Cite as: McMaster, K., Sambasivam, S., & Wolthuis, S. (2014). Software development using C++: Beauty and the beast. *Issues in Informing Science and Information Technology, 11,* 73-84. Retrieved from http://iisit.org/Vol11/IISITv11p073-084McMaster0442.pdf

Software Development Using C++: Beauty-and-the-Beast

Kirby McMaster
Lake Forest College, Lake Forest, IL, USA
kmcmaster@weber.edu

Samuel Sambasivam
Azusa Pacific University, Azusa, CA, USA
ssambasivam@apu.edu

Stuart Wolthuis
BYU – Hawaii, Laie, HI, USA
stuart.wolthuis@byuh.edu

Abstract

Good programming style plays an important role in producing better software. Good style makes source code easy to read and to understand. This will usually reduce programming errors and simplify maintenance. We discuss popular style practices in C++ software development. Then we present a software program we have developed called UglyCode. UglyCode can be used by instructors to demonstrate the effect of various programming style options on code readability. This software converts "beautiful" C++ source code into "beastly" versions that exaggerate bad programming style. Specific examples to illustrate the use of UglyCode are shown. With UglyCode, programming style effects can be viewed interactively by showing the results when style features are changed in existing code.

Keywords: Programming style, layout, ugly code, algorithm, C++.

Introduction

Teaching Computer Science students how to become competent programmers must go beyond explaining the syntax of a programming language. In programming courses, the early focus is on teaching a computer how to solve a problem (Shustek, 2008). This includes a description of the main features of a higher-level language (e.g., C++, Java, or Python), along with ways to organize language statements into modules and working programs.

As students gain programming knowledge, they are introduced to design and implementation of algorithms (Dijkstra, 1971). Characteristics of algorithms that receive sustained emphasis are *correctness*, *performance*, and *efficiency*. Eventually, to become professional programmers, students must learn to develop systems that satisfy additional objectives, such as maintainability, usability, reliability, and security. These properties have become increasingly important as systems have become larger, more complex, and interconnected.

Programming Objectives

With regard to *correctness*, programmers always desire to write software that is error-free. Computing professionals with strong mathematics backgrounds tend to focus on *logical* correctness, as determined by *proofs*. A software development group in Australia (Klein et al., 2009) recently announced that they have finally proven their micro-kernel operating system to be correct.

Textbooks on algorithms provide proofs for many common algorithms (Cormen, Leiserson, Rivest, & Stein, 2009; Sedgewick & Wayne, 2011). However, proving an algorithm to be correct does not guarantee that the source code will be without errors. Proofs also do not ensure that the program meets customer requirements.

Another way to verify program correctness is to perform thorough testing of the software while it is being developed. A well-designed *test plan* consists of a broad range of tests, both for individual parts of the system and for the system as a whole.

Programmers also want to write programs that *perform* rapidly and make *efficient* use of computer resources. There are almost always trade-offs between performance and efficiency. In a system where multiple processes run concurrently, the primary responsibility for managing these tradeoffs is delegated to the operating system (Silberschatz, Galvin, & Gagne, 2012). Beyond the operating system, programmers can improve performance and efficiency through their choice of *algorithms* and *data structures*.

Programmers are introduced to the benefits of *modular* code in early programming courses (Liang, 2013: Stroustrup, 2013) and data structures courses (Dale, 2011; Drozdek, 2012). The initial modules are functions and procedures. In object-oriented programming, the design and use of classes, objects, and encapsulation is a necessary way to manage complexity in larger programs.

Programming courses spend little time directly on *maintainability*. Most software is not maintained by the original developer. *Readability* is essential for continual code maintenance. The Department of Defense estimates that 60-80% of software life cycle costs are for maintenance.

Modular code improves maintenance efforts, but other programming practices, such as agile development and configuration management, can also make code easier to correct and modify. Detailed discussions appear in Software Engineering textbooks (Pressman, 2009; Somerville, 2011).

Programming Style

In *The Practice of Programming*, Kernighan and Pike (1999) ask why we should "bother" with programming style. "Why worry about style? Who cares what a program looks like if it works? Doesn't it take too much time to make it look pretty? Aren't the rules arbitrary anyway?"

