



Program and Programming Style

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

For Your Amusement



“Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” -- Martin Fowler

“Good code is its own best documentation. As you’re about to add a comment, ask yourself, ‘How can I improve the code so that this comment isn’t needed?’ ” -- Steve McConnell

“Programs must be written for people to read, and only incidentally for machines to execute.” -- Abelson / Sussman

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

“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

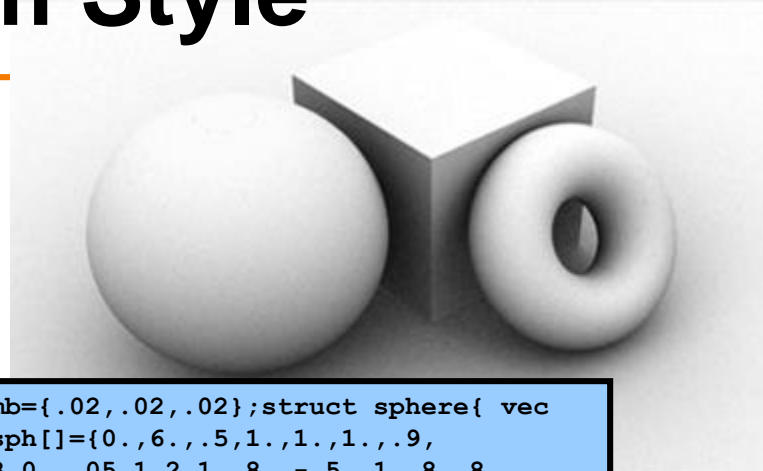
- How to compose a good program quickly



Motivation for Program Style

Who reads your code?

- The compiler
- Other programmers



```
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(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):1e31,u=b-u*le-7?b-u:b+u,tmin=u*le-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(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))))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,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 \approx 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 that do something

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

Not necessarily for functions that are something: `sin()`, `sqrt()` 8



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

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!”

Revealing Structure: Expressions



Parenthesize to resolve ambiguity

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

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

Does this code work?

- 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!

Revealing Structure: Expressions



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-ish code

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

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

Revealing Structure: Expressions



Break up complex expressions

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

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

- Good code – lining up things helps

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

- Very common, though, to elide parentheses

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



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();  
}
```



Revealing Structure: Spacing

Use readable/consistent spacing

- Example: Assign each array element $a[j]$ to the value j .
- Bad code

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

- Good code

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

- Often can rely on auto-indenting feature in editor



Revealing Structure: Indentation

Use readable/consistent/correct indentation

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

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

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?



Revealing Structure: Indentation

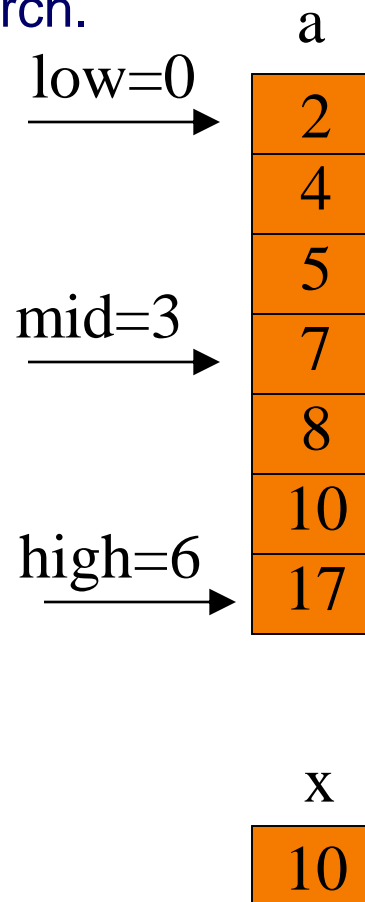
Use “else-if” for multi-way decision structures

- 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;
```



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; */
    }
    ...
```

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;
}
```



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

```
#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\n",  
       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**

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

30% of the class lost points on assignment 1 for a “how” instead of “what” comment on main()

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



Using Modularity

Abstraction is the key to managing complexity

- Abstraction is a tool (the only one???) that people use to understand complex systems
- Abstraction allows people to know *what* a (sub)system does without knowing *how*

Proper modularity is the manifestation of abstraction

- Proper modularity makes a program's abstractions explicit
- Proper modularity can dramatically increase clarity
- ⇒ Programs should be modular

However

- *Excessive* modularity can *decrease* clarity!
- *Improper* modularity can *dramatically decrease* clarity!!!
- ⇒ Programming is an art



Modularity Examples

Examples of **function**-level modularity

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

Examples of **file**-level modularity

- (See subsequent lectures)

