**Program and Programming Style**

The material for this lecture is drawn, in part, from *The Practice of Programming* (Kernighan & Pike) Chapter 1.

---

**Motivation for Program Style**

**Who reads your code?**

- The compiler
- Other programmers

This is a working ray tracer! (courtesy of Paul Heckbert)

---

**For Your Amusement**

Any fool can write code that a computer can understand. Good programmers write code that humans can understand. -- Martin Fowler

Good code is its own best documentation. As you’re about to add a comment, ask yourself, “How can I improve the code so that this comment isn’t needed?” -- Steve McConnell

Programs must be written for people to read, and only incidentally for machines to execute. -- Abelson / Sussman

---

**Goals of this Lecture**

Help you learn about:

- Good program style
- Good programming style

Why?

- A well-styled program is more likely to be correct than a poorly-styled program
- A well-styled program is more likely to stay correct (i.e. is more maintainable) than a poorly-styled program
- A power programmer knows the qualities of a well-styled program, and how to compose one quickly

---

**Agenda**

Program style

- Qualities of a good program

Programming style

- How to write a good program quickly

---

**Motivation for Program Style**

Why does program style matter?

- Correctness
  - The clearer a program is, the more likely it is to be correct
- Maintainability
  - The cleaner a program is, the more likely it is to stay correct over time

Good program ≈ clear program
Choosing Names

Use descriptive names for globals and functions
  • E.g., display, CONTROL, CAPACITY
Use concise names for local variables
  • E.g., i (not arrayIndex) for loop variable
Use case judiciously
  • E.g., Stack_push (Module function)  
    CAPACITY (constant)  
    buf (local variable)
Use a consistent style for compound names
  • E.g., frontsize, frontSize, front_size
Use active names for functions that do something
  • E.g., getchar(), putchar(), Check_octal(), etc.
  Not necessarily for functions that are something: sin(), sqrt()

Using C Idioms

Use C idioms
  • Example: Set each array element to 1.0.
  • Bad code (complex for no obvious gain)
  • Good code (not because it’s vastly simpler— it isn’t!— but because it uses a standard idiom that programmers can grasp at a glance)

Don’t feel obliged to use C idioms that decrease clarity

Revealing Structure: Expressions

Parenthesize for correctness and to resolve ambiguity
  • Example: read and print character until end-of-file
  • Broken code

      while (c = getchar() != EOF)
          putchar(c);

  • Working, idiomatic code

      while ((c = getchar()) != EOF)
          putchar(c);

iClicker Question

Q: Does the following code work to check if integer \( n \) satisfies \( j < n < k \)?

A. No, needs to be \( \text{if} \ (j < n \ && \ n < k) \)

B. Correct, but I’d leave it alone

C. Correct, and I’d never write such a monstrosity
Revealing Structure: Expressions

Use natural form of expressions
- Example: Check if integer n satisfies \( j < n < k \)
  - Bad code
    ```
    if (!((n >= k) || (n <= j)))
    ```
  - Good code
    ```
    if ((j < n) && (n < k))
    ```
  - Conditions should read as you’d say them aloud
    - Not “Conditions shouldn’t read as you’d never say them in other than a purely internal dialog!”

Revealing Structure: Expressions

Break up complex expressions
- Example: Identify chars corresponding to months of year
  - Bad code
    ```
    if (c == 'J') || (c == 'F') || (c == 'M') || (c == 'A') || (c == 'S') || (c == 'O') || (c == 'N') || (c == 'D'))
    ```
  - Good code – lining up things helps
    ```
    if (c == 'J' || (c == 'F') || (c == 'M') || (c == 'A') || (c == 'S') || (c == 'O') || (c == 'N') || (c == 'D'))
    ```
  - Very common, though, to elide parentheses
    ```
    if (c == 'J' || c == 'F' || c == 'M' || c == 'A' || c == 'S' || c == 'O' || c == 'N' || c == 'D')
    ```

Revealing Structure: Spacing

Use readable/consistent spacing
- Example: Assign each array element \( a[j] \) to the value \( j \)
  - Bad code
  ```
  for (j=0;j<100;j++) a[j]=j;
  ```
  - Good code
  ```
  for (j = 0; j < 100; j++)
    a[j] = j;
  ```