Programming style involves ways that a programmer can organize and present code to make it more understandable to other programmers. By making code easier to understand, style improvements can contribute to other desirable program characteristics. For example, readable code is more likely to be correct when initially written. It is easier to modify when changes are required. Programming style can also assist software testing to verify program correctness.

Customers want programs that are correct, perform well, make efficient use of resources, and meet requirements. Good programming style helps programmers provide these properties in the software they develop. We teach programming style because it helps students acquire the ability to write professional quality code.

The remainder of this paper covers programming style concepts, our UglyCode software, and C++ code examples. In the next section, we describe style concepts that involve source code *layout* and *content*. In the third section, we introduce our UglyCode software, which can be used by

instructors to display the effect of different style choices on code readability. The fourth section presents C++ style examples that demonstrate the use of UglyCode.

C++ Programming Style

The purpose of programming style is to help programmers interpret what the code of other programmers (and their own code) is actually doing. But which programming style is best? Expert programmers have their own preferred styles for writing code. One common point of consensus is that style choices should be used consistently.

In *The Elements of Programming Style,* Kernighan and Plauger (1978) describe a number of style choices for programmers. We discuss a partial list of their style topics, organized into layout and content groups.

Program Layout

Program layout consists of techniques to rearrange source code to make it more readable. No content is added to the code--just changes in spacing.

Blank lines: Blank lines can be added to source code between functions and to group lines of code together that perform some computing activity. This makes the organization of the code easier to discern.

Indenting: Another way to visually present lines of code that "belong together" is to use the same level of indentation for the lines. Because source code can have nested blocks, more than one level of indentation can be helpful. Indenting is also used to indicate that a statement wraps over more than one line.

One question that always generates a variety of responses is "how many spaces should I indent?" Each programmer will have a preferred answer, and many software development environments provide explicit standards.

Block layout: A block of code is a sequence of statements having the behavior that either all statements are executed, or none are. In a conditional statement, the block is executed only when the condition is true. In iterative statements, the block will be executed repeatedly until the continuation status changes.

An important part of block layout is placing marks in the code where each block starts and ends. Formatting conventions for blocks depend on the programming language. In a language with *fully-bracketed syntax* (e.g., if-endif), the statements include markers for the start and end of blocks. Some recent languages such as C, C++, and Java use curly braces (e.g., "{ ... }") to mark blocks (Kernighan & Ritchie, 1988). For these languages, there are differences of opinion on where to place the curly braces.

Statement length: Early fixed-format higher level languages such as FORTRAN and COBOL were designed with punched cards in mind, having a maximum of 80 characters per card. In these languages, a statement will continue across more than one card only if marked in a special way. Most recent languages are free-format, in that a statement is assumed to continue across multiple lines until a termination character (e.g., ";") appears. The programmer has the choice of how wide each line of code should be. Multiple lines, with carefully selected break points, can be used for longer statements.

Providing Content

A programmer can also improve the readability of code by adding information within the statements. Common ways to provide this information are through the choice of names for variables and the inclusion of comments at appropriate locations in the code.

Variable names: The value of a variable changes during the execution of a program. To allow references to the current value, a variable must be given a name. Ideally, the name will describe the attribute represented by the variable. Very short names and heavily abbreviated names can be cryptic to readers.

Comments: Comments can be placed in source code for most programming languages. Some special marking is usually required (e.g., "//") to indicate that the comment will not be executed. McConnell (2004) recommends that comments be included only if they (1) describe the code's intent, (2) provide information not in the code, or (3) summarize a section of code. C++ allows several types of comments: full-line comments, end-line comments, and multiple-line comments.

UglyCode Software

We have written a program called UglyCode to assist instructors in teaching programming style. The UglyCode software teaches programming style concepts in *reverse*. The usual *forward* teaching approach displayed in textbooks shows examples of "ugly" (*beast*) code, before applying desirable style principles to generate "pretty" (*beauty*) code.

