Object-Oriented Programming

Rainfall Coding Practices

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Purpose

- The following is a list of coding practices
- We will demonstrate the violation using the starting Rainfall exercise code
- For each violation, we will show the proper form by making changes to the Rainfall code
- Each fixed violation is a separate `git` commit. The *commit message* will describe the purpose of the code change
- The git commit messages have a particular format and wording style. Use the messages *exactly* as I show in class.

Guidelines Followed in Starting Code

```cpp
#include <iostream>
#include <iomanip>
#include <vector>

using namespace std;

int main() {
····vector<float> rainfall;
····float t, m, n;
····while (cin >> n) {
········rainfall.push_back(n);
····}
····if (!rainfall.size()) {
········cout << "Error: no rainfall data";
········return 1;
····}
····t = rainfall[0];
····m = rainfall[0];
····for (vector<float>::size_type i = 1; i < rainfall.size(); ++i) {
········t += rainfall[i];
········if (rainfall[i] > m)
············m = rainfall[i];
····}
····cout << "| Hourly Rainfall | Inches in 100s |" << '\n';
····cout << "|:---------------|---------------:|" << '\n';
····cout << "| Average ·····| ····" << left << setw(10) << fixed << setprecision(2) << (t / rainfall.size()) << " |" << '\n';
····cout << "| Heaviest ····| ····" << left << setw(10) << m << " |" << '\n';
}

```

- Key: '·' is a single space
- Program builds (compiles and links) and runs correctly
- Indentation consistently follows the *flow of control*
- Indentation is all spaces and not mixed spaces/tabs
- File ends in a newline

Header Comment

Every program file needs a header comment

- Minimally includes the name of the file
- Minimally describes what the program does (for a program with a `main()`) or what the file is for
- *May* include a *license*
- Do not include your name (what version control is for)
- Do not include a list of dates/changes (what version control is for)
- May include a *copyright*

Side Note: License

```
/**
 * @file CPUCount.hpp
 *
 * @copyright Copyright (C) 2014-2019 srcML, LLC. (www.srcML.org)
 *
 * This file is part of the srcml command-line client.
 *
 * The srcML Toolkit is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * The srcML Toolkit is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with the srcml command-line client; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
 */
```

- Software licenses are not used for most classroom assignments

- However, outside of classwork, all files should contain a license

- License may be free and open, e.g., GPL2, GPL3, MIT License, Popular Licenses, or non-free, e.g., a commercial license

Side Note: SPDX Licenses

```
// SPDX-License-Identifier: GPL-3.0-only
/**
 * @file CPUCount.hpp
 *
 * @copyright Copyright (C) 2021 srcML, LLC. (www.srcML.org)
 *
 * This file is part of the srcml command-line client.
 */
```

- The older style was to include the license in the *Header Comment*
- The current style includes the license in a machine-readable SPDX comment *before* the header comment.

Order of Include Files

Include files should be at the top of the file before any other statements

- An include file *should be* able to be at any position of the list and *cannot depend on previous includes*

- The order should be logical

- Lots of ways to organize, most general first, etc.

- Whatever the organization, order them *logically* not *historically*

Final `return` In `main()`

`main()` requires a `return` for all code paths

- The declaration of `main()` has a return type of `int`
- No final `return` leads to an undefined return value when the program runs
- A return of `0` indicates that the program was successful
- A non-zero return indicates that the program had a runtime error
- Can use different positive values for different runtime errors

`using namespace std;`

Do not use `using namespace std;`

- Causes significant problems if used in an include file (*.hpp)
- Code is *read* much more often than code is *written*
- Does not save time and causes potential confusion
- Loses the grouping that the namespace prefix provides
- To use correctly, must declare inside the `main()` function

Error and Log Messages

Write error and log messages to standard error

- `std::cout` is for the *standard output* stream

- `std::cerr` is for the *standard error* stream

- Write the typical output of a program to *standard output*

- Write error and log output to *standard error*

Side Note: Streams

C++ Stream	Standard Stream	Standard File Descriptors	Buffered
std::cin	standard input	stdin	Yes
std::cout	standard output	stdout	Yes
std::cerr	standard error	stderr	No
std::clog	standard error	stderr	Yes

End Output

End final output of the program with a newline

- Easy to see with a shell
- Integrates better with other tools

Sections of Statements

Separate each section of statements with a single, blank line

- A program does a wide variety of things

- Each thing it does often requires multiple lines of code

- Separate the sections of code with a *single*, blank line

Comments for Sections of Code

Every section of code needs a comment describing the purpose

- Can be seen as *pseudocode*

- Comment goes before the section of code it describes

- Allows a developer to find the code of current interest more quickly

- Adding the comments before the code is written serves to design the overall algorithm

Declare Variables Individually

Declare only one variable in a declaration statement

- Just because variables have the same type, it does not mean that they should be declared together

- Types (and positions) of where variables are declared can change as the program progresses

- A single declaration allows room for comments

- With pointers, references, and modifiers, easy to make mistakes, e.g.,

Standard Number Type: `int`

Prefer `int` for the integer variable type

- `int` is the type of a literal integer, e.g., `123`

- Use `int` as the standard C++ integer number type. Only use `long`, e.g., `123l`, `unsigned int`, e.g., `123u`, and `unsigned long long`, e.g., `123ull`, etc., if specifically needed.