Revealing Structure: Indentation

Use readable/consistent/correct indentation
- Example: Checking for leap year (does Feb 29 exist?)
  - Bad code
    ```
    if (month == FEB)
      if ((year % 4) == 0)
        if (day > 29)
          legal = FALSE;
        else
          if (day > 28)
            legal = FALSE;
      else
        legal = TRUE;

    else
      legal = FALSE;
    ```
  - Good code
    ```
    legal = TRUE;
    if (month == FEB)
      if ((year % 4) == 0)
        if (day > 29)
          legal = FALSE;
        else
          if (day > 28)
            legal = FALSE;
      else
        legal = TRUE;
    ```
    ```
    if (month == FEB)
      if ((year % 4) == 0)
        low = mid + 1;
      else
        return mid;
    ```
  - Often can rely on auto-indenting feature in editor

Revealing Structure: Indentation

Use "else-if" for multi-way decision structures
- Example: Comparison step in a binary search.
  - Bad code
    ```
    if (x < a[mid])
      high = mid - 1;
      else
        if (x > a[mid])
          low = mid + 1;
        else
          return mid;
    ```
  - Good code
    ```
    if (x < a[mid])
      high = mid - 1;
    else
      if (x > a[mid])
        low = mid + 1;
      else
        return mid;
    ```
Revealing Structure: “Paragraphs”

Use blank lines to divide the code into key parts

```c
#include <stdio.h>
#include <stdlib.h>
/* Read a circle’s radius from stdin, and compute and write its
diameter and circumference to stdout. Return 0 if successful. */
int main(void)
{  const double PI = 3.14159;
   int radius;
   int diam;
   double circum;
   printf("Enter the circle's radius:\n");
   if (scanf("%d", &radius) != 1)
   {  fprintf(stderr, "Error: Not a number\n");
      exit(EXIT_FAILURE);  /* or:  return EXIT_FAILURE; */
   }
   ...
   diam = 2 * radius;
   circum = PI * (double)diam;
   printf("A circle with radius %d has diameter %d\n", radius, diam);
   printf("and circumference %f.\n", circum);
   return 0;
}
```

Composing Comments

Master the language and its idioms
- Let the code speak for itself
- And then...

Compose comments that add new information

```c
int i;  /* Add one to i. */
```

Comment paragraphs of code, not lines of code
- E.g., “Sort array in ascending order”

Comment global data
- Global variables, structure type definitions, field definitions, etc.

Compose comments that agree with the code!!!
- And change as the code itself changes!!!

Composing Function Comments

Describe what a caller needs to know to call the function properly
- Describe what the function does, not how it works
- Code itself should clearly reveal how it works...
- If not, compose “paragraph” comments within definition

Describe input
- Parameters, files read, global variables used

Describe output
- Return value, parameters, files written, global variables affected

Refer to parameters by name
Composing Function Comments

Bad function comment

```c
/* decomment.c */
/* Read a character. Based upon the character and 
the current DFA state, call the appropriate 
state-handling function. Repeat until 
end-of-file. */
int main(void)
{
    ...
}
```

Describes how the function works

Good function comment

```c
/* decomment.c */
/* Read a C program from stdin. Write it to 
stdout with each comment replaced by a single 
space. Preserve line numbers. Return 0 if 
successful, EXIT_FAILURE if not. */
int main(void)
{
    ...
}
```

Describes what the function does

Using Modularity

Abstraction is the key to managing complexity
- Abstraction is a tool (the only one?) that people use to 
understand complex systems
- Abstraction allows people to know what a (sub)system does 
without knowing how

Proper modularity is the manifestation of abstraction
- Proper modularity makes a program's abstractions explicit
- Proper modularity can dramatically increase clarity
  ⇒ Programs should be modular

However
- Excessive modularity can decrease clarity!
- Improper modularity can dramatically decrease clarity!!!
  ⇒ Programming is an art

Modularity Examples

Examples of function-level modularity
- Character I/O functions such as `getchar()` and `putchar()`
- Mathematical functions such as `sin()` and `gcd()`
- Function to sort an array of integers

Examples of file-level modularity
- Assignment 3.
- And all the other assignments.

