores. This is known as *snake_case*. All C++ file names should end with the extension `.cpp`. For example, you might have a file name like `my_cool_project.cpp`

- Standard variable names in C++ should also be in snake case:

```
int my_num = 66;
string instructor_name = "Kercheval";
```

- Function names are not variables, so you have the option to format them in camel case:

```
void AddNewStudent()
int GetStudentCount()
string PrintSchoolName()
```

  When writing small programs, you may format function names in snake case, but it's a good idea to switch to camel case when you start writing more complex software.

- *All* programmer-chosen names within a file, including variable names, must be relevant to the program. This will take practice, but here are some general guidelines to be aware of:
  - **Function names** usually involve verbs, like `CountItems()` or `PopulateTable()`.
  - **Variable names** are usually nouns, like `index` or `user_count`.
  - When naming something, consider: what does it do? What does it represent? How does it fit into the context of the overall program?
  - That said, extremely long and wordy variable names become detrimental to a program's legibility. *Always try to keep variable names descriptive but short.*
  - The only exception to this rule is that very short variable names are allowed when their contexts are very limited, like when naming a counter variable in a `for` loop (i.e. it's OK to say `for (int i = 0; i < 10; i++)` instead of `for (int counter = 0; counter < 10; counter++)`.

3 Formatting

Consistent, sane formatting helps make your code readable. C++, interestingly, couldn't care less about formatting and will compile your code even if it's all written on one enormous line. Other programmers, the College Board, and the expectations of this class do care about formatting, though. You should too.

- **Your program must never be more than 80 columns wide.** This is a fairly standard, accepted convention across many programming languages. It ensures that your code can be read in many different editors and in terminals, and nothing breaks a train of thought quicker than having to scroll horizontally when reading someone else's code.
  - Where to break a line of code that exceeds 80 columns is generally up to the programmer, but it usually is best done before operators, with the new line indented in such a way that it is vertically aligned with variables or statements of the same context. For example:

```
int some_result = LongFunctionName(var_one, var_two,
                                   var_three, var_four);
```

  - When using operations on multiple variables, chain the additional content on multiple lines with the operation leading the next element, instead of following the previous one. For example, do this:

```
string the_raven = "Once upon a midnight dreary, "
                 + "while I pondered, weak and weary, "
                 + "over many a quaint and curious volume "
                 + "of forgotten lore";
```

  and not this:

```
string the_raven = "Once upon a midnight dreary, " +
                   "while I pondered, weak and weary, " +
                   "over many a quaint and curious volume " +
                   "of forgotten lore";
```

- **Your program's functions must never be more than 50 lines long.** This helps prevent the reader from getting lost, and it helps you keep your functions concise and on-task.
- Your `main` function must either be placed at the very top of your program, with all other function *signatures* (that is, their names) listed above it and their implementations below it, or your `main` method must be placed at the very bottom of your file.
- There is some debate among programmers as to whether to format curly braces like this

  ```
  while (condition) {
      // do something...
  }
  ```

  or like this

  ```
  while (condition)
  {
      // do something...
  }
  ```

  It does not matter which you choose for your code, as long as you are consistent. However, this is only relevant for the placement of the opening curly brace. The closing curly brace must always be on a new line.
- There is also significant debate about tab formatting. Should they be tab characters or spaces? Should they be 2, 4, or 8 characters wide? Again, in this case it does not matter which convention you use, as long as you are consistent. (Kercheval's personal preference is 4 spaces.)
- Use spaces to pad mathematical operators. This makes complex operations significantly more legible. For example, write `int nine = 4 + 5` instead of `int nine=4+5`.
  - The only exceptions to this rule are the `++` and `--` operators, which should immediately follow the variable they operate on, without a space.
- Use spaces liberally when writing `if/for/while` blocks in your code as well. For example:

  ```
  if (condition) {
      for (int i = 0; i < limit; i++) {
          // do something...
      }
  }
  ```

  is significantly easier to read than:

  ```
  if(condition){
      for(int i=0;i<limit;i++){
          // do something...
      }
  }
  ```

4 Bad Habits

Like glitter, bad programming habits are easy to pick up and hard to get rid of. Some languages are very permissive of these habits, including C++. Indeed, many of these habits make programming easier, since they reduce the burden on you as the programmer. Don't be tempted. The dangerous programming practices listed below, while perhaps legal according to C++, will cause debugging mayhem and hideous code for you down the line if you depend on them, and in this class they will result in a reduction of credit on assignments.

