Style Guide

Written by Julie Zelenski, based on earlier work by Marty Stepp and Keith Schwarz

You may think the motivation for good style is earning that ✓+ from your section leader, but the most important beneficiary of your efforts is **you yourself**. Committing yourself to writing tidy, well-structured code from the start sets you up for good times to come. Your code will be easier to test, will have fewer bugs, and what bugs there are will be more isolated and easier to track down. You finish faster, your results are better, and your life is more pleasant. What's not to like?

The guidelines below identify some of style qualities we will be looking for when grading your programs. As with any complex activity, there is no one "best" style, nor a definitive checklist that covers every situation. That said, there are better and worse choices and we want to guide you toward the better choices.

In grading, we will expect that you make a concerted effort to follow these practices. While it is possible to write code that violates these guidelines and yet still exhibits good style, we recommend that you adopt our habits for practical purposes of grading. If you have theoretical points of disagreement, come hash that out with us in person. In most professional work environments you are expected to follow that company's style standards. Learning to carefully obey a style guide, and writing code with a group of other developers where the style is consistent among them, are valuable job skills.

This guide gives our general philosophy and priorities, but even more valuable will be the guidance on your own particular style choices. Interactive grading with your section leader is your chance to receive one-on-one feedback, ask questions, and learn about areas for improvement. Don't miss out on this opportunity!

Layout, Indentation, and Whitespace

- **Indentation:** Consistent use of whitespace/indentation always! Proper whitespace/indentation illuminates the structure of your program and makes it easier to follow the code and find errors.
    - Increment indent level on each opening brace `{`, and decrement on each closing brace `}`.
    - Chose an increment of 2-4 spaces per indent level. Be consistent.
    - Do not place more than one statement on the same line.

```
// confusing, hard to follow
while (x < y) {
   if (x != 0) { binky(x);
   }
else { winky(y);
    y--; }}
  return x;
```

```
// indentation follows structure
while (x < y) {
    if (x != 0) {
        binky(x);
    } else {
        winky(y);
        y--;
    }
}
return x;
```

- **Long lines:** When any line is longer than 100 characters, break it into two lines. Indent the overflow text to align with text above.

```
result = longFunctionName(argument, 106 * expression * variable) + variable - longerFunctionName()
+ otherFunction(variable);
```

```
result = longFunctionName(argument, 106 * expression * variable) + variable -
         longerFunctionName() + otherFunction(variable);
```

- **Blank lines:** Use blank lines to separate functions and logical groups of statements within a function.

- **Whitespace:** Add space between operators and their operands. Add parentheses to show grouping where precedence might be unclear to reader.

```
int root = (-b+sqrt(b*b-4*a*c))/2*a;
```

```
int root = (-b + sqrt((b * b) - (4 * a * c))) / (2 * a);
```

Names

Choose meaningful identifiers. This reduce the cognitive load for reader and self-documents the purpose of each variable and function.

- **Nouns for variable names:** For variables, the question is "What is it?" Use a noun (`name`,`scores`) add modifier to clarify (`courseName`, `maxScore`). Do not repeat the variable type in its name (not `titleString`, just `title`). Avoid one-letter names like `a` or `p` (exceptions for loop counters `i`, `j` or, coordinates `x` and `y`). Never name a variable `l`, much too easily confused with the number one.
- **Verbs for function names:** For functions, the question is "What does it do?" Functions which perform actions are best identified by verbs (`findSmallest`, `stripPunctuation`, `drawTriangle`). Functions used primarily for their return value are named according to property being returned (`isPrime`, `getAge`).
- **Use named constants:** Avoid sprinkling magic numbers throughout your code. Instead declare a named `const` value and use where that value is needed. This aids readability and gives one place to edit value when needed.

  ```
  const int VOTING_AGE = 18;
  ```

- **Capitalization:** Use camel-case for names of functions and variables (`countPixels`), capitalize names of classes/types (`GridLocation`), and uppercase names of constants (`MAX_WIDTH`). Conventions allows reader to quickly determine which category a given identifier belongs to.

