Coding Conventions

A Local Standard

CIS*2750

Professional Aspect of Software Engineering

Goal: Self-Documenting Code

- **Self-documenting** explains *itself* without need for external documentation, like flowcharts, UML diagrams, process-flow diagrams, etc.
  - *Doesn't imply we don't like/use those documents!*
- **Coding conventions** target:
  - How you write **statements** in the language, **organize** them into "modules," **format** them in the source files
    - *Module:* generic term meaning C function, Java/C++ class, etc.
  - How you create **names**
  - How you write **comments**

Motivation

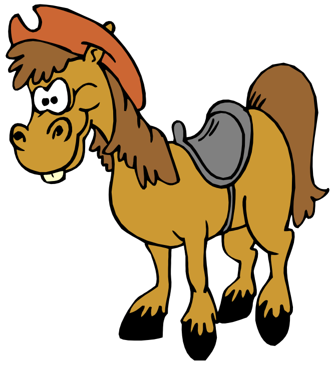
- You have a **team** working on a project that will extend over a moderately **long** period of time. Over the course of the project, **people** will come and go.
- The project needs to progress as **fast** as possible
  - You want to implement only what you really need
  - *Strategy:* **reuse** as much *legacy code* as possible
  - New functionality will be built as the need for it emerges.

Style of Working

- Teams will be looking at many modules as they craft simple enhancements to make the system do what is needed.
  - They will **read** many modules, and **edit** many.
  - They will “factor out” **duplicate** code and move it to a single place in the system (=refactoring).
  - Over time, few modules will be able to be said to have *an* **author**.

Standard Coding Practices

- Teams strive to use the same **coding conventions** in every regard:
  - **Name** your classes similarly, your variables, your functions.
  - **Comment** the same way, **format** your code the same way.
    - By doing this, you ensure rapid understanding of whatever module needs changing, and as they evolve, your modules will not degenerate into a *HorseByCommittee* appearance.



Benefits

- Projects **benefit** from having strong Coding Conventions/Standards because...
  - People can stop **reformatting** code and **renaming** variables and methods whenever working on code written by **other** people.
  - It's slightly **easier to understand** code that is consistently formatted and uses a consistent naming standard.
  - It's **easier to integrate** modules that use a common consistent naming standard -- less need to look up and cross-reference the different names that refer to the same thing.

Which Coding Conventions?

- Doesn't really matter that much, as long as everyone follows them (and stops *arguing* about them!)
- One sometimes encounters **Bad Coding Standards**, arbitrarily imposed restrictions destructive to the development process, but that's the **exception** rather than the **rule**.

2750 doesn’t use “teams,” but…

- Trying to instill good practices *now*
  - Even on single-author projects, often have to return later and read/understand/modify your own code
  - TA needs to understand your code
- CIS*3750 & 3760 *do* have team projects
- Practice adapting to local standards
  - Coop term, later jobs, mandated by company
  - No place for “hotshot” sailing in and insisting on doing everything his/her own way → better self-employed

Coding Conventions Apply To…

- Comments, 3 types:
  - **File** headers
  - **Function** headers
  - Explanations of **variables** and **statements**
- Names (chosen by programmer)
- Statements
  - Organization: files, “modules,” nesting
  - Format: spacing and alignment

Room for “Taste”?

- Not trying to squelch *all* personal style
- Proper style can be in the eye of the beholder

Style 1:

```
variable1 = variable2 + other + long + stuff
  + function3();
```

Style 2:

```
variable1 = variable2 + other + long + stuff +
  function3();
```

Organization of Program

- *Analogy:* Organize programs for readability, as you would organize a book:
  - Title page & Table of contents → **File header**
  - Chapter → **Module** (function or logical group of functions)
  - Paragraph → **Block of code**
  - Index & Glossary → can be generated automatically if comments are used wisely (Javadoc, doxygen)
  - Cross references → ctags.sourceforge.net free tool
    - "man ctags" in lab; works with vi(m), gedit. nedit

Organization of Modules

- Apply comp. sci. principle of information hiding
  - Hide details of implementation that users don’t need to know
- Divide each module into a **public** part and a **private** part.
  - public part goes into an *include* (.h) file
  - private part goes into a *source* (.c) file

How Many Source Files?

