Jeffrey's Programming Rants, Tips, and Textbook Corrections*

Jeffrey D. Oldham

1999 Dec 17

Here are some of my pet programming peeves, tips how to avoid them, and a link to errata for the textbook. Please avoid the pet peeves, follow the tips, and download the errata now.

1 Textbook Resources and Errata

WWW material for the textbook includes code from the text, sample course materials, and text corrections. To determine your textbook's printing, look at the last line on the back of the title page. Your printing is the largest single digit number not in the list. For example, my textbook has "7 8 9 10" so I have the sixth printing.

2 Programming Peeves and Tips to Avoid Them

2.1 Do Not Depend on ANSI Character Codes or No Hard-Coded Constants in Code

As written in Appendix A of the textbook, one's code should not rely on the presence of the ASCII character encoding. Another encoding called Unicode to support many different languages is growing in popularity. Use code like `if (c == 'A')`, not `if (c == 97)`.

2.2 Ask the Compiler for Warnings

When compiling code, ask for as many warnings as it can generate. Think of the compiler as a friend who can find some programming mistakes. Compile using `g++ -Wall -W -Wstrict-prototypes -Wpointer-arith -Wcast-align -Wconversion -Wnested-externs -Wundef -Winline`. See Appendix D for some examples of using `g++ -Wall`.

2.3 Use String Objects Whenever Possible, But, If You Must, Use `setw`

Use string objects whenever possible; everything you can do with C-style character array strings, you can do with string objects. Thus, the string functions listed in Appendix G are deprecated. If you absolutely must use C-style strings and do input, use the `setw` manipulator to ensure the character array is not overflowed. For example, the code at the bottom of p. 729 should be `cin >> setw(100) >> message`.

2.4 Do Input in While Loop Conditionals, or Avoid Using `eof`

Do all input in the conditional portion of a while loop. For example, `while (cin >> x >> y >> z)`. This does the input, entering the loop only if the input is successful.

Using the `eof` istream member function is problematic for two reasons:

- `eof` yields true only after attempting to read past the end of input and failing. It yields false even if all the input has been read but no attempt has been made to read more data (which does not exist.)
- All input needs to be checked for success before use. Many programmers forget to do this if it is not in the while loop's condition.

For an example violating this, see Appendix F, p. 731. Notice that, if `cin >> next` fails, an undefined value is added to the sum. This is definitely not desired.

Here is a corrected while loop:

```
while (cin >> next) {
```

```
  // We successfully read the integer.
  sum += next;
}
// We have 1) finished reading the file xor
//    2) found a line not containing an integer.
// We will assume finished reading the file.
cout << "The total of all numbers is " << sum << ".\n";
```

If there are multiple items on each line, use one variable per item. For example,

```
int first;
int second;
double third;
while (cin >> first >> second >> third) {
  // We reach here only if successfully read the line.
  // ... put more code here ...
}
// We have 1) finished reading the file xor
//    2) found a line not containing an integer.
// We will assume finished reading the file.
```

3 Where the Textbook is Wrong

`asserts` and Opening Files Do Not Mix Despite the code in Appendix F, p. 732, I do not recommend using `assert` statements to test whether opening a file stream was successful. `assert` statements should be used to check invariants such as preconditions and postconditions when developing code, but the code's correctness should not rely on their presence. In other words, `assert` statements should check for programmer mistakes, not user or input mistakes. For example, check for successfully opening a file stream using `if (outfile)` or `if (!outfile.fail())`.

4 Conditional Compilation and Debugging

Yet another part of C++ for which we do not have time to discuss in class is conditional compilation. Using *preprocessor symbols*, we can selectively omit or include pieces of code. This is particularly useful when we would like to have some code run only while debugging.

The debug.cc program illustrates conditional compilation. We surround the debugging code with lines

```
#if defined(DEBUG)
```

and

```
#endif
```

These debugging lines will be included in the program if and only if the compiler knows that the `DEBUG` identifier is defined.

To tell the compiler that we want `DEBUG` defined, use a compile line like

```
g++ -DDEBUG debug.cc
```

`-D` precedes whatever term we want defined when compiling.