For UglyCode, the input is a C++ program that has been written to illustrate good programming style. Using UglyCode, options can be exercised on how to "degrade" the code. Students can see how much harder it is to understand source code when good style features are removed. Instructors can demonstrate programming style concepts using both "forward" (textbook) and "reverse" (UglyCode) examples.

Our explanation of how to use the UglyCode software is organized according to the user-interface controls that appear on the main screen. The controls include a File menu, six sets of checkboxes to select code style changes, and two command buttons to transform and restore the original code.

File Menu

This is the only menu choice on the UglyCode screen. It includes the following submenu options.

1. *Open*: Open an existing source code file, using a "file-chooser" input control. UglyCode is designed for C++ (and Java) programs.

2. *Save As*: After style selections are made and viewed, the resulting "ugly" version of the original program can be saved as a text file. To avoid overwriting the input file, the name for the saved file should differ from the input file name.

3. *Exit*: This option ends the UglyCode program.

Command Buttons

Two buttons on the lower right-hand corner of the screen are used to invoke actions on the source code.

Ugly It!: After one or more style options have been selected using the checkboxes, this button should be clicked to activate the changes on the original source code. The restyled "ugly" code then appears in the main window.

Reset Text: Clicking this button will restore the code in the window to its original form.

Checkboxes

Checkboxes are organized into groups, based on style category. Within each group, the checkboxes act like command buttons, in that at most one box can be checked. The following checkbox groups are listed on the right-hand side of the UglyCode main screen.

Line spacing

Line spacing choices display how the inclusion or exclusion of blank lines in code can affect program readability. Blank lines can make it easier to see which parts of the code belong together. We include three spacing options.

Remove Blank Lines: Selecting this checkbox causes all blank lines to be removed from the code. This choice is equivalent to single-spacing.

Double Space Code: Double-spacing rarely appears in production code, although a few developers embrace it. This style choice is included to contrast with single-spacing. In a work environment, the extra blank lines in code could be used for inserting notes during code reviews. Students often use double-spacing when writing term papers for non-computer courses. We have seen the equivalent of double-spacing in code from students, when they copy their code into Microsoft Word. The current default spacing in Word places 10 points at the end of each line, which students (and lab administrators) often fail to change.

Random Blank Lines: Blank lines do not improve program readability when the lines are inserted in apparently random locations. This option inserts blank lines randomly into the code. After each non-blank line, the probability is 1/3 that the next line will be blank. Combined with random indenting (described below), this option can lead to "beastly" looking code.

Indenting

We provide three options for the number of spaces that occur on the left side of each line of code.

Remove Indents: With this option, all spaces on the left of each line are trimmed off. All code starts at the left-side margin. This makes it hard to identify where branching and looping control structures start and end.

Add Fixed-Size Indents: The "best" size for indenting is an individual preference among programmers. With this option, the instructor can demonstrate the readability of code with various indenting sizes. Choosing zero spaces is equivalent to removing all indents. A pop-up window allows the user to enter the desired number of spaces per indent. Within nested control structures, multiple levels of indenting can accumulate on a single line.

Add Random Indents: In this selection, each line receives a random indent size of 0 to 16 spaces. This is certainly not a recommended way to indent, but a similar result sometimes occurs in student assignments. Suppose that code is written using an editor with a fixed-size indent (say 4), but the programmer mixes spaces with tabs. When the code is later brought into a different editor (e.g., Notepad with tab size 8), the mixture of new tab sizes and old spaces can yield a ragged left margin for the code. Randomly-indented code can be a "beast" to debug.

Curly Braces

A common layout decision is "where to put curly braces to designate the start and end of blocks." The decision can be guided by language traditions, as well as by programmer preferences. C, C++, and Java have separate histories, with different preferred block marking rules. This option allows the instructor to compare the traditional C-style braces with Java-style braces, allowing students to form their own opinions.

Change to Java Style: With this choice, curly braces defining blocks for loops (while, for) and branches (if-else) are formatted to have the opening brace on the *same* line as the decision expression. Curly braces in other parts of the code are not changed.

Change to C Style: With this choice, curly braces are formatted with each opening brace on its own *separate* line. Closing braces in the code are not changed.

