Use of Problem Solving Skills in an Introductory Microprocessor Course

Dr. Ronald H. Rockland
New Jersey Institute of Technology

Abstract

This paper describes a ten-step approach for solving computer application problems in an introductory microprocessor course at New Jersey Institute of Technology. The steps include problem solving techniques, algorithm development, flowcharting and anticipating potential problems. Students were not expected to start developing code until the ninth step. Student experiences with these concepts and a project lab that incorporates these steps are also discussed.

Introduction

Technical people do not think effectively in assembly or higher level programming languages, but rather in their native language. However, it is too easy for students taking an introductory microprocessor course to start programming first by thinking in a computer language, without any planning. This can lead to poor programming skills, as well as frustration with programming in general.

In a third year introductory course to microprocessors in the electrical engineering technology program at NJIT, assembly language was introduced as a tool to understand the x86 structure. In the past students were given assignments to enhance their programming skills, and developed increasingly difficult programs. Over the past year, changes were made to enhance the use of problem-solving skills prior to programming.

To better understand how to approach a problem that needs to be coded, a 10-step method was created. Several of the steps are approaches used in higher level languages, and can be applied to assembly language programming. The approach could easily be applied to other types of technical problems beyond programming. This paper will describe that 10-step method, and how it was incorporated in the microprocessor course.

It should be noted that in this process, step 9 is the actual writing of code. The concept of this paper is that writing code is just one small portion of developing a program, and if all the other pre-coding steps are followed, the chances of having developed a working, rugged, and easy to debug program are greatly enhanced.

During the course, examples were given to the students to illustrate many of steps that are discussed in this paper. One of the examples that were given is detailed below.

Goal: To create a computer game with the following characteristics:

1. Computer will generate a random number, and then ask you for your name, and whether you want to play the game.
2. You need to guess that number by inputting a number and pressing the Enter key.
3. If you are higher or lower then the computer will notify you.You then will keep on trying until you guess the number. When you guess the number the computer will notify you, and then ask if you want to play again.
4. The computer will keep track of how many guesses it takes you each time, the average number of guesses each person has made, and the number of times each person has played the game.
5. You need to indicate to the computer when you want to quit (and not by turning off the computer, or throwing it against the wall).

This example will be used in several of the following steps.

Step 1 – The problem statement

If a student does not know what needs to be solved, it is impossible for that student to create a program. However, during the first time that the author taught the course, that was exactly how the students approached the development of code. When asked what they were trying to solve, most student groups had a difficult time stating the problem.

While laboratories might have a "problem statement" already listed, the expectation for each student should be to express the problem statement in his/her own words. As part of a pre-lab assignment, each group must define what is really required, as opposed to what do they want to do. Problem statements should be one or two sentences describing what needs to be done.

For the computer game example, most students had a difficult time wording the problem statement. Some would simply rewrite the problem as it appears, not presenting the problem statement in a concise manner. One possible problem statement would be "To create a random number game program where you need to guess the correct answer". This statement would then lead to a description of the specific problem and the objectives.

Step 2 – Identify Input and Output Parameters

In any technical problem, there will always be inputs and outputs. In between these elements is a processing element[1]. The inputs may include keyboard inputs, inputs that are embedded in the data segment, or inputs that come from an external source such as a transducer. However, the inputs are the parameters that are being inputted, not the methodology of inputting.

A similar statement exists for the outputs. Outputs are not the method of outputting (i.e. the monitor) but the parameters outputted. The code that will be eventually written will include a processing portion to this input, resulting in an output.

To adequately solve a problem, the inputs and outputs must be identified, their ranges specified, and their respective units known. Also, there should be a determination as to whether the outputs are a result of calculations that will need to be made. The best way to understand what are the inputs and outputs would be to draw a box, or series of interrelated boxes, that show lines from the left as inputs and lines to the right as outputs.

For the game example, the inputs would be the numbers that are guessed, the name, whether one wants to play, or whether one wants to quit. The outputs will include how many guesses it takes each time, the average number of guesses each person has made, the number of times each person has played the game, the introduction screen, whether the guess is higher or lower, whether one has identified the number, a statement whether one wants to quit, and possibly even a good-bye statement.

By using the inputs and outputs, the data segment can be adequately defined, and the number of different sub-routines that are needed to be calculated can also be determined.

Step 3- Try a sample calculation/Draw a sample screen

Visualization of how to do a problem is key to solving it. One method of visualization is to actually go through the steps by trying a sample calculation or by drawing the different screen displays. For example, if a student were required to develop a program to average numbers, this step would require the student to write down a series of numbers and then perform the average. As the student is performing the calculations, a description of each process should be written.