- Matter of policy and/or taste
  - One extreme: just one module per file
    - OO programming: 1 class per file is common
    - Large project → explosion of files! .h .c .o
  - Other extreme: all modules in one file
    - Reasonable for quite small project
    - Large project → lose benefit of separate compilation
- Middle way: group *related* modules in one file
  - **calutil.h/c: all iCalendar utility functions**

>2-3000 lines is getting too large

File Headers

Tells at a glance what file you're looking at and what's in there.

- *Filename*
  - Suggests the purpose of the file's module(s)
  - Could be class name (Java/C++)
- *Description*
  - This is the main reason to have the header. It clearly tells the purpose of the module(s), their role in the system. If you can't describe the purpose in one or two sentences, you've thrown unrelated modules together.

File Headers

- *Creation date*
  - Provides a creation timestamp for copyright purposes, but it does more than that. It provides a quick clue to the *context* that existed at the time the module was created. Not as accurate as automatic source control, but *quick* and *maintenance free*.
- *Author's Name or Initials*
  - *2750 convention:* give **Student Number**
- *Copyright banner*
  - This identifies the uses to which this code can be put.
  - *2750 convention:* optional for assignments

File Headers

- *E-mail address*
  - If this is publicly released code, then the author should be contactable.
- *The lack of anything else*
  - More info likely has to be kept up to date.
  - Maintenance free, *simplicity* and *clarity* are the goals.

Function Headers

- Commenting **interfaces** is an especially good thing to do.
  - Function headers describe purpose of function and use of arguments, return value, also pre-/post-conditions.
    - **public** interface function headers & prototypes go in .h files
    - **internal** helper functions & headers go in .c files (unless called from multiple .c files)
      - No need for internal function prototype if function precedes use
- *2750 convention:* no function headers needed in our supplied .h (would just duplicate spec)
  - But you must provide headers for any functions *you* add

Commenting Implementations

- **Implementations** (.c file, etc.) don't need to be commented heavily if their interfaces are well-documented.
  - It is possible to *overcomment* → creates big maintenance burden
- A well-written implementation for a given function is rarely more than 100 language statements. Thus, if you know what the function is supposed to do, it should be straightforward to understand the mechanics of it.

Meaningful Comments

- **Comments** provide **meta** information about the program, reasons for choosing this algorithm or implementation, known issues, hints for **future readers** (including yourself).
- "Meaningful" comment:
  - Can be **understood** by readers.
  - Says something that is **likely *not* to be understood** by the same readers unless it was present.
  - Don't "parrot the code":

```
x = 2 + y;        // add 2 to y
```

- Avoid comments needing heavy maintenance

Psychological Factors

- **Redundancy**

  Cxn yxx rxxd thxs sxntxnce?

  Cn y rd ths sntnce?

- More effort is required as redundancy is removed.
- Something to consider when creating variable names and writing comments.

Variable Names

- Use simple, descriptive variable names.
- Good names can be created by using one word or putting multiple words together joined by underscores or caps (“camel” case)
  - prefer usual English word order

```
#define MAX_FIELD 127
int numStudents, studentID;
char *homeAddr;
```

Variable Names

- Be careful of lower-case L (l) or upper-case O in variable or constant names

```
int l, O;  l = O + l/0.1; //bad

int length; FILE *outfile; //OK
#define KG_PER_TON 907.18474;
```

- Do not use names of existing library functions or constants → multiply-defined externals or worse
  - do not use the names **argc**, **argv** for any other purpose except command line argument manipulation

Variable Names

- Avoid variable names that differ by only one or two characters.
- Short names such as `x`, `n`, `i` are acceptable when their meaning is clear and a longer name would not add any more information.
- *Trade off:* long, unabbreviated names $\rightarrow$ statements become too long, hard to follow
  - studentIdentificationNumber, arraySubscript