- **No global variables! Never!** Global variables are a very easy solution to the classic programming question of "where do I put this variable and how do I pass it around?" However, they make it practically impossible to debug which process might be editing them, and heaven forbid you accidentally pick a variable name in a function that you've already defined globally. I will show you how global variables work in the name of giving you the truth, but I will be vocally upset if you use them in a program.
  - If you really, really want to, you may define "constants" (not "variables") at the beginning of a C++ file. They are called constants because they may never change, and every function in the file may depend on that. Their names must be entirely capitalized, with underscores separating words. For example, if you want a variable to hold the value of the speed of light (which will never change), you could write: `const int SPEED_OF_LIGHT = 299752458;`
- **No do-while! Only while!** Do-while loops are the coding equivalent of a cartoon animal running off the edge of a cliff before looking down and realizing there's no ground beneath it. Only use standard `while` loops in C++, so that you always check that you can do something before you do it.
- **No `while (true)`!** It's is not only lazy, it's a one-way ticket to infinite loop town. Always use a condition to quantify your `while` loops' endings. They make your code safer, more predictable, and easier to read.
- **Be careful with conditionals.** Large blocks of labyrinth-like nested `if/else` statements are a telltale sign of a program that has not been well designed. Always consider ways to be concise with your conditionals. This is as simple as asking yourself questions like "do I really need `if/else if` in this case, or can I just use `if/else`? Do I even need an `else`?"

5 Idioms and Syntactic Sugar

Programming is full of *idioms*, which are language-specific constructs for representing different actions. It is also full of *syntactic sugar*, which are complex or tedious operations wrapped in simple syntax. C++ is loaded with both, which is part of the reason why it's so popular and so versatile. Both idioms and syntactic sugar make writing code quicker and usually more concise, but they can quickly make your code impossible to decipher to anyone who doesn't know them. For that reason, we will generally avoid using idioms in this course, and we will learn the syntactically "sour" equivalent of all sugary expressions first. Some of these expressions, however, are so common in programming that they come up more often than their idiom- or sugar-free counterparts. Acceptable idioms and sweetened expressions in the scope of this course, then, include:

- `i++` and `i--`
  - These represent `i = i + 1` and `i = i - 1`, respectively. Either way of incrementing or decrementing a value by 1 is acceptable in this course.
- `n += 3`
  - This represents `n = n + 3`. Writing the operation either way is acceptable in this course.

- `k /= 2`
  - This represents `k = k / 2`. Writing the operation either way is acceptable in this course.
- `if (condition)`
  - This is an acceptable way to check whether `condition` is true, and is in fact significantly preferred to writing `if (condition == true)`, which is redundant.
- `if (!condition)`
  - This is an acceptable way to check whether `condition` is false, but writing `if (condition == false)` is fine as well, since it is not redundant.
- Sugary printing and reading
  - C++ is famous for its iconic input and output operators, respectively `>>` and `<<`. They are not the only ways to achieve input and output in C++, but they are certainly the easiest to read and write. For example, this:

    ```
    cout << "hello, world!" << endl;
    ```

    Is the same as:

    ```
    printf("%s\n", "hello, world!");
    ```

    Which is a bit stranger to interpret, especially for those new to C++. I encourage you to use the sugary form when you wish, though we will explore `printf` and its history and surprisingly sublime behavior.
- `for-each` loops
  - These are somewhat common in C++ programs, and can be pretty helpful when you're doing standard work with lists of variables. For example:

    ```
    int[] data = {1, 2, 3, 10, 20, 30};
    for (int item : data) {
        item += 5;
    }
    ```

    Is an idiomatic way of saying:

    ```
    int[] data = {1, 2, 3, 10, 20, 30};
    for (int i = 0; i < data.length; i++) {
        data[i] = data[i] + 5;
    }
    ```

    You should be familiar with both ways of writing a `for` loop, for the sake of being able to read code in other languages, which tend to follow the latter example's syntax.

