EECS 280 C++ Coding Standards

The goal of coding standards is to make code easier to understand and maintain. Refer to these guidelines as you are writing code, not just at the end, to develop good coding habits and to receive a high handgrading score.

Names

1. Use meaningful variable names. If a variable name is not descriptive and its purpose is not clear by convention, choose a different name.
2. Avoid single-letter variable names unless their meaning is very clear by convention, mathematical definition, or documentation.
    a. "i" as a counter variable name is OK.
    b. Especially avoid letters that can be confused for numbers or other letters, such as "l" (that's lowercase L).
3. Longer names are usually better, i.e. `spammify` instead of `sfy`.
    a. Avoid abbreviations unless there is context that makes the abbreviation easy to decipher. For example, "`img`" is a reasonable abbreviation in an image processing program.
4. Avoid acronyms unless there is very strong context or documentation as to their meaning.
    a. Use "`left_bower`" instead of "`lb`", "`rodent_of_unusual_size`" instead of "`rous`."
5. Global constant names should use a consistent naming convention to make it clear that you are dealing with a global constant. Some possible naming conventions are:
    a. All caps: `const int MY_CONSTANT = 42;`
    b. A leading "`c_`": `const int c_my_constant = 42;`
6. It's common to use `snake_case` or `camelCase` for variable and function names, but be consistent.

7. Start custom type names (classes, structs, and type aliases) with a capital letter.

Comments

1. Comments help people understand your code. They are for humans, not for computers.
2. Comments go on their own line(s) unless they are *extremely* short.
3. Don't comment basic stuff such as "`// looping from 0 to size - 1`."
4. If you find yourself wanting to comment a complex or long piece of code, consider putting the code into a function.
5. Use comments to help your future self and other programmers. If you looked at your code a month from now, would you remember what it did?
6. You may use `/* … */` or `//` for comments spanning multiple lines, just be consistent.

Functions

1. Use functions to improve code readability and avoid duplication.
2. Document functions with an "RME" (requires, modifies, effects) to indicate what they do.
3. Make functions short and targeted.
    a. A function has a specific purpose and shouldn't try to be a "god" function doing it all. This also applies to your main() function.
    b. It is perfectly valid and encouraged for functions to call other functions!
4. If a function is complicated, try to make it less complicated by splitting it up.
5. Avoid "single point of return." Prefer multiple return statements.
    a. If you ever reach a point in a function where you know what the return value will be, just return it.

Change:	To:

`int spammify(int egg) {`
`  int result = 0;`
`  if (egg < 42) {`
`  result = egg * 2;`
`  }`
`  else {`
`  result = egg - 1;`
`  }`
`  return result;`
`}`
	`int spammify(int egg) {`
	`  if (egg < 42) {`
	`  return egg * 2`
	`  }`
	`  return egg - 1;`
	`}`

Formatting

1. Curly braces can be on the same line or their own line, just be consistent.

This is fine:	This is also fine:

`if (i > 1) {`
`    ...`
`}`
	`if (i > 1)`
	`{`
	`    ...`
	`}`

2. Don't mix tabs and spaces. Most editors and IDEs allow you to set this ahead of time, so do that and be consistent. You can also configure them to insert the right number of spaces when you press the tab key.
3. Avoid excessively long lines of code. 80 characters is a traditional limit, but some teams will choose to increase this slightly.
    a. The important thing is to avoid cramming a ton of expressions into one line.
4. Include spaces between operators (`<<`, `>`, `<=`, `+`, `-`, `*`, `%`, `||`, `&&`, etc).

This is easier to read:	Bad:

`if (foo >= (bar * 2) && baz) {`
`    cout << a + (b % c);`
`}`
	`if(foo>=(bar*2)&&baz) {`
	`    cout<<a+(b%c);`
	`}`

Note: Some operators (`.`, `->`, `++`, `--`) are traditionally used without spaces:

```
spam.egg
spam_ptr->egg
cheese++
cheese--
```

Other ways to make your code better

1. Avoid using "magic" numbers, strings, etc. Prefer to use a constant instead.
    a. Do NOT use `#define` to define constants. Declare them as `const` variables instead.
    b. Avoid constants where the name is the same as the value:

Change:	To:
`const int TWO = 2;`	`const int num_players = 2;`

2. Avoid using the "`this`" pointer for member access.
    a. Only use the "`this`" pointer when you need a pointer or reference to the object a member function is operating on.
3. If you have an "`if`" statement whose body ends with a return statement, you can omit the "`else` block."

Change:	To:

`if (condition) {`
`  return 42;`
`}`
`else {`
`  return 43;`
`}`
	`if (condition) {`
	`  return 42;`
	`}`
	`return 43;`

4. When checking bools, pointers, and streams in conditionals, prefer checking the "truthiness" of the value.

Change:	To:
`if (my_bool == true) { ... }`	`if (my_bool) { ... }`
`if (my_ptr == nullptr) { ... }`	`if (not my_ptr) { ... }`

`if (my_stream.good()) { ... }`
`if (my_stream.bad()) { ... }`
`if (my_stream.fail()) { ... }`
`if (my_stream.eof()) { ... }`
	`if (my_stream) { ... }`
	`if (not my_stream) { ... }`
	`if (not my_stream) { ... }`
	`if (not my_stream) { ... }`

Note: The keywords "`and`", "`or`", and "`not`" can be used instead of "`&&`", "`||`", and "`!`" in C++. Which ones you use is up to you as long as you do so consistently.

5. If you need to return true or false based on the result of a conditional, just return the result of the conditional instead of explicit true/false

Change:	To:

`if (spam == 42) {`
`  return true;`
`}`
`else {`
`  return false;`
`}`
	`return spam == 42;`

6. Avoid passing large objects/containers by value.
    a. Copying large objects is expensive.
    b. If a function is not supposed to modify the object, pass by const reference!
    c. This includes when using a range-based for loop:
        i. `for (const auto& thing : things) { ... }`
7. Avoid using public member variables.
    a. Breaks abstraction and encapsulation.
    b. If you have plain old data (POD), consider using a struct.
8. Don't explicitly implement the Big 3 if the compiler-supplied versions will work.

Dangerous things that you should never do

1. `goto` - just don't use it. Use functions, loops, conditionals, and exceptions instead.
2. Global variables (global *constants* are different, see bullet 5 under Names section).
    a. Dangerous because they can be changed at any point in your program. Use function parameters instead.
3. Including `.cpp` files - e.g. `#include "file.cpp"`.
4. `using namespace std` inside of a header file.
    a. If someone else includes your header and if you have namespace using statements in it, then these statements become part of their code, appearing at the point your header file was included. They are stuck with whatever namespace using decision you made, and can't override it with their own.

Project-specific ways to make your code better

1. P2 Image: Respect the interfaces. That includes in your unit tests!
2. P3 Euchre:
    a. Don't copy paste calls to deal_one(). Write a function.
    b. Write moar functions.
3. P4 Calculator: It's okay to put your switch/if block in main, but use helper functions for the individual cases (even if the cases are short!).
4. P5 Machine Learning: Write moar functions.

Further reading

EECS 381 - http://umich.edu/~eecs381/handouts/C++_Coding_Standards.pdf
Google - https://google.github.io/styleguide/cppguide.html
Mozilla - https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style
PEP 8, a widely used standard for Python - https://www.python.org/dev/peps/pep-0008/