Maintainability and Readability

- elements of maintainability
- program readability
  - module structure
  - commenting
- coding standards and tools
- code ownership

What makes code “understandable”?

- good architecture
  - intuitive components, well chosen interfaces
  - straight-forward hierarchial structure
- good specifications
  - good overview of system structure & operation
  - clear descriptions of each component
- good design
  - good modularity and cohesion
  - well abstracted interfaces
- readable code
  - it is relatively obvious how the code works

What makes code “readable”?

- module organization
  - order in which we describe/define routines, variables
- visual layout
  - use white-space to delimit functional units
  - use consistent visual metaphors to convey structure
- commenting
  - to further accentuate structure
  - to guide us though non-obvious parts of the code
- variable and routine naming conventions
  - to make meaning of code more obvious
- these aren’t merely style; they involve technique

Understandable code

- Get together in groups of 3-4
  - each identify a personal experience of:
    - some very hard to understand code
    - some very easy to understand code
  - list the factors that made them so
  - reasons why they were done that way
  - prepare a summary of your conclusions
  - you have 5 minutes

General Module Structure

- standard preamble
  - copyrights, version, module overview
- include files, type definitions
- static global, then private data declaration
- routine pre-declarations (if required)
- constructors and destructors (if required)
- public, then private routine definitions
  - in some logical order (e.g. top-down, alphabetical order, temporal call order, etc)

General Routine Structure

- standard preamble
  - description of purpose, context, limitations
- routine declaration
  - return type, name, parameters & types
- local variables
  - by type, one per line, with descriptions
- blank-separated paragraphs of code
  - preceded, if necessary, with comments describing, in general, what each does
Explanations and Excuses
- often non-obvious code is bad
  - complexity results from a bad approach
  - it is usually better to fix it than to explain it
- some non-obvious approaches are good
  - eliminate problems, improve performance
  - explain problem and the novel approach
- sometimes we leave code incomplete
  - designed features that aren’t yet required
  - work we plan to complete later
  - leave warning, excuse, advice, comment

Commenting Data Items
- we mostly talk about commenting code?
- many data items also need comments
  - who uses it for what purpose
  - units, range, and meanings of its values
  - validity assertions, synchronization rules
- can be true for all types of declarations
  - basic types, structures, bit-fields, unions, etc.
- associate comments with declarations
  - end-line comments for simple declarations
  - large block comments for complex items

non-Readability Comments
- mandatory module preambles
  - copyright notices, legal disclaimers
  - title, version, and authorship information
- information for use by CAD tools
  - semantic interface descriptions
    - for documentation and test generation tools
    - notes for static analysis tools
    - information that is not statically determinable
    - correctness assertions
    - for automated testing or run-time checking

Coding Standards - scope
- naming conventions
  - generation, use of case, prefixes, suffixes
- usage conventions
  - e.g. versions, defines, include file processing
- commenting conventions
  - standard module preamble
  - standard routine preamble
- formatting conventions
  - indentation and commenting style

Code Ownership
- Form teams
  - When is code “personal property”? 
  - When is code “public property”? 
  - Where is the line between “creators’ prerogative” and “team responsibility”? 
  - What should we do if we are writing “personal code” that might one day become “public”?
- You have 5 minutes to prepare positions

For Next Lecture
- McConnell chapter 27 – project size
- McConnell section 28.3
  - estimation approaches
- Kampe: S/W estimation principles
- Wiki: COnstructive COst mOdel
- Peters: S/W Project Estimation
  - the ongoing process of estimation
- Wiegers: Risk Assessment &Management
  - overview of basic planning methodology
For Next Lecture
McConnell 21.1-2 - collaborative development
McConnell 28.1, 28.5 – good practices
McConnell 33 – Personal character
Wikipedia: XP Practices
Williams: Pair Programming
  – how well it can work
Rosenberg: Problems w/Pair Programming
  – how it can not work
CACM: Global Software Development (eres)
  – how it challenges development practices

Product Documentation
• design documentation
  – architectural/design introduction/overview
  – component level design specifications
  – design rationale folders
• end-user documentation
  – manuals on how to use the product
• support documentation
  – installation/configuration guidelines
  – trouble-shooting guidelines
  – detailed technical descriptions

Design Rationale(s)
• author - architects & designers
• audience - other architects and designers
• form - usually just a collection of notes
  – some may eventually turn into white papers
• content
  – issues, problems, options, decisions
• role in code maintainability
  – explain non-obvious features of the design
  – keep important lessons from being forgotten

