$Id: coding-style.mm,v 1.241 2014-04-03 19:43:01-07 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps012b-wm/Coding-style
URL: http://www2.ucsc.edu/courses/cmps012b-wm/:/Coding-style/

Programs are a communication device not only from a person to a computer, but from a person to another person, and should thus be readable. Time spent in making a program readable makes it easier for others to read your program.

This document provides some general coding style guidelines. They are necessarily incomplete and not every statement is applicable to every programming language. Examples are restricted to the C and Java languages, since those are the only programming languages seen by UCSC students in introductory courses.

Most text is in Roman or **Bold** font for the narrative. In coding examples, **`Courier Bold`** is mostly used to indicate what you type in exactly, and *Italic* to indicate something you substituted. In computer interaction listings, `Courier Roman` is used to indicate what the computer displays and **`Courier Bold`** to indicate what you type in yourself at the keyboard.

This document is expected to be marginally comprehensible to students at the beginning of CMPS-012A, mostly comprehensible to students at the end of CMPS-012A or the beginning of CMPS-012B, and completely comprehensible to students who have completed CMPS-012B, or any course for which it is a prerequisite.

1. General introduction

(1) The first line of every text file you submit for evaluation should contain your personal name and *username*`@ucsc.edu` at the top of the file. If it is program file, this must be in a comment. A text file is any file readable with an editor, `cat`(1), `more`(1), etc., be it a program source file, a `Makefile`, or a `README`. In the case of a shell or Perl script, it must be the second line, the first line being the hash-bang (`#!`) line, as in

```
#!/bin/bash
```

(2) When resizing an `xterm` using your window manager, look for the little box that shows its dimension. It might say something like `80x75`. Make sure the `xterm` is 80 columns wide. Choose whatever height is convenient for your screen. Taller is better. Taller than the screen is not good. History: The original DEC VT-100 terminal, which `xterm` emulates had 80 columns and 24 lines, which is why the default `xterm` uses `-geometry 80x24`.

(3) Messages sent in email should have lines limited to 65 characters, so that when quoted in a followup, lines tend not to wrap and are still shorter than 72 characters after several followups. History: A page is 8½ inches wide and with 1 inch margins leaves 6½ inches for text. A standard typewriter font prints 10 characters per inch, which is 65 characters per line.

(4) To reformat a paragraph from inside `vi`(1), the `fmt`(1) command may be used in several ways, sometimes with the crown margin (`-c`) option. A couple of examples follow:

```
{!}fmt -65
:10,20!fmt -65
```

(5) If you insist on having a signature keep it to four or fewer lines. No one wants to see an ASCII cartoon or box of asterisks. And, of course, a `.gif` or `.jpg` is quite obnoxious.

(6) If you use `vi`(1), create a file called `.exrc` in your home directory. A sample `.exrc` file is shown in the `examples/` subdirectory. Your `.exrc` should contain at least the following lines:

```
set number
set showmode
```

2. File format

(1) Lines of source code should never be longer than 72 characters. The standard `xterm`(1) is 80 columns wide, and when the `:set number` option is used in `vi`(1), line numbers are displayed in a field of width 8 on the left side of the screen. Lines longer than 72 characters thus cause ugly and unreadable wraparound. History: Punched cards were 80 columns wide, with 72 characters being used for data and the remaining 8 characters used for sequence numbers (in case you dropped your deck).

(2) Text files should contain only visible printable characters, spaces, and newlines. Specifically, there should ***never*** be carriage return (`^M`) characters in a file. If you have the extremely bad judgement to edit using bloatware from M*cr*$*ft, you may delete them with such programs as `dos2unix`(1), by using the `vi` command

```
:g/^V^M/s///
```

or with the shell command

```
perl -pi -e 's/\r//g' filename
```

(3) Do not use tabs in files. When used for indentation, they cause the code to move too far to the right, thus making lines appear longer than 72 characters. Never use an editor command to change the length of a tab, since that will not necessarily be the same when viewed by someone else using a different editor. The command `expand`(1) can be used to remove tabs from files.

