

Choosing Names



Use descriptive names for globals and functions

- E.g., `display`, `CONTROL`, `CAPACITY`

Use concise names for local variables

- E.g., `i` (not `arrayIndex`) for loop variable

Use case judiciously

- E.g., `Stack_push` (Module_function)
`CAPACITY` (constant)
`buf` (local variable)

Use a consistent style for compound names

- E.g., `frontsize`, `frontSize`, `front_size`

Use active names for functions that do something

- E.g., `getchar()`, `putchar()`, `Check_octal()`, etc.

Not necessarily for functions that are something: `sin()`, `sqrt()` 7

Length of name
≈ proportional to
scope of variable

Using C Idioms



Use C idioms

- Example: Set each array element to 1.0.
- Bad code (complex for no obvious gain)

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

- Good code (not because it's vastly simpler—it isn't!—but because it uses a standard idiom that programmers can grasp at a glance)

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

- Don't feel obliged to use C idioms that decrease clarity

8

Revealing Structure: Expressions



Parenthesize for correctness and to resolve ambiguity

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

- Broken code

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

- Working, idiomatic code

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

9

▶ iClicker Question

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

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

- A. No, needs to be `if ((j < n) && (n < k))`
- B. Correct, but I'd parenthesize anyway
- C. Correct, and I'd leave it alone

Revealing Structure: Expressions



Parenthesize for correctness and to resolve ambiguity

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

- Common code

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

- Clearer code (maybe)

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

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!

11

▶ iClicker Question

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

```
if (!(n >= k) || (n <= j))
```

- A. No, incorrect
- B. Not sure – I can't be expected to apply DeMorgan's laws during a 10 AM lecture
- C. Correct, but I'd never write such a monstrosity

Revealing Structure: Expressions



Use natural form of expressions

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

```
if (!(n >= k) || (n <= j))
```

- Good code

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

- Conditions should read as you'd say them aloud
 - Not "Conditions shouldn't read as you'd never say them in other than a purely internal dialog!"

13

Revealing Structure: Expressions



Break up complex expressions

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

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

- Good code – lining up things helps

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

- Very common, though, to elide parentheses

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

14

Revealing Structure



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

Perhaps better in this case: a switch statement

```
switch (c) {
case 'J': case 'F': case 'M':
case 'A': case 'S': case 'O':
case 'N': case 'D':
do_this();
break;
default:
do_that();
}
```

15

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

16

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;
}
```

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?

17

Revealing Structure: Indentation



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

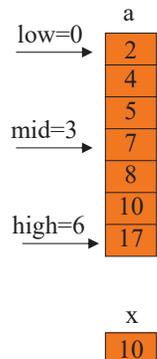
- Example: Comparison step in a binary search.

- Bad code

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

- Good code

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



18

Revealing Structure: “Paragraphs”



Use blank lines to divide the code into key parts

```
#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; */
    }
    ...
}
```

19

Revealing 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;
}
```

20

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!!!

21

Composing Comments



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

```
#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; */
    }
    ...
}
```

22

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\n",
    radius, diam);
printf("and circumference %f.\n", circum);

return 0;
}
```

23

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**

24

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

25

Composing Function Comments



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

26

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

27

Modularity Examples



Examples of **function-level** modularity

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

Examples of **file-level** modularity

- Assignment 3.
- And all the other assignments.

28

Program Style Summary



Good program \approx clear program

Qualities of a clear program

- Uses appropriate names
- Uses common idioms
- Reveals program structure
- Contains proper comments
- Is modular

29

Agenda



Program style

- Qualities of a good program

Programming style

- **How to write a good program quickly**

30

Bottom-Up Design

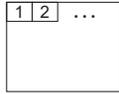


Bottom-up design ☹️

- Design one part of the system in detail
- Design another part of the system in detail
- Combine
- Repeat until finished

Bottom-up design in painting

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



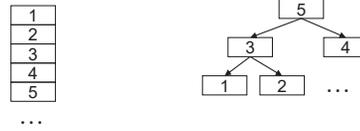
31

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*



32

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



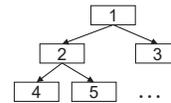
33

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**



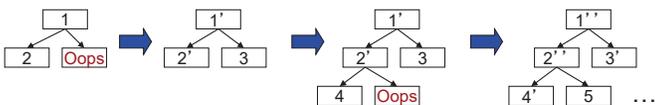
34

Top-Down Design in Reality



Top-down design in programming in reality

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



35

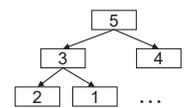
Aside: Least-Risk Design



Design process should minimize risk

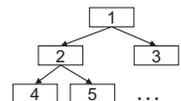
Bottom-up design

- 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



36

Aside: Least-Risk Design



Recommendation

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

37

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

38

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

```
"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
```

39

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

40

The main() Function



```
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;
}
```

41

The main() Function



```
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;
}
```

42

The main() Function



```
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;
}
```

43

The main() Function



```
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;
}
```

44

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 (<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;
}
```

45

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 (<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;
}
```

46

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 = 0;
    int wordCount = 0;
    <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;
}
```

