Program and Programming Style

The material for this lecture is drawn, in part, from *The Practice of Programming* (Kernighan & Pike) Chapter 1
“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
“Programming in the Large” Steps

Design & Implement
- Program & programming style <-- we are here
- Common data structures and algorithms
- Modularity
- Building techniques & tools (done)

Debug
- Debugging techniques & tools

Test
- Testing techniques (done)

Maintain
- Performance improvement techniques & tools
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

```c
typedef struct{double x,y,z}vec;vec U,black,amb={.02,.02,.02};struct sphere{ vec cen,color;double rad,kd,ks,kt,kl,ir}*s,*best,sph[]={0.,6.,.5,1.,1.,1.,.9,.05,.2,.85,0.,1.7,-1.,8.,-.5,1.,5.,2.1.,7.,3.0,.05,1.2,1.,8.,-.5,1.,8.,8.,1.,3.,7.,0.,0,1.2,3.,-6.15,1.,.8,1.,7.,0.,0.,0.,6,1.5,-3.12,1.,8.1,1.,5.,0.,0.,0.,5,1.5};
yx;double u,b,tmin,sqrt(),tan();double vdot(A,B)vec A,B;
{return A.x*B.x+A.y*B.y+A.z*B.z;}vec vcomb(a,A,B)double a;vec A,B;
{B.x+=a*A.x;B.y+=a*A.y;B.z+=a*A.z;return B;}
vec vunit(A)vec A;
{return vcomb(1./sqrt( vdot(A,A)),A,black);}struct sphere*intersect(P,D)vec P,D;
{best=0;tmin=1e30;s=sph+5;while(s--sph)b=vdot(D,U=vcomb(-1.,P,s-cen)),u=b*b-vdot(U,U)+s-rad*s-rad,u=u0?
{return A.x*B.x+A.y*B.y+A.z*B.z;}
vec vunit(A)vec A;
{vec vcomb(l./sqrt( vdot(A,A)),A,black);}struct sphere*intersect(P,D)vec P,D;
{best=0;tmin=1e30;s=sph+5;while(s--sph)b=vdot(D,U=vcomb(-1.,P,s-cen)),u=b*b-vdot(U,U)+s-rad*s-rad,u=u0?
{sqrt(u):u1e-7?b-u:b+u,tmin=u1e-7&tmin?best=s,u: return best;}
vec trace(level,P,D)vec P,D;
{double d,eta,e;vec N,color;struct sphere*s,*l;if(!level--)return black;if(s=intersect(P,D));else return amb;color=amb;eta=s-ir;d=
-vdot(D,N=vunit(vcomb(-1.,P=vcomb(tmin,D,P),s-cen )));if(d<0)N=vcomb
(-1.,N,black),eta=1/eta,d=-d;l=sph+5;while(l--sph)if((e=1-kl*vdot(N,U=vunit
(vcomb(-1.,P,l-cen)))&intersect(P,U)==1)color=vcomb(e,1-color,color);U=s-
color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta* eta*(l-d)*d;return vcomb(s-
kt,e0?trace(level,P,vcomb(eta*D,vcomb(ea*d-qrt (e),N,black)));black,vcomb(s-
k,s,trace(level,P,vcomb(Z*d,N,D)),vcomb(s-kd, color, vcomb(s-kl,U,black)))));}main()
{printf("%d %d\n",32,32);while(yx<32*32)U.x=yx%32-32/2,U.z=32/2-yx++,32,U.y=32/2/
tan(25/114.5915590261),U=vcomb(255., trace(3,black,vunit(U)),black),printf("%.0f
%.0f %.0f\n",U);}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
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
  • E.g., getchar(), putchar(), Check_octal(), etc.
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

\[
\text{for } (i=0; i<n; i++) \\
\quad \text{array}[i] = 1.0;
\]

• 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

\[
\text{if } (! (n >= k) \&\& !(n <= j))
\]

• 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 aloud”!
Parenthesize to resolve ambiguity

- Example: Check if integer $n$ satisfies $j < n < k$

- Common code

  ```java
  if ((j < n) && (n < k))
  ```

- Clearer code

  ```java
  if (((j < n) && (n < k))
  ```

Does this code work?
Parenthesize to resolve ambiguity (cont.)

- Example: read and print character until end-of-file

- Bad code

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

- Good code

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

Does this code work?
Revealing Structure: Expressions

Break up complex expressions

- Example: Identify chars corresponding to months of year
- Bad code

```java
if ((c == 'J') || (c == 'F') || (c == 'M') || (c == 'A') || (c == 'S') || (c == 'O') || (c == 'N') || (c == 'D'))
```

- Good code – lining up things helps

```java
if ((c == 'J') || (c == 'F') || (c == 'M') || (c == 'A') || (c == 'S') || (c == 'O') || (c == 'N') || (c == 'D'))
```

- Very common, though, to elide parentheses

```java
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

```c
for (j=0; j<100; j++) a[j]=j;
```

- Good code

```c
for (j = 0; j < 100; j++)
    a[j] = j;
```

- Often can rely on auto-indenting feature in editor
Use readable/consistent/correct indentation

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

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

Does this code work?