Program Style Summary



Good program \approx 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



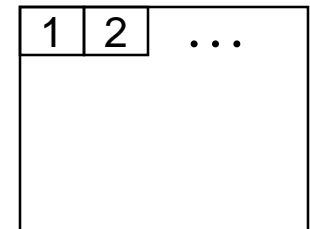
Bottom-Up Design

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*

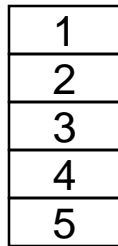




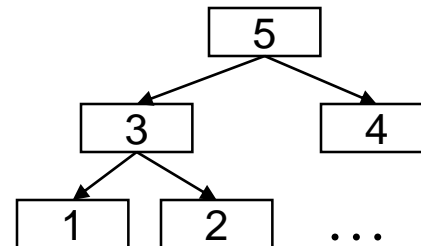
Bottom-Up Design

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



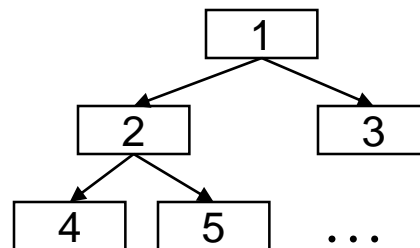


Top-Down Design

Top-down design in **programming**

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

- Bonus: Product is naturally **modular**

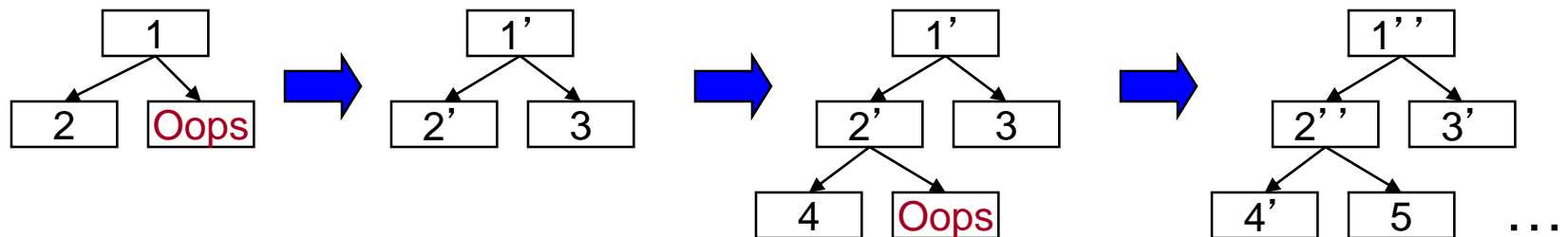


Top-Down Design in Reality



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



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



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
```

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



```
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

```
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;
}
```



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;
}
```




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;
}
```



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;
}
```



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;
}
```



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;
}
```





The main() Function

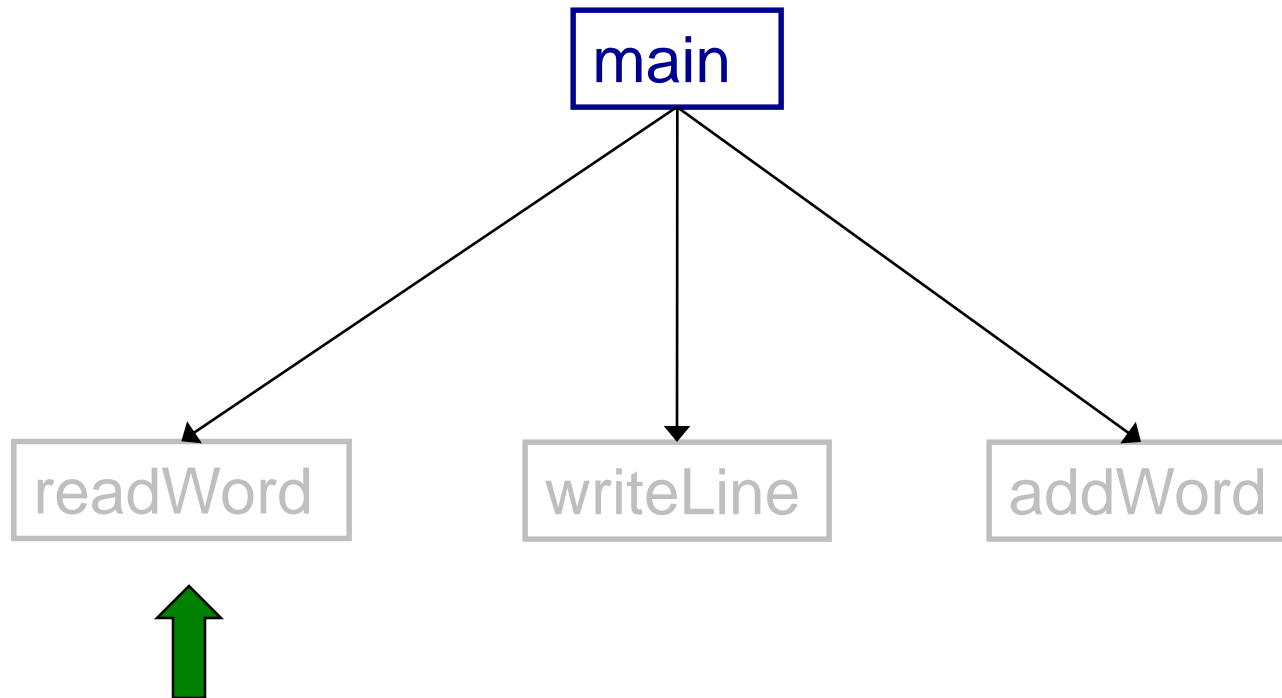
```
enum {MAX_WORD_LEN = 20};
enum {MAX_LINE_LEN = 50};
int main(void)
{  char word[MAX_WORD_LEN+1];
   char line[MAX_LINE_LEN+1];
   int wordLen;
   int lineLen = 0;
   int wordCount = 0;
   <clear line> ←
   wordLen = readWord(word);
   while (wordLen != 0)
   {  if ((wordLen + 1 + lineLen) > MAX_LINE_LEN)
      {  writeLine(line, lineLen, wordCount);
         <clear line> ←
      }
      lineLen = addWord(word, line, lineLen);
      wordLen = readWord(word);
   }
   if (lineLen > 0)
       puts(line);
   return 0;
}
```



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;
}
```

