Program and Programming Style

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

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

"Everything should be built top-down, except the first time.” - Alan Perlis

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 compose a good program quickly

Motivation for Program Style

Who reads your code?

- The compiler
- Other programmers

This is a working ray tracer! (courtesy of Paul Heckbert)
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 clearer a program is, the more likely it is to stay correct over time

Good program = clear program

Program Style Outline

Good program = clear program
Qualities of a clear program
• Uses appropriate names: descriptive, concise for local variables, case, consistent for compound names, active names for functions
• Uses common idioms
• Reveals program structure (natural expressions, parenthesis, breaking complex expressions, spacing, indentation, paragraphs using blank lines)
• Contains proper comments (function comments describe what, not how; refer to parameters by name/type, and describe return value)
  • Is modular

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(), sort()

Using C Idioms

Use C idioms
• Example: Set each array element to 1.0.
  • Bad code (complex for no obvious gain)
    \[
    i = 0;
    \text{while}(i \leq n - 1) \quad \text{array}[i++] = 1.0;
    \]
  • Good code (not because it's vastly simpler—tiny!—but because it uses a standard idiom that programmers can grasp at a glance)
    \[
    \text{for } (i=0; i<n; i++) \quad \text{array}[i] = 1.0;
    \]
  • Don't feel obliged to use C idioms that decrease clarity

Revealing Structure: Expressions

Use natural form of expressions
• Example: Check if integer \( n \) satisfies \( j < n < k \)
  • Bad code
    \[
    \text{if }(!(n \geq k) \&\& !(n \leq j))
    \]
  • Good code
    \[
    \text{if }((j < n) \&\& (n < k))
    \]
  • 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

Parenthesize to resolve ambiguity
• Example: Check if integer \( n \) satisfies \( j < n < k \)
  • Common code
    \[
    \text{if } (j < n \&\& n < k)
    \]
  • Clearer code (maybe)
    \[
    \text{if } ((j < n) \&\& (n < k))
    \]

Does the code work?

It's clearer depending on whether your audience can be trusted to know the precedence of all the C operators. Use your judgment on this.
Revealing Structure: Expressions

Parenthesize to resolve ambiguity (cont.)

• Bad code

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

• Good-ish code

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

• (Code with side effects inside expressions is never truly “good”,
but at least this code is a standard idiomatic way to write it in C)

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

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;

• Often can rely on auto-indenting feature in editor

Revealing Structure: Indentation

Use readable/consistent/correct indentation

• Example: Checking for leap year (does Feb 29 exist?)

Legal = TRUE;
if (month == FEB)
{  if ((year % 4) == 0)
    if (day > 29)
      legal = FALSE;
  else
    if (day > 28)
      legal = FALSE;
}

• Use "else-if" for multi-way decision structures

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:

    if (scanf("%d", &radius) != 1)
    {  fprintf(stderr, "Error: Not a number

        exit(EXIT_FAILURE);  /* or:  return EXIT_FAILURE; */
    }
    ...
    diam = 2 * radius;
    circum = PI * (double)diam;
    printf("A circle with radius %d has diameter %d

        and circumference %f.

    return 0;
```
Composing Function Comments

Bad function comment

/* 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

/* 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

• (See subsequent lectures)

Program Style Summary

Good program = clear program

Qualities of a clear program

• Chooses appropriate names (for variables, functions, …)
• Uses common idioms (but not at the expense of clarity)
• Reveals program structure (spacing, indentation, parentheses, …)
• Composes proper comments (especially for functions)
• Uses modularity (because modularity reveals abstractions)

Agenda

Program style

• Qualities of a good program

Programming style

• How to compose 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

**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

---

**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 with minimal detail
- Refine each pseudocode statement
- Small job ⇒ replace with real code
- Large job ⇒ replace with function call
- Repeat in (mostly) breadth-first order until finished

- Bonus: Product is naturally modular

---

**Bottom-Up Design**

**Bottom-up design in programming**
- Compose part of program in complete detail
- Compose another part of program in complete detail
- Combine
- Repeat until finished
- Unlikely to produce a good program
"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

Example Input and Output

Input
"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

Output

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

```c
enum {MAX_WORD_LEN = 20};
int main(void)
{
    char word[MAX_WORD_LEN+1];
    int wordLen;
    <clear line>
    wordLen = readWord(word);
    while (wordLen != 0)
    {
        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;
    }
}
```

The main() Function