Variable scope

- **Scope:** Declare variables in the narrowest possible scope. For example, if a variable is used only inside a loop, declare it inside the scope for the loop body rather than at the top of the function or at the top of the file.
- **Don't reuse same name in inner scope**: Declaring a variable in inner scope with same name as a variable in outer scope will cause inner use of name to "shadow" the outer definition. Not only is this confusing, it often leads to difficult bugs.
- **No global variables:** Do not declare variables at global scope. When there is information to be shared across function calls, it should flow into and out via parameters and return values, **not** reach out and access global state.

Use of C++ language features

- **Prefer C++ idioms over C idioms:** Since C++ is based on C, there is often a "C++ way" to do a given task and also a "C way". For example, the "C++ way" to print output is via the output stream `cout`, while the "C way" is using `printf`. C++ strings use the `string` class, older code uses the C-style `char*`. Prefer the modern C++ way.

  ```
  // old school
  char* str = "My message";
  printf("%s\n", str);
  ```

  ```
  // modern and hip
  string str = "My message";
  cout << str << endl;
  ```

- **for vs while:** Use a `for` loop when the number of repetitions is known (definite); use a `while` loop when the number of repetitions is unknown (indefinite).

```
// loop exactly n times
for (int i = 0; i < n; i++) {
    ...
}

// loop until there are no more lines
string str;
while (input >> str) {
    ...
}
```

- **break and continue in loops:** Wherever possible, a loop should be structured in the ordinary way with clear loop start, stop, advance and no disruptive loop control. That said, there are limited uses of `break` that are okay, such as loop-and-a-half (`while(true)` with `break`) or need to exit loop mid-iteration. Use of `continue` is quite rare and often confusing to reader, better to avoid.
- **Use of fallthrough in switch cases:** A switch case should almost always end with a `break` or `return` that prevents continuing into the subsequent case. In the very rare case that you intend to fallthrough, add a comment to make that clear. Accidental fallthrough is the source of many a difficult bug.

```
switch (val) {
    case 1:
        handleOne();
        break;
    case 2:
        handleTwo();
        // NOTE: fallthrough ***
    case 3:
        handleTwoOrThree();
```

- **return statements** Although it is allowed for a function to have multiple `return` statements, in most situations it is preferable to funnel through a single `return` statement at the end of the function. An early `return` can be a clean option for a recursive base case or error handled at the beginning of a function. `return` can also serve as a loop exit. However, scattering other `return` throughout the function is not a good idea– experience shows they are responsible for a disproportionate number of bugs. It is easy to overlook the early-return case and mistakenly assume the function runs all the way to its end.
- **Always include `{}` on control statements:** The body an `if/else`, `for`, `while`, etc., should always be wrapped in `{}` and have proper line breaks, even if the body is only a single line. Using braces prevents accidents like the one shown below on left.

```
// ugh
if (count == 0) error("not found");
for (int i = 0; i < n; i++) draw(i);
if (condition)
    doFirst();
    doSecond(); // inside? Indent looks so, b
ut no braces!
```

```
// better
if (count == 0) {
    error("not found");
}
for (int i = 0; i < n; i++) {
    draw(i);
}

if (condition) {
    doFirst();
    doSecond();
}
```

- **Booleans:** Boolean expressions are prone to redundant/awkward constructions. Prefer concise and direct alternatives. A boolean value **is** true or false, you do not need to further compare to true/false or convert a boolean expression to true/false.

```
if (isWall == true) {
   ...
}

if (matches > 0) {
    return true;
} else {
    return false;
}
```

```
// better

if (isWall) {
    ...
}


return (matches > 0);
```

- **Favor `&&`, `||`, and `!` over `and`, `or`, and `not`:** For various reasons mostly related to international compatibility, C++ has two ways of representing the logical connectives AND, OR, and NOT. Traditionally, the operators `&&`, `||`, and `!` are used for AND, OR, and NOT, respectively, and the operators are the preferred ways of expressing compound booleans. The words `and`, `or`, and `not` can be used instead, but it would be highly unusual to do so and a bit jarring for C++ programmers used to the traditional operators.

```
// non-standard
if ((even and positive) or not zero) {
    ...
}
```

```
// preferred
if ((even && positive) || !zero) {
    ...
}
```

- **Use error to report fatal conditions:** The `error` function from the Stanford library can be used to report a fatal error with your custom message. The use of `error` is preferred over throwing a raw C++ exception because it plays nicely with the debugger and our SimpleTest framework.

```
// raw exception
if (arg < 0) {
    throw arg;
}
```

```
// preferred
if (arg < 0) {
    error("arg must be positive!");
}
```

Efficiency

In CS106B, we value efficient choices in data structure and algorithms especially where there is significant payoff, but are not keen on micro-optimizations that serve to clutter the code for little gain.

