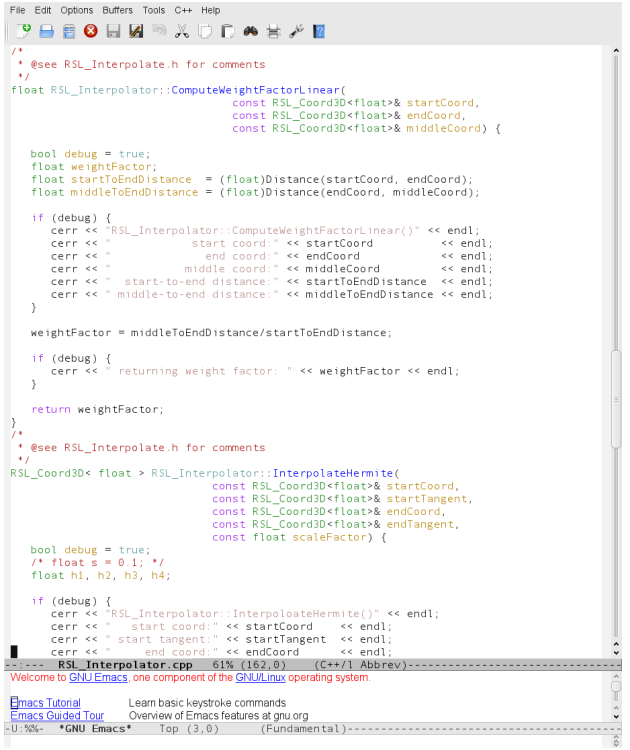
Bob's Concise Coding Conventions



Robert S. Laramee

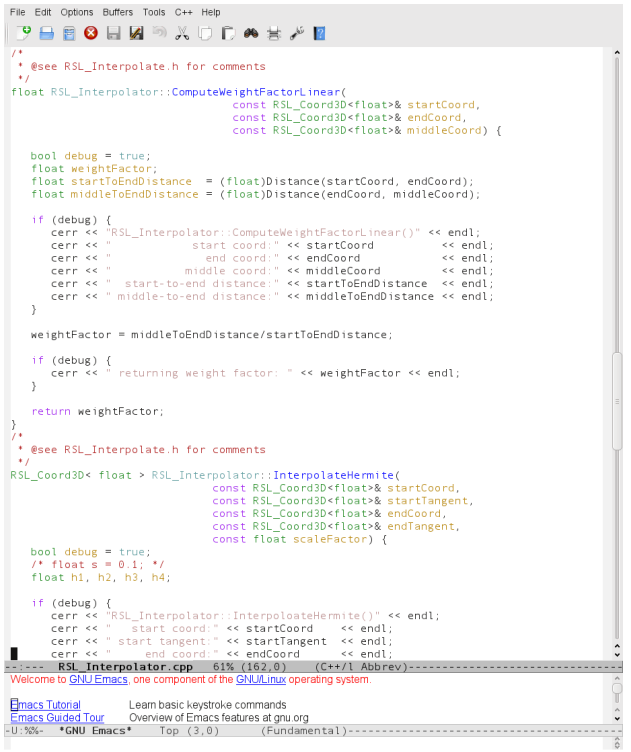
Visual and Interactive Computing Group
Computer Science Department
Swansea University, UK
r.s.laramee “at” swansea.ac.uk

Swansea University
Prifysgol Abertawe

Introduction and Motivation

Writing a useful software application is difficult.

- For some, this may be first time implementing larger, long-term project.
- Developing large software application requires more knowledge than implementing small one.
- Following guidelines, combined with good practices, facilitates success



Why Coding Conventions?