(4) ***Exception:*** When typing in a `Makefile`, tabs are required before commands, but never elsewhere. Use exactly one tab at the beginning of a shell command in a `Makefile`. Making tabs significant was a design flaw in `make`(1), which `gmake` follows.

(5) The last character of every line in a text file must always be a newline character. This includes the last line of a file. The `vi`(1) editor automatically ensures that this is true, but `emacs`(1) does not. If you use `emacs`, make sure your home directory has a file called `.emacs` (note the initial dot) containing the line

```
(setq require-final-newline t)
```

(6) Every file should usually contain an `RCS $Id$` string. This identifies a copy of any printed listing with a specific version number. This identifier should be the first (after your name and username) or last line of a file, and, for any given project, be consistently placed. For a program, it usually appears in a comment, but it may instead appear in a macro or string in order to propagate it into an object file. All versions of all files must be archived in an `RCS`, `CVS`,

SVN, or Git repository.

(7) SVN users should Google for "svn:keywords" to find out how to do this. See [http://svnbook.red-bean.com/en/1.0/ch07s02.html] for a command like

```
svn propset svn:keywords "Id" filename
```

(8) One of the following commands can be used to check for invalid unprintable characters in a file (usually tabs and carriage returns) and for a missing newline at end of file. It cats the file using the -vt option which displays invisible characters visibly, and pipes the output into diff(1) to check it against the original. A minus sign (-) argument to diff causes it to compare stdin. It also complains about missing newlines, if any. We specifically omit the -t option for Makefiles, since tabs can not be avoided.

```
cat -vt filename 2>&1 | diff - filename
cat -v  Makefile 2>&1 | diff - Makefile
```

(9) The following command cats a file into expand(1), cut(1)s the first 72 characters of the file, and grep(1)s for any remaining line with nine or more characters. I.e., the line numbers and excess characters are printed. It prints nothing if all lines are within bounds. The grep(1) command has 9 dots, which will match the 8-position line number plus one character.

```
cat -n filename | expand | cut -c1-8,81- | grep .........
```

(10) Run the program checksource (in the bin subdirectory of the course volume) and see that it does not complain.

3. Filenames and identifiers

(1) Filenames should be spelled using only lower case letters, digits, periods, and underscores or minus signs. Upper case letters in filenames are generally to be avoided, except for special names such as Makefile and README, which must be spelled exactly that way. Commands like ls(1) sort filenames lexicographically and thus list capitalized names before lower case names. Use this as a sorting tool. The following non-alphanumeric characters generally to not cause problems when used in filenames:

```
% + , - . : = @ _
```

(2) Shell metacharacters are prohibited in filenames. A slash (/) is a directory separator, so Unix will not let you use it in a filename, even if you quote it. Following is a list of shell metacharacters:

```
! " # $ & ' ( ) * / ; < > ? [ \ ] ^ ` { | } ~
```

The tilde (~) only has special meaning when it is the first character in a filename, in which case it causes username interpolation. This, along with the plus (+) and minus (-) characters, may appear in a filename, but never as the first character.

(3) When Java compiles inner classes, it creates filenames with the dollar sign ($), which is a very bad design decision, but there is nothing you can do about it.

(4) Never use one-letter identifiers. Using i, for example, means that when searching for the variable using an editor, the search will stop at keywords such as if and while. Instead, use a name related to the variable being scanned, as in the use of argi as an index into argv. Never use an identifier

which is a substring of a reserved word.

(5) On the other hand, excessively long identifiers are not a good thing either. Names like `ArrayIndexOutOfBoundsException`, `NoSuchElementException`, `NullPointerException`, `StringIndexOutOfBoundsException`, are excessively long and uglier than a camel. Kernighan & Ritchie's laconic style as used in the C library is better, e.g., `fclose`, `fopen`, `printf`, `strcmp`.