```c
enum {MAX_WORD_LEN = 20};
int main(void)
{
    char word[MAX_WORD_LEN+1];
    int wordLen;
    <clear line>
    wordLen = readWord(word);
    while (wordLen != 0)
    {
        if (word doesn’t fit on line)
        {
            <write justified line>
            <clear line>
        }
        <add word to line>
        wordLen = readWord(word);
    }
    if (<line isn’t empty>)
        <write 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;
  <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);
    }
    if (lineLen > 0)
      <write line>
      return 0;
  }
  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;
  <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);
    }
    if (lineLen > 0)
      <write line>
      return 0;
  }
  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;
  <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);
    }
    if (lineLen > 0)
      <write line>
      return 0;
  }
  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'; lineLen = 0; wordCount = 0;
  wordLen = readWord(word);
  while (wordLen != 0)
  {  if ((wordLen + 1 + lineLen) > MAX_LINE_LEN)
    {  writeLine(line, lineLen, wordCount);
      line[0] = '\0'; lineLen = 0; wordCount = 0;
    }
    lineLen = addWord(word, line, lineLen);
    wordLen = readWord(word);
  }
  if (lineLen > 0)
    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'; lineLen = 0; wordCount = 0;
  wordLen = readWord(word);
  while (wordLen != 0)
  {  if ((wordLen + 1 + lineLen) > MAX_LINE_LEN)
    {  writeLine(line, lineLen, wordCount);
      line[0] = '\0'; lineLen = 0; wordCount = 0;
    }
    lineLen = addWord(word, line, lineLen);
    wordLen = readWord(word);
  }
  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 in targeting things that are already built.

---

### Status

```
main
  |  readWord
  |   writeLine
  |      addWord
```

### The addWord() Function

```c
int addWord(const char *word, char *line, int lineLen) {
    if (line already contains words, then append a space)
        append word to line
    return the new line length
}
```
The addWord() Function

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

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. */
            extraSpaces += wordCount;
        }
    }
    putchar(' 
');
}
```
The writeLine() Function

```c
void writeLine(const char *line, int lineLen, int wordCount)
{
    int i, extraSpaces, spacesToInsert, j;
    /* 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;
            wordCount--;
        }
    }
    putchar('
');
}
```

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

Top-Down Design and Modularity

Note: Top-down design naturally yields modular code

Much more on modularity in upcoming lectures

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 (see Appendix)
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 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

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

Challenge problem

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

(1) The jars should add up to $n$, that is, $\sum_{i=0}^{k-1} \text{spacesHere}(i,k,n) = n$
(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.

My solution used floating point division and rounding; do “man round” and pay attention to what the 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};

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

    /* Return length of word. */
    return pos;
}
```

Appendix: The “justify” Program

```c
/* Read a word from stdin. Assign it to word. Return the length of the word, or 0 if no word could be read. */
int readWord(char *word)
{  int ch, pos = 0;
    /* Skip over white space. */
    ch = getchar();
    while ((ch != EOF) && isspace(ch))
      ch = getchar();
    /* Store chars up to MAX_WORD_LEN in word. */
    while ((ch != EOF) && (! isspace(ch))
      {  if (pos < MAX_WORD_LEN)
          {  word[pos] = (char)ch;
              pos++;
          }
      ch = getchar();
    }
    word[pos] = '\0';
    /* Return length of word. */
    return pos;
}
```
Appendix: The “justify” Program

```c
/* Append word to line, making sure that the words within line are separated with spaces. lineLen is the current line length. Return the new line length. */
int addWord(const char *word, char *line, int lineLen)
{
    int newLineLen = lineLen;
    /* If line already contains some words, then append a space. */
    if (newLineLen > 0)
    {  strcat(line, " ");
       newLineLen++;
    }
    strcat(line, word);
    newLineLen += strlen(word);
    return newLineLen;
}
Continued on next slide
```

Appendix: The “justify” Program

```c
/* Write line to stdout, in right justified form. lineLen indicates the number of characters in line. wordCount indicates the number of words in line. */
void writeLine(const char *line, int lineLen, int wordCount)
{
    int extraSpaces, spacesToInsert, i, j;
    /* 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;
             wordCount--;
             }  
    putchar('n');
}
Continued on next slide
```

Appendix: The “justify” Program

```c
/* Read words from stdin, and write the words in justified format to stdout. Return 0. */
int main(void)
{
    /* Simplifying assumptions:
       Each word ends with a space, tab, newline, or end-of-file.
       No word is longer than MAX_WORD_LEN characters. */
    char word[MAX_WORD_LEN + 1];
    char line[MAX_LINE_LEN + 1];
    int wordLen;
    int lineLen = 0;
    int wordCount = 0;
    line[0] = '\0'; lineLen = 0; wordCount = 0;
    ...
    wordLen = readWord(word);
    while ((wordLen != 0)
    {  /* If word doesn't fit on this line, then write this line */
       if ((wordLen + 1 + lineLen) > MAX_LINE_LEN)
       {  writeLine(line, lineLen, wordCount);
          line[0] = '\0'; lineLen = 0; wordCount = 0;
       }
       lineLen = addWord(word, line, lineLen);
       wordCount++;
       wordLen = readWord(word);
    }
    if (lineLen > 0)
        puts(line);
    return 0;
}
```

Aside: Least-Risk Design

Design process should minimize risk

Bottom-up design

- Compose each child module before its parent
- 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

Least-risk design

- The module to be composed next is the one that has the most risk
- The module to be composed next is the one that, if problematic, will require redesign of the greatest number of modules
- The module to be composed next is the one that poses the least risk of needing to redesign other modules
- The module to be composed next is the one that poses the least risk to the system as a whole
- Risk level: minimal (by definition)
Aside: Least-Risk Design

Recommendation
- Work mostly top-down
- But give high priority to risky modules
- Create scaffolds and stubs as required