- **Better BigO class**: Given a choice of options for implementing an algorithm, the preference is generally for the one with better Big O, i.e. an O(NlogN) algorithm is preferable to quadratic O(N^2), constant O(1) or logarithmic O(logN) beats out linear O(N).
- **Choose best performing ADT for situation:** For example, if you need to do many lookup operations on collection, `Set` would be preferable to `Vector` because of efficient `contains` operation. All Stack/Queue operations are O(1) making `Stack` an ideal choice if you only add/remove at top or `Queue` perfect if you remove from head and add at tail. There is a small win for choosing `HashSet`/`HashMap` over `Set`/`Map` when you do not require access to elements in sorted order.
- **Save expensive call result and re-use:** If you are calling an expensive function and using its result multiple times, save that result in a variable rather than having to call the function multiple times. This optimization is especially valuable inside a loop body.

```
// computes search twice
if (reallySlowSearch(term) >= 0) {
    remove(reallySlowSearch(term));
}
```

```
// avoid recompute
int index = reallySlowSearch(term);
if (index >= 0) {
    remove(index);
}
```

- **Avoid copying large objects:** When passing an object as a parameter or returning an object from a function, the entire object must be copied. Copying large objects, such as collection ADTs, can be expensive. Pass the object by reference avoid this expense. The client and the function then share access to the single instance.

```
// slow because of copying

void process(Set<string> data) {
    ...
}

Vector<int> fillVector() {
    Vector<int> v;
    // add data to v
    ...
    return v; // makes copy
}
```

```
// improved efficiency

void process(Set<string>& data) {
    ...
}

// shares vector without making copy
void fillVector(Vector<int>& v) {
    // add data to v
    ...
}
```

Unify common code, avoid redundancy

When drafting code, you may find that you repeat yourself or copy/paste blocks of code when you need to repeatedly perform the same/similar tasks. Unifying that repeated code into one passage simplifies your design and means only one piece of code to write, test, debug, update, and comment.

- **Decompose to helper function:** Extract common code and move to helper function.

```
// repeated code
if (g.inBounds(left) &&
    g[left] && left != g[0][0] ) {
    return true;
} else if g.inBounds(right) &&
    g[right] && right != g[0][0] ) {
    return true;
}
```

```
// unify common into helper
bool isViable(GridLocation loc,
              const Grid<bool>& g)
{
    return g.inBounds(loc) &&
      g[loc] && loc != g[0][0];
}

...

return isViable(left, g) ||
       isViable(right, g);
```

- **Factoring out common code:** Factor out common code from different cases of a chained if-else or switch.

```
// repeated code
if (tool == CIRCLE) {
    setColor("black");
    drawCircle();
    waitForClick();
} else if (tool == SQUARE) {
    setColor("black");
    drawSquare();
    waitForClick();
} else if (tool == LINE) {
    setColor("black");
    drawLine();
    waitForClick();
}
```

```
// factor out common

setColor("black");

if (tool == CIRCLE) {
    drawCircle();
} else if (tool == SQUARE) {
    drawSquare();
} else if (tool == LINE) {
    drawLine();
}

waitForClick();
```

Function design

A **well-designed function** exhibits properties such as the following:

- Performs a single independent, coherent task.

- Does not do too large a share of the work.
- Is not unnecessarily entangled with other functions.
- Uses parameters for flexibility/re-use (rather that one-task tool).
- Clear relationship between information in (parameters) and out (return value)

Function structure: An overly long function (say more than 20-30 lines) is unwieldy and should be decomposed into smaller sub-functions. If you try to describe the function's purpose and find yourself using the word "and" a lot, that probably means the function does too many things and should be subdivided.