(6) Many Java programmers like using CamelCase. CamelCase is ugly and unnatural because it has interior capital letters. When using the Java API, it is not possible to avoid CamelCase. Generally it is better for variable, class, and function names to be writting entirely in lower case with underscores separating words.

(7) Identifiers should be long enough that the command `grep`(1) will print out only those lines containing those identifiers, and few other spurious lines. Do not use excessively long names, nor interior capital letters. Use underscores to separate words when that enhances readability.

(8) Enumeration constants, preprocessor symbols, and `final` variables in Java should be spelled in `UPPER_CASE`. No other identifiers should be spelled in upper case.

(9) It is a C convention that no name may end with the characters `_t` except for symbols defined via `typedef`.

(10) If an identifier consists of more than one word, and there is no ambiguity, such as `strcmp`, etc., just concatenate the words. If it is difficult to see the word boundary, use an underscore (_) to separate the words, as in `max_incr`, for example.

4. Indentation and layout

(1) Top level constructs should always be against the left margin. For Java, these are `import` and `class` statements. For C, they are `#includes`, global variables, and functions.

(2) An opening brace (`{`) should never be on a line by itself. Always place it at the end of the line after the control construct. The corresponding closing brace (`}`) should be in the same column as the keyword or type that opened the block.

(3) ***Exception:*** Some C programmers like putting the opening brace of a ***function*** in the leftmost column immediately underneath the return type. This is the only case where an opening brace may appear on a line by itself.

(4) All code between the braces should be indented by three spaces, as shown in the example C and Java programs.

(5) When using an `if`...`else` structure, the `else` should be on the same line as the closing brace of the corresponding `if`, When using `else if` the entire construct from the closing brace before the `else` to the opening brace after the condition should be on the same line, if it fits. Following is an example of the expected layout:

```
if (condition) {
   statements ...
}else if (condition) {
   statements ...
}else {
   statements ...
}
```

(6) An infinite loop with a **`break`** or **`return`** in the middle is frequently written as an empty **`for`**-construct, as in

```
for (;;) {
   statements ...
   if (condition) break;
   statements ...
}
```

(7) Occasionally, both branches of the conditional will be very short and then may be written in a slightly different style without braces, with each statement being aligned :

```
if (prev == null) head = newnode;
             else prev.link = newnode;
```

(8) In C, always use file guards in a header file. So, for example, if you have a file called `foo.h`, make a preprocessor symbol from the filename (upper case), as in

```
#ifndef __FOO_H__
#define __FOO_H__
header file declarations ...
#endif
```

(9) In C, use the preprocessor symbol **`NULL`** and not a 0, because it is more readable.

(10) Never omit the braces if the statements being controlled by a construct are written on a different line than the control construct. If the **`if`**-part has braces then so must the **`else`**-part and vice versa.

(11) Never omit the braces if the statement being controlled by a construct is itself a control construct. This is especially important if the keyword **`else`** is involved. The dangling **`else`** can introduce hard-to-find bugs.

(12) Parentheses should be used consistently. Either put a space immediately outside of each parenthesis, opening or closing, as is illustrated in the example in this document, and never inside ; or put a space immediately inside each parenthesis, but not outside. Every parenthesis must have a space before it or after it but not both.

(13) When defining a parameterized macro in C, the language prohibits a space in front of the open parenthesis in the left side of the definition.

(14) Always put a space after a semi-colon in the control part of a **`for`** statement. Do not put more than one statement on a line without a very good reason.

(15) Do not use parentheses at the top level of a **`return`** statement. If the **`return`** statement is complicated, inner parentheses might be needed.

(16) Sometimes a statement is too long for a line. In that case, the best place to break a line is after a comma, as shown here. The next line should then be indented a little further and aligned at a sensible spot.

```
out.printf ("value = %d, string = %s, truth = %b%n",
            value, this.string, the.truth);
```

(17) For a very long assignment statement, break just in front of a very low precedence operator and align that operator under the equal sign.