A similar concept exists when program development involves screen interactions. If a sample screen, whether it is an introductory screen or output screen, is drawn, then the steps necessary to develop the code become easier.

Step 4- Develop the pseudocode

Pseudocode is an artificial and informal language, similar to English[2]. Therefore, the pseudocode is written in English. The method of writing it should be top-down, where the first approach is to write approximately 4-8 lines (can be more if the program is more complex) describing the major tasks. These tasks should come from performing the previous step. The objective of writing this pseudocode is to translate each phrase to one or more lines of assembly code.

After the top level is written, expand each line down to the most basic steps. Writing pseudocode requires an understanding of what level of programming, i.e. assembly language or

high-level language, is involved. When writing in assembly language, there is a need to describe more detail than in a high level language. Getting the students to write the level of detail that is required is a challenge. Pseudocode should not include any code, such as mov ax, 01H. This forces the student to think about the problem rather than start writing code.

One method of getting the students to write the detail necessary is to tell them that the pseudocode needs to be detailed enough so that they can describe to someone else the pseudocode, and that person can write the entire program from that description. Students need to review the pseudocode several times, and challenge themselves as to whether there is more detail needed. During the semester, students are given in-class examples where they only need to develop pseudocode for the problem. After approximately 10-15 minutes, their code is reviewed, and they are shown how much additional pseudocode still needs to be written.

Many of these examples can be non-formula based. Some of these examples include:

a. Describe in pseudocode how one would visit a person whose office is located somewhere in a 50-story office building, where one only has a name.
b. Describe in pseudocode how one would give directions to a person to a specific location, i.e. a building on campus.
c. Describe in pseudocode how one would get out of a parking space.

In problem a, many students would simply state; "look up the name in a directory and go up to the office". While that might be a good top level pseudocode, more detail is needed. For example, how does one find the directory, how does one look up the name, how do you locate the elevator, how do you get up to the required floor (push the elevator button), and how do you find the office when you get off the floor. Once the students see that the level of detail is missing, they become more proficient at writing the detailed pseudocode.

For the example given in the beginning of this paper, the initial pseudocode would be to list each of the steps. However, there is a need for more detail. For example, in the first line, "Computer will generate a random number, and then ask you for your name, and whether you want to play the game", the detail pseudocode should consist of the following statements:

a. Generate a random number
b. Display the random number
c. Ask for the players name
d. Store the players name
e. Ask whether the person wants to play the game – include the person's name in the phrase
f. If the answer is yes, then continue with the rest of the program. If no, ask for a new player.

Step 5 – Develop the flowchart

For any complex program, the one failing with pseudocode is that it doesn't show the relationship between each line. A simple flowchart, with either process boxes or decision boxes, can help show the program flow. While there are many shapes that constitute a good flowchart, the students should only concentrate on a few[3]:

a. The action box (blank rectangular box), which would contain statements from the pseudocode.
b. A diamond shape decision box, which would have flow coming from the top, list the decisions inside, and have the decision flow coming out of either side (either true/false or yes/no).
c. The bubble, or small circles, which indicate the beginning and end of the programs. They can also be used as markers to indicate where a line leaves an area.

However, working with flowcharting is difficult for students not accustomed to thinking in this manner. That is why it is important that the pseudocode be developed first.

Step 6 – Develop the methodology

Once you have the detailed pseudocode, there is a need to list the methodology that will be used. For assembly language, this would include what will be in the data segment, what registers will be used, what type of addressing will be used, the types of subroutines, will a table be used, etc. This is also the area where students should decide which grouping of code should be subroutines.

This step is not where code is written, but rather where the elements of the specific code will be determined.

Step 7- Anticipate potential problems

One of the axioms that designers should follow is, "test for failure, not for success". Test for success would involve inputting data into your program, and seeing it work. Test for failure means determining what inputs or conditions cause failure, and correct these problems. In doing this, the design of the program is more rugged.

A typical example might be to develop a program that averages 5 numerical grades entered from the keyboard. Potential problems could include

a. Inputting a letter instead of a number
b. Inputting a grade greater than 100 (assuming that 100 is the highest grade)
c. Not putting in a grade at all.

Changes to the pseudocode (programming has not begun yet) could include determining that a number is being inputted rather than a letter (item a above) and checking that the grade is between 0 and 100 (items b and c above). Once the pseudocode is changed, the students should then review the methodology again.

Also, these problems should be ranked as to how reasonable it is to expect that problem, and which problems might cause the coding to increase beyond a reasonable length. If so, then the reason not to include any additional code for solving these problems should be discussed in the project report.