A Tool to Combat Problem Described By Bob's Theory of Software Redevelopment

- Illegible code is default-quickly turns into *legacy* code
- In “reality” most software projects fail [Ellis 2008, Krigsman 2008]
- Basic philosophy behind conventions is to maximize legibility
- Legible software is better software
- Legible software contains fewer bugs, more stable
- Legible software is more flexible, encourages re-use
- Two other key ingredients: Software Design and Comment Conventions

Overview

Part 1: Bob's Concise Coding Conventions

- Influences
- General set of guidelines for C++
- Applicable to imperative, object-oriented languages
- Concise, fit on one page
- Background (and references) behind each rule provided

Part 2: Bob's Theory of Software Re-Development

- Motivation behind the conventions
- Describes a frequently occurring development cycle

  Conventions intend to combat problems

Some Important Influences

- The Visualization Toolkit (VTK) Coding Conventions
- Java Coding Conventions from Sun Microsystems
- S. Meyers. **More Effective C++, 35 New Ways to Improve Your Programs and Design**, Addison-Wesley, 1996 (336 pages)
- S. Meyers. **Effective C++, 55 Specific Ways to Improve Your Programs and Designs**, Addison-Wesley, 2005 (320 pages)
- H. Sutter and A. Alexandrescu, **C++ Coding Standards, 101 Rules, Guidelines, and Best Practices**, Addison-Wesley, 2005 (220 pages)
- B. Stroustrup, **The C++ Programming Language, Special Edition**, Addison-Wesley, 2000 (1018 pages)
- Personal industry development experience

Rule 1: Method Length

1. Methods are 75 lines or less

- Method is visible on a single screen/page.
- Possible to see whole method from start to finish (without scrolling).

Exception(s): Methods with case tables (switch statements) and perhaps main method.

Motivation

- The longer a method is, the less re-usable andmore difficult it is to modify.
- The longer a procedure is, more likely it is to contain bugs and more difficult it is to debug.
- By confining method to one screen, it gives programmer (at least)

a chance to keep track of variables from beginning to end.

- Conformance to this rule facilitates code optimization with profiler [Meyers '96]

Rule 2: Indentation

2. No methods shall use more than five levels of indentation.

Exception(s): none

Motivation

Too many levels of indentation quickly renders code illegible.

Rule 3: Line Length

3. No line of code exceeds 80 characters.

It should not be necessary to expand code editor to entire screen width in order to read single line of code.

Exception(s): none

Motivation

- Lines that are too long are less legible and more difficult to debug.
- The longer a line is, the more difficult it is for eyes to move from end of one line to next.
- Good publishers use a guideline of approximately 66 characters per line of text (so 80 is generally too much) [Oetiker et al, 2008].
- Reason why most newspapers and magazines are multi-column
- Object-oriented programming requires multiple windows to be open simultaneously.

  Having one window open occupying entire screen makes the mechanics of programmer's job much more difficult [Sun Microsystems, 1999].

Rule 4: Class Variable Names

4. All class variables start with the two character sequence "m_"

(as in "member" variable) e.g., `m_ClassVariable`.

Exception(s): symbolic constants. Symbolic constants are written in `ALL_CAPITALS`.

Motivation

Class variables should be easily distinguishable from local variables or other types of variables.

Rule 5: Accessor Methods

5. All class variables are accessed with accessor methods,

i.e. `Get()` and `Set()` methods, e.g.,

`GetClassVariable()`, `SetClassVariable(int newValue)` .

Exceptions: none

Motivation

- Enforces encapsulation: extremely important concept in object-oriented methodology. (Wirfs-Brock et al. '90)
- Accessing member variables with methods makes implementation easy to change, e.g., a `float` to an `int`.
- Prevents unwieldy (or even impossible) search-and-replace operations [VTK Coding Standards '09, Sun Microsystems '99].

