



Programming and Program Style

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

1



Goals of this Lecture

- Help you learn about:
 - Good **programming** (verb) style
 - Good **program** (noun) style
- Why?
 - A well-styled program is **easier to maintain** and **more likely to be correct** than a poorly-styled program
 - A power programmer knows the qualities of a well-styled program, and how to develop one

2

Lecture Overview



- **Programming style:** how to create a good program
 - Top-down design
 - Successive refinement
 - Example: left and right justifying text
- **Program style:** qualities of a good program
 - Well structured
 - Uses common idioms
 - Uses descriptive names
 - Contains proper comments
 - Modular

3



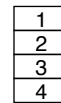
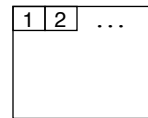
Part 1: Programming Style

4

Bottom-Up Design is Bad



- Bottom-up design ☹️
 - Design one part in detail
 - Design another part in detail
 - Repeat until finished
- Bottom-up design in **painting**
 - Paint upper left part of painting in complete detail
 - Paint next part of painting in complete detail
 - Repeat until finished
 - *Unlikely to produce a good painting*
- Bottom-up design in **programming**
 - Write first part of program in complete detail
 - Write next part of program in complete detail
 - Repeat until finished
 - *Unlikely to produce a good program*

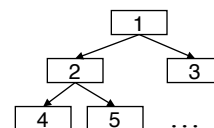


5

Top-Down Design is Good



- 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 the entire painting
- 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
 - Recurse in (mostly) breadth-first order
 - **Bonus: Product is naturally modular**



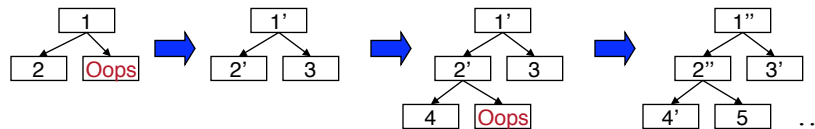
6

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
- Recurse in (mostly) breadth-first order, until all functions are defined



7

Example: Text Formatting



- Goals of the example

- Illustrate good programming style
 - Especially function-level modularity and top-down design
- Illustrate how to go from problem statement to code
 - Review and illustrate C constructs

- Text formatting (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 the very last line
- Simplifying assumptions
 - Word ends at white space or end-of-file
 - No word is longer than 20 characters

8

Example Input and Output



I Tune every heart and every voice.
N Bid every bank withdrawal.
P Let's all with our accounts rejoice.
U In funding Old Nassau.
T In funding Old Nassau we spend more money every year.
Our banks shall give, while we shall live.
We're funding Old Nassau.

O Tune every heart and every voice. Bid every bank
U withdrawal. Let's all with our accounts rejoice.
T In funding Old Nassau. In funding Old Nassau we
P spend more money every year. Our banks shall give,
U while we shall live. We're funding Old Nassau.
T

9

Thinking About the Problem



- I need a notion of “word”
 - Sequence of characters with no white space
 - All characters in a word must be printed on the same line
- I need to be able to read and print words
 - Read characters from `stdin` till white space or `EOF`
 - Print characters to `stdout` followed by space(s) or newline
- I need to deal with poorly-formatted input
 - I need to remove extra white space in input
- Unfortunately, I can't print the words as they are read
 - I don't know # of spaces needed till I read the future words
 - Need to buffer the words until I can safely print an entire line
- But, how much space should I add between words?
 - Need at least one space between adjacent words on a line
 - Can add extra spaces evenly to fill up an entire line

10

Writing the Program



- Key constructs
 - Word
 - Line
- Next steps
 - Write pseudocode for `main()`
 - Successively refine
- 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, much backtracking would occur

11

The Top Level



- First, let's sketch `main()` ...

```
int main(void) {
    <Clear line>
    for (;;) {
        <Read a word>
        if (<No more words>) {
            <Print line with no justification>
            return 0;
        }
        if (<Word doesn't fit on this line>) {
            <Print line with justification>
            <Clear line>
        }
        <Add word to line>
    }
    return 0;
}
```

12

Reading a Word



```
...
enum {MAX_WORD_LEN = 20};
int main(void) {
    char word[MAX_WORD_LEN + 1];
    int wordLen;
    <Clear line>
    for (;;) {
        wordLen = ReadWord(word);
        if (<No more words>) {
            <Print line with no justification>
            return 0;
        }
        if (<Word doesn't fit on this line>) {
            <Print line with justification>
            <Clear line>
        }
        <Add word to line>
    }
    return 0;
}
```

```
int ReadWord(char *word) {
    <Skip over whitespace>
    <Store chars up to MAX_WORD_LEN in word>
    <Return length of word>
}
```

- Now let's successively refine. What does <Read a word> mean? The job seems complicated enough that it should be delegated to a distinct function...