Supplementary Slides

Architectural Overviews
• author - lead engineers
• audience - initially developers, later everyone
• form - eventually a polished report/presentation
• content
  – describe overall structure of product
  – goals, principles, components, interfaces
  – later, a technical introduction to the product
• role in code maintainability
  – lays conceptual foundations, intro to design

Installation/Configuration
• author - responsible engineers
• audience - support and customers
• form - external product documentation
• content
  – process for installing product
  – product configuration options
  – performance considerations and tuning
• role in code maintainability
  – explain how the product is managed
Trouble-Shooting Guidelines

- **author**: development and support engineers
- **audience**: technical support
- **form**: internal product documentation
- **content**:
  - how to diagnose likely problems
  - how to fix or get around them
- **role in code maintainability**:
  - direct how-to guidance for support engineers
  - acquaint maintenance engineers w/problems

Detailed Technical Descriptions

- **author**: responsible engineers
- **audience**: support/maintenance engineers
- **form**: internal product documentation
- **content**:
  - detailed component design descriptions
  - may be at routine level, or even more detailed
- **role in code maintainability**:
  - training for new code maintainers

NOTE: these are not common

Detailed Technical Descriptions

- **often mandated by large organizations**
  - with very long maintenance commitments
    - e.g. aerospace, government
  - when many maintainers will be trained
    - e.g. military, telecommunications
  - they do improve product maintainability
  - they are very expensive to produce
    - documentation takes longer than the code
    - they have to change whenever the code does
  - easier just to make code more readable

javadoc

- **A commenting discipline and tool for automatically generating documentation from the code.**
- **A set of standard tags**
  - @param, @return, @throws, @serial
  - @link, @see, @version, @since, @author
- **A convention for descriptions**
  ```
  /**
  * stuff for javadoc
  */
  ```

choosing good names

- **well chosen routine names** ...
  - describe what the routine does
  - suggest the meaning of their return value
- **well chosen variable names** ...
  - tell us what the variable means
  - suggest its scope and general class
- **good names are better than comments**
  - because they take up much less space
  - because they appear on every usage

mnemonic naming conventions

- **describe the entity/action they represent**
  - integer: linesPerPage
  - boolean: outOfSpace
  - routine: pushStack()
- **follow recognizable patterns**
  - scoreMax, scoreMin, scoreMean, ...
  - addStudent(), dropStudent(), ...
- **are long enough to make sense**
  - but short enough to be manageable
    - easy to type, don’t take up the whole line
syntactic naming conventions
• can suggest general variable classes
  – UpperMixed classes and defined types
  – lowerMixed locals and parameters
  – UPPERCASE constants and macros
• can suggest scope
  – _m_mixedCase member-private data
  – G_mixedCase global data
• can suggest class from which they come
  – base_Value enumerated types

white-space & program structure
• blank lines make block separation clear
• indentation makes block nesting clear
• visually clearer than keywords or braces

standard (K&R) C indentation
• most braces on same line as keyword
• closing braces, un-indented, on own line
• motivation - ease of editing

Indentation Guidelines
• use consistent indentations for all ...
  – lines within a single loop, block, or sub-case
  – loops, blocks or sub-cases at the same level,
    (especially siblings within a larger block)
  – labels are allowed to hang to the left
• avoid excessively large indentations
  – no more than 50-75% of a line
  – use a smaller indent (e.g. 4 vs 8)
  – put the sub-block into a sub-routine
  – as a last resort, shift whole block left

Line Formatting
• expressions
  – use parenst to make precedence explicit
  – use spaces to make sub-expressions obvious

simpler code: boolean exprs

Useful Code Commenting

• prose or pseudo-code summaries
  – explain the purpose of the code that follows
  – high level overview of the algorithm
  – enumerate pre-conditions that must hold
• rationale and references
  – remind reader of important issues
  – explain non-obvious choices
  – refer reader to more detailed discussions
• draw attention to module sub-sections
  – start of a new class or routine

Labels for non-obvious targets

```c
if (queue.numEntries > 0) {
    ...
} else /* (queue.numEntries <= 0) */

    ...
    if (request.status == ERR_FATAL)
        break; /* NextRequest */
    ...
} /* NextRequest */

...
```

Standards - tools

• static analysis tools (e.g. lint)
  – stricter checking than the C compiler
• style checkers (e.g. cstyle, hdrchk, libchk)
  – audit code against specified standards
• pretty-printers (e.g. indent, C beautifier)
  – reformat code w/standard indents/spacing
• auto-documenters (e.g. javadoc)
  – generate routine documents from comments
• code browsers (e.g. cxref, source navigator)