(18) Do not declare more than one variable per line, except for formal parameters, which are normally declared all on the same line. If that line is long, break after a comma, and indent the next line to the indentation of the first parameter.

5. Comments

(1) Every program must also contain an RCS `$Id$` string in a comment, macro, or string. When you initially type it in, just type the four characters. The RCS `ci` operation will edit that string.

(2) The beginning of a program containing a main function should have a large comment in the format of a Unix `man`(1) page, so shown in the examples.

(3) If you use `//` comments, Each of the comment markers should be lined up at the beginning of each line, as is shown in the example C program.

(4) If you use `/*`. . . `*/` comments, the opening `/*` and closing `*/` should be on lines by themselves with a column of asterisks (`*`) connecting them, as is shown in the example C program. It is also acceptable, for short comments to put both the `/*` and `*/` on the same line.

(5) Comments on code or functions should precede that which they describe and be indented to the same level as the code they describe.

(6) If code is tricky, and a comment is short, the comment may appear after the code on the same line, provided that it fits. Tricky code is usually bad.

6. Process requirements

(1) Error messages should always be printed to `stderr` (`System.err`), never to `stdout` (`System.out`), and an error message (whether a warning or a fatal error) should cause a non-zero exit code.

(2) Debug messages and trace chatter should also be written to `stderr`, not `stdout`, since they are not properly part of the output stream of a program.

(3) An exit status of 0 should always be returned if the program worked. A non-zero exit status between 1 and 255 should be returned if the program failed. In C, use the preprocessor symbols `EXIT_SUCCESS` for 0 and `EXIT_FAILURE` for 1, rather than the numbers.

(4) In Java, the `main` function is of type `void` and returns an exit status of 0 if it returns. To return a non-zero exit status, use the function `System.exit`. Define the following constants :

```
public static final int EXIT_SUCCESS = 0;
public static final int EXIT_FAILURE = 1;
```

7. Coding strategy

(1) Avoid confusing uses of the `++` and `--` operators. The following is very bad:
```
data[++index] = other[++index];
```

(2) Never[1] use a `goto`[2]. Well, almost never[3]. Most programmers, upon being told to go to hell[4], will object more to the method than the destination. The `break`, `continue`, and `return` statements make a `goto` absolutely unnecesary for well-written code. Java has, in addition to these, `try`, `catch`, and `throw`.

(3) In C, `gdb`(1) and `valgrind`(1) to check for bad memory accesses and memory leak.

(4) If you use `gcc` to compile your programs, use the following options:
```
gcc -g -O0 -Wall -Wextra -std=gnu99
```

(5) Study the Ten Commandments for C Programmers (Annotated Edition), by Henry Spencer, available at [`http://www.lysator.liu.se/c/ten-commandments.html`]

8. Example code

Following are some example programs and code. The files are all in a subdirectory `examples/`. The line numbers are present for ease of classroom discussion, and are not actually in the document. The leftmost characters in the text of the programs is actually flush against the left margin, indented code being moved over a little. Each example is preceded by a shell pipelines that test for basic formatting sanity.

An example `Makefile` is shown at the end of this document. but a detailed discussion is beyond the scope of this document. Refer to the classroom presentation at the appropriate time. In general, it is a better idea to use `gmake` rather than `make`, since it is more powerful, and has builtin as well as user-defined functions.