13

Reading a Word (cont.)



- `ReadWord()` seems easy enough to design. So let's flesh it out...

```
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;
}
```

14

Saving a Word



```
...
enum {MAX_WORD_LEN = 20};
enum {MAX_LINE_LEN = 50};
int main(void) {
    char word[MAX_WORD_LEN + 1];
    int wordLen;
    char line[MAX_LINE_LEN + 1];
    int lineLen = 0;
    <Clear line>
    for (;;) {
        wordLen = ReadWord(word);
        if (<No more words>) {
            <Print
            return
        }
        if (<Word
        <Print
        <Clear
    }
    AddWord(word, line, &lineLen);
}
return 0;
}
```

```
void AddWord(const char *word, char *line, int *lineLen) {
    <if line already contains some words, append a space>
    strcat(line, word);
    (*lineLen) += strlen(word);
}
```

- Now, back to main(). What does <Add word to line> mean? The job seems complicated enough to demand a distinct function...

15

Saving a Word (cont.)



- AddWord() is almost complete already, so let's get that out of the way...

```
void AddWord(const char *word, char *line, int *lineLen) {
    /* If line already contains some words, append a space. */
    if (*lineLen > 0) {
        line[*lineLen] = ' ';
        line[*lineLen + 1] = '\0';
        (*lineLen)++;
    }
    strcat(line, word);
    (*lineLen) += strlen(word);
}
```

16

Printing the Last Line



```
...
int main(void) {
    char word[MAX_WORD_LEN + 1];
    int wordLen;
    char line[MAX_LINE_LEN + 1];
    int lineLen = 0;
    <Clear line buffer>
    for (;;) {
        wordLen = ReadWord(word);

        /* If no more words, print line
           with no justification. */
        if ((wordLen == 0) && (lineLen > 0)) {
            puts(line);
            return 0;
        }
        if (<Word doesn't fit on this line>) {
            <Print line with justification>
            <Clear line buffer>
        }
        AddWord(word, line, &lineLen);
    }
    return 0;
}
```

- Again, back to main(). What do <No more words> and <Print line with no justification> mean? Those jobs seem easy enough that we need not define additional functions...

17

Deciding When to Print



```
...
int main(void) {
    char word[MAX_WORD_LEN + 1];
    int wordLen;
    char line[MAX_LINE_LEN + 1];
    int lineLen = 0;
    <Clear line buffer>
    for (;;) {
        wordLen = ReadWord(word);

        /* If no more words, print line
           with no justification. */
        if ((wordLen == 0) && (lineLen > 0)) {
            puts(line);
            return 0;
        }
        /* If word doesn't fit on this line, then... */
        if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {
            <Print line with justification>
            <Clear line buffer>
        }
        AddWord(word, line, &lineLen);
    }
    return 0;
}
```

- What does <Word doesn't fit on this line> mean? That's somewhat tricky, but involves little code...

18

Printing with Justification



- Now, to the heart of the program. What does <Print line with justification> mean? Certainly that job demands a distinct function. Moreover, it's clear that the function must know how many words are in the given line. So let's change `main()` accordingly...

```
...
int main(void) {
    ...
    int numWords = 0;
    <Clear line>
    for (;;) {
        ...
        /* IF word doesn't fit on this line, then... */
        if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {
            WriteLine(line, lineLen, numWords);
            <Clear line>
        }

        AddWord(word, line, &lineLen);
        numWords++;
    }
    return 0;
}
```

19

Printing with Justification (cont.)



- And write pseudocode for `WriteLine()` ...

```
void WriteLine(const char *line, int lineLen, int numWords) {
    <Compute number of excess spaces for line>

    for (i = 0; i < lineLen; i++) {
        if (<line[i] is not a space>)
            <Print the character>
        else {
            <Compute additional spaces to insert>

            <Print a space, plus additional spaces>

            <Decrease extra spaces and word count>
        }
    }
}
```

20

Printing with Justification (cont.)



```
void WriteLine(const char *line, int lineLen, int numWords)
{
    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 / (numWords - 1);

            /* Print a space, plus additional spaces. */
            for (j = 1; j <= spacesToInsert + 1; j++)
                putchar(' ');

            /* Decrease extra spaces and word count. */
            extraSpaces -= spacesToInsert;
            numWords--;
        }
    }
    putchar('\n');
}
```

- Let's go ahead and complete WriteLine ()...