Program Style Summary

Good program = clear program

Qualities of a clear program
- Uses appropriate names
- Uses common idioms
- Reveals program structure
- Contains proper comments
- Is modular

Agenda

Program style
- Qualities of a good program

Programming style
- How to write a good program quickly
**Bottom-Up Design**

**Bottom-up design**
- Design one part of the system in detail
- Design another part of the system in detail
- Combine
- Repeat until finished

**Bottom-up design in painting**
- Paint part of painting in complete detail
- Paint another part of painting in complete detail
- Combine
- Repeat until finished
- Unlikely to produce a good painting (except sometimes: see the movie "Tim's Vermeer")

**Top-Down Design**

**Top-down design**
- Design entire product with minimal detail
- Successively refine until finished

**Top-down design in painting**
- Sketch the entire painting with minimal detail
- Successively refine until finished

**Top-Down Design in Reality**

**Top-down design in programming in reality**
- Define main() function in pseudocode
- Refine each pseudocode statement
  - **Oops!** Details reveal design error, so...
  - Backtrack to refine existing (pseudo)code, and proceed
  - Repeat in (mostly) breadth-first order until finished

**Aside: Least-Risk Design**

**Design process should minimize risk**

**Bottom-up design**
- Compose each module before its children
  - **Risk level:** high
    - May compose modules that are never used

**Top-down design**
- Compose each parent module before its children
  - **Risk level:** low
    - Compose only those modules that are required
Aside: Least-Risk Design

Recommendation
- Work mostly top-down
- But give high priority to risky modules (that may result in major rewrites)
- Create scaffolds and stubs as required

Example: Text Formatting

Functionality (derived from King Section 15.3)
- Input: ASCII text, with arbitrary spaces and newlines
- Output: the same text, left and right justified
- Fit as many words as possible on each 50-character line
- Add even spacing between words to right justify the text
- No need to right justify last line

Assumptions
- “Word” is a sequence of non-white-space chars followed by a white-space char or end-of-file
- No word is longer than 20 chars

Example Input and Output

C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.”

C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments.”

--- Dennis Ritchie

Caveats

Caveats concerning the following presentation
- Function comments and some blank lines are omitted
- Because of space constraints
- Don’t do that!!!
- Design sequence is idealized
- In reality, typically much backtracking would occur

The main() Function

int main(void)
{
    char word[MAX_WORD_LEN+1];
    int wordLen;
    <clear line>
    wordLen = readWord(word);
    while (<there is a word>)
    {
        if (<word doesn’t fit on line>)
        {
            <write justified line>
            <clear line>
        }
        <add word to line>
        <read a word>
    }
    if (<line isn’t empty>)
    <write line>
    return 0;
}
enum {MAX_WORD_LEN = 20};
enum {MAX_LINE_LEN = 50};
int main(void)
{
    char word[MAX_WORD_LEN+1];
    int wordLen;
    int lineLen = 0;
    int wordCount = 0;
    <clear line>
    wordLen = readWord(word);
    while (wordLen != 0)
    {
        if (word doesn’t fit on line)
        {
            <write justified line>
            <clear line>
            lineLen = addWord(word, line, lineLen);
            wordLen = readWord(word);
        } else
        {
            <add word to line>
            <write line>
        }
        if (lineLen > 0)
        {
            <write line>
            puts(line);
            return 0;
        }
    }
}
The main() Function

```c
enum {MAX_WORD_LEN = 20};
enum {MAX_LINE_LEN = 50};
int main(void)
{
    char word[MAX_WORD_LEN + 1];
    char line[MAX_LINE_LEN + 1];
    int wordLen;
    int lineLen = 0;
    int wordCount = 0;
    line[0] = '
0';
    while (wordLen != 0)
    {
        if ((wordLen + 1 + lineLen) > MAX_LINE_LEN)
            writeLine(line, lineLen, wordCount);
        line[0] = '
0';
        lineLen = addWord(word, line, lineLen);
        wordLen = readWord(word);
        if (lineLen > 0)
            return 0;
    }
    if (lineLen > 0)
        puts(line);
    return 0;
}
```