Value vs. reference parameters: Use reference parameters when need to modify value of parameter passed in, or to send information out from a function. Don't use reference parameters when it is not necessary or beneficial. Notice that `a`, `b`, and `c` are not reference parameters in the following function because they don't need to be.

```
/*
 * Solves a quadratic equation ax^2 + bx + c = 0,
 * storing the results in output parameters root1 and root2.
 * Assumes that the given equation has two real roots.
 */
void quadratic(double a, double b, double c,
               double& root1, double& root2) {
    double discr = sqrt((b * b) -(4 * a * c);
    root1 = (-b + discr) / (2 * a);
    root2 = (-b - discr) / (2 * a);
}
```

Prefer return value over reference 'out' parameter for single value return: If a single value needs to be sent back from a function, it is cleaner to do with return value than a reference `out` parameter.

```
// harder to follow
void max(int a, int b, int& result) {
    if (a > b) {
        result = a;
    } else {
        result = b;
    }
}
```

```
// better as
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Avoid "chaining" calls, where many functions call each other in a chain without ever returning to `main`. Here is a diagram of call flow with (left) and without (right) chaining:

```
// chained control flow
main
|
+-- doGame
    |
    +-- initAndPlay
        |
        +-- configureAndPlay
            |
            +-- readCubes
            |
            +-- playGame
                |
                +-- doOneTurn
```

```
// better structured as
main
|
+-- welcome
|
+-- initializeGame
|   |
|   +-- configureBoard
|       |
|       +-- readCubes
|
+-- playGame
|   |
|   +-- doOneTurn
```

Commenting

Some of the best documentation comes from giving types, variables, functions, etc. meaningful names to begin and using straightforward and clear algorithms so the code speaks for itself. Certainly you will need comments where things get complex but don't bother writing a large number of low-content comments to explain self-evident code.

The audience for all commenting is a C++-literate programmer. Therefore you should not explain the workings of C++ or basic programming techniques.

Some programmers like to comment before writing any code, as it helps them establish what the program is going to do or how each function will be used. Others choose to comment at the end, now that all has been revealed. Some choose a combination of the two, commenting some at the beginning, some along the way, some at the end. You can decide what works best for you. But do watch that your final comments do match your final result. It's particularly unhelpful if the comment says one thing but the code does another thing. It's easy for such inconsistencies to creep in the course of developing and changing a function. Be careful to give your comments a once-over at the end to make sure they are still accurate to the final version of the program.

- **File/class header:** Each file should have an overview comment describing that file's purpose. For an assignment, this header should include your name, course/section, and a brief description of this file's relationship to the assignment.
- **Citing sources:** If your code was materially influenced by consulting an external resource (web page, book, another person, etc.), **the source must be cited**. Add citations in a comment at the top of the file. Be explicit about what assistance was received and how/where it influenced your code.
- **Function header:** Each function should have a header comment that describes the function's behavior at a high level, as well as information about:
  - **Parameters/return:** Give type and purpose of each parameter going into function and type and purpose of return value.
  - **Preconditions/assumptions:** Constraints/expectations that client should be aware of.("this function expects the file to be open for reading").
  - **Errors:** List any special cases or error conditions the function handles (e.g. "…raises error if divisor is 0", or "… returns the constant NOT_FOUND if the word doesn't exist").
- **Inline comments:** Inline comments should be used sparingly where code complex or unusual enough to warrant such explanation. A good rule of thumb is: explain what the code accomplishes rather than repeat what the code says. If what the code accomplishes is obvious, then don't bother.

```
// inline babbling just repeats what code already says, don't!

int counter;                // declare a counter variable
counter++;                  // increment counter
while (index < length)      // while index less than length
```

- **TODOs:** Remove any `// TODO:` comments from a program before turning it in.
- **Commented-out code:** It is considered bad style to submit a program with large chunks of code "commented out". It's fine to comment out code as you are working on a program, but if the program is done and such code is not needed, just remove it.
- **Doc comments:** You can use "doc" style (`/** ... */`) comment style if you like, but it is not required for this class. A style of commenting using `//` or `/* ... */` is just fine.