Status





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>
}
```


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.



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> ←
}
```



The readWord() Function

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

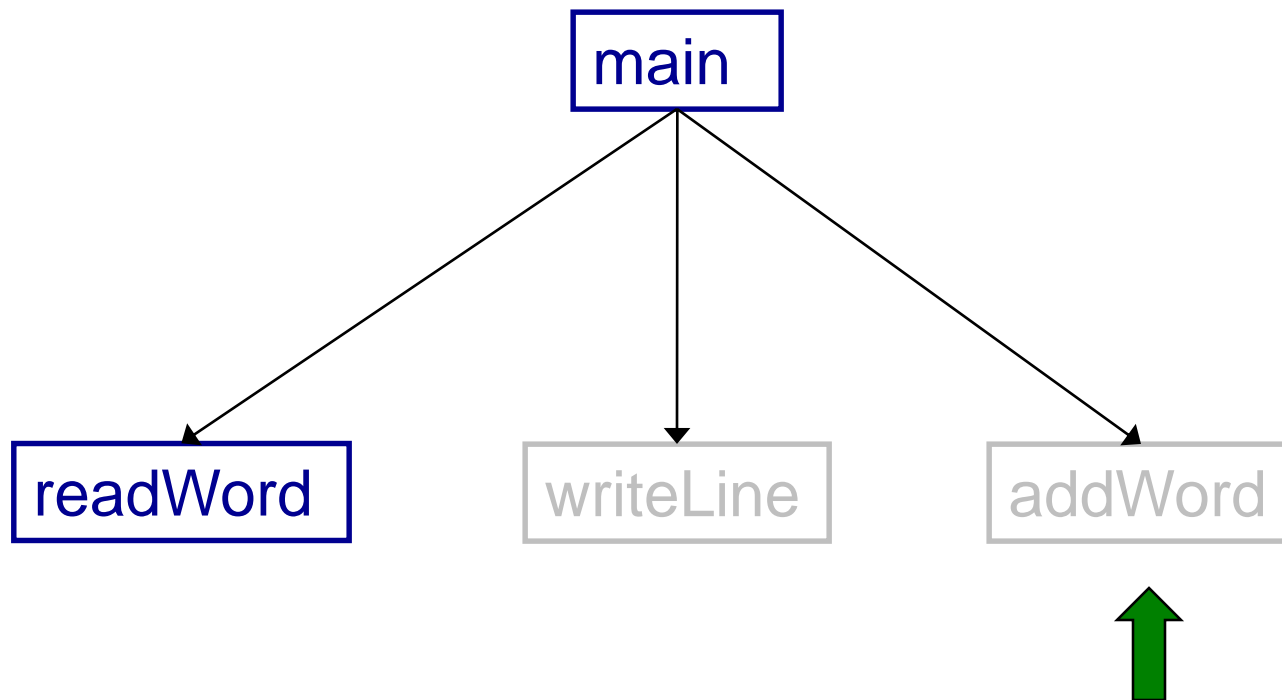
    /* Skip over white space. */
    while ((ch != EOF) && isspace(ch))
        ch = getchar();

    /* 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.