Step 8 – Develop a test methodology for each area/ subroutine

If the program consists of many subroutines, test methods should be developed for each subroutine. All these test methods, which include sample data and expected outcomes, should be developed before starting to write code.

Step 9 – Develop the code

Once steps 1-8 are complete, the students can begin to develop the code. For assembly language, this should be in a top-down approach, where a basic structure is first coded, including calls to subroutines. However, the subroutines should have no coding.

Then each subroutine can be developed, tested, and added to this basic structure once the subroutine has been verified. There should be testing of the basic structure after adding each subroutine.

If the pseudocode was written properly and with enough detail, much of the coding can simply be translation of each line of pseudocode to one or more lines of actual code. If the student needs to write a fair amount of code with no corresponding pseudocode, then this should be a learning experience, since the pseudocode was not written with enough detail.

Step 10 – Debug, Rewrite, and Test

Besides the test methodology for each subroutine, a test methodology needs to be developed for the overall program. During the debugging process, code may need to be rewritten. If so, the students had to review their original pseudocode to see why the changes were being made.

Discussion

The pseudocode development was considered the pre-lab portion of the lab experiment, and each student had to develop an individual pseudocode. Then the team (usually two students) would review the pseudocode, develop a single revised pseudocode, and then continue with the process.

Initially, the pre-lab consisted of just writing the pseudocode. As students felt more comfortable with this process, the pre-labs were expanded to include writing the pseudocode, flowcharts, methodology and potential problems. For the final project lab, each student was also required to develop a test methodology for the subroutines.

Initially, this was not an easy sell to the students. They had difficulty with problem statements and getting into enough detail with the pseudocode. Some students who had prior experience with computer programming felt that the whole process was a waste of time.

The last laboratory, which was a multi-week project lab, changed the mindset of most of the latter group of students. Since these requirements were more extensive, the students realized that the program was going to be much larger than previous programs. One group started to program right away, and began to have problems immediately. When they started to use the development process outlined in this paper, they realized where the problems were, and successfully finished the project lab.

Project Lab

The overall goal of this lab is to create a generalized grading system for a class. Every week there might be a test, with the grades from 0 to 100 (no extra credit problems). There will be no more than 5 tests in the semester, and from 10-12 students in the class. There will also be a final exam, which will count as 20% of the final grade. However, the program should account for the fact that there might be less than 5 tests (there will always be a final).

You will be entering all the data via the keyboard, and display the results on the monitor. The type of analysis will be based on a command that you issue via the keyboard. The commands that can be issued are AVERAGE, MAX and MIN. You want to be able to find these numbers for individual tests, or for all the tests (including the final).

You want to curve the final grade, and then assign a letter to each grade. The curve will be based on the adding two times the difference between the maximum average grade for the semester in that class and 100 (I am being very generous – don't expect this in real life). Once you have curved the numerical grades, assign the letter grades based on the following range.

Letter	**Range**
A	90-100
B	80-89

Letter	Range
C	70-79
D	55-69
F	Below 55

The initial screen should enable you to enter a student (just by the initials), the test number (i.e. test 1, test 2) and the scores for each student. You should make up test scores to verify the program (make sure that there is a range of grades when you give the final grades.

How the user works with the keyboard, what questions are asked, and how the information is displayed is strictly up to you. While creativity would be important, ease of programming might make some decisions as to how the user interface should look like.

Conclusion

A ten step method of developing programs in an assembly language class was presented. Student assessment, at the end of the semester, was very favorable to the method. It is hoped that these students in future classes will use this problem solving approach.

Bibliography

1. D.I. Schneider, *Essentials of Visual Basic 5.0 Programming*, Prentice Hall, Upper Saddle River, NJ 1999
2. H.M.Deitel, *C, How to Program*, Prentice Hall, Upper Saddle River, NJ, 1994
3. K.J. Ayala, The 8086 Microprocessor, West Publishing Company, St. Paul, MN, 1995

RONALD H. ROCKLAND

Dr. Ronald H. Rockland is an assistant professor in EET at New Jersey Institute of Technology. He received his M.S. and Ph.D. degrees in biomedical/electrical engineering from New York University, and an M.B.A. from the University of St. Thomas. His interests are in signal processing of biomedical waveforms and computer aided learning for technology students. Prior to NJIT, Dr. Rockland was a computer training consultant, and also an adjunct professor of marketing at two community colleges. He has over twenty years of industrial experience, with positions in R&D, engineering management, sales and marketing management and general management.