Programming Style

- Writing good programs is like writing good prose; the object is to \textit{communicate} concise, straightforward, no unnecessary parts

- Principles of good programming style are \textit{language independent}
  some languages have features that \textit{encourage} good style, e.g. structured loops
  some have features that \textit{discourage} good style, e.g. gotos, anemic data types
  modern block-structured languages are better than older unstructured languages
  but \textit{bad} programs can be written in \textit{any} language

- Benefits of good style
  code that is easy to \textit{understand}
  code that is more likely to \textit{work}
  code that is easy to \textit{maintain} and change

- Method to develop good programming style
  \textit{read} code written by good programmers
  Ask: Will I understand this program in two years?
Names

• Pick names that capture the use of the variable or function, e.g. `addElement`
  - nouns for variables
  - verbs for functions
  - adjectives for boolean, conditions, and some enumeration constants
• Use **descriptive** names for global variables and functions, e.g. `elementCount`
• Use **concise** names for local variables that reflect **standard notation**
  
  prefer to
  
  ```
  for (i = 0; i < n; i++)
  a[i] = 0;
  ```
  ```
  for (arrayindex = 0; arrayindex < arraysize; arrayindex++)
  array[arrayindex] = 0;
  ```
• Use **case** judiciously
  - use all capitals for constants
  - don’t rely on only case to distinguish names
• Use a consistent style for **compound** names
  ```
  printword
  PrintWord
  print_word
  ```
• Module-level prefixes help distinguish names, e.g. `Strset_T, Strset_add`
• Don’t use nerdy abbreviations and acronyms
Layout and Indentation

• Use *white space* judiciously
  separate code into “paragraphs”
  make expressions more readable

• Use *indentation* to emphasize *structure*
  use editor “autoindent” facilities and a consistent amount of space
  watch for extreme 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
Clear Expression

• Compare:

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

vs.

```c
/* 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 clarify structure
  - don’t sacrifice clarity for “efficiency”
Clear Expression, cont’d

• Compare:
  
  \[
  \text{if } (\neg (i > 10 \lor 0 > i)) \ldots \quad \text{vs.} \quad \text{if } (0 \leq i \land i \leq 10) \ldots
  \]

• Compare:
  
  \[
  \text{for } (neg = 0; \,*s1 == \,*s2++; )
  \]
  \[
  \quad \text{if } (*s1++ == '\0')
  \]
  \[
  \quad \quad \text{break;}
  \]
  \[
  \quad neg = \,*s1 - (**-s2);
  \]
  \[
  \quad \text{if } (!neg) \ldots
  \]

  \[
  \quad \text{vs.}
  \]
  \[
  \text{while } (*s1 == *s2 \land *s1 != '\0') \{ s1++; s2++; \}
  \]
  \[
  \quad \text{if } (*s1 == *s2) \ldots
  \]

  \[
  \quad \text{vs.}
  \]
  \[
  \text{if } (\text{strcmp}(s1, s2) == 0) \ldots
  \]

• Rules:
  
  avoid double negation
  avoid temporary variables
  use library functions
  let the compiler do the dirty work
Clear Expression, cont’d

• Compare:

```plaintext
if (a > b)
    if (b > c)
        z = c;
    else
        z = b;
else
    if (a > c)
        z = c;
    else
        z = a;
```

```plaintext
vs.

if (a < b)
    if (b <= c)
        z = a;
    else
        z = c;
else /* a >= b */
    if (b <= c)
        z = b
    else
        z = c;
```

better yet:

```plaintext
z = min(a, min(b, c));
```

• Rules:

  lay out expressions according to standard conventions
  rearrange logic so it is easy to understand
  follow each decision with a matching action
Control Structure

- Flow of control should be written for human understanding

```c
for (i = 0; i < n; i++) {
    if (strcmp(table[i].word, word))
        continue;
    table[i].count++;
}
```

Better:

```c
for (i = 0; i < n; i++)
    if (strcmp(table[i].word, word) == 0)
        table[i].count++;  
```

- Avoid `continue; break` is OK, but use it sparingly; “breaking” out of functions is OK, if used with care

```c
func(a) {
    if (isbad(a))
        return;
    ...
}
```

- Rules:
  - use structured control constructs
  - don’t make the reader jump around or decrypt convoluted flow of control
  - avoid long blocks
  - avoid complicated, nested blocks
  - minimize the use of `return` and `break`
• “Comb” structures

\[
\begin{align*}
\text{compare:} & \\
\text{vs:} & \\
\text{if } (x < v[mid]) & \\
& \quad \text{high} = \text{mid} - 1; \\
\text{else if } (x > v[mid]) & \\
& \quad \text{low} = \text{mid} + 1; \\
\text{else} & \\
& \quad \text{return mid;} \\
\text{if } (x < v[mid]) & \\
& \quad \text{high} = \text{mid} - 1; \\
\text{else if } (x > v[mid]) & \\
& \quad \text{low} = \text{mid} + 1; \\
\text{else} & \\
& \quad \text{return mid;}
\end{align*}
\]

• Ditto for \textit{switch}

• Rules:

- implement multiway branches with \textit{if ... else if ... else}
- emphasize that only one of the actions is performed
- avoid empty \textit{then} and \textit{else} actions
- handle default action, even if it “can’t happen;” use \texttt{assert(0)}
- avoid nesting
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
  \[ i = i + 1; \quad /* \text{add one to } i */ \]
• Comments and code must agree; if they disagree, odds are they are both wrong
• 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
Program Organization

• Good, consistent organization makes programs easier to read and modify

• Pick a consistent program layout style for
  
  - functions
  - statements
  - expressions
  - comments

• **Small** programs (~ few **hundred** lines, maximum) can fit into one file
  
  - opening explanatory comments
  - purpose
  - author and history (handled better by tools like RCS)
  - `#includes` (i.e. imports)
  - `#defines` (i.e. constants)
  - type definitions (e.g. `typedef`, `struct`, etc.)
  - global variables
  - `main`
  - functions in alphabetical or logical order

• Maximize “data ink”
Program Organization, cont’d

- 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
- Interfaces
  use separate headers for separate interfaces, but don’t over-modularize
  permit multiple inclusion
  do not define variables
- Global variables and functions
  declared in interfaces, so all clients see the same declaration
  defined and initialized in an implementation
- What about large programs, say, more than 50,000 lines? Another course...
Efficiency and Style

• If a program doesn’t work, it doesn’t matter how fast it is!

• Rules:
  make it clear before you make it faster
  make it correct before you make it faster
  see if it’s fast enough before you make it faster
  keep it correct while you make it faster
  ill-conceived attempts to increase efficiency usually lead to bad code; gains are usually small or non-existent

• 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”

• Rules:
  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