Status



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>
}
```



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>
}
```

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> ←
}

```

The addWord() Function



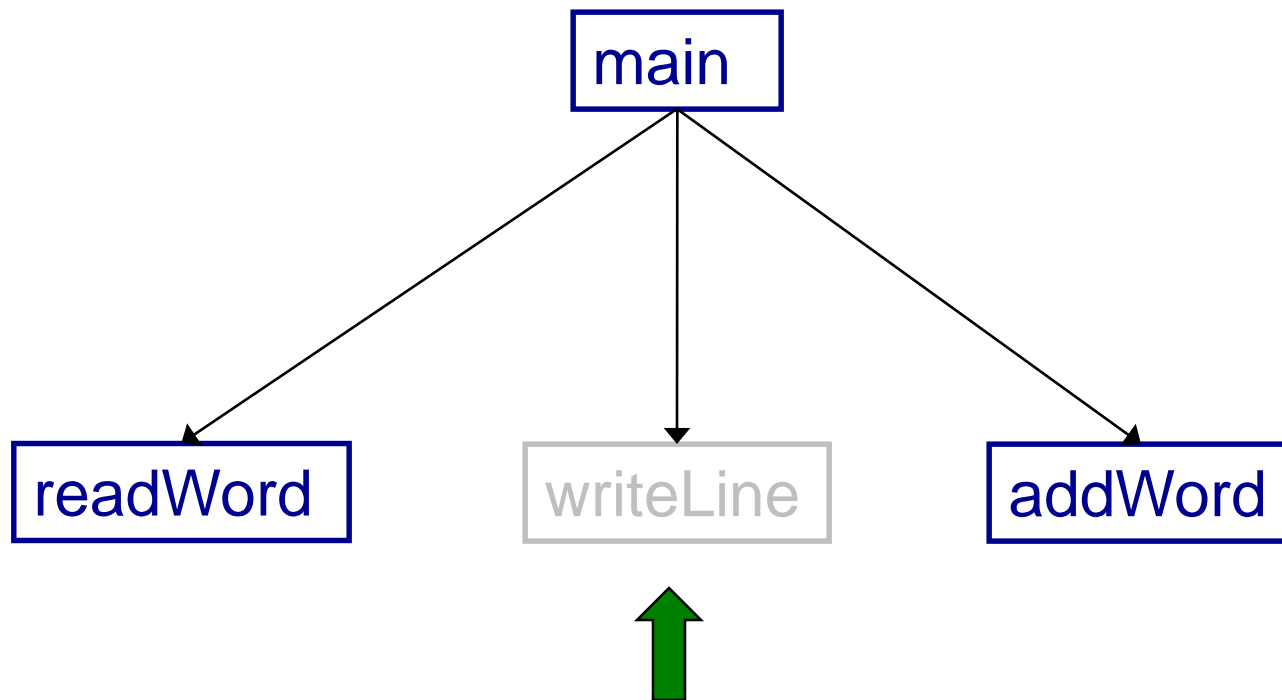
```
int addWord(const char *word, char *line, int lineLen)
{
    int newLineLen = lineLen;

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

    strcat(line, word);

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


Status



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');
}
```



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');
}
```



The writeLine() Function

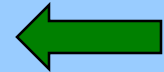
```
void writeLine(const char *line, int lineLen, int wordCount)
{
    int i, extraSpaces, spacesToInsert;

    /* 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





The writeLine() Function

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

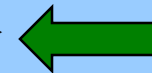
    /* Compute number of excess spaces for line. */
    extraSpaces = MAX_LINE_LEN - lineLen;

    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





The writeLine() Function

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

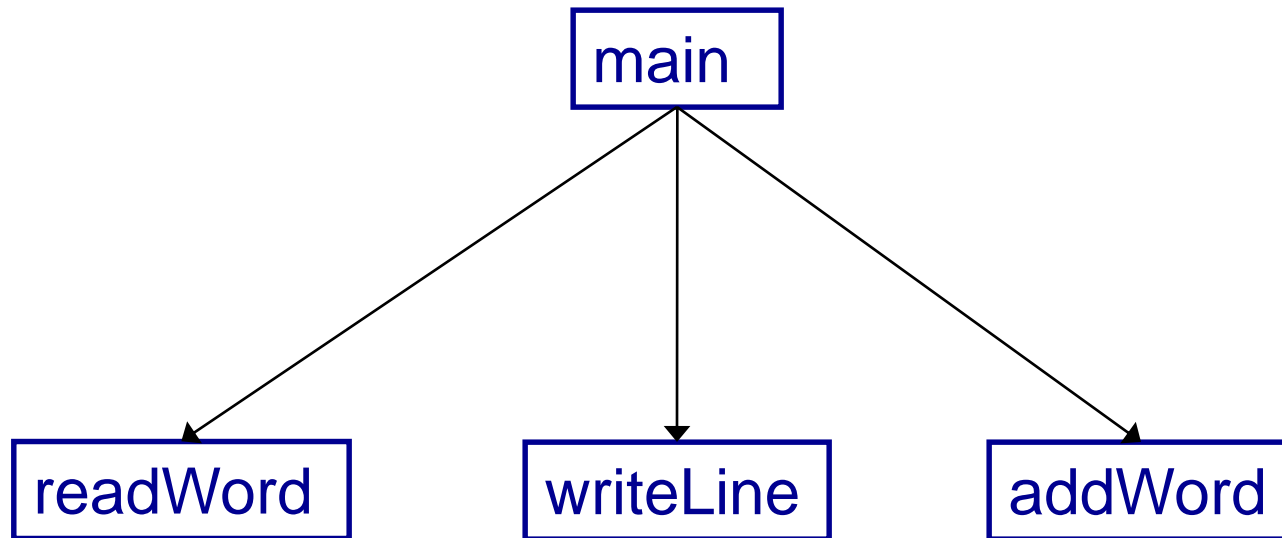
  /* Compute number of excess spaces for line. */
  extraSpaces = MAX_LINE_LEN - lineLen;

  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');
}
```

Status



Complete!