The number of gaps

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

21

Clearing the Line



- One step remains. What does <Clear line> mean? It's an easy job, but it's done in two places. So we probably should delegate the work to a distinct function, and call the function in the two places...

```
...
int main(void) {
    ...
    int numWords = 0;
    ClearLine(line, &lineLen, &numWords);
    for (;;) {
        ...
        /* If word doesn't fit on this line, then... */
        if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {
            WriteLine(line, lineLen, numWords);
            ClearLine(line, &lineLen, &numWords);
        }
        addWord(word, &numWords);
    }
    return 0;
}

void ClearLine(char *line, int *lineLen, int *numWords) {
    line[0] = '\0';
    *lineLen = 0;
    *numWords = 0;
}
```

22

Modularity: Summary of Example



- To the user of the program
 - Input: Text in messy format
 - Output: Same text left and right justified, looking mighty pretty
- Between parts of the program
 - Word-handling functions
 - Line-handling functions
 - `main()` function
- The many benefits of modularity
 - Reading the code: In small, separable pieces
 - Testing the code: Test each function separately
 - Speeding up the code: Focus only on the slow parts
 - Extending the code: Change only the relevant parts

23

Part 2: Program Style



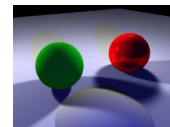
24

Program Style



• Who reads your code?

- The compiler
- Other programmers



```
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.,-3.,12.,.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(F,D)vec F,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=0?sqrt(u):1e31,u=b-ule-7?b-u:b+u,tmin=u-1e-7&&u<tmin?best=s,u:tmin;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(F,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))))0&&intersect(P,U)=l)color=vcomb(e,l-color,color);U=s-color;color.x*=U.x;color.y*=U.y;color.z*=U.z;e=1-eta* eta*(1-d*d);return vcomb(s-kt,e0?trace(level,P,vcomb(eta,D,vcomb(eta*d-sqrt(e),N,black))):black,vcomb(s-ks,trace(level,P,vcomb(2*d,N,D)),vcomb(s-kd,color,vcomb(s-kl,0,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(vcomb(1.,U,black)),black,amb));}
```

This is a working ray tracer. (courtesy of Paul Heckbert) 25

Program Style



• Why does program style matter?

- Bugs often caused by programmer's misunderstanding
 - What does this variable do?
 - How is this function called?
- Good code = human readable code

• How can code become easier for humans to read?

- Convey program structure
- Use common idioms
- Choose descriptive names
- Compose proper comments
- Use modularity

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

27

Structure: Indentation (cont.)



- 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;  
}
```

Does this
code work?

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

Does this
code work?

28

Structure: Indentation (cont.)



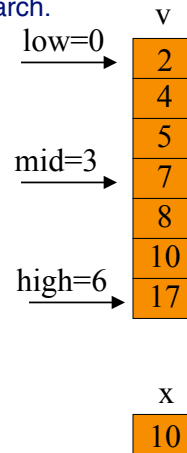
- Use “else-if” for multi-way decision structures

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

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

- Good code

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



29

Structure: “Paragraphs”



- Use blank lines to divide the code into key parts

```
#include <stdio.h>
#include <stdlib.h>

int main(void)

/* Read a circle's radius from stdin, and compute and write its
diameter and circumference to stdout. Return 0 if successful. */

{
    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; */
    }
    ...
```

30

Structure: “Paragraphs”



- Use blank lines to divide the code into key parts

```
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;
}
```

31

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 aloud”!

32

Structure: Expressions (cont.)



- Parenthesize to resolve ambiguity
 - Example: Check if integer n satisfies $j < n < k$

- Bad code

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

Does this code work?

- Good code

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

33

Structure: Expressions (cont.)



- Parenthesize to resolve ambiguity (cont.)
 - Example: read and print character until end-of-file

- Bad code

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

Does this code work?

- Good code

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

34

Structure: Expressions (cont.)



- 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

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

- Lining up the parallel structures is helpful, too!

35

C Idioms



- Use C idioms
 - Example: Set each array element to 1.0.
 - Bad code (or, perhaps just “so-so” code)

```
i = 0;  
while (i <= n-1)  
    array[i++] = 1.0;
```

- Good code

```
for (i=0; i<n; i++)  
    array[i] = 1.0;
```

- We'll see many C idioms throughout the course
- Don't feel obliged to use C idioms that decrease clarity

36

Naming



- 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., `Buffer_insert` (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.

37

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 sections (“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. ☺

38

Comments (cont.)



- Comment sections (“paragraphs”) of code, not lines of code

```
#include <stdio.h>
#include <stdlib.h>

int main(void)

/* Read a circle's radius from stdin, and compute and write its
diameter and circumference to stdout. Return 0 if successful. */

{
    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; */
    }
    ...
}
```

39

Comments (cont.)



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

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

return 0;
}
```

40

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**

41

Function Comments (cont.)



