Programming Style and Program Style

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

Goals of this Lecture

• Help you learn about:
  • Good programming style
  • Good program 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
Lecture Overview

- **Programming style**: how to write a good program
  - Top-down design
  - Successive refinement
  - Example: left and right justifying text
  Properties of the process

- **Program style**: qualities of a good program
  - Well structured
  - Uses common idioms
  - Uses descriptive names
  - Contains proper comments
  - Modular
  Properties of the resulting artifact

Part 1: Programming Style

How to write a good program
You don’t Paint a Painting Bottom-up

• 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*

• Top-down design
  • Sketch the entire painting with minimal detail
  • Successively refine the entire painting

Or Design a Program Bottom-up

• 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 efficiently*

• Top-down design and successive refinement
  • 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: End product is naturally modular
Top-Down Design in Reality

- In reality, we make mistakes and have to backtrack
  - 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

- But this is a downer, so let’s ignore it for now …

Example: 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 last line
  - That sounds hard. Let’s simplify it a little:
    - Word ends at white space or at end-of-file
    - No word is longer than 20 characters
Example Input and Output

<table>
<thead>
<tr>
<th>INPUT</th>
<th>OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tune every heart and every voice.</td>
<td>Tune every heart and every voice. Bid every bank withdrawal.</td>
</tr>
<tr>
<td>Bid every bank withdrawal.</td>
<td>Let’s all with our accounts rejoice.</td>
</tr>
<tr>
<td>Let’s all with our accounts rejoice.</td>
<td>In funding Old Nassau.</td>
</tr>
<tr>
<td>In funding Old Nassau we spend more money every year.</td>
<td>Our banks shall give, while we shall live.</td>
</tr>
<tr>
<td>Our banks shall give, while we shall live.</td>
<td>We’re funding Old Nassau.</td>
</tr>
</tbody>
</table>

Thinking About the Problem

- I need the concept of a “word”
  - Sequence of characters with no white space
- I need the concept of a “line”
  - Sequence of characters of fixed max size, separated by newlines
  - Words separated by spaces
  - All characters in a word must be printed on the same line
- I have to be able to read and print words
- I have 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 until 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
  - No. of gaps = Line size divided by number of words in line
Writing the Program

• Write pseudocode for main()
• Successively refine

• Caveats concerning the following presentation
  • Function comments and some blank lines are omitted because of space constraints on slides
  • Do as I say, not as I do …
  • Design sequence is idealized
  • In reality, much backtracking would happen

The Top Level

• First, let’s sketch main()…

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

```c
int main(void) {
    while (true) {
        if (no more words) {
            return 0;
        }
        if (word doesn't fit on this line) {
            return 0;
        }
        add word to line
    }
    return 0;
}
```

```c
enum {MAX_WORD_LEN = 20};
int main(void) {
    char word[MAX_WORD_LEN + 1];
    int wordLen;
    while (true) {
        wordLen = ReadWord(word);
        if (no more words) {
            return 0;
        }
        if (word doesn't fit on this line) {
            return 0;
        }
        add word to line
    }
    return 0;
}
```

```c
int ReadWord(char *word) {
    skip over whitespace
    store chars up to MAX_WORD_LEN in word
    return length of word
}
```
Reading a Word (cont.)

• `ReadWord()` seems easy enough to design. So let’s flesh it out…

```c
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;
}
```

Adding Word to Line Buffer

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

```c
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;
    }
}
Adding Word to Line Buffer

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;
    for (;;) {
        wordLen = ReadWord(word);
        if (<No more words>) {
            return 0;
        }
        if (<Word doesn’t fit on this line>) {
            AddWord(word, line, &lineLen);
        }
        AddWord(word, line, &lineLen);
    }
    return 0;
}

Adding Word to Line Buffer (cont.)

• AddWord() is almost complete, so let’s get that out of the way...

void AddWord(const char *word, char *line, int *lineLen) {
    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);
    }
}
• 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...

```
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;
}
```
Deciding When to Print

```c
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? Also involves little code...

