Contents

General Coding Standards

Comments

All assignments, labs, or homework should have, at least, the following comment at the top of the main class:

1. Your name
2. The course number and name
3. Brief summary of file or project as applicable

There is no standard in how comments are used. There are a few guidelines though, and using any of these will be accepted for the course (if consistent in your code):

- Use /* */ style for multiline comments. Use // for single line.
- Use // for everything (multiline are self-spaced with multiple //)
- Use /* */ for everything (single line = */* a comment */*)

Ensure all functions & classes contain a summary comment (can be single line). With classes it should go below the class name and with functions directly above the function but avoid obvious comments with identifiers and variables. Only use them to clarify if needed (like multiple declaration on single line):

```
// This function takes in fahrenheit temp as double and returns the temp in celsius
double temp_to_celsius(double fahr) { … }

int course_num, section, degree_num;     // holds basic info on course (good comment)
double GPA;                              // holds the grade point average (no, obvious)
int ranking;                             // ranking within freshman class (may be needed)
```

Attempt to line up comments like above (and all) examples in this document.

General Formatting Structure

Separate different "sections" (blocks) of code with blank lines. For Example:

```
// Infinite loop until all grades added to vector
while (true) {
  int grade {0};
  std::cout << "Please enter your grade 0-5: ";
  std::cin >> grade;

  // if grade in range add to vector otherwise ignore it
  if (grade >= 0 and grade <= 5) {
    grades.push_back(grade);
  }

  // Ask if they have more grades and leave on "n"
  std::string leave {};
  std::cout << "\nDo you have more grades? (y/n) ";
  std::cin >> leave;

  if (leave[0] == 'n' or leave[0] == 'N') {
    break;
  }
}
```

Code Length and Indentation

Always indent the content of blocks (either 1 tab or 2-4 spaces).

Code should have either 80 or 120 character code limit per line (80 is recommended but won't count off until 125). This improves readability, copyability (email someone a 220 character line of code and you'll see), and is easier on the eyes (meaning fatigue)

Classes

General

Classes should follow the principles of SOLID. Starting with each class **only handling a single set of responsibilities.**

Naming

Class names should always Uppercase the first letter of each word. Older libraries (and code) may have class names in ALL CAPS – avoid this today as these can conflict:

```
class Reports          // Single word class – good
class GradeReports     // Fine for both too
```

Enum Classes and their properties are the EXCEPTION, they should be lowercase:

```
enum class directions {north, south, east, west};
```

Encapsulation

Minimize the exposure of members by using information hiding to reduce chances of unintended access. Use private and protected member access where needed – **prefer private**.

Order of membership should decrease: public -> protected -> private

```
class Foo {
  public:
    int bar {0};
  protected:
    std::string convert_bar(int x);
  private:
    // … other data …
}
```

Functions

Any set of operations you complete repeatedly should be considered for making into a function – bundle them so they only handle a single operation. An example of bad then good code follows:

```
// read and print an int – REALLY BAD CODE
// 3 operations (read,write,error handling) & only handles ints
void read_and_print(istream& is) {
  int x;
  if (is >> x)
    cout << "the int is " << x << '\n';
  else
    cerr << "no int on input\n";
}
```

```
// Better would be:
int read_stream(istream& is) {
   int x;
   is >> x;
   return x;
}

void print_stream(ostream& os, int x) {
   os << x << "\n";
}

// Then we combine these when needed
void read_and_print() {
   auto x = read(cin);
   print(cout, x);
}
```

Naming Identifiers and Functions

These standards come from various backgrounds and the only rule is follow the standard of the OS you're coding toward (if coding toward single OS). For here, use **camelCase (lower first letter)** or **snake_case** and ensure the names are descriptive (verb,noun)**:**

```
calculateGrade   // descriptive camelCase so fine
calc_grade       // calc is standard jargon so again fine
// though okay we can do better (calc give what – "A,B,C"; "4.0,3.7,2.2")?
sum_grades          // clear – returns the sum of grades
calc_letter_grade  // Again clearer & not to long
calcLetterGrade    // Or in camelcase
```

Structures (Structs)

Struct naming conventions are the same as functions and standard variables. Avoid making variable declarations with global structs.

```
struct point {
    int x;
    int y;
    int z;
} points;  // Fine in local context – terrible in global (remove points)
```

If you need a non-public member (private or protected) – don't use a struct use a class.

Variables

Using either **camelCase** or **snake_case** is fine. I only require consistency: when you start with one in a program – use that one. Names should be meaningful and descriptive but as short as possible:

```
std::string lastName               // okay
std::string last_name              // still okay
std::string ln                     // not descriptive
std::string last_name_of_client // too long
// (if multiple last names should use a data structure)
```

Exceptions do exist, such as loops, constants, and "known" jargon

```
for (int i=0; i<10; i++) {…}      // fine standards include i,e,j,k,x
float GPA;                        // Standard Jargon so can be all upper and short

#define TAXRATE .839          // const using preprocessing
const float TAXRATE = .839  // const using standard assignment
```

Variable Assignment

Variables should have a consistent assignment (use either {} or = but be consistent).

```
int grade = 9;
int classes {9};  // use either but don't mix and match
```

Class variable naming

Protected and Private variables may start with an underscore. **You may use "_" or nothing, do not use "_CapitalLetter" or "__doubleunderscore": this will make the variables macros.**

```
private:
   int _accessories;     // Fine, must be private or protected
   int accessories;      // also fine but this could be seen as public by dev
   int __accessories;  // might be reserved word or makes it a macro – not good
   int _Accessories;   // Same thing here – not good
```