Comments

Comments can improve understanding of what the original programmer intended to do, but only if the comments are well-placed and are "helpful". UglyCode options include (1) removing comments and (2) replacing existing comments with "useless" comments. UglyCode acts only on single-line comments that start with "//". Other comment delimiters (e.g., /* and */) are ignored.

Remove Comments: With this option, all comments starting with "//" are removed from the source code. For full-line comments, the entire line is removed. For an end-line comment, the comment is removed, but not the source code before the comment.

Change to Useless Comments: This option replaces all full-line and end-line comments with "useless" comments. For full-line comments, UglyCode chooses randomly from a list of 29 computer-humor statements we found on the web. For example, one of our favorite full-line comments is, "When your hammer is C++, every problem looks like a thumb."

End-line comments are usually shorter, so the program chooses randomly from a list of 11 popular desserts (to sidetrack hungry programmers). For example, one end-line dessert is: "Chocolate Mousse".

For both single-line and end-line comments, the replacement comments are not relevant to the code. Repeating this option will give a different random sample of useless comments.

Variable Names

Many variable naming styles are language specific (e.g., the preference for lower case names in C). Most naming conventions recommend the use of "meaningful names", subject to possible name length restrictions. For example, "computedTax" is self-descriptive, while “CT” could be construed as an eastern US state or a medical diagnostic procedure.

The variable naming options in UglyCode show how atypical naming rules can hinder program understanding.

Change Case: This option demonstrates how case differences can affect readability. For variables with common type declarations (e.g., char, int, long, float, double) that appear at the *start* of a line, the case of various characters in the variable name are changed (from lower-case to upper-case, or vice-versa). For example, a lower-case name such as "job_cost" might be changed to "JoB_CoSt". A name such as "netIncome" could become "NeTINcOmE".

Use Meaningless Names: Whether a variable name is considered "meaningless" depends on the context. There are many ways to create meaningless names. We chose a simple encryption algorithm, a Caesar cipher, because it is easy to program and generates strange looking names. Each case-sensitive letter in a name is changed to the letter K positions later in the alphabet (with wrap-around). The value of shift K varies each time this option is invoked. Non-letters are unchanged. For example, the variable "best2BUY" using shift K = 3 would become "ehvw2EXB", which looks quite meaningless.

Line Breaks

In free-format languages such as C++, the programmer can choose how much of each statement to place on a line. For long statements, line breaks within a statement can improve the readability of the code. For short statements, more than one statement can be included on a single line.

Set Line Length: This option shows what the code will look like if a line length is specified. A pop-up window allows the user to enter a desired *minimum* line length (e.g., 40). The code is then reformatted so that when concatenated statements exceed this length, they are split over additional lines. A line break is placed in the first "safe" position at or beyond the minimum length. "Safe" is defined to be immediately after the first semicolon (";"), left brace ("{"), right brace ("}"), or plus-sign with space ("+ ") at or beyond the minimum length. These break points are usually safe, but in some situations they can lead to compile errors.

Remove Line Breaks: This is the ultimate reformatting of source code. All line breaks are removed and replaced with spaces. The program now consists of a single long line, which is what a compiler sees. Because the UglyCode window shows this line without word-wrap, the window's bottom slider control must be used to view the entire program. The revised one-line program can be saved and then viewed in an editor that provides word-wrap (e.g., Notepad).

Note: After making programming style changes, the resulting C++ program can be saved as a text file. If you are fortunate, the revised program will compile and run. For example, if Remove Blank Lines, Remove Indents, Remove Comments, Change Case (for Variable Names), and Remove Line Breaks are all checked, the reformatted program should execute exactly as before the changes. The source code is *much* more "beastly", but the compiler/computer doesn't care.

Code style changes are not cumulative. Each set of selected changes is applied to the original source code. If an instructor wants to demonstrate the cumulative effects of style changes, she/he should plan a sequence of changes and then mark cumulative sets of checkboxes for Ugly It! button clicks.