Top-Down Design and Modularity



Note: Top-down design naturally yields modular code

Much more on modularity in upcoming lectures

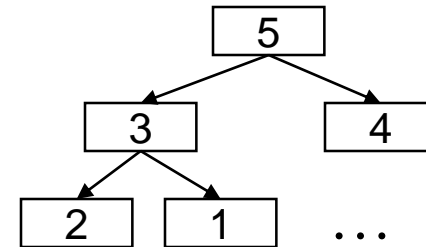


Aside: Least-Risk Design

Design process should minimize risk

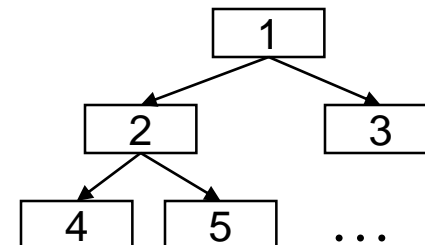
Bottom-up design

- 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

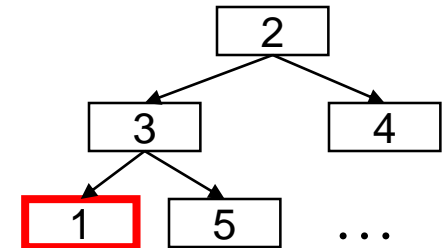


Aside: Least-Risk Design



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

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
```

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

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.

My solution used floating-point division and rounding; do "man round" and pay attention to where that man page says "include <math.h>".

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

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

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

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

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

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;  
}
```



Debugging (Part 1)



The material for this lecture is drawn, in part, from
The Practice of Programming (Kernighan & Pike) Chapter 5

For Your Amusement



“When debugging, novices insert corrective code; experts remove defective code.”

-- Richard Pattis

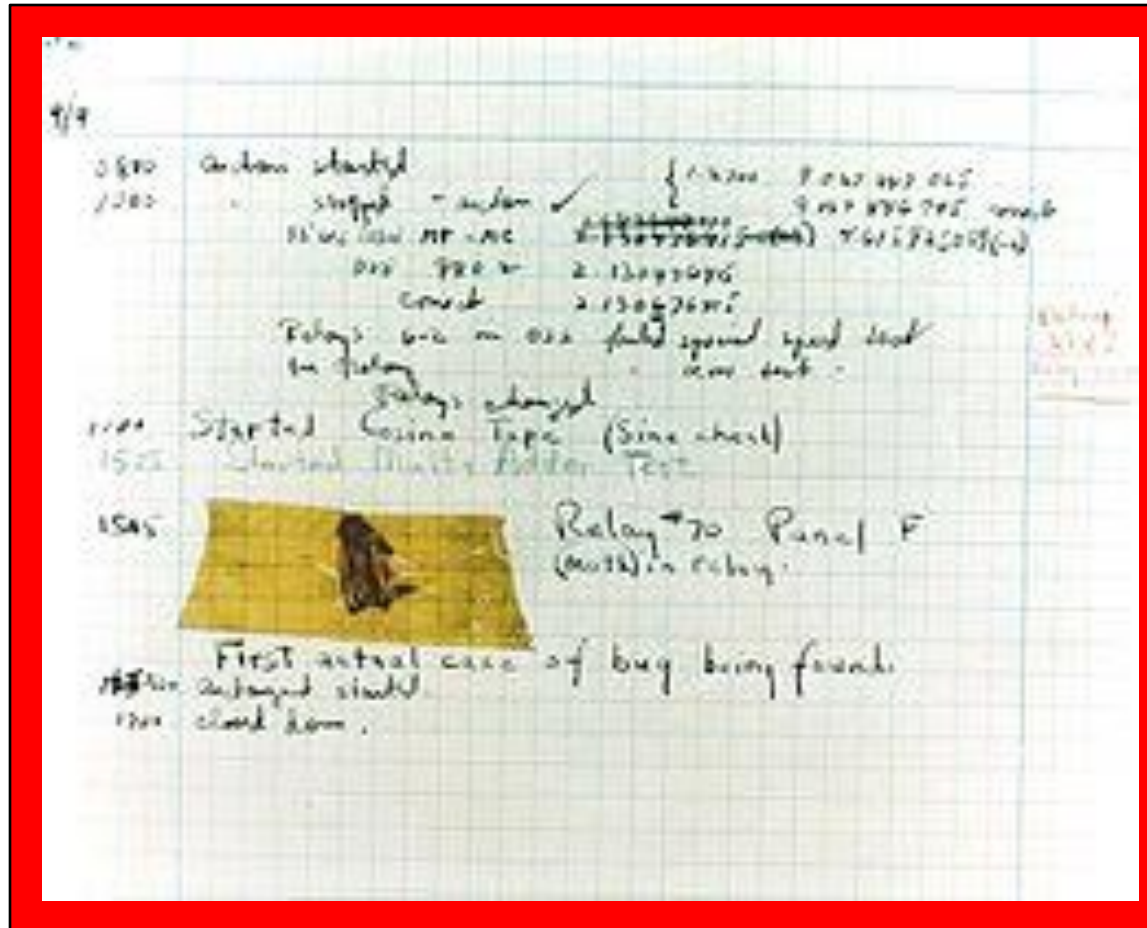
“If debugging is the act of removing errors from code, what's programming?”

