Good Programming

CS 217

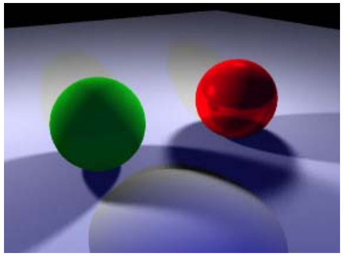
Read: chapters 1 and 4 of “The Practice of Programming”

Overview of Today's Class

- Programming style
  - Layout and indentation
  - Variable names
  - Documentation
- Modularity
  - Modules
  - Interface and implementation
  - Example: left and right justifying text

Programming Style

- Who reads your code?
  - Compiler
  - Other programmers



- Which of these cares about style?

```
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-u1e-7?b-u:b+u,tmin=u=1e-7&&u<tmin?best=s,u:
tmin;return best;}vec trace(level,P,D)vec P,D;{double d,eta,e;vec N,color;
struct sphere*s,*l;if(!level--)return black;if(s=intersect(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=l -
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)

Programming Style

- Why does programming style matter?
  - Bugs are 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?
  - Structure
  - Conventions
  - Documentation
  - Modularity

Convey Structure: Space and Indenting

- 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;
```

- Can often rely on auto-indenting feature in editor

Represent Code in “Paragraphs”

- Use blank lines to divide the code into key parts

```
#include <termios.h>
#include <unistd.h>

int main(int argc, char **argv) {

  /* Set the input to no-echo, character-at-time
   * ("cbreak") mode,
   * and remember the old mode in t0 */
  struct termios t0, t1;
  tcgetattr(0,&t0);
  t1 = t0;
  t1.c_lflag &= !(ECHO|ICANON);
  tcsetattr(0,0,&t1);

  run();

  /* Set the terminal back to its original mode */
  tcsetattr(0,0,&t0);

  return 0;
}
```

Use Natural Form for Expressions

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

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

- Good code

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

- Conditions should read like you'd say them aloud
  - Not "Conditions shouldn't read like you'd never say them aloud"!

Parenthesize to Resolve Ambiguity

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

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

- Good code

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

- Better to make the groupings explicit
  - Relational operators (e.g., ">") have precedence over logical operators (e.g., " &&"), but who can remember these things?

Another Example With Parentheses

- Example: Read and print character until the end-of-file.
- Right code

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

- Wrong code (what will it do???)

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

- Must make the grouping explicit
  - Logical operators (e.g., “!=“) have precedence over assignment (“=“)

Break Up Complex Expressions

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

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

- Good code

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

- Lining up the parallel structures is helpful, too!

Use Consistent Indentation

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

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

Wrong code
(else matches "if day > 29")

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

Right code

Note: The "&" means "mod"