47

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 = 0;
    int wordCount = 0;
    <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;
}
```

48

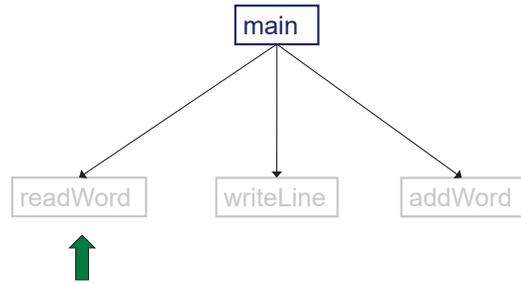
The main() Function



```
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 = 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;
}
```

49

Status



50

The readWord() Function



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

51

The readWord() Function



```
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>
  <return length of word>
}
```

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

52

The readWord() Function



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

  /* 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 length of word>
}
```

53

The readWord() Function



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

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

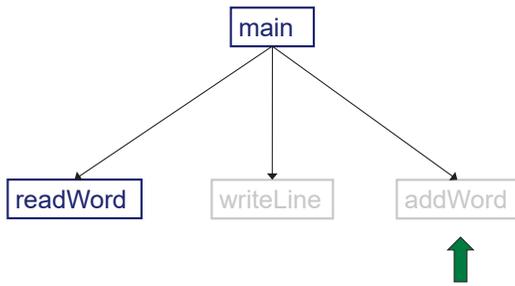
  /* Read up to MAX_WORD_LEN chars into word. */
  while ((ch != EOF) && (! isspace(ch)))
  { if (pos < MAX_WORD_LEN)
    { word[pos] = (char)ch;
      pos++;
    }
    ch = getchar();
  }
  word[pos] = '\0';

  return pos;
}
```

readWord() gets away with murder here, consuming/discarding one character past the end of the word.

54

Status



55

The addWord() Function



```
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>
}
```

56

The addWord() Function



```
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>
}
```

57

The addWord() Function



```
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>
}
```

58

The addWord() Function



```
int addWord(const char *word, char *line, int lineLen)
{
    int newLineLen = lineLen;

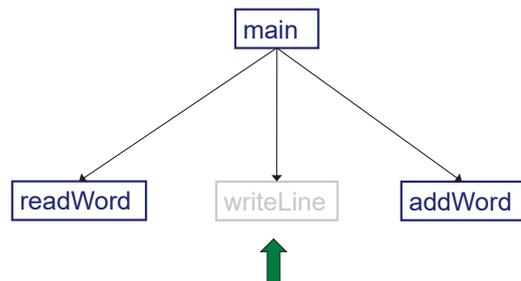
    /* If line already contains some words, then append a space. */
    if (newLineLen > 0)
    {
        strcat(line, " ");
        newLineLen++;
    }

    strcat(line, word);

    newLineLen += strlen(word);
    return newLineLen;
}
```

59

Status



60

The writeLine() Function



```
void writeLine(const char *line, int lineLen, int wordCount)
{
    int i;

    <compute number of excess spaces for line>
    for (i = 0; i < lineLen; i++)
    {
        if (line[i] != ' ')
            putchar(line[i])
        else
        {
            <compute additional spaces to insert>
            <print a space, plus additional spaces>
            <decrease extra spaces and word count>
        }
    }
    putchar('\n');
}
```

61

The writeLine() Function



```
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>
            <print a space, plus additional spaces>
            <decrease extra spaces and word count>
        }
    }
    putchar('\n');
}
```

62

The writeLine() Function



```
void writeLine(const char *line, int lineLen, int wordCount)
{
    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(line[i])
        else
        {
            /* Compute additional spaces to insert. */
            spacesToInsert = extraSpaces / (wordCount - 1);

            <print a space, plus additional spaces>
            <decrease extra spaces and word count>
        }
    }
    putchar('\n');
}
```

The number of gaps

63

The writeLine() Function



```
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>
        }
    }
    putchar('\n');
}
```

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

64

The writeLine() Function



```
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;

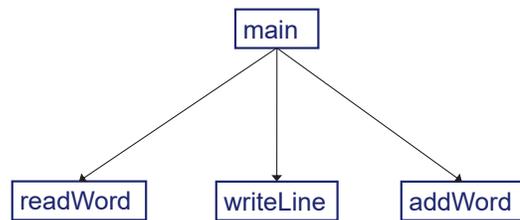
    for (i = 0; i < lineLen; i++)
    {
        if (line[i] != ' ')
            putchar(line[i])
        else
        {
            /* Compute additional spaces to insert. */
            spacesToInsert = extraSpaces / (wordCount - 1);

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

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

65

Status



Complete! And modular!

66

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

67

Are we there yet?



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

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

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

What’s wrong with this output?



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

```
"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
```

69

What’s better with this output?



Adequate

```
"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
```

Better

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

70

Challenge problem



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

(1) the jars should add up to n , that is,

```
{s=0; for(i=0;i<k;i++) s+=spacesHere(i,k,n); assert (s==n);}
```

or in math notation, $\sum_{i=0}^{k-1} \text{spacesHere}(i,k,n) = n$

(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.

One solution uses floating-point division and rounding; do “man round” and pay attention to where that man page says “include <math.h>”.

“Programming in the Large” Steps



Design & Implement

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

Debug

- Debugging techniques & tools

Test

- Testing techniques & tools

Maintain

- Performance improvement techniques & tools

72

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

73

Appendix: The “justify” Program



```
/* 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

74

Appendix: The “justify” Program



```
/* 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

75

Appendix: The “justify” Program



```
/* 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

76

Appendix: The “justify” Program



```
/* 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

77

Appendix: The “justify” Program



```
...

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;
}
```

78