Deskchecking and Debugging Output

September 5, 2013

Contents

1 The Program **2**

2 Debugging Output **6**

3 Desk Checking **10**

3.1 Example: 11

3.2 Tips on Desk Checking 14

Basic techniques of debugging.

1 The Program

The Program

The program we will work with computes some basic statistics on plain text.

```
Enter some text, terminated by an empty line:
In this lab, we will look at the basic use of an
automated debugger.  The program we will work with
computes some basic statistics on plain text.

There are 27 words in 2 sentences.
The average word length is 4.3 characters.
```

....

The Code

The main routine (`textstats.cpp`) looks like this:

```
#include <iomanip>
#include <iostream>
#include <string>

#include "words.h"


using namespace std;


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

```
  initializeWords(10000);

  cout << "Enter some text, terminated by an empty line:" << endl;
  string line;
  getline (cin, line);
  while (line != "")
    {
      addText (line);
      getline (cin, line);
    }

  cout << "There are " << getNumberOfWords()
       << " words in " << getNumberOfSentences()
       << " sentences." << endl;

  double avg = ((double)getNumberOfAlphabeticCharacters()) / getNumberOfWords();

  cout << "The average word length is "
       << setprecision(1) << fixed << avg
       << " characters." << endl;


  return 0;
}
```

....

words.h

The functions used in that code are declared in `words.h`:

```
#ifndef WORDS_H
#define WORDS_H
```

```
#include <string>

void initializeWords (int MaxWords);

void addText (std::string lineOfText);

int getNumberOfWords();

int getNumberOfSentences();

int getNumberOfAlphabeticCharacters();


#endif
```

. . . .

words.cpp

and defined in

```
#include "words.h"
#include <cstdlib>
#include <sstream>

using namespace std;

int numWords;

string* words;

void initializeWords (int MaxWords)
{
```

```
  words = new string[MaxWords];
  numWords = 0;
}

void addText (std::string lineOfText)
{
  istringstream in (lineOfText);
  while (in >> words[numWords])
    ++numWords;
}



int getNumberOfWords()
{
  return numWords;
}

int getNumberOfSentences()
{
  int numSentences = 0;
  for (int i = 0; i < numWords; ++i)
    {
      char lastChar = words[i][words[i].size()-1];
      if (lastChar == '.' || lastChar = '?' || lastChar == '!')
        ++numSentences;
    }
  return numSentences;
}


int getNumberOfAlphabeticCharacters()
```

```
{
  int numChars = 0;
  for (int i = 0; i < numWords; ++i)
    {
      for (int k = 0; k < words[i].size(); ++k)
        {
          char c = words[i][k];
          if ((c  >= 'A' && c <= 'Z') || (c  >= 'a' && c <= 'z'))
            ++numChars;
        }
    }
  return numChars;
}
```

:

. . . .

2 Debugging Output

Something's Wrong

Let's suppose that we have run this program and discovered that the values being printed for the average word length are incorrect. Looking that relevant portion of the main routine:

```
double avg = ((double)getNumberOfAlphabeticCharacters())
                     / getNumberOfWords();

cout << "The average word length is "
     << setprecision(1) << fixed << avg
     << " characters." << endl;
```

We can see that there are two different functions that contribute to this average.

- Which one is at fault?

Which Value is Wrong

- Which one is at fault?
  - The easiest way to tell is to add debugging output to print the two component values:

```
cerr << "getNumberOfAlphabeticCharacters(): "
     << getNumberOfAlphabeticCharacters() << endl;
cerr << "getNumberOfWords(): "
     << getNumberOfWords() << endl;
double avg = ((double)getNumberOfAlphabeticCharacters())
                        / getNumberOfWords();

cout << "The average word length is "
     << setprecision(1) << fixed << avg
     << " characters." << endl;
```

Once we see the output of these two lines, we can determine which function needs to be explored further.

cerr

For debugging output, we usually write to *cerr* (the *standard error* stream) rather than to *cout* (the *standard output* stream).

- *cerr* normally appears on your screen/console, just like *cout* usually does.
  - And if you have redirected *cout* to a file, *cerr* will still appear on your console
- Makes it easier to later hunt for the debugging output statements and remove them or comment them out without accidentally removing the "real" outputs.

Refactoring

Sometimes we can make it easier to add debugging output by *refactoring*, rearranging the code slightly in a way that should not affect what it does.

For example, it's easier to add debugging output if we refactor this:

```
double avg = ((double)getNumberOfAlphabeticCharacters())
                    / getNumberOfWords();

cout << "The average word length is "
     << setprecision(1) << fixed << avg
     << " characters." << endl;
```

into

```
int numChars = getNumberOfAlphabeticCharacters();
int numWords = getNumberOfWords();
double avg = ((double)numChars) / numWords;

cout << "The average word length is "
     << setprecision(1) << fixed << avg
     << " characters." << endl;
```

and then adding the debugging output:

```
int numChars = getNumberOfAlphabeticCharacters();
int numWords = getNumberOfWords();
cerr << "numChars: " << numChars << endl;
cerr << "numWords: " << numWords << endl;
double avg = ((double)numChars) / numWords;

cout << "The average word length is "
     << setprecision(1) << fixed << avg
     << " characters." << endl;
```

. . . .

Where to From There?

Once we determine which function if giving incorrect answers, we can add debugging output there until we have tracked down the problem.

Tips:

- If you believe the problem lies in a particular variable, insert debugging output after every statement that alters the value of that variable
- If there's a lot of such statements, try doing a "binary search" by inserting debugging output halfway through the chain of changes, then determining if things go wrong before or after that location.

. . . .

Tips (cont.)

- Loops can drown you in debugging data. But if you suspect that the problem occurs in a particular iteration of the loop, limit your output.
  - Suppose that you had a failure on input "This is very strange! Isn't it?" and believed that the problem actually occurred on the 4th word.

    Then avoid drowning yourself in data by changing

```
int getNumberOfAlphabeticCharacters()
{
  int numChars = 0;
  for (int i = 0; i < numWords; ++i)
    {
      for (int k = 0; k < words[i].size(); ++k)
        {
          char c = words[i][k];
          if ((c  >= 'A' && c <= 'Z') || (c  >= 'a' && c <= 'z'))
            ++numChars;
        }
```

```
    }
  return numChars;
}
```

to this:

```
int getNumberOfAlphabeticCharacters()
{
  int numChars = 0;
  for (int i = 0; i < numWords; ++i)
    {
      for (int k = 0; k < words[i].size(); ++k)
        {
          char c = words[i][k];
           if (i == 3) cerr << "Before: c=" << c << "  numChars=" << numChars << endl;
          if ((c  >= 'A' && c <= 'Z') || (c  >= 'a' && c <= 'z'))
            ++numChars;
           if (i == 3) cerr << "After: c=" << c << "  numChars=" << numChars << endl;
        }
    }
  return numChars;
}
```

. . . .

3 Desk Checking

Desk Checking

In *desk checking*, we draw a "picture" of the input data, then step through the code, line by line, updating the picture as the data values change

- a.k.a *walkthroughs* (Malik, chapter 2)

- paper-and-pencil debugging
- One of the fundamental approaches to debugging, but often under-utilized

. . . .

3.1 Example:

Example of Desk-Checking

Let's desk check the `getNumberOfSentences()` function:

```
int getNumberOfSentences()
{
  int numSentences = 0;
  for (int i = 0; i < numWords; ++i)
    {
      char lastChar = words[i][words[i].size()-1];
      if (lastChar == '.' || lastChar = '?' || lastChar == '!')
        ++numSentences;
    }
  return numSentences;
}
```

Start with a "picture: of some input data:

```
numWords: 3
words: "Hello.", "What's", "up?"
```

The go line by line though the code, updating our picture.

. . . .

Desk Checking, Line by Line

```
numWords: 3
words: "Hello.", "What's", "up?"
```

```
int numSentences = 0;
```

```
numWords: 3
words: "Hello.", "What's", "up?"
numSentences: 0
```

```
for (int i = 0; i < numWords; ++i)
```

```
numWords: 3
words: "Hello.", "What's", "up?"
numSentences: 0
i: 0
```

. . . .

Desk Checking, Line by Line (cont.)

