Appendix $\quad

Page 336 $\quad $\quad $\quad $\quad $\quad | $\quad $\quad $\quad $\quad $\quad 4.6 Standard Template Library $\quad $\quad $\quad $\quad $\quad | $\quad $\quad $\quad $\quad $\quad Appendix $\quad $\quad $\quad $\quad $\quad | $\quad $\quad $\quad $\quad $\quad C++ Object Oriented Programming

Appendix $\quad

A. Elements of Style

While the ultimate test of a program is how well it performs for the user, the value of the program is greatly limited if it is difficult to understand or update. For this reason, it is very important for programmers to write their code in the most clear and understandable way possible. We call this “programming style.”

Elements of Style

Perhaps the easiest way to explain coding style is this: **give the bugs no place to hide**. When our variable names are clearly and precisely named, we are leaving little room for confusion or misinterpretation. When things are always used the same way, then readers of the code have less difficulty understanding what they mean.

There are four components to our style guidelines: variable and function names, spacing, function and program headers, general comments, and other standards.

Variable and Function Names

The definitions of terms and acronyms of a software program typically consist of variable declarations. While variables are declared in more than one location, the format should be the same. Using descriptive identifiers reduces or eliminates the need for explanatory comments. For example: `sum` tells us we are adding something; `sumOfSquares` tells us specifically what we are adding. Use of descriptive identifiers also reduces the need for comments in the body of the source code. For example: `sum += x * x;` requires explanation. On the other hand, `sumOfSquares += userInput * userInput;` not only tells us where the item we are squaring came from, but also that we are creating a sum of the squares of those items. If identifiers are chosen carefully, it is possible to write understandable code with very few, if any, comments. The following are the University conventions for variable and function names:

Identifier	Example	Explanation
Variable	`sumOfSquares`	Variables are nouns so it follows that variable names should be nouns also. All the words are TitleCased except the first word. We call this style camelCase.
Function	`displayDate()`	Functions are verbs so it follows that function names should also be verbs. Like variables, we camelCase funcitons.
Constant	`PI`	Constants, include `#defines` are ALL_CAPS with underscore between the words.
Data types	`Date`	Classes, enumeration types, type-defs, and structures are TitleCased with the first letter of each word capitalized. These are OS 165 constructs.

Rule	Example	Explanation
Operators	`tempC = 5.0 / 9.5 (tempF – 32.0)`	There needs to be one space between all operators, including arithmetic (+ and %), assignment (= and +=) and comparison (>= and !=).

Indention	`{`
	`  int answer = 42;`
	`  if (answer > 100)`
	`  cout << "Wrong answer!";`
	`}`
		With every level of indention, add three white spaces. Do not use the tab character to indent.

Functions		Put one blank line between functions. More than one results in unnecessary scrolling, less feels cramped

Related code	`// get the data`
	`float income;`
	`cout << "Enter income: ";`
	`cin  >> income;`
		Much like an essay is sub-divided into paragraphs, a function can be sub-divided into related statements. Each statement should have a blank line separating them.

Function and Program Headers

It takes quite a bit of work to figure out what a program or function is trying to do when all the reader has is the source code. We can simplify this process immensely by providing brief summaries. The two most common places to do this are in function and program headers.

A function header appears immediately before every function. The purpose is to describe what the program does, any assumptions made about the input parameters, and describe the output. Ideally, a programmer should need no more information than is provided in the header before using a function. An example of a function header is the following:

```
/*************************************************
 * GET YEAR
 * Prompt the user for the current year. Error checking
 * will be performed to ensure the year is valid
 *    INPUT:  None (provided by the user)
 *    OUTPUT: year
 *************************************************/
```

Program Header Example

A program header appears at the beginning of every file. This identifies what the program does, who wrote it, and a brief description of what it was written for. Our submission program reads this program header to determine how it is to be turned in. For this reason, it is important to start every program with the template provided at `/home/cs165/template.cpp`. The header for Assignment 1.0 is:

```
/*************************************************************************
 * Program:
 *    Assignment 10, Hello World
 *    Brother Helfrich, CS165
 * Author:
```

General Comments

We put comments in our code for several reasons:

- To describe what the next few lines of code do
- To explain to the reader of the code why code was written a certain way
- To write a note to ourselves reminding us of things that still need to be done
- To make the code easier to read and understand