Rule 6: Accessor Methods

6. Accessor methods come at top of both header files and implementation files.
Exception(s): none

Motivation

- Accessor methods are most common to use, as such, it is most convenient when defined at the “top” of the file or class definition.

```
/*
 * @see RSL_Grid.h for comments
 */
bool RSL_Grid::SetMinMaxCoordsInXYplane(const RSL_Coord3D<float> newXYmin,
                                        const RSL_Coord3D<float> newXYmax) {
   bool debug = false;
   if (debug) {
      cerr << "RSL_Grid::SetMinMaxCoordsInXYplane()" << endl;
      cerr << " min in XY plane: " << newXYmin << endl;
      cerr << " max in XY plane: " << newXYmax << endl;
   }

   m_MinCoordInXYplane = newXYmin;
   m_MaxCoordInXYplane = newXYmax;

   if (m_MinCoordInXYplane > m_MaxCoordInXYplane) {

      cerr << "*** Warning, RSL_Grid::SetMinMaxCoordsInXYplane() " << endl;
      cerr << "new minimum is greater than maximum. " << endl;
   }

   return true;
}
```

Rule 7: Class Variables

7. All member class variables are private.

Exception(s): symbolic constants

Motivation

- Keeping class variables private enforces encapsulation.
- Only the class itself should know about the specific implementation details of its own data [Meyers 2005].

Rule 8: Method Naming

8. Private methods begin with a lower-case letter.

Public methods begin with an upper-case letter.

Exception(s): none

Motivation

- It is very nice to tell whether method is private or public simply by looking at it (without having to look it up) [Sun Microsystems 1999].
- Even in presence of tools.

Rule 9: Method Parameters

9. In general, methods do not require more than 5 parameters.

Exception(s): very rare

Motivation

- The more parameters a method takes, the less re-usable it is.
- Have different implementations of same method taking different (but only a few)

parameters.

- Too many method parameters, say six or more, may indicate problem(s) with software design.
- A long list of parameters may indicate that changes to design are necessary, e.g., the introduction of a new class(es) or re-arrangement of existing classes [Sun Microsystems 1999].

Rule 10: Symbolic Constants

10. Do not use numbers in your code, but rather symbolic constants.

Exception(s): 0 and 1.

Motivation

- One 6 may not be same as another 6. [Sutter and Alexandrescu 2005]
- Using symbolic constants instead of typing numbers makes code much more legible.
- Even original author eventually forgets what number is.
- Values of symbolic constants are easy to change.
- Changing values of numbers directly in the code causes bugs, especially when the number appears in multiple places [Sun Microsystems 1999].
- Horstmann articulates rule as “Do Not Use Magic Numbers” [Horstmann 2003].

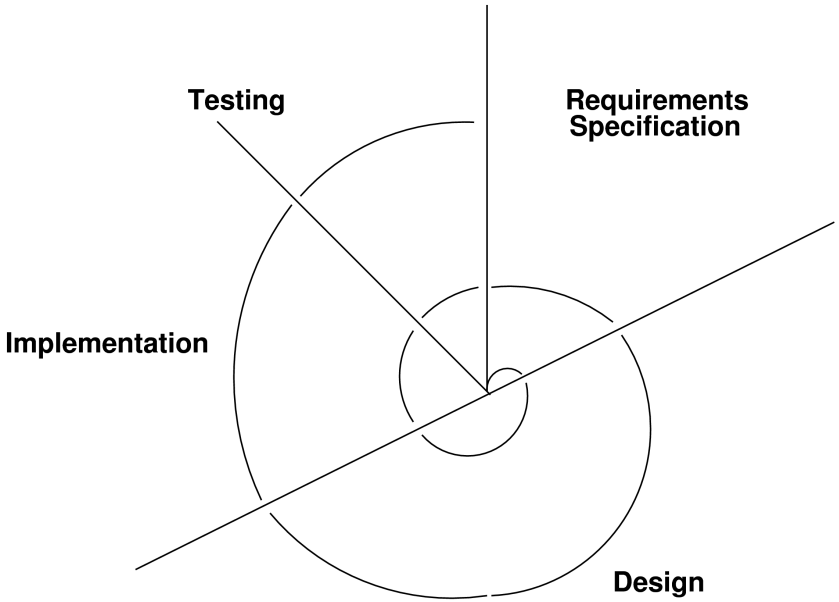
Rule 10: Symbolic Constants Example with Magic Numbers