- Possible exceptions: `size_t` and `std::size_type`

Standard Number Type: `double`

Prefer `double` for the floating-point variable type

- `double` is the type of a literal floating-point value, e.g., `1.23`

- `double` is the type for all of the standard math functions, e.g., `log()`

- Use `double` as the standard C++ floating-point number type. Only use `float` if specifically needed, e.g., SSE instructions

Declarations of Local Variables

Initialize local variables where they are declared

- The purpose of temporary variables is to store a result for further use in the program

- Declaring them right where they are needed puts them in the correct section

- The "declare all variables first" is an old style from the C-language

- Always initialize temporary variables with a proper value

Variable Names

All variable names should be descriptive of what they store

- Naming is one of the hardest problems in any software design

- Single-variable names for general purposes lose the documentation aspect of a name

- We will discuss naming in more detail later on

`auto` Keyword

Use `auto` when initializing a declaration that includes a non-literal expression

- Adds flexibility if types change

- Exception: Use an explicit type instead of `auto` when initializing with a literal value to make certain you get the intended type

  - `int total = 0;`

  - `double x = 0;`

Cohesive Sections

Put the calculations of each result in a separate section

- If calculating more than one result at a time, put in separate sections of code

- Concentrate on functionality first to organize the program, then on statements needed

- Students often overestimate the computation cost of the loop structure

Cohesive Sections: Heaviest Slice

```
// calculate the average and heaviest rainfall
// auto total = rainfall[0];
auto heaviest = rainfall[0];
for (std::vector<double>::size_type i = 1; i <
    // total += rainfall[i];
    if (rainfall[i] > heaviest)
        heaviest = rainfall[i];
}

// output the rainfall report
std::cout << "| Hourly Rainfall | Inches in 100s |" <<
std::cout << "|:----------------|----------------|" <<
// std::cout << "| Average         |     " << std::left
std::cout << "| Heaviest        |     " << std::left
```

```
// calculate the heaviest rainfall
auto heaviest = rainfall[0];
for (std::vector<double>::size_type i = 1; i <
    if (rainfall[i] > heaviest)
        heaviest = rainfall[i];
}

// // calculate the average rainfall
// auto total = rainfall[0];
// for (std::vector<double>::size_type i = 1; i <
//     total += rainfall[i];
// }

// output the rainfall report
std::cout << "| Hourly Rainfall | Inches in 100s |" <<
std::cout << "|:----------------|----------------|" <<
// std::cout << "| Average         |     " << std::left
std::cout << "| Heaviest        |     " << std::left
```

Cohesive Sections: Average Slice

```
// calculate the average and heaviest rainfall
auto total = rainfall[0];
// auto heaviest = rainfall[0];
for (std::vector<double>::size_type i = 1; i <
    total += rainfall[i];
    // if (rainfall[i] > heaviest)
    //     heaviest = rainfall[i];
}

// output the rainfall report
std::cout << "| Hourly Rainfall | Inches in 100s |" <<
std::cout << "|:----------------|----------------|" <<
std::cout << "| Average         |     " << std::left
// std::cout << "| Heaviest        |     " << std::left
```

```
// // calculate the heaviest rainfall
// auto heaviest = rainfall[0];
// for (std::vector<double>::size_type i = 1; i <
//     if (rainfall[i] > heaviest)
//         heaviest = rainfall[i];
// }

// calculate the average rainfall
auto total = rainfall[0];
for (std::vector<double>::size_type i = 1; i <
    total += rainfall[i];
}

// output the rainfall report
std::cout << "| Hourly Rainfall | Inches in 100s |" <<
std::cout << "|:----------------|----------------|" <<
std::cout << "| Average         |     " << std::left
// std::cout << "| Heaviest        |     " << std::left
```

Expressions in Output Statements

Use only simple expressions in output statements

- Separate *output* from *calculation*

- Simple expression: Single variable or literal

- Simple expression: Single call

Loop Traversal

Prefer range-based `for` over indexing

- The `for` statement with indexing can lead to an infinite loop

- Range-based for is safer and does not lead to infinite loops with standard containers