Since a comment can be easily read by a programmer and source code, in many cases, must be decoded, one purpose of comments is to clarify complicated code. Comments can be used to convey information to those who will maintain the code. For example, a comment might provide warning that a certain value cannot be changed without impacting other portions of the program. Comments can provide documentation of the logic used in a program. Above all else, comments should add value to the code and should not simply restate what is obvious from the source code. The following are meaningless comments and add no value to the source code:

```
int i; // declare i to be an integer
i = 2; // set i to 2
```

On the other hand, the following comments add value:

```
int i; // indexing variable for loops
i = 2; // skip cases 0 and 1 in the loop since they were processed earlier
```

With few exceptions, we use line comments (`//`) rather than block comments (`/* ... */`) inside functions. Please add just enough comments to make your code more readable, but not so many that it is overly verbose. There is no hard-and-fast rule here.

"Commenting out" portions of the source code can be an effective debugging technique. However, these sections can be confusing to those who read the source code. The final version of the program should not contain segments of code that have been commented out.

Other Standards

Because of the way printers and video displays handle text, readability is improved by keeping each line of code less than 80 characters long.

Subroutines and classes should be ordered in a program such that they are easy to locate by the reader of the source code. For larger programs, an alphabetical arrangement works well. For shorter programs, following the order in which the subroutines are invoked or objects are instantiated is useful.

Each curly brace should be on its own line; this makes them easier to match up.

Just as one would expect correct spelling in a technical paper, one should also find correct spelling within a program

Style Checklist

Comments

- program introductory comment block
- identify program
- identify instructor and class
- identify author
- brief explanation of the program
- brief explanation of each class
- brief explanation of each subroutine

Variable declarations

- declared on separate lines
- comments (if necessary)

Identifiers

- descriptive
- correct use of case
- correct use of underscores

White space

- white space around operators
- white space between subroutines
- white space after key words
- each curly brace on its own line

Indentation

- statements consistently indented
- block of code within another block of code further indented

General

- code appropriately commented
- each line less than 80 characters long
- correct spelling
- no unused (e.g. commented out) code

Example 1: Method

The following is an example of a class method with excellent style.

```
/*************************************************************************
 * Extraction    cin >> x;
 * RETURN:    istream by reference  (so we can say (cin >> x) >> y;)
 * PARAMETER: istream by reference  (we do not want a copy of cin)
 *            by reference          (no copies but we do want to change this)
 *************************************************************************/
istream & operator >> (istream & in, Card & card)
{
   // input comes in the form of a string
   string input;
   in >> input;

   // do the actual work
   card.parse(input);
   assert(card.validate());

   // return the input stream
   return in;
}
```

Page 340 $\quad $\quad $\quad $\quad $\quad | $\quad $\quad $\quad $\quad $\quad Appendix $\quad $\quad

Example 2: Header

The following example is a header file with excellent style.

```
/***********************************************************************
 * This file describes the WRITE interface, a class
 *      designed to write data to a file
 ***********************************************************************/
#ifndef WRITE_H
#define WRITE_H

#include <fstream>        // necessary for the ofstream object
#include <string>         // necessary for the string parameters

/*****************************************************
 * WRITE
 * This class will write text to a file without having
 * the client have to worry about opening or closing it
 *****************************************************/
class Write
{
public:
   Write() : isOpen(false)      {       }
   Write(const std::string & fileName);
  ~Write();

   void writeToFile(const std::string & text);

private:
   bool          isOpen;           // did we successfully open the file?
   std::ofstream fout;             // the file stream object
};
#endif // WRITE_H
```

Example 3: Makefile

The following example is a makefile with excellent style.

```
################################################################
# Program:
#     Example 1.5, Complex Numbers demo
#     Brother Helfrich, CS165
# Author:
#     Br. Helfrich
# Summary:
#     This program will demonstrate how to break a large program into several files
################################################################

################################################################
# The main rule
################################################################
a.out: complexTest.o complex.o
   g++ -o a.out complexTest.o complex.o
   tar -cf example14.tar *.h *.cpp makefile

################################################################
# The individual components
#      complex.o      : Compile only if complex.cpp or complex.h changed
#      complexTest.o  : Compile only if complextTest.cpp or complex.h changed
################################################################
```