6 Comments

Appropriate use of comments (or *documentation*) is *the* way to help others read your code. Good commenting will ensure that your code is thought out well, that it's maintainable in the future, and that you yourself can read your old programs and understand why you made the design choices you did.

Quality comments are required for the code you write in this class. Lack of documentation in code will be considered a bug, and code that is not properly documented will not receive full credit.

Comments in C++ can take two forms: *block* or *in-line* comments, which span multiple lines or just one line, respectively. Block comments are opened with `/*` and closed with `*/`. For example:

```
/* This is a block comment.
 *
 * The comment continues until it is explicitly closed.
 */
```

In-line comments only last until the end of the line they're on, and must be prepended with `//`:

```
// This is an in-line comment.
// All in-line comments must begin with two slashes.
int one_six = 2 * 3;            // Multiplication
int another_six = 3 + 3;        // Addition
```

Note that the comments are capitalized and punctuated, and that the in-line comments are a healthy horizontal distance away from the code they comment. I expect your comments to look the same. Don't be lazy!

When and where to leave comments is generally up to the programmer, but in this course (and in enterprise software development) we will impose some conventions. Those conventions are:

- All `.cpp` files define a program. At the top of each file, you must include information about the program defined there, in the following format:

  ```
  /* program_name
   * Author: [your name]
   * Date of most recent edit
   *
   * Brief description of what the program is/represents, how it fits within
   *     the project as a whole
   *
   * Function signatures:
   * Signatures (but not descriptions) of class's methods, for example:
   * void set_value(int new_value)
   * string another_function()
   * bool is_equal_to(string thing_to_compare)
   */
  ```

- Similarly, all functions defined within a file must be commented, directly above their implementation. A function's documentation should be in the following format:

  ```
  /* function_name
   * Description of what the function does, *not* how it does it, including
   *     contracts it upholds, assumptions it makes
   *
   * Parameters:
   * parameter 1, type and brief description for context, if needed
   * parameter 2, type and brief description for context, if needed
   *
   * Returns:
   * type of and context for any returned values
   *     (say none for void methods)
   */
  ```

- Along with class and method comments, in-line comments are welcome for new "paragraphs" in your code, that is, places in your program where your process changes direction, as well as for places in your code where you perform an operation or calculation that may not be immediately obvious to the reader, where you want to help them to understand your thought process.

Effective commenting, especially in-line commenting, is a delicate art, and it requires you to be able to tell when your comments are useful and when they are clutter. You can generally consider comments to be "the more the merrier" for this course, but if you find yourself constantly leaving comments explaining your work, reflect on whether or not you can change your code's algorithms and structure so that it's understandable without an excess of comments.

7 Notes on linguistic precision

You will find throughout your experience with technology that computer scientists tend to be nitpicky, pedantic grammar freaks. While sometimes annoying and counterproductive (see the Linux vs GNU/Linux debate), this quality is important. Typos, a lack of precision, and incorrect word choice might be inconsequential for small projects, but on large-scale programming tasks, they can seriously hinder a user's ability to understand what's going on, as well as your ability as a programmer to effectively communicate about your work. In this course, therefore, you are expected to be as precise with your English as you are with your C++. This is a broad expectation, but some specifics include:

- Run `aspell check [filename]` on your program prior to submission to catch typos.
- An equals sign by itself is called the "assignment operator" and is pronounced "gets," not "equals." `int x = 5` is pronounced "int x gets five."
- Two equals signs represents a check for equality and is pronounced "equals." `thing1 == thing2` is pronounced "thing1 equals thing2."
- Using brackets to access an array element is pronounced "sub." `my_list[3]` is pronounced "my_list sub three."