1. Edsger W. Dijkstra: Go To Statement Considered Harmful. [`http://www.acm.org/classics/oct95/`]

2. Edsger W. Dijkstra: EWD 214: A Case against the GO TO Statement. [`http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD215.PDF`]

3. Donald E. Knuth: Structured Programming with go to Statements. ACM Computing Surveys, vol. 6, December 1974, pp. 261–301. [`http://portal.acm.org/citation.cfm?id=356640`]

4. ‹*Lasciate ogne speranza, voi ch'intrate.*› — Dante Alighieri, *La Divina Commedia, Inferno, canto iii.*

8.1 An example Java program

```
bash-% cat -vt examples/args.java | diff - examples/args.java
bash-% cat -n  examples/args.java | expand | cut -c1-8,81- | grep .........
bash-% cat -nv examples/args.java | expand
     1  // Author:  Your Name, username@ucsc.edu
     2  // $Id: args.java,v 1.1 2009-12-17 18:04:46-08 - - $
     3  //
     4  // NAME
     5  //    args - print out the class path and the value of args
     6  //
     7  // SYNOPSIS
     8  //    args [string...]
     9  //
    10  // DESCRIPTION
    11  //    The class path is printed, followed by each of the arguments
    12  //    given in the command line vector.  When a Java program is run
    13  //    from a jar, the classpath is the name of the jar.
    14  //
    15
    16  import static java.lang.System.*;
    17
    18  class args {
    19
    20     public static void main (String[] args) {
    21        String classpath = getProperty ("java.class.path");
    22        out.printf ("classpath=\"%s\"%n", classpath);
    23        for (int argi = 0; argi < args.length; ++argi) {
    24           out.printf ("args[%d]=\"%s\"%n", argi, args[argi]);
    25        }
    26     }
    27
    28  }
    29
```

8.2 An example C program

```
bash-% cat -vt examples/argv.c | diff - examples/argv.c
bash-% cat -n  examples/argv.c | expand | cut -c1-8,81- | grep ........
bash-% cat -nv examples/argv.c | expand
     1  // Author:  Your Name, username@ucsc.edu
     2  // $Id: argv.c,v 1.1 2009-12-17 18:04:46-08 - - $
     3  //
     4  // NAME
     5  //    argv - print out the strings in argv
     6  //
     7  // SYNOPSIS
     8  //    argv [string...]
     9  //
    10  // DESCRIPTION
    11  //    Print out the strings from the command line argument vector.
    12  //
    13
    14  #include <stdio.h>
    15  #include <stdlib.h>
    16
    17  int main (int argc, char **argv) {
    18     int argi;
    19     for (argi = 0; argi < argc; ++argi) {
    20        printf ("argv[%d]=\"%s\"\n", argi, argv[argi]);
    21     }
    22     return EXIT_SUCCESS;
    23  }
    24
```

8.3 An example `Makefile`

Note that tabs only appear in the left margin to indent commands, not comments, macro definitions, or dependency specifications. When displayed at a terminal or on paper, tabs appear to be eight spaces.

```
bash-% cat -v  examples/Makefile | diff - examples/Makefile
bash-% cat -n  examples/Makefile | expand | cut -c1-8,81- | grep .........
bash-% cat -nv examples/Makefile | expand
     1  # $Id: Makefile,v 1.1 2009-12-17 18:04:46-08 - - $
     2
     3  SOURCES = Makefile args.java argv.c
     4  LINTOPT = -D__EXTENSIONS__ -Xa -fd -m -u -x -errchk=%all,no%longptr64
     5  CCOPT   = -D__EXTENSIONS__ -Xc -v -g -xO0
     6  CHECKIN = LOGNAME=- ci -m- -q -s- -t- -u -zLT ${1} ; \
     7            rcs -q -U ${1} ; chmod u+w ${1}
     8
     9  all : args argv
    10
    11  args : args.class
    12          echo Main-class: args >Manifest
    13          jar cvfm args Manifest args.class
    14          - rm Manifest
    15          chmod +x args
    16
    17  args.class : RCS args.java
    18          ${call CHECKIN, args.java}
    19          javac args.java
    20
    21  argv : argv.o
    22          cc ${CCOPT} argv.o -o argv
    23
    24  argv.o : RCS argv.c
    25          ${call CHECKIN, argv.c}
    26          cc ${CCOPT} -c argv.c
    27
    28  lint : argv.c
    29          lint ${LINTOPT} argv.c
    30
    31  ci : RCS ${SOURCES}
    32          ${call CHECKIN, ${SOURCES}}
    33
    34  RCS :
    35          mkdir RCS
    36
    37  ident : ${SOURCES}
    38          ident ${SOURCES}
    39
    40  clean :
    41          - rm args.class argv.o
```

```

spotless : clean
        - rm args argv