C++ Style Examples

Several examples of how style concepts can reduce the clarity of a C++ program are described in this section. Each example uses the sample code for the GCD (greatest common divisor) function listed below.

```
long GCD(long n1, long n2)
{
  // Euclidean algorithm for GCD
  long High, Low, Remainder;

  // Initialize High and Low
  if(n1 > n2)
  {
    High = n1;  Low = n2;
  }
  else
  {
```

```
    High = n2;  Low = n1;
  }

  // Loop until remainder is 0
  Remainder = High % Low;
  while(Remainder > 0)
  {
    High = Low;
    Low = Remainder;
    Remainder = High % Low;
  }
  return Low;
} // end GCD
```

Figure 1: Original C++ Code for GCD Function.

The GCD function calculates the greatest common divisor of two integers n1 and n2. We display UglyCode output to show the *beast* effect of selected programming style choices on this code.

Remove Blank Lines and Comments

The first C++ example demonstrates the combined effect of removing all blank lines and comments. The code with the comments and blank lines removed is shown in Figure 2.

```
long GCD(long n1, long n2)
{
  long High, Low, Remainder;
  if(n1 > n2)
  {
    High = n1;  Low = n2;
  }
  else
  {
    High = n2;  Low = n1;
  }
  Remainder = High % Low;
  while(Remainder > 0)
  {
    High = Low;
    Low = Remainder;
```

```
    Remainder = High % Low;
  }
  return Low;
} // end GCD
```

Figure 2: Ugly C++ Code With Blank Lines and Comments Removed.

The function name (GCD acronym) suggests the purpose of the function. The shorter transformed code for the function is still readable, but more effort is required to understand the structure of the algorithm.

Java Curly Braces with No Indenting

The next C++ example shows how readability suffers when all indenting is removed. The code without indenting but with Java-style braces is presented in Figure 3.

```
long GCD(long n1, long n2) {
// Euclidean algorithm for GCD
long High, Low, Remainder;

// Initialize High and Low
if(n1 > n2) {
High = n1;  Low = n2;
}
else {
High = n2;  Low = n1;
}

// Loop until remainder is 0
Remainder = High % Low;
while(Remainder > 0) {
High = Low;
Low = Remainder;
Remainder = High % Low;
}
return Low;
} // end GCD
```

Figure 3: Ugly C++ Code With Java-style Braces and No Indenting.

The code is reasonably clear, thanks to the comments. The change to Java style braces has little effect on readability, but it does reduce the number of lines. However, the lack of indentation makes it harder to distinguish the boundaries of the blocks for the *if* statement and *while* loop.

Meaningless Names & Useless Comments

The third C++ example demonstrates how poorly chosen variable names and unhelpful comments can affect program readability. The transformed code is shown in Figure 4.

```
long JFG(long q1, long q2)
{
  // Beware of bugs in this code; I have only proved it correct.
  long Kljk, Orz, Uhpdlqghu;

  // You can't make a good program without breaking some egos.
  if(q1 > q2)
  {
    Kljk = q1;  Orz = q2;
  }
  else
  {
    Kljk = q2;  Orz = q1;
  }

  // When your hammer is C++, every problem looks like a thumb.
  Uhpdlqghu = Kljk % Orz;
  while(Uhpdlqghu > 0)
  {
    Kljk = Orz;
    Orz = Uhpdlqghu;
    Uhpdlqghu = Kljk % Orz;
  }
  return Orz;
} // Hot Fudge Sundae
```

Figure 4: Ugly C++ Code With Meaningless Names and Useless Comments.

The UglyCode program scrambles the names of variables (and functions) that are type-declared at the start of a statement. We do not scramble the names of variables declared within statements, such as argument names and counter variables declared inside loops. The scrambled name "Uhpdlqghu" is virtually meaningless.

For comments to be effective, they must provide information that is helpful to programmers who later read the source code. Instead of removing the comments, we replace them with comments that have nothing to do with the code (e.g., "A million monkeys...").

The above programming style examples illustrate the utility of the UglyCode software for interactive classroom use. With six style groups and 2-3 choices per group, there are a total of 14 possible cases involving a single style change. When several style changes are combined, or when the cumulative effects of a sequence of changes are examined, the number of cases increases dramatically. Not all cases will be of equal interest. A variety of sample programs can be prepared to demonstrate specific style combinations to students.