-- Tom Gilb

“Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”

-- Brian Kernighan

For Your Amusement



“Programming in the Large” Steps



Design & Implement

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

Test

- Testing techniques (done)

Debug

- Debugging techniques & tools <-- we are here

Maintain

- Performance improvement techniques & tools

Goals of this Lecture



Help you learn about:

- Strategies and tools for debugging your code

Why?

- Debugging large programs can be difficult
- A power programmer knows a wide variety of debugging **strategies**
- A power programmer knows about **tools** that facilitate debugging
 - Debuggers
 - Version control systems



Testing vs. Debugging

Testing

- What should I do to try to **break** my program?

Debugging

- What should I do to try to **fix** my program?

Agenda



(1) Understand error messages

(2) Think before writing

(3) Look for familiar bugs

(4) Divide and conquer

(5) Add more internal tests

(6) Display output

(7) Use a debugger

(8) Focus on recent changes



Understand Error Messages

Debugging at **build-time** is easier than debugging at **run-time**, if and only if you...

Understand the error messages!

```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0.
int main(void)
{   printf("hello, world\n");
    return 0;
}
```

What are the errors? (No fair looking at the next slide!)

Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0.
int main(void)
{ printf("hello, world\n");
  return 0;
}
```

Which tool
(preprocessor,
compiler, or
linker) reports
the error(s)?

```
$ gcc217 hello.c -o hello
hello.c:1:20: error: stdio.h: No such file or
directory
hello.c:2:1: error: unterminated comment
hello.c:7: warning: ISO C forbids an empty
translation unit
```

Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{ printf("hello, world\n")
  return 0;
}
```

What are the errors? (No fair looking at the next slide!)

Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{ printf("hello, world\n")
  return 0;
}
```

Which tool
(preprocessor,
compiler, or
linker) reports
the error?

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:6: error: expected ';' before 'return'
```

Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{   printf("hello, world\n");
    return 0;
}
```

What are the errors? (No fair looking at the next slide!)

Understand Error Messages



```
#include <stdio.h>
/* Print "hello, world" to stdout and
   return 0. */
int main(void)
{ printf("hello, world\n")
  return 0;
}
```

Which tool
(preprocessor,
compiler, or
linker) reports
the error?

```
$ gcc217 hello.c -o hello
hello.c: In function 'main':
hello.c:5: warning: implicit declaration of function
'printf'
/tmp/ccLSPMTR.o: In function `main':
hello.c:(.text+0x1a): undefined reference to `printf'
collect2: ld returned 1 exit status
```

Understand Error Messages



```
#include <stdio.h>
#include <stdlib.h>
enum StateType
{ STATE_REGULAR,
  STATE_INWORD
}
int main(void)
{ printf("just hanging around\n");
  return EXIT_SUCCESS;
}
```

What are the errors? (No fair looking at the next slide!)



Understand Error Messages

```
#include <stdio.h>
#include <stdlib.h>
enum StateType
{ STATE_REGULAR,
  STATE_INWORD
}
int main(void)
{ printf("just hanging around\n");
  return EXIT_SUCCESS;
}
```

What does
this error
message even
mean?

```
$ gcc217 hello.c -o hello
hello.c:7: error: two or more data types in declaration specifiers
hello.c:7: warning: return type of 'main' is not 'int'
```



Understand Error Messages

Caveats concerning error messages

- Line # in error message may be approximate
- Error message may seem nonsensical
- Compiler may not report the real error

Tips for eliminating error messages

- Clarity facilitates debugging
 - Make sure code is indented properly
- Look for missing semicolons
 - At ends of structure type definitions
 - At ends of function declarations
- Work incrementally
 - Start at first error message
 - Fix, rebuild, repeat

Agenda



- (1) Understand error messages
- (2) Think before writing**
- (3) Look for familiar bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes

Think Before Writing



Inappropriate changes could make matters worse, so...

Think before changing your code

- Explain the code to:
 - Yourself
 - Someone else
 - A Teddy bear?
- Do experiments
 - But make sure they're disciplined



Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs**
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes



Look for Common Bugs

Some of our favorites:

```
switch (i)
{ case 0:
  ...
  break;
  case 1:
  ...
  case 2:
  ...
}
```

```
if (i = 5)
  ...
```

```
if (5 < i < 10)
  ...
```

```
int i;
...
scanf("%d", i);
```

```
char c;
...
c = getchar();
```

```
while (c = getchar() != EOF)
  ...
```

```
if (i & j)
  ...
```

What are the errors?

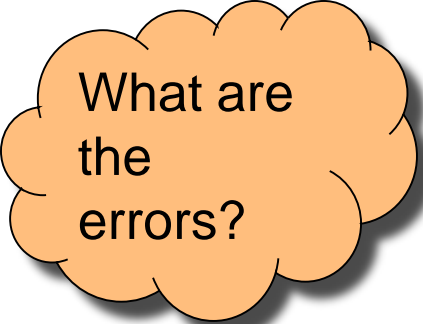
Look for Common Bugs



Some of our favorites:

```
for (i = 0; i < 10; i++)
{
  for (j = 0; j < 10; i++)
  {
    ...
  }
}
```

```
for (i = 0; i < 10; i++)
{
  for (j = 10; j >= 0; j++)
  {
    ...
  }
}
```



What are the errors?

Look for Common Bugs



Some of our favorites:

```
{  int i;
  ...
  i = 5;
  if (something)
  {  int i; ←
    ...
    i = 6;
    ...
  }
  ...
  printf("%d\n", i);
  ...
}
```

What value is written if this statement is present? Absent?

Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer**
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes

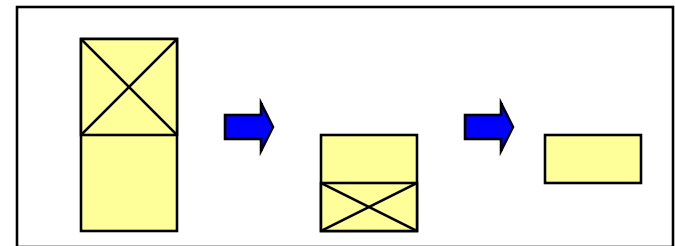
Divide and Conquer

Divide and conquer: To debug a **program...**

- Incrementally find smallest **input file** that illustrates the bug

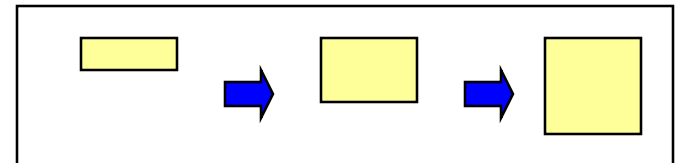
- Approach 1: **Remove** input

- Start with file
- Incrementally remove lines until bug disappears
- Examine most-recently-removed lines



- Approach 2: **Add** input

- Start with small subset of file
- Incrementally add lines until bug appears
- Examine most-recently-added lines





Divide and Conquer

Divide and conquer: To debug a **module**...

- Incrementally find smallest **client code subset** that illustrates the bug
- Approach 1: **Remove** code
 - Start with test client
 - Incrementally remove lines of code until bug disappears
 - Examine most-recently-removed lines
- Approach 2: **Add** code
 - Start with minimal client
 - Incrementally add lines of test client until bug appears
 - Examine most-recently-added lines

Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests**
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes

Add More Internal Tests



(5) Add more internal tests

- Internal tests help **find** bugs (see “Testing” lecture)
- Internal test also can help **eliminate** bugs
 - Validating parameters & checking invariants can eliminate some functions from the bug hunt

Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output**
- (7) Use a debugger
- (8) Focus on recent changes



Display Output

Write values of important variables at critical spots

- Poor:

```
printf("%d", keyvariable);
```

`stdout` is buffered;
program may crash
before output appears

- Maybe better:

```
printf("%d\n", keyvariable);
```

Printing '`\n`' flushes
the `stdout` buffer, but
not if `stdout` is
redirected to a file

- Better:

```
printf("%d", keyvariable);  
fflush(stdout);
```

Call `fflush()` to flush
`stdout` buffer
explicitly



Display Output

- Maybe even better:

```
fprintf(stderr, "%d", keyvariable);
```

Write debugging output to **stderr**; debugging output can be separated from normal output via redirection

- Maybe better still:

```
FILE *fp = fopen("logfile", "w");  
...  
fprintf(fp, "%d", keyvariable);  
fflush(fp);
```

Bonus: **stderr** is unbuffered

Write to a log file

Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger**
- (8) Focus on recent changes

Use a Debugger



Use a debugger

- Alternative to displaying output

The GDB Debugger



GNU Debugger

- Part of the GNU development environment
- Integrated with Emacs editor
- Allows user to:
 - Run program
 - Set breakpoints
 - Step through code one line at a time
 - Examine values of variables during run
 - Etc.

For details see precept tutorial, precept reference sheet,
Appendix 1

Agenda



- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
- (8) Focus on recent changes**

Focus on Recent Changes



Focus on recent changes

- Corollary: Debug now, not later

Difficult:

- (1) Compose entire program
- (2) Test entire program
- (3) Debug entire program

Easier:

- (1) Compose a little
- (2) Test a little
- (3) Debug a little
- (4) Compose a little
- (5) Test a little
- (6) Debug a little
- ...

Focus on Recent Changes



Focus on recent change (cont.)

- Corollary: Maintain old versions

Difficult:

- (1) Change code
- (2) Note new bug
- (3) Try to remember what changed since last version

Easier:

- (1) Backup current version
- (2) Change code
- (3) Note new bug
- (4) Compare code with last version to determine what changed

Maintaining Old Versions



To maintain old versions...

Approach 1: Manually copy project directory

```
...  
$ mkdir myproject  
$ cd myproject
```

Create project files here.

```
$ cd ..  
$ cp -r myproject myprojectDateTime  
$ cd myproject
```

Continue creating project files here.

```
...
```



Maintaining Old Versions

Approach 2: Use a **Revision Control System** such as subversion or git

- Allows programmer to:
 - **Check-in** source code files from **working copy** to **repository**
 - **Commit** revisions from **working copy** to **repository**
 - saves all old versions
 - **Update** source code files from **repository** to **working copy**
 - Can retrieve old versions
- Appropriate for one-developer projects
- Extremely useful, almost *necessary* for multideveloper projects!

Not required for COS 217, but good to know!

Google “subversion svn” or “git” for more information.

Summary



General debugging strategies and tools:

- (1) Understand error messages
- (2) Think before writing
- (3) Look for common bugs
- (4) Divide and conquer
- (5) Add more internal tests
- (6) Display output
- (7) Use a debugger
 - Use GDB!!!
- (8) Focus on recent changes
 - Consider using RCS, etc.

Appendix 1: Using GDB



An example program

File testintmath.c:

Euclid's algorithm;
Don't be concerned
with details

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

```
int gcd(int i, int j)
```

```
{ int temp;
```

```
  while (j != 0)
```

```
  { temp = i % j;
```

```
    i = j;
```

```
    j = temp;
```

```
  }
```

```
  return i;
```

```
}
```

```
int lcm(int i, int j)
```

```
{ return (i / gcd(i, j)) * j;
```

```
}
```

```
...
```

```
...
```

```
int main(void)
```

```
{ int iGcd;
```

```
  int iLcm;
```

```
  iGcd = gcd(8, 12);
```

```
  iLcm = lcm(8, 12);
```

```
  printf("%d %d\n", iGcd, iLcm);
```

```
  return 0;
```

```
}
```

The program is correct

But let's pretend it has a
runtime error in **gcd()**...

Appendix 1: Using GDB



General GDB strategy:

- Execute the program to the point of interest
 - Use breakpoints and stepping to do that
- Examine the values of variables at that point

Appendix 1: Using GDB



Typical steps for using GDB:

(a) Build with `-g`

```
gcc217 -g testintmath.c -o testintmath
```

- Adds extra information to executable file that GDB uses

(b) Run Emacs, with no arguments

```
emacs
```

(c) Run GDB on executable file from within Emacs

```
<Esc key> x gdb <Enter key> testintmath <Enter key>
```

(d) Set breakpoints, as desired

```
break main
```

- GDB sets a breakpoint at the first executable line of `main()`

```
break gcd
```

- GDB sets a breakpoint at the first executable line of `gcd()`

Appendix 1: Using GDB



Typical steps for using GDB (cont.):

(e) Run the program

run

- GDB stops at the breakpoint in main()
- Emacs opens window showing source code
- Emacs highlights line that is to be executed next

continue

- GDB stops at the breakpoint in gcd()
- Emacs highlights line that is to be executed next

(f) Step through the program, as desired

step (repeatedly)

- GDB executes the next line (repeatedly)

• Note: When next line is a call of one of your functions:

- **step** command *steps into* the function
- **next** command *steps over* the function, that is, executes the next line without stepping into the function

Appendix 1: Using GDB



Typical steps for using GDB (cont.):

(g) Examine variables, as desired

```
print i  
print j  
print temp
```

- GDB prints the value of each variable

(h) Examine the function call stack, if desired

```
where
```

- GDB prints the function call stack
- Useful for diagnosing crash in large program

(i) Exit gdb

```
quit
```

(j) Exit Emacs

```
<Ctrl-x key> <Ctrl-c key>
```

Appendix 1: Using GDB



GDB can do much more:

- Handle command-line arguments
`run arg1 arg2`
- Handle redirection of stdin, stdout, stderr
`run < somefile > someotherfile`
- Print values of expressions
- Break conditionally
- Etc.