```

8.4 An example `.exrc`

Note the output of the invalid character checking shell command. As a special case, the **`set`** commands in the **`.exrc`** need to actually terminate with a carriage return character. In input mode, they can be typed in with **`^V^M`** (Ctrl/V, Ctrl/M). The leading quote (") is a comment marker. The command **`set showmatch`** is useful for typing in programs when you need to see matching parentheses, brackets, and braces as you type them in.

```
bash-% cat -nv examples/.exrc | expand
     1  " $Id: .exrc,v 1.1 2009-12-17 18:04:46-08 - - $
     2  "set autoindent
     3  "set modeline
     4  "set showmatch
     5  map ;; 1G
     6  map ;a :set all^M
     7  map ;f {!}fmt -c -65^M
     8  map ;g {!}fmt -c -50^M
     9  map ;i :set ignorecase^M
    10  map ;I :set noignorecase^M
    11  map ;M :set noshowmatch^M
    12  map ;m :set showmatch^M
    13  map ;N :set nonumber^M
    14  map ;n :set number^M
    15  map ;s {!}sort^M
    16  map ;S {!}sort -f^M
    17  map ;W :set wrapmargin=0^M
    18  map ;w :set wrapmargin=1^M
    19  set autoprint
    20  set ignorecase
    21  set noslowopen
    22  set number
    23  set report=1
    24  set shell=/bin/sh
    25  set showmode
    26  set wrapmargin=1
```

8.5 A format-checking script `check.sh`

Following is a script that performs basic format checking.

```
bash-% cat -nv examples/check.sh
     1	#!/bin/sh
     2	# $Id: check.sh,v 1.1 2009-12-17 18:04:46-08 - - $
     3	#
     4	# NAME
     5	#    check.sh - basic file format checks
     6	#
     7	# SYNOPSIS
     8	#    check.sh [filename...]
     9	#
    10	# DESCRIPTION
    11	#    Checks up on basic file formatting by expanding tabs to 8
    12	#    character-stops and checking for a line length of <= 72
    13	#    characters.  It prints out all lines longer than 72.
    14	#    It then checks for any invisible characters by comparing
    15	#    the visible display with the original file.  Tabs are
    16	#    noted, except for a Makefile.
    17	#
    18	
    19	for file in $*
    20	do
    21	   echo "********" testing $file
    22	   cat -n $file | expand | cut -c1-8,81- | grep .........
    23	   case $file in Makefile) opt=-v;; *) opt=-vt;; esac
    24	   cat $opt $file | diff - $file
    25	   echo ================================================================
    26	done
```

8.6 A file format fixing script `fixfile.sh`

A script that removes carriage returns, expands tabs, and adds a trailing newline if necessary.

```
bash-% cat -nv examples/fixfile.sh
     1	#!/bin/sh
     2	# $Id: fixfile.sh,v 1.1 2009-12-17 18:04:46-08 - - $
     3	#
     4	# NAME
     5	#    fixfile.sh - fix basic formatting in a text file
     6	#
     7	# SYNOPSIS
     8	#    fixfile.sh [filename...]
     9	#
    10	# DESCRIPTION
    11	#    Fixes basic formatting on a text file:
    12	#    Expands tabs to 8 spaces using expand(1).
    13	#    Deletes trailing carriage return characters.
    14	#    Adds a final newline if missing from the file.
    15	#
    16	# BUGS
    17	#    Does not fix lines longer than 72 characters.
    18	#
    19	
    20	for file in $* ; do
    21	   /usr/bin/echo ’%!expand\n%s/\r//\nw\nq’ | /usr/bin/ex $file
    22	done
```