Programming Style

· Writing good programs is like writing good prose; the object is to communicate concise, straightforward, no unnecessary parts.

· Principles of good programming style are language independent. Some languages have features that encourage good style, e.g., structured loops. Some have features that discourage good style, e.g., gotos, anemic data types. Modern block-structured languages are better than older unstructured languages. But bad programs can be written in any language.

· Benefits of good style:
  - Code that is easy to understand.
  - Code that is easy to modify and change.
  - Code that is easy to understand.
  - Better programs can be written in any language.

Writing good programs is like writing good prose. The object is to communicate.

Layout and Indentation

· Use white space judiciously.
  - Separate code into "paragraphs" and make expressions more readable.

· Use indentation to emphasize structure.
  - Use editor "autoindent" facilities and a consistent amount of space.
  - Watch for excessive indentation — signals excessive nesting.

· Line up parallel structures.
  - Alpha = angle(p1, p2, p3);
  - Beta = angle(p2, p3, p1);
  - Gamma = angle(p3, p1, p2);

· One statement per line.

· Be consistent, but use variation for emphasis.

· Break long lines at logical places, e.g., by precedence; indent continuations.

· Use tabular input and output formats.

Names

· Pick names that capture the use of the variable or function, e.g., addElement.

· Nouns for variables, verbs for functions, adjectives for booleans, conditions, and some enumeration constants.

· Use descriptive names for global variables and functions, e.g., external.

· Protect your code from local variables that reflect standard notation.

· Use descriptive names for global variables and functions, e.g., address.

· Write names that capture the use of the variable or function, e.g., address.

Clear Expression

· Compare:

  - for(i=1; i<=n; i++)
    for(j=1; j<=n; i++)
      v[i-1][j-1] = (i/j)*(j/i);

  - /* make v the identity matrix */
    for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
        v[i][j] = 0.0;
      v[i][i] = 1.0;

· Rules:
  - Be clever, but don't be too clever.
  - Say what you mean, simply and directly.
  - Use parentheses to emphasize precedence and braces to display structure.
  - Use white space and indentation to display structure.

Programmable Style

· Don't use esoteric abbreviations and acronyms.

· Module-level parameters help distinguish names. e.g., strset-addr.

· Use a consistent style for compound names.

· Don't rely on only case to distinguish names.

· Use all capitals for constants.

· Use variation to indicate differences.

· Break long lines at logical places, e.g., by precedence; indent continuations.

· Use tabular input and output formats.
Control Structure

- Flow of control should be written for human understanding.

Rules:
- Avoid double negation.
- Avoid temporary variables.
- Use library functions.
- Let the compiler do the dirty work.

- Avoid the use of `return` and `break`.
- Avoid complicated nested blocks.
- Avoid long blocks.
- Don't make the reader jump around or decrypt control flow of control.
- Use structured control constructs.

Clear Expression, cont'd

- Rules:
  - Lay out expressions according to standard conventions.
  - Rearrange logic so it is easy to understand.
  - Follow each decision with a matching action.
  - Handle default action, even if it can't happen.

Rules:
- Avoid `continue` and `break`.
- Favor `if ... else if ... else`.
- Emphasize that only one of the actions is performed.
- Implement multiway branches with `if ... else if ... else`.
- Avoid empty `then` and `else` actions.
- Handle default action, even if it can't happen.

Control Structure, cont'd

- Rules:
  - Use library functions.
  - Avoid empty `then` and `else` actions.
  - Favor `if ... else if ... else`.
  - Emphasize that only one of the actions is performed.

Clear Expression, cont'd

- Rules:
  - Lay out expressions according to standard conventions.
  - Rearrange logic so it is easy to understand.
  - Avoid long blocks.
  - Avoid complicated, nested blocks.
  - Minimize the use of `return` and `break`.

Control Structure, cont'd

- Rules:
  - Use library functions.
  - Avoid empty `then` and `else` actions.
  - Favor `if ... else if ... else`.
  - Emphasize that only one of the actions is performed.

Clear Expression, cont'd

- Rules:
  - Lay out expressions according to standard conventions.
  - Rearrange logic so it is easy to understand.
  - Avoid long blocks.
  - Avoid complicated, nested blocks.
  - Minimize the use of `return` and `break`.

Program Structure

- Rules:
  - Modularize; use interfaces
  - Every function/interface should do one thing well
  - Every function/interface should hide something
  - Replace repetitious code with calls to functions
  - Let the data structure the program
  - Make sure your code "does nothing" gracefully
  - Don't patch bad code — rewrite it
  - Don't strain to reuse code — reorganize it
  - Watch for "off-by-one" errors

Documentation

- Best program documentation includes:
  - Clean structure
  - Consistent programming
  - Good mnemonic identifiers
  - Smattering of enlightening comments

- Comments should add new information
  - i = i + 1; /* add one to i */

- Comments and code must agree; if they disagree, odds are they are both wrong
  - Comments and code should "speak the same language"
  - Don't comment bad code — rewrite it

- Comment algorithms, not coding idiosyncracies
  - Comment procedural interfaces and data structures liberally

- Master the language and its idioms; let the code speak for itself
  - "Arrays are more general than lists; they are just a list with each entry having a name."

Program Organization

- Good, consistent organization makes programs easier to read and modify
- Pick a consistent program layout style for:
  - Functions
  - Statements
  - Expressions
  - Comments
  - #includes (i.e., imports)
  - #defines (i.e., constants)
  - Type definitions (e.g., typedef, struct, etc.)
  - Global variables
  - Main
  - Functions in alphabetical or logical order

- Divide medium-size programs (~ few thousand lines, maximum) into modules
- Use established interfaces and implementations
- Implementations organized around data or function
- Organize each implementation as a "small" program
- Use separate headers for separate implementations but don't "over-modularize"
- Group related headers in namespaces

- What about large programs, say, more than 50,000 lines? Another course...
If a program doesn't work, it doesn't matter how fast it is! Make performance improvements only if they are really needed, and if there are objective measurements that identify the sources of inefficiency. Intuitions are notoriously bad; they aren't objective measurements. Keep it simple to make it faster. Let the compiler do the simple optimizations. Don't diddle code to make it faster — find a better algorithm.

Rules:

· If a program doesn't work, it doesn't matter how fast it is!

· Make it clear before you make it faster.

· Make it correct before you make it faster.

· Make performance improvements only if they are really needed, and if there are objective measurements that identify the sources of inefficiency.

· Intuitions are notoriously bad; they aren't "objective measurements."

· Keep it simple to make it faster.

· Let the compiler do the simple optimizations.

· Don't diddle code to make it faster — find a better algorithm.

· Intuitions are notoriously bad; they aren't "objective measurements."

· Keep it simple to make it faster.

· Make performance improvements only if they are really needed, and if there are objective measurements that identify the sources of inefficiency.

· Intuitions are notoriously bad; they aren't "objective measurements."

· Keep it simple to make it faster.