Good Coding Practices

It is critical in our work to produce clean, dependable products, that are "***easy to use properly, difficult to use improperly***". For standard software products, they can just release a patch. If our code fails, our customer's mission may fail. Here are just a few of the good coding habits you can use to help create and maintain good code.

- **Commit changes early and often** – even for small coding tasks, such as this class, use version control to snapshot changes or additions in case your code takes a wrong path, is damaged, or you just want to see the history. This way, you can freely change files, or even delete them, without worrying that your changes are irreversible.

- **Use Branches**, but ***keep them short-lived and pull in changes from master often*** – even if programming alone, it is sometimes desirable to set one piece of work aside and work on another. For example, you notice a change that would improve the code, that is simple to complete and test and merge, and unrelated to your current work. Don't mix them together, and don't stash your changes in order to work on the new item, just pull a new branch off master, work on it until it is merged, and then pull it into your original branch.

  Note: You can commit frequently and frivolously, and then squash the commits to one or a few before merging.

- **Code Reviews** – Not only does an extra pair of eyes on the code spot more bugs, but it provides two very different perspectives on the code. When we review our own code, *we see what we meant to code*, and are often blinded to errors and typos. When someone else reviews our code, they have to actually read and interpret it to figure out what it does, so *they end up seeing what the code actually does*. Even a novice programmer adds value by doing code reviews. Not to mention that both sides learn from reviews, from how to make code intent clear, to sharing coding knowledge and methods.

- Choose **function and variable names** to make your code readable (***see naming.pdf***)

  - Using comments to document your code requires future readers/modifiers to interpret every line of code and comment; when a function is called, the function name should make its purpose clear, as should the parameter names.

    E.g. while (moreRecordsExist(queryResults)) {/*process a record*/}.

  - Providing a full function comment block (Doxygen format is widely supported) also allows your code to be used without having to read every line of it, by providing pop-up tool tips for functions and variables, and inline assistance entering parameters.

  - No 'magic numbers' - Use of literal numbers, characters, and most strings, is almost always a mistake. The number 42 is meaningless, but ANSWER_TO_LIFE is not. Switch(object->mode) {case 0:, case 1:. …} is code obfuscation, requiring reverse engineering to interpret. It's very simple to make these readable with simple defines, enumerations, or const variables.

  - Single letter names are difficult to search for, and provide no code meaning; don't use them even as a loop index

- Keep *variable scope* to a minimum. Don't declare variables at the top of a function unless they are needed at the top of the function, use inline loop index declaration unless the index is needed after the loop, etc.
- Always use curly braces even if only one statement. It makes the code easier to interpret, and avoids the common mistake, especially with python coders, of seeing indented lines as grouped, when they are not.
- If negative values don't make sense, use unsigned.

- **SOLID code**
  - SRP – Single Responsibility Principle – a function or class should have only one purpose. Don't use a function to "see if your token can move to the next space, and if it cannot, change directions". Use one to determine canMoveForward() and another to turn(). This avoids misinterpretation due to unexpected hidden "features", and makes your code easier to maintain and modify.
  - OCP – Open/Closed Principle – All published interfaces should be closed to modification, but open to extension. There is no sense in requiring a search and replace overhaul to all of your code, or even worse, external users of your code, because you fundamentally changed the nature of a function or property, or because you decided to change the order or number of parameters.

  --- The remainder are for object oriented design, and are not covered in this course ---

  - LSP – Liskov Substitution Principle – Any superclass of a class should be able to be substituted for an object without breaking the code. Subclasses should not return a type different from the superclass version of the function, nor take different parameters, or change the semantics of the function, or introduce new side effects, etc. The classic example is a Bird superclass used for rubber ducks, and the toy manufacturer coming in one morning to find the rubber ducks flying. Sure, that is a silly example, but a superclass that implements fly() requires that all subclasses implement fly() or use the default behavior. In the rubber duck example, it would be better to have a subclass FlyingBirds that implement the Flying interface. (For a full explanation, see https://reflectoring.io/lsp-explained/)
  - ISP – Interface Segregation Principle – Don't expose unnecessary methods or properties, and segregate interfaces into limited, specialized interfaces. Never require client code to take a bulky interface littered with members it won't use. (For a full explanation, see https://reflectoring.io/interface-segregation-principle/)
  - Dependency Inversion (or Injection of Control) – Use an interface or abstract object to define dependent objects, provide a factory to create objects when needed, provide a null/mock instance for testing. For example, your code has a command processing thread that requires a comm object and a logger. Provide those to the thread function as abstractions, then have a factory to create a socket based, or web socket based, or serial, or pipe, etc. comm object that implements the comm interface, and when a new connection comes in, generate a comm object for it, and start a command processor thread passing that and the appropriate logger.

- **DRY** is simply "Don't Repeat Yourself". Code duplication needlessly amplifies the amount of code you have to maintain. If you reuse a largish chunk of code, or even a small piece 3+ times, consider putting that code in a function that can be called from each place.

- **Use Unit Tests** (where possible) – Small scale white-box tests can accomplish what large scale black-box testing can never achieve. Consider a function that has 10 possible paths through it, including edge cases. A quick unit test with 10 members provides 100% coverage. Now imagine a fairly simple app with 10 such functions. A test engineer would be playing a losing game to try to run that app with enough test cases to cause each of those functions to pass through each of their 10 paths, and could never know how much coverage they've achieved. I was once told by a tech lead that if you write code well enough, there are no bugs, so code reviews and unit tests are a waste of time. Hopefully you see the humor in that statement.