CSCI 200 - Fall 2023
Foundational Programming Concepts & Design

Best Practices

Best Practices To Follow

· **Code Style** · **Code Correctness** · **Code Structure** · **Dynamic Memory Management** · **Software Engineering Design Principles** ·

Code Style

The following set of guidelines ensure all code in this class will be written in a similar and consistent manner, allowing any reader to understand the program's intent and contents.

- One clear and consistent coding style used (such as **K&R**, **1TBS**, or **Allman**).
- Course naming scheme is followed for variable, function, class, and other identifiers. See the **course style guide** for more specifics.
- Code is self-documenting. Variables sensibly named, function names descriptive of their purpose.

Code Correctness

The following set of guidelines ensure all programs written in this class behave properly without side effects.

- Code compiles and links without any errors or warnings.
- Program runs without any run time errors. Exceptions are properly caught, user input is validated appropriately, and program exits successfully without error.
- Use `const` wherever possible:

RMP-H

- If you declare a variable and that variable is never modified, that variable should be `const`.
- If your function takes a parameter and does not modify that parameter, that parameter should be `const`.
- If a member function does not modify the callee, that member function should be `const`.
- If you are pointing at a value that does not change, the pointer should point at a constant value (e.g. `const T*`).
- If the pointer itself is never modified, the pointer should be a constant pointer (e.g. `T* const`).
- If the pointer itself is never modified AND the value pointed at does not change, the pointer should be a constant pointer AND the pointer should point at a constant value (e.g. `const T* const`).

Code Structure

The following set of guidelines ensure all programs written in this class are done in an abstracted, modular, extendable, and flexible manner.

- Do not use global variables unless absolutely necessary. Instead, encapsulate them and design your interfaces effectively. If there is no way around using a global variable, be prepared to defend and justify its usage.
- Program flow uses structural blocks (conditionals/loops) effectively, appropriately, and efficiently.
- Keep your headers clean. Put the absolute minimum required in your headers for your interface to be used. Anything that can go in a source file should. Do not `#include` any system headers in your .h files that are not absolutely required in that file specifically. Do not add `using namespace` in headers.
- Use header guards correctly and appropriately.
- Place templated class and function definitions in a `*.hpp` file.
- Place static class and function definitions in abstracted `*.h` and `*.cpp` files.
- Place each class and structure in their own files as appropriate based on their makeup.

Dynamic Memory Management

The following set of guidelines ensure all programs written in this class behave properly without side effects.

- Implement the Big-3 as appropriate.
- Do not leak memory. Every allocation using `new` needs to have a corresponding `delete`.

Software Engineering Design Principles

The following set of guidelines ensure all program components written in this class are done in an abstracted, modular, extendable, and flexible manner.

- Follow and apply the following design principles:
  - **Write Once, Use Many / Write Once, Read Many (WORM) / Don't Repeat Yourself (DRY)**: Use loops, functions, classes, and `const` as appropriate.
  - **Encapsulate what varies**: Use functions and classes as appropriate. Identify the aspects that vary and separate them from what stays the same.
  - **Favor composition over inheritance**.
  - **Program to an interface, not an implementation** & **SOLID Principles**: When using object-oriented inheritance & polymorphism, do the following:
    - No variable should hold a reference to a concrete class.
    - No class should derive from a concrete class.
    - No method should override an implemented method from any of its base classes.
  - Use appropriate inheritance access. Only expose necessary members to derived classes and/or publicly.
  - Use `virtual` and `override` as appropriate. Mark members as `final` wherever possible and/or appropriate on derived classes.

Last Updated: 05/12/23 09:19

Any questions, comments, corrections, or request for use please contact jpaone {at} mines {dot} edu.

[Jump to Top] [Site Map]