

Design variations and data abstraction

Thoughts on Assignment 1

The problem (review)

- Write a program that reads text and “fills” the lines to make them roughly the same length.
- Assume that a line with no text on it begins a new paragraph.

Example

- Input:

As I was going down the stair,
I saw a man who wasn't there.

He wasn't there again today;
he must be from the CIA.

- Output:

As I was going down the stair, I saw
a man who wasn't there.

He wasn't there again today; he must
be from the CIA.

How do we approach such a problem?

- Understand the problem thoroughly
- Design a solution
- Implement the solution (often the easiest part!)
- Figure out why what you did wasn't what you really wanted
- Repeat until satisfied or out of time

Understanding the problem

- Output lines should be no longer than a given limit
- Input is divided into paragraphs; output must also be divided into paragraphs that correspond to the input
- What else is there to know?
 - What about the rest of the relationship between the input and the output?

What we know already

- Every input character appears in the output, except possibly for white-space characters (spaces, new-lines, etc.)
- White space must be rearranged to ensure that output lines are as long as possible without being too long
- Rearrangement might replace spaces by new-lines and vice versa

What we don't know

- How do the white-space characters in the output correspond to those in the input?
 - Several different correspondences are possible
 - Even if we pick one correspondence, there is no guarantee that it is the right one
- A good solution will allow alternatives

Sample questions with unknown answers

- If an input line begins with spaces, should the output line do so as well?
 - Even after the beginning of a paragraph?
 - What if it begins with too many spaces?
- If two words have more than one space between them, should the output preserve those spaces?
 - What if the output line breaks there?
 - What if there are too many spaces?

Two possible viewpoints

- All the characters in each line are significant. We want to rearrange those lines, changing as little as we can, to meet the length requirements.
- Each line consists of words with space between them. We want to keep the words, but we can change the spaces.
- How do we choose a viewpoint?

Two approaches to design

- Preserve as much flexibility as possible
 - When you find you did the wrong thing, it will be easy to change
 - The program is likely to be complicated
- Look for the simplest definitions
 - If they're wrong, it will be easier to find what's right
 - A simple program costs less to throw away

Problems should reflect purposes

- One way to decide what problem to solve is to ask “How will we use the solution?”
 - Reformatting email messages
 - Preparing text to be printed
- Does the usage say anything about what we want the program to do?
 - In this case, not really

When in doubt, start with simple definitions

- They are easier to work with
- Maybe they will be good enough
- It is easier to make them more complicated later than to simplify them

Simple definitions for this exercise

- Every input character is either *significant* or *insignificant*.
- A *word* is a maximal sequence of one or more significant characters.
- A *paragraph break* is a maximal sequence of insignificant characters that includes two or more new-line characters.

Defining the problem

- We can now view the input as a sequence of words intermixed with paragraph breaks
- The characters between two adjacent words are interesting only if they are a paragraph break
- We can define a *token* as either a word or a paragraph break

Sketching the solution (find the bugs!)

```
n = 0    /* chars written on current line */
while ( we can read a token ) {
    if ( the token is a paragraph break ) {
        start a new output paragraph
    } else {
        lw = length(word)
        if (n+lw > max) {
            start a new line; n = 0
        }
        write the word; n += lw
    }
}
```

What bugs are there?

- The computation $n += 1w$ gives the wrong answer
 - Redefine the computation to account for spaces between words
- We never finish the last paragraph
 - Flush the output at the end
- We never break a word in the middle, even if the word is huge
 - Is this really a bug?

A bug that might not be there

- If the input begins or ends with a paragraph break, so will the output
 - This may be a bug in something, but it's not clear that it is a bug in the program
 - Nevertheless, we failed to think about it in our definitions

Where did the bugs come from?

- We wrote "`lw+=length(word)`" without thinking about whether that was what we really wanted
- We started writing programs to write paragraphs without thinking first about what paragraphs are
- In both cases, we rushed into implementation too fast

What can we do differently?

- Be even more careful about *what* we want before we think about *how* to get it.

OK, what *do* we want?

- The key notions seem to be
 - Will this word fit on the current line?
 - Append a word to the line
 - Print a line
 - Set the line to blank
- If l is a line, we will use notation like $l.append(w)$ to append word w to line l

With these notions, we can rewrite...

```
l.reset()
while (we can read a token) {
    if (token is a paragraph break) {
        l.print(); l.reset();
    } else {
        word = token;
        if (!l.canfit(word)) {
            l.print(); l.reset()
        }
        l.append(word)
    }
}
l.print(); l.reset()
```

Where are we now?

- We have defined two concepts, *line* and *token*, that are
 - abstract enough to be independent of any particular programming language and implementation
 - concrete enough that we can write and discuss programs that use them
- We still have an implementation job ahead of us

Abstract data types

- Our lines and tokens are examples of abstract data types
 - Their users know them by their properties
 - The implementation details are hidden
- Some languages have direct support for data abstraction
- Other languages make you work at it

Typical support for data abstraction

- An abstract data type is
 - a data structure
 - a collection of operations
- The implementations of the operations are allowed to know about the data structure
- The user cannot get at the data structure directly

Terminology

- The part of a program that describes an abstract data type is often called a *class definition*
- The type itself is often called a *class*
- A given class can have *objects* of that class

Homework (due Monday)

- Implement the *line* abstract data type in any language you like *except C++* (you can use the C subset of C++ if you like)
- How does your implementation perform (*i.e.* linear, quadratic, $n \log n$, etc.)? Measure it.
- (You may have some trouble doing this until after Wednesday's lecture)