Use Common C Idioms

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

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

- Good code

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

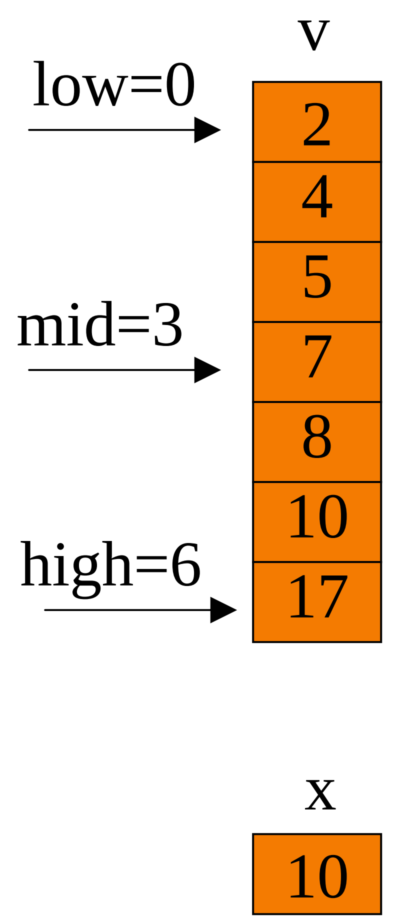
Use “else-if” for Multi-way Decision

- 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;
```



Follow Consistent Naming Style

- Descriptive names for globals and functions
  - E.g., **`display, CONTROL, CAPACITY`**
- 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)
- Consistent style for compound names
  - E.g., **`frontsize`**, **`frontSize, front_size`**
- Active names for functions
  - E.g., **`getchar()`**, **`putchar()`**, **`check_octal()`**, etc.

Documentation

- Comments should add new information
  - `i = i + 1;   /* add one to 1 */`
- Comments must agree with the code
  - And change as the code itself changes… ☺
- Comment procedural interfaces liberally
  - Inputs, outputs, and what’s going to happen
- Comment sections of code, not each line of code
  - E.g., “Sort array in ascending order”
- Master the language and its idioms
  - Let the code speak for itself

Modularity

Dividing Programs into Modules

- Big programs are harder to write than small ones
  - “You can build a dog house out of anything.” – Alan Kay
  - “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
  - Understanding *what* something does without knowing *how*
  - Separation of the *interface* (.h) from the *implementation* (.c)
    - Client must use the interface correctly
    - Implementations must do what they say they will do
- Examples
  - Sorting an array of integers
  - Character I/O, like **`getchar()`** and **`putchar()`**
  - Mathematical functions, like **`lcd()`** and **`gcm()`**
  - Set, stack, queue, list, tree, hash, etc.

An Example: Text Formatting

- Goals of the example
  - Illustrate the concept of modularity
  - Demonstrate how to go from problem statement to code
  - Review and illustrate C constructs from earlier lectures
- Text formatting (from Section 15.3 of the King book)
  - Input: ASCII text, with arbitrary spaces and newlines
  - Output: the same text, left and right justified
    - Fit as many words as possible on each 50-character line
    - Add even spacing between words to right justify the text
    - No need to right justify the very last line
  - Simplifying assumptions
    - Word ends with space, tab, newline, or end-of-file
    - Truncate any word longer than 20 characters

Example Input and Output

I N P U T

Tune   every   heart   and   every   voice.
Bid every   bank withdrawal.
Let's all   with our   accounts rejoice.
In funding Old Nassau.
In funding Old Nassau we spend more money every year.
Our banks   shall give, while   we   shall   live.
We're funding   Old Nassau.

O U T P U T

Tune every heart and every voice. Bid every bank withdrawal. Let's all with our accounts rejoice. In funding Old Nassau. In funding Old Nassau we spend more money every year. Our Banks shall give, while we shall live. We're funding Old Nassau.

Thinking About the Problem

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

Subdividing the Program

- Key constructs
  - Word
  - Line
- Source files
  - word.c (and word.h)
  - line.c (and line.h)
  - fmt.c, the main program
- Next steps
  - Write psuedocode for main program
  - Identify necessary *word* and *line* functions
  - Start writing (and testing) individual functions

Pseudocode for the Main Program

```
for ( ; ; ) {
    read a word;
    if (can’t read any more words) {
        print last line with no justification;
        terminate the program;
    }
    if (word doesn’t fit on this line) {
        print current line with justification;
        clear the line buffer;
    }
    add the new word to the line buffer;
}
```

Main Program: Format Text

```
#include <string.h>
#include “line.h”
#include “word.h”

#define MAX_WORD_LEN 20

main() {
   char word[MAX_WORD_LEN + 1];
   int word_len;

   clear_line();
   for ( ; ; ) {
      read words and do stuff;
   }
}
```

Main Program: “Do Stuff”

```
read_word(word, MAX_WORD_LEN+1);
word_len = strlen(word);

/* If reached the end, print last line */
if (word_len == 0) {
   flush_line();
   return 0;
}

/* If the word won’t fit, print the line */
if ((word_len + 1) > space_remaining()) {
   write_line();
   clear_line();
}
add_word(word);
```

Words: Reading a Character

- Words are pretty easy
  - Just need to read from stdin one word at a time
  - Though, we want to convert newlines and tabs to white spaces
- Reading a character