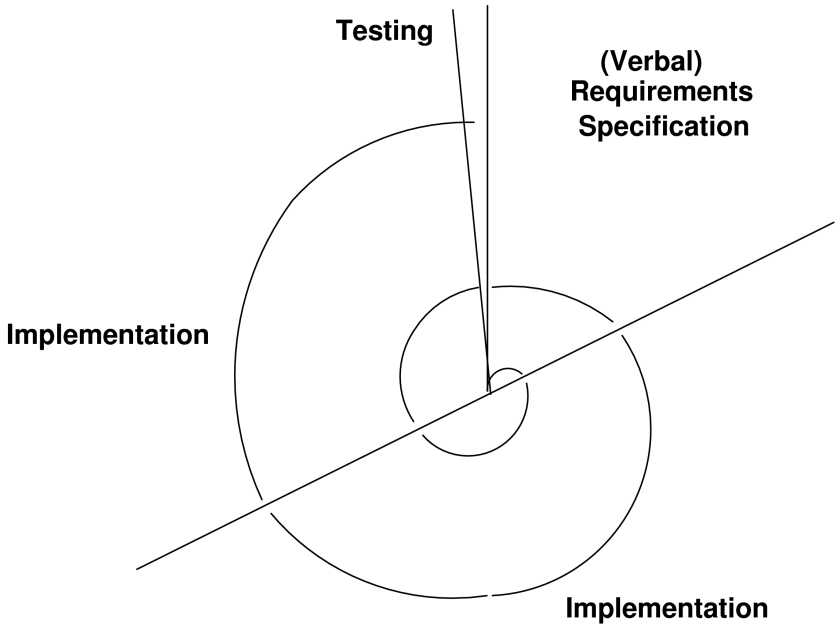
```
void RSL_OglTexture::CopyImageData(FXuchar* textureData) {

  bool debug = false;
  int  currentRow, currentCol, textureOffset, dataOffset;
  int lengthOfOneRow      = this->GetWidth();
  int heightOfOneColumn = this->GetHeight();

  if (debug) {
    cerr << "RSL_OglTexture::CopyImageData() name: " << this->GetName() << endl;
    cerr << " width: " << this->GetWidth() << ", height: " << this->GetHeight() << endl;
  }
  for (currentRow = 0; currentRow < heightOfOneColumn; currentRow++) {
    for (currentCol = 0; currentCol < lengthOfOneRow; currentCol++) {

      textureOffset = currentRow * lengthOfOneRow * 4 + currentCol * 4;

      dataOffset = currentRow * lengthOfOneRow * 3 + currentCol *3;

      this->GetBufferDataPtr()[textureOffset + 0] = textureData[dataOffset + 0];
      this->GetBufferDataPtr()[textureOffset + 1] = textureData[dataOffset + 1];
      this->GetBufferDataPtr()[textureOffset + 2] = textureData[dataOffset + 2];
      this->GetBufferDataPtr()[textureOffset + 3] = 255;
    }
  }
  if (debug) cerr << "RSL_OglTexture::CopyImageData() END" << endl;
```