Variable Names

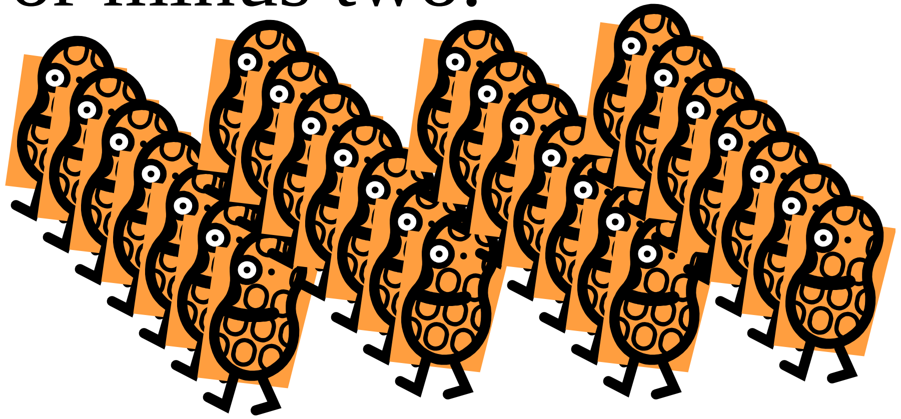
- Follow every variable declaration with a comment that explains it.
- Group similar variables together.
  - use similar names for variables that perform similar functions

Statement Style

- Put each statement on a line by itself.
- Avoid very long statements. Use two shorter statements instead.
- Keep all lines to 80 characters or less.
- Group statements in logical “chunks” separated with white space (blank lines)
  - Helps eye follow logic without getting overwhelmed

Psychology of "Chunking"

- Recognizable pieces of information that can be fitted into one *slot* of (human) short term memory.
- Seven plus or minus two.



Use Vertical Alignment (Type A)

- Makes lines at same level of nesting stand out.

```
if ( flag == 0 ) {
    var1 = 0;
    if ( var2 > level1 ) {
        var2 = level1;
        level1 = 0;
    }
    printf ( "%d/n", var2 );
}
```

Ugly Code

- *How about this?*

```
if (WndHt < WIN_MIN)
{
  ResetWin(WPtr );
  while( WndHt> WinHt)
  {
  WinHt =getWindow( pWin );
  if ( WinHt== ( winhite *
 WND_CORR )) {
     stepup (wdwhght );
     STEPUP( wndwh);
  }
  }
}
```

- **Align Consistently!**

```
if ( WndHt < WIN_MIN ) {
     ResetWin(WPtr);
     while ( WndHt > WinHt ) {
         WinHt = getWindow(pWin);
         if ( WinHt == (winhite *
                WND_CORR) ) {
             stepup (wdwhght);
             STEPUP(wndwh);
         }
     }
}
```

Vertical Alignment (Type B)

- Makes tabular information easier to read.

```
int LineFactors[3][5] = { { 19,  2, 22, 32,  5 },
                          { 99, 33, 55, 45,  4 },
                          { 32,  6, 14, 21, 15 } };
```

- Hint: to make lovely alignment "bulletproof" set editor to **convert tabs to spaces**

{{{Nested Blocks}}}

- Increases apparent complexity non-linearly.
- 3 levels is enough for major nesting.
- Ways to reduce nesting (goto-less jumps):
  - function **return** from nested code
  - loops:
    - **continue** skips to end-of-loop test
    - **break** exits off bottom of loop
  - techniques avoid piling up deep “if/else” blocks, and creating extra “done” flags

Good General Coding Principle

- **KISS**
  - Keep it simple - always easier to read and maintain (and debug!)
- **Be Explicit**
  - SWYM - Say What You Mean

`if ( WordCount )` vs. `if ( WordCount != 0 )`

`n+3*x-5/y` vs. `n + (3*x) - (5/y)`