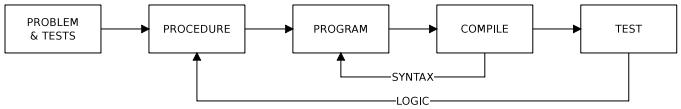
How to Write Programs

1. Introduction

There are many ways to write programs. Each person has to develop a style that works. Some methods work for some people, other methods work for other people, and one method works for almost nobody. This document will explain the method that most beginners try and will then show other methods used by people who write software for real.

2. Software Design Big Picture

This diagram shows all the steps for building software:



PROBLEM

We start with a problem to solve. Often the problem involves sample data to process. For example, the problem might be: read in text from standard input word by word, where a word is a space-separated string of non-space characters. Sample data might look like:

```
word1    word2-@#$%.txt
  9volt-alkaline    .....dots...
```

In this case, the program would read, one by one, four separate words. Sample data is an important part of the statement of a problem. Designing test data helps you refine and clarify the details of the problem. In many cases, you, the developer, are presented with sample input and asked to process it. Expanding the set of sample input helps you think through the problem. "Test-driven development" is a serious, useful technique.

PROCEDURE/PLAN

Once the problem statement begins to become clearer and some sample input is created, you move to design. Write out in plain English in outline form the steps that will solve the problem. Imagine you are explaining a procedure to another person. Use regular words. If you like, use pseudo code but do not write computer code at this point.

Carpenters often make diagrams to plan their work. Cooks often write down a schedule of when to start preparing and cooking different dishes so the meal is ready at the right time. Before the age of GPS, people actually wrote down or printed driving directions and maps before setting out. And some people make lists before they go shopping. Why would you do that? Is it not easier just to go to the market and roam up and down the aisles picking things you think you need as you see them? What could go wrong? If a carpenter/builder starts to nail boards and beams before having a plan, things might not go well.

If you hired someone to redesign your kitchen or bathroom, you should be worried if that person just showed up with some sledge hammers, boards and saws and hammers and got started. You might ask that person if he or she has a plan you can review and that he/she can follow.

Writing software is like building a new bathroom or redesigning a kitchen. It is an archtectural undertaking and requires planning and design. In real software projects, people spend a lot of time proposing, revising, discussing plans on whiteboards and in documents.

The plan should include a description of the main functional units of the system and the data

structures the system will use. An outline or call graph can help show which functions depend on which other functions. If a function seems complicated, decompose it into smaller functions. This is often easier to do in the planning stage.

PROGRAM

Once you have an outline of a procedure, you can translate that plan into code. Write functions that correspond