Rule 10: Symbolic Constants Example with Symbolic Constants

```
void RSL_OglTexture::CopyImageData(FXuchar* textureData) {

  bool debug = false;
  int  currentRow, currentCol, textureOffset, dataOffset;
  int lengthOfOneRow      = this->GetWidth();
  int heightOfOneColumn = this->GetHeight();

  if (debug) {
    cerr << "RSL_OglTexture::CopyImageData() name: " << this->GetName() << endl;
    cerr << " width: " << this->GetWidth() << ", height: " << this->GetHeight() << endl;
  }
  for (currentRow = 0; currentRow < heightOfOneColumn; currentRow++) {
    for (currentCol = 0; currentCol < lengthOfOneRow; currentCol++) {

      textureOffset = currentRow * lengthOfOneRow * NUM_RGBA_COMPONENTS +
                  currentCol * NUM_RGBA_COMPONENTS;
      dataOffset = currentRow * lengthOfOneRow * NUM_RGB_COMPONENTS +
                currentCol * NUM_RGB_COMPONENTS;

      this->GetBufferDataPtr()[textureOffset + 0] = textureData[dataOffset + 0];
      this->GetBufferDataPtr()[textureOffset + 1] = textureData[dataOffset + 1];
      this->GetBufferDataPtr()[textureOffset + 2] = textureData[dataOffset + 2];
      this->GetBufferDataPtr()[textureOffset + 3] = MAX_ALPHA;
    }
  }
  if (debug) cerr << "RSL_OglTexture::CopyImageData() END" << endl;
```

Bob's Theory of Software Redevelopment





(left) The software development cycle presented in a typical object-oriented software engineering course at university [Wirfs-Brock et al. 1990],

(right) an often-used software development cycle.

Bob's Theory of Software Redevelopment: Stages 1-2

Stage 1-The Start

- Idea for a new product
- Idea is expressed *verbally*
- Pitched by an enthusiastic salesperson

Stage 2-The Implementation

- Starts immediately
- Core development carried out by 1-2 lead developers
- Lead developers work *hard* for v1.0 release in 1 year

Bob's Theory of Software Redevelopment: Stages 3-4

Stage 3-One Year Later v1.0

- Version 1.0 due, however, program has become *large*
- Implementation is more difficult than anticipated
- Large size is causing many problems: bugs, broken features, code needs organization, not all features are in place
- Therefore, release date needs to be *delayed*.

Stage 4-Two Years Later, v1.0

- V1.0 is released two years after start (one year delay)
- Product must be released after so much delay
- Delivery is not quite the success as imagined originally
- Bugs will be fixed for next version + lots of new features
- Additional engineers are assigned to the project to give it a push

Bob's Theory of Software Redevelopment: Stages 5-6

Stage 5-Three Years Later, v2.0, and Decline

- Many bugs need to be fixed, software should stabilize, should be nice new features
- But engineers experience *problems*, code base grows rapidly, little coordination amongst engineers, no software design, no coding conventions
- Software seems like a leaky barrel
- Engineers and mangers are *frustrated*.

Stage 6-The Departure

- Original lead engineers are frustrated: project is not success as anticipated
- Code base has grown out of control, introducing more bugs and problems
- Serious efforts invested have not brought anticipated reward
- Conflict between engineers and mangers: reward versus product delivery
- Original lead engineers **quit**.

Bob's Theory of Software Redevelopment: Stages 7-8

Stage 7-The Rescue Attempt

- No documentation
- Management will not let 3+ years of effort go to waste
- Replacement engineers, 2rd generation, are hired to save product
- 6 month grace period (for learning) follows

Stage 8-A Slow Death

- Bugs are fixed, but fixes and new features introduce new problems
- 2rd generation of engineers will eventually reach same conclusion as 1st
- Product cannot be rescued
- They either (1) quit-bringing in 3rd generation or (2) try starting a new project

GOTO Stage 1

Summary and Conclusions

We have introduced Bob's Concise Coding Conventions

- General set of guidelines for imperative, object-oriented languages
- Very easy to follow, concise, fit on 1 page
- Background (references) behind each rule provided

Bob's Theory of Software Re-Development

Motivation behind conventions

Describes a frequently occurring, failing, development cycle

Conventions combat perpetual re-invention of wheel

The more legible code is, the better it is.

Acknowledgements

Thanks to:

Tony McLoughlin and Ed Grundy of Swansea University

Further recommended reading:

Robert S. Laramee, **Bob's Concise Coding Conventions (C^3)** , in *Advances in Computer Science and Engineering (ACSE)*, Vol. 4, No. 1, February 2010, pages 23-36 (available online)