- Bad function comment

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

- Describes **how the function works**

42

Function Comments (cont.)



- Good function comment

```
/* decomment.c */

int main(void) {

    /* 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. */

    ...
}
```

- Describes **what the function does**

43

Modularity



- Big programs are harder to write than small ones
 - “A dog house can be built without any particular design, using whatever materials are at hand. A house for humans, on the other hand, is too complex to just throw together.” – K. N. King
- Abstraction is the key to managing complexity
 - Abstraction allows programmer to know *what* something does without knowing *how*
- Examples of function-level abstraction
 - Function to sort an array of integers
 - Character I/O functions such as `getchar ()` and `putchar ()`
 - Mathematical functions such as `lcm ()` and `gcd ()`
- Examples of file-level abstraction
 - (Described in a later lecture)

44

Summary



- Programming style
 - Think about the problem
 - Use top-down design and successive refinement
 - But know that backtracking inevitably will occur

45

Summary (cont.)



- Program style
 - Convey program structure
 - Spacing, indentation, parentheses
 - Use common C idioms
 - But not at the expense of clarity
 - Choose consistent and descriptive names
 - For variables, functions, etc.
 - Compose proper comments
 - Especially for functions
 - Divide code into modules
 - Functions and files

46

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

47

Appendix: The “justify” Program



```
int ReadWord(char *word) {
    /* Read a word from stdin. Assign it to word. Return the length
       of the word, or 0 if no word could be read. */
    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;
}
```

Continued on next slide

48

Appendix: The “justify” Program



```
void ClearLine(char *line, int *lineLen, int *numWords) {
    /* Clear the given line. That is, clear line, and set *lineLen
    and *numWords to 0. */

    line[0] = '\0';
    *lineLen = 0;
    *numWords = 0;
}

void AddWord(const char *word, char *line, int *lineLen) {
    /* Append word to line, making sure that the words within line are
    separated with spaces. Update *lineLen to indicate the
    new line length. */

    /* If line already contains some words, append a space. */
    if (*lineLen > 0) {
        line[*lineLen] = ' ';
        line[*lineLen + 1] = '\0';
        (*lineLen)++;
    }
    strcat(line, word);
    (*lineLen) += strlen(word);
}
```

Continued on next slide

49

Appendix: The “justify” Program



```
void WriteLine(const char *line, int lineLen, int numWords) {
    /* Write line to stdout, in right justified form. lineLen
    indicates the number of characters in line. numWords indicates
    the number of words in line. */

    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 / (numWords - 1);

            /* Print a space, plus additional spaces. */
            for (j = 1; j <= spacesToInsert + 1; j++)
                putchar(' ');

            /* Decrease extra spaces and word count. */
            extraSpaces -= spacesToInsert;
            numWords--;
        }
    }
    putchar('\n');
}
```

Continued on next slide

50

Appendix: The “justify” Program



```
int main(void) {  
  
    /* Read words from stdin, and write the words in justified format  
    to stdout. */  
  
    /* 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];  
    int wordLen;  
  
    char line[MAX_LINE_LEN + 1];  
    int lineLen = 0;  
    int numWords = 0;  
  
    ClearLine(line, &lineLen, &numWords);  
  
    ...  
}
```

Continued on next slide

51

Appendix: The “justify” Program



```
...  
  
for (;;) {  
    wordLen = ReadWord(word);  
  
    /* If no more words, print line  
    with no justification. */  
    if ((wordLen == 0) && (lineLen > 0)) {  
        puts(line);  
        break;  
    }  
  
    /* If word doesn't fit on this line, then... */  
    if ((wordLen + 1 + lineLen) > MAX_LINE_LEN) {  
        WriteLine(line, lineLen, numWords);  
        ClearLine(line, &lineLen, &numWords);  
    }  
  
    AddWord(word, line, &lineLen);  
    numWords++;  
}  
return 0;  
}
```

52