The readWord() Function

```c
int readWord(char *word)
{
    int ch;
    /* Skip over white space. */
    ch = getchar();
    while ((ch != EOF) && isspace(ch))
        ch = getchar();
    /* Read up to MAX_WORD_LEN chars into word. */
    while ((ch != EOF) && (! isspace(ch)))
    {
        if (pos < MAX_WORD_LEN)
            word[pos] = (char)ch;
        pos++;
        ch = getchar();
    }
    word[pos] = '\0';
    return pos;
}
```

Note the use of a function from the standard library. Very appropriate for your top-down design to target things that are already built.
The `addWord()` Function

```c
int addWord(const char *word, char *line, int lineLen)
{
    /* If line already contains some words, then append a space. */
    if (lineLen > 0)
    {  strcat(line, " ");
        lineLen++;
    }
    strcat(line, word);
    lineLen += strlen(word);
    return lineLen;
}
```

Status

```
main
readWord
writeLine
addWord
```
The writeLine() Function

```c
void writeLine(const char *line, int lineLen, int wordCount)
{
    int i, extraSpaces;
    /* Compute number of excess spaces for line. */
    extraSpaces = MAX_LINE_LEN - lineLen;
    for (i = 0; i < lineLen; i++)
    {
        if (line[i] == ' ')
            putchar(line[i]);
        else
        {
            /* Compute additional spaces to insert. */
            spacesToInsert = extraSpaces / (wordCount - 1);
            /* Print a space, plus additional spaces. */
            for (j = 1; j <= spacesToInsert + 1; j++)
                putchar(' '); // Decrease extra spaces and word count.
            extraSpaces -= spacesToInsert;
        }
    }
    putchar('
');
}
```

Example:
If extraSpaces is 10 and wordCount is 5, then gaps will contain 2, 2, 3, and 3 extra spaces respectively.

Status

```
main
readWord
writeLine
addWord
```

Complete! And modular!
Summary

Program style
- Choose appropriate names (for variables, functions, …)
- Use common idioms (but not at the expense of clarity)
- Reveal program structure (spacing, indentation, parentheses, …)
- Compose proper comments (especially for functions)
- Use modularity (because modularity reveals abstractions)

Programming style
- Use top-down design and successive refinement
- But know that backtracking inevitably will occur
- And give high priority to risky modules

Are we there yet?

Now that the top-down design is done, and the program "works," does that mean we’re done?

No. There are almost always things to improve, perhaps by a bottom-up pass that better uses existing libraries.

The second time you write the same program, it turns out better.

What’s wrong with this output?

"C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments."

Better

"C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments."

Better

Better

What’s better with this output?

"C is quirky, flawed, and an enormous success. While accidents of history surely helped, it evidently satisfied a need for a system implementation language efficient enough to displace assembly language, yet sufficiently abstract and fluent to describe algorithms and interactions in a wide variety of environments."

-- Dennis Ritchie

Adequate

Adequate

Adequate

“Programming in the Large” Steps

Design & implement
- Program & programming style
- Common data structures and algorithms
- Modularity
- Building techniques & tools

Debug
- Debugging techniques & tools

Test
- Testing techniques & tools

Maintain
- Performance improvement techniques & tools

Challenge problem

Design a function int spacesHere(int i, int k, int n)
that calculates how many marbles to put into the i-th jar, assuming that there are n marbles to distribute over k jars.

(1) the jars should add up to n, that is, (n=0; for(i=0;i<k;i++) e+spacesHere(i,k,n); assert (n==n);)
or in math notation, $\sum_{i=0}^{k-1} spacesHere(i,k,n) = n$

(2) marbles should be distributed evenly—the "extra" marbles should not bunch up in nearby jars.

HINT: You should be able to write this in one or two lines. Without any loops.

One solution uses floating-point division and rounding: do "man round" and pay attention to where that man page says "include "math.h"."
# Appendix: The “justify” Program

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

enum {MAX_WORD_LEN = 20};
enum {MAX_LINE_LEN = 50};

Continued on next slide
```

Continued on next slide