```
#include <stdio.h>
#include “word.h”

int read_char(void) {
   int ch = getchar();

   if ((ch == ‘\n’) || (ch == ‘\t’))
     return ‘ ‘;
   return ch;
}
```

Words: Reading a Word

```
void read_word(char *word, int len) {
   int ch, pos = 0;

   /* Skip the blanks between words */
   while ((ch = read_char()) == ' ')
      ;

   /* Store characters up to max length */
   while ((ch != ' ') && (ch != EOF)) {
      if (pos < len)
         word[pos++] = ch;
      ch = read_char();
   }
   word[pos] = '\0'; /* End the word */
}
```

Lines: Key Functions

- Clear the line buffer (clear_line)
  - Set line to string '\0' (length 0, with 0 words)
- Check amount of space left on a line (space_remaining)
  - Extra room left before reaching MAX_LINE_LEN
- Add new word to line buffer (add_word)
  - Add a blank space, unless this is the first word on the line
  - Add the new word to the end of the line
- Print line with no justification (flush_line)
  - Print the line, if length is greater than zero
- Print line with justification (write_line)
  - Determine the number of extra space in the line
  - Add extra white spaces while printing each word
  - (This is really the most challenging part of the code)

Lines: Getting Started

- Global variables, to keep it simple
  - **line**: string of the characters on the line
  - **line_len**: current number of characters on the line
  - **num_words**: current number of words on the line

```
#include <stdio.h>
#include <string.h>
#include “line.h”

#define MAX_LINE_LEN 50

char line[MAX_LINE_LEN + 1];
int line_len = 0;
int num_words = 0;
```

Lines: Simple Book-keeping

- Clearing the line buffer

```
void clear_line (void) {
   line[0] = '\0';
   line_len = 0;
   num_words = 0;
}
```

- Checking for space remaining

```
int space_remaining (void) {
   return MAX_LINE_LEN – line_len;
}
```

Lines: Add a Word to a Line

```
void add_word(char *word) {
    /* Add space after existing word */
    if (num_words > 0) {
        line[line_len] = ' ';
        line[line_len+1] = '\0';
    }

    /* Concatenate line with the new word */
    strcat(line, word);
    line_len += strlen(word);
    num_words++;
}
```

Lines: Print Without Justification

- Printing without justification
  - If line is empty, print nothing
  - Otherwise, simply print the line with the current spacing

```
void flush_line(void) {
   if (line_len > 0)
      puts(line);
}
```

Lines: Print Line With Justification

- Print-with-justification is the hardest part of the program
  - So, write as pseudocode first

```
void write_line(void) {
  compute number of excess spaces for line;
  for (i = 0; i < line_len; i++) {
    if (line[i] is not a white space)
      simply print the character;
    else {
      compute additional blanks to insert;
      print a blank, plus additional ones;
      decrease extra spaces and word count;
    }
  }
}
```

Lines: Print Line With Justification

```
void write_line(void) {
   int extra, insert, i, j;

   extra = MAX_LINE_LEN – line_len;
   for (i = 0; i < line_len; i++) {
      if (line[i] != ‘ ‘)
         putchar(line[i]);
      else {
         insert = extra/(num_words – 1);
         for (j = 0; j <= insert; j++)
            putchar(‘ ‘);
         extra -= insert;
         num_words--;
      }
   }
}
```

Modularity: Summary of Example

- To the user of the program
  - Input: text in messy format
  - Output: same text left and right justified, looking mighty pretty
- Between parts of the program
  - Word
  - Line
  - Main routine
- 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
  - Compiling the code: compile each part separately

Conclusions

- Programming style
  - Add spaces and blank lines to enhance readability
  - Pick variable and function names to enhance readability
  - Document the code to make it self-explanatory
- Modularity
  - Divide large programs into separate modules
  - Separate the interface from the implementation
  - Example: left and right justifying of text
- For more details
  - “The Practice of Programming”: chapters 1 and 4
  - “C Programming: A Modern Approach”: chapter 15, and perhaps 19
- On Thursday
  - No office hours before lecture on Thursday, for this week
  - Lecture on variable scoping and memory management