```
char lastChar = words[i][words[i].size()-1];
```

- Now, `words[i]` is "Hello.".
  - `words[i].size()` is, therefore, 6.
- So `words[i][words[i].size()-1]` is `words[i][5]`,
  - which in turn is `"Hello."[5]`
    - which, finally, is '.'

```
numWords: 3
words: "Hello.", "What's", "up?"
numSentences: 0
i: 0
lastChar: '.'
```

. . . .

Desk Checking, Line by Line (cont.)

```
    if (lastChar == '.' || lastChar = '?' || lastChar == '!')
```

This condition is true. Our data picture does not change.

```
        ++numSentences;
```

```
numWords: 3
words: "Hello.", "What's", "up?"
numSentences: 0 1
i: 0
lastChar: '.'
```

. . . .

Here, for the first time, we had a variable change value instead of simply adding new variables with initial values. If we were doing this with paper and pen, we would simply strike out the old value and write in the new one.

Sometimes I do desk-checking with a text editor, in which case I would overwrrite the old value with the new one.

Keep On Checkin'

How long do we keep this up?

- We might continue all the way to the end of the algorithm,
- or until we have convinced ourselves that things are working properly,
- or until we actually find a bug.

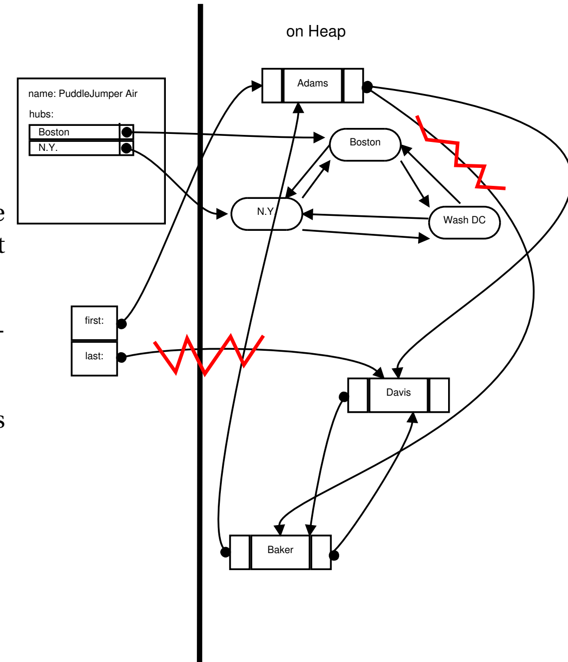
A great deal depends on the goal we had in mind when we first started desk-checking.

- Are we searching for a bug?
- Or just trying to understand some unfamiliar code?

. . . .

3.2 Tips on Desk Checking

Tips



- The example above was easy because the data was simple. There were no pointers. The only compound data structures were short arrays.
- Once we start dealing with data containing pointers, my "data pictures" will start being real pictures instead of text.
  - A white board is good for this kind of desk checking, as it allows you to erase old lines and draw new ones.

Tips (cont.)

- It helps to be very detail-oriented.
  - Look carefully at each statement to see what it *really* does, not what you *intended* it to do when you wrote it.
- Don't drown yourself in data - keep the sample inputs small and manageable

- A useful design technique: do desk checking *before* you write the code.
  - Draw a series of pictures showing how you intend to manipulate the data, step by step
  - Then figure out the code statements that will take you there.

. . . .

Walk-Throughs

In some companies, programmers are required to desk check as a team.

- You walk through your code while managers or team members watch and ask questions
- Only if you convince them that your code/design is likely to work, are you allowed to move forward and actually add it to the project.
- Often called *structured walkthroughs*

. . . .