---

Deciding When to Print

```c
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;
}
```

• Now, to the heart of the program. What does <Print line with justification> mean? Certainly that demands a distinct function.
Printing with Justification

• That WriteLine function must know how many words are in the given line (why?)

```c
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;
}
```

Printing with Justification (cont.)

• Pseudocode for WriteLine()...

```c
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 some number of additional spaces>
            <Decrease extra spaces count and word count>
        }
    }
}
```
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');
}

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

• Let’s go ahead and complete WriteLine ()…

Clearing the Line

• Easy, but 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;
    <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>
        } else {  
            AddWord(word, line, &lineLen);
            numWords++;
        }
        return 0;
    }
Clearing the Line

```c
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, line, &lineLen);
        numWords++;
    }
    return 0;
}

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

    return 0;
}
```

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

- Modularity within the program
  - `main()` function
  - Line-handling functions
  - Word-handling functions

- 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
Part 2: Program Style

Program Style

- Who reads your code?
- The compiler
- Other programmers

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

\[
\text{for } (j=0; j<100; j++) \ a[j]=j;
\]

• Good code

\[
\text{for } (j = 0; j < 100; j++) \\
\quad a[j] = j;
\]

• Often can rely on auto-indenting feature in editor
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?

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;

low=0
mid=3
high=6

v
2
4
5
7
8
10
17

x
10
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;
}
```
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”

Structure: Expressions (cont.)

- Parenthesize to resolve ambiguity
  - Example: Check if integer \( n \) satisfies \( j < n < k \)

- Bad code

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

- Good code

\[
\text{if } ((j < n) \&\& (n < k))
\]
Structure: Expressions (cont.)

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

  - Bad code
    
    ```c
    while (c = getchar() != EOF)
    putchar(c);
    ```

  - Good code
    
    ```c
    while ((c = getchar()) != EOF)
    putchar(c);
    ```

- Break up complex expressions stylistically
  - Example: Identify chars corresponding to months of year

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

  - Good code
    
    ```c
    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
C Idioms

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

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

• Good code

```c
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

Naming (read for yourselves)

• 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.
Comments (reprise; read yourselves)

• 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

Comments (cont.)

• Comment sections (“paragraphs”), 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; */
    }

    diam = 2 * radius;
    circum = 2 * PI * radius;
    printf("The circle’s diameter is %d, its circumference is %.2f\n", diam, circum);
}
```
/* 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;
}

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
Function Comments (cont.)

• 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

Function Comments (cont.)

• 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
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)

Summary

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

Appendix: The “justify” Program

/*-------------------------------------------------------*/
/* justify.c                                             */
/* Author:  COS 217 Instructors                          */
/*-------------------------------------------------------*/
#include <stdio.h>
#include <ctype.h>
#include <string.h>

/* The maximum number of characters in a word. */
enum {MAX_WORD_LEN = 20};

/* The maximum number of characters in a line. */
enum {MAX_LINE_LEN = 50};

Continued on next slide
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;
}
```

Continued on next slide

Appendix: The “justify” Program

```c
/* Clear the given line. That is, clear line, and set *lineLen and *numWords to 0. */
void ClearLine(char *line, int *lineLen, int *numWords) {
    line[0] = '\0';
    *lineLen = 0;
    *numWords = 0;
}

/* Append word to line, making sure that the words within line are separated with spaces. Update *lineLen to indicate the new line length. */
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);
}
```

Continued on next slide
Appendix: The “justify” Program

/* Write line to stdout, in right justified form. lineLen indicates the number of characters in line. numWords indicates the number of words in line. */
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++) {
        /* 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');
}

/* 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.
  -- No line is longer than MAX_LINE_LEN characters. */
int main(void) {
    char word[MAX_WORD_LEN + 1];
    int wordLen;
    char line[MAX_LINE_LEN + 1];
    int lineLen = 0;
    int numWords = 0;
    ClearLine(line, &lineLen, &numWords);
    ...
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;