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

Does this code work?
Revealing Structure: Indentation

Use “else-if” for multi-way decision structures

- Example: Comparison step in a binary search.
- Bad code

```c
if (x < a[mid])
    high = mid - 1;
else
    if (x > a[mid])
        low = mid + 1;
    else
        return mid;
```

- Good code

```c
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 = 2 * PI * radius;

    printf("The circle's diameter is \n%d\n", diam);
    printf("and circumference is \n%.2f\n", circum);
}
```

Revealing Structure: “Paragraphs”

Use blank lines to divide the code into key parts

```c
#include <stdio.h>

int main()
{
    int radius = 5;
    int diam, circum;

    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
  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 Comments

Comment sections ("paragraphs") of code, not lines of code

```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;

    /* Read the circle's radius. */
    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; */
    }
    ...
```
Composing Comments

/* Compute the diameter and circumference. */
diam = 2 * radius;
circum = PI * (double)diam;

/* Print the results. */
printf("A circle with radius %d has diameter %d
",
        radius, diam);
printf("and circumference %f.\n", circum);

return 0;
}
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**
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**
Composing Function Comments

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

Examples of **file**-level modularity
- (See subsequent lectures)
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 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*
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*
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

Top-down design in **programming**
- 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**

![Diagram](image.png)
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

Diagram:

```
  1
   / \  
  2   1'
     /   
    2'   3
   /     
  4      Oops
```

...
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
"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

```c
int main(void)
{
    <clear line>
    <read a 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;
    }
}
```
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 (<there is a word>)
    {
        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;
    }
}
```
enum {MAX_WORD_LEN = 20};
int main(void)
{
  char word[MAX_WORD_LEN+1];
  int wordLen;
  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;
}
enum {MAX_WORD_LEN = 20};
int main(void)
{
    char word[MAX_WORD_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>
        }
        <add word to line>
        wordLen = readWord(word);
    }
    if (lineLen > 0)
        <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;
    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;
}
```
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)
        puts(line);
    return 0;
}
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>)
        {
            writeLine(line, lineLen, wordCount);
            <clear line>
        }
        lineLen = addWord(word, line, lineLen);
        wordLen = readWord(word);
    }
    if (lineLen > 0)
        puts(line);
    return 0;
}
The main() Function

```
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 ((wordLen + 1 + lineLen) > MAX_LINE_LEN)
        {
            writeLine(line, lineLen, wordCount);
            <clear line>
        }
        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;
    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;
}
```
Status

main

- readWord
- writeLine
- addWord
The readWord() Function

```c
int readWord(char *word)
{
    <skip over white space>
    <read chars, storing up to MAX_WORD_LEN in word>
    <return length of word>
}
```
int readWord(char *word)
{
    int c;

    /* Skip over white space. */
    c = getchar();
    while ((c != EOF) && (! isspace(c)))
        c = getchar();

    /* Read up to MAX_WORD_LEN chars into word */

    /* Return length of word */
}
The readWord() Function

```c
int readWord(char *word)
{
    int c;
    int pos = 0;

    /* Skip over white space. */
    c = getchar();
    while ((c != EOF) && (! isspace(c)))
    {
        c = getchar();
    }

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

    <return length of word>
}
```
The readWord() Function

```c
int readWord(char *word)
{
    int c;
    int pos = 0;
    c = getchar();

    /* Skip over white space. */
    while ((c != EOF) && (! isspace(c))
        c = 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;
}
```
Status

Diagram:

- 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 words, then append a space. */
    if (newLineLen > 0)
    {
        strcat(line, " ");
        newLineLen++;
    }

    <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 words, then append a space. */
    if (newLineLen > 0)
    {
        strcat(line, " ");
        newLineLen++;
    }

    strcat(line, word);

    <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;
}
void writeLine(const char *line, int lineLen, int numWords)
{
    int i;

    <compute number of excess spaces for line>

    for (i = 0; i < lineLen; i++)
    {
        if (line[i] != ' ')
            putchar(' ')
        else
        {
            <compute additional spaces to insert>

            <print a space, plus additional spaces>

            <decrease extra spaces and word count>
        }
    }
    putchar('\n');
}
void writeLine(const char *line, int lineLen, int numWords)
{
    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(' ');
        else
        {
            // compute additional spaces to insert
            // print a space, plus additional spaces
            // decrease extra spaces and word count
        }
    }
    putchar('\n');
}
void writeLine(const char *line, int lineLen, int numWords)
{
    int i, extraSpaces, spacesToInsert;

    /* Compute number of excess spaces for line. */
    extraSpaces = MAX_LINE_LEN - lineLen;

    for (i = 0; i < lineLen; i++)
    {
        if (line[i] != ' ')
            putchar(' ')
        else
        {
            /* Compute additional spaces to insert. */
            spacesToInsert = extraSpaces / (wordCount - 1);

            /* print a space, plus additional spaces */
            <print a space, plus additional spaces>

            /* decrease extra spaces and word count */
            <decrease extra spaces and word count>
        }
    } putchar('\n');
}
The writeLine() Function

```c
void writeLine(const char *line, int lineLen, int numWords)
{
    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(' ')
        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>
        }
    }
    putchar('
');
}
```

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

```c
void writeLine(const char *line, int lineLen, int numWords)
{  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(' ')
   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('
');
}
```
Status

Complete!
Note: Top-down design naturally yields modular code

Much more on modularity in upcoming lectures
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
Aside: Least-Risk Design

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
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
Appendix: The “justify” Program

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

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

Continued on next slide
int readWord(char *word)
{
    int ch, pos = 0;

    /* Skip over white space. */
    ch = getchar();
    while (!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
/ * 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('
');
}
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;

    ... 
```

Continued on next slide
Appendix: The “justify” Program

```c
... 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;
}```