Summary and Conclusions

In this paper, we discussed how programming style can affect source code understanding. We related programming style to program features such as correctness, performance, efficiency, and maintainability. We presented several style options for C++ program layout, such as blank lines, indenting, and block layout. We also described style features such as variable names and comments that add content to improve readability.

We described a program we have written called UglyCode, which allows an instructor to demonstrate how style changes affect the clarity of code. Instead of showing an example of "bad" (*beast*) code and then making it "pretty" (*beauty*), UglyCode works in the opposite direction. Program input should be a C++ program written in "good" style. Style changes are then requested, and the resulting degradation of the code can be viewed immediately.

The UglyCode software allows students to see the effect of individual style changes and groups of changes. Transformed source code can be saved in a text file. Students can attempt to compile and run the modified code. In this way, they can determine whether or not the style changes disrupt the compiler. It is surprising to see how often style changes are ignored by C++ compilers.

Future Research

We have demonstrated prototypes of the UglyCode software in Programming and Software Engineering courses. The data we have collected from students is largely anecdotal. With a completed version of UglyCode now available, we plan to measure how well this tool helps students appreciate the importance of good programming style.

Note: An executable version of the UglyCode program, along with the sample C++ program presented in this paper, can be obtained from the authors.

References

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). *Introduction to algorithms* (3rd ed). MIT Press.

Dale, N. (2011). *C++ plus data structures* (5th ed). Jones & Bartlett.

Dijkstra, E. W. (1971). A short introduction to the art of programming. *E. W. Dijkstra Archive*. Retrieved from http://www.cs.utexas.edu/users/EWD/

Drozdek, A. (2012). *Data structures and algorithms in C++* (4th ed). Cengage Learning.

Kernighan, B. W., & Pike, R. (1999). *The practice of programming*. Addison-Wesley.

Kernighan, B. W., & Plauger, P. J. (1978). *The elements of programming style* (2nd ed). McGraw-Hill.

Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language* (2nd ed). Prentice Hall.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., ... & Winwood, S. (2009, October). Formal verification of an OS kernel. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principle*s (pp. 207-220). ACM.

Liang, Y. D. (2013). *Introduction to programming with C++* (3rd ed). Prentice Hall.

McConnell, S. (2004). *Code complete* (2nd ed). Microsoft Press.

Pressman, R. (2009). *Software engineering: A practitioner's approach* (7th ed). McGraw-Hill.

Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed). Addison-Wesley.

Shustek, L. (2008). Donald Knuth: A life's work interrupted. *Communications of the ACM*, *51*(8), 31-35.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2012). *Operating system concepts* (9th ed). Wiley.

Somerville, I. (2011). *Software engineering* (9th ed). Addison-Wesley.

Stroustrup, B. (2013). *The C++ programming language* (4th ed). Addison-Wesley.

Biographies

Dr. Kirby McMaster recently retired from the Computer Science Department at Weber State University. To remain active, he is currently a visiting professor in Computer Science at Lake Forest College in Illinois. His primary research interests are in database systems, software engineering, and frameworks for Computer Science and Mathematics.

Dr. Samuel Sambasivam is Chairman and Professor of the Computer Science Department at Azusa Pacific University. His research interests include optimization methods, expert systems, client/server applications, database systems, and genetic algorithms. He served as a Distinguished Visiting Professor of Computer Science at the United States Air Force Academy in Colorado Springs, Colorado for a year. He has conducted extensive research, written for publications, and delivered presentations in Computer Science, data structures, and Mathematics. He is a voting member of the ACM and is a member of the Institute of Electrical and Electronics Engineers (IEEE).

Stuart L. Wolthuis is Assistant Professor in the Computer & Information Sciences Department at Brigham Young University--Hawaii. His teaching focus includes software engineering, HCI, and information assurance. He brings almost 24 years of service in the USAF to the classroom with real world experiences as a program manager and software engineer. When not enjoying Hawaii's great outdoors, his research interests include melding together information systems and marine biology. His current project, Ocean View, will link land-locked educators and students to live underwater ocean views via an educational website.