- Extension of using iterators

Semantics of the `for` Statement

```
int total = 0;
for (int i = 0; i < v.size(); i++) {
    total += v[i];
}
```

```
int total = 0;
for (int i = 0; i < v.size(); ++i) {
    total += v[i];
}
```

```
for (INIT; CONDITION; INCR) {
    STATEMENT;
}
```

```
int total = 0;
{
    int i = 0;
    while (i < v.size()) {
        total += v[i];

        i++;
    }
}
```

```
int total = 0;
{
    int i = 0;
    while (i < v.size()) {
        total += v[i];

        ++i;
    }
}
```

```
// EXCEPTION: continue statement
{
    INIT;
    while (CONDITION) {
        STATEMENT;

        INCR;
    }
}
```

`const` Variables

Use the `const` specifier on a type whenever possible

- Always use when a variable is initialized in the declaration with its final value
- You might discover that a variable you thought was unchanged later in the program actually is changed

C++ Standard Library

Prefer `std::istream_iterator` over an input loop

- Look for existing algorithms in the standard library of C++ instead of writing your own loop
- Use in declarations often allows the use of `const`
- Just one of many examples
- Use of other libraries, e.g., boost, may cause problematic dependencies and depends on the project restrictions
- In this class, only use the standard libraries of C++ unless told otherwise

Extract Method

Create new functions for sections of code with loops and multiple calculations

- Official name is the "Extract Method" refactoring

- Creates an *abstraction* by hiding explicit code behind a name and parameters

- Abstractions have a cost, so whether to extract or not is a *design decision*, and is not always a straightforward decision

- E.g., don't extract the `// output the rainfall report` because that code belongs to `main()`

Include Guards

```
/*
    Functions.hpp

    Include file for ...
*/

#ifndef INCLUDED_FUNCTIONS_HPP
#define INCLUDED_FUNCTIONS_HPP

// Insert: Include files necessary for compiling
//         function declarations

// Insert: Function declarations

#endif
```

- All include files need include guards

- Generate the name based on the name of the file: All uppercase, and replace dot '`.`' with an underscore '`_`'

- May include a standard prefix (separated by an underscore). I use the prefix `INCLUDED_` (see [Lakos])

- If there are underscores in the name, use them. Do not add underscores for anything else

Example Include Guards

Filename	Include Guard Name

XMLReader.hpp	`#ifndef INCLUDED_XMLREADER_HPP`
	`#define INCLUDED_XMLREADER_HPP`
	`// ...`
	`#endif`

xmlReader.hpp	`#ifndef INCLUDED_XMLREADER_HPP`
	`#define INCLUDED_XMLREADER_HPP`
	`// ...`
	`#endif`

xml_reader.hpp	`#ifndef INCLUDED_XML_READER_HPP`
	`#define INCLUDED_XML_READER_HPP`
	`// ...`
	`#endif`

Function declarations

```cpp
// Sum of two integers
int add(int n1, int n2);

// Multiply two floating-point numbers
double multiply(double factor1, double factor2);

// Divide the numerator by the denominator
double divide(double numerator, double denominator);

// Determine if a character is a vowel
bool isVowel(char c);

// Reverse a string inplace
void reverse(std::string& s);

// Find the maximum value
int max(const std::vector<int>& numbers);

// Update the value at a pointer if the pointer is not null
void updateValue(int* valuePtr, int newValue);

// Add a value to each element
void addToElements(std::vector<int>& vec, int valueToAdd);

// Concatenate two strings
std::string concatenate(std::string_view s1, std::string_view s2);

// Determine if two strings are equal
bool equal(std::string_view s1, std::string_view s2);

// Print a message to the console a specific number of times
void printMessageNTimes(std::string_view message, int nTimes);

// Area of a rectangle
double rectangleArea(double length, double width);

// Determine if a number is even
bool isEven(int number);

// Swap the values of two integers
void swap(int& a, int& b);
```

- Always include a comment on the line before
- Functions are typically actions or the result of a calculation, so name them accordingly
- Remember the proper types for IN, OUT, and IN/OUT
- Name all parameters for any multiple-parameter function
- Name the parameter for a single-parameter function if you can find a good name

C++ Standard Library

When performing a standard operation on a container, look for existing solutions

- The C++ Standard Library is full of algorithms to work on the standard containers

- Look for those algorithms instead of creating your own

- Replacing your custom code with a general solution is a win

- Good developers are happy when they can delete code

Cost of Abstraction

If a function is not doing very much, consider replacing the call with the calculation

- Official name is the "Inline Method" refactoring

- By "inline," we do **not** mean the `inline` specifier, which often has no effect at all

- Only do this with straightforward calculations. Would only rarely do this for a calculation that involves a loop.

- Design is *iterative*, so as the program is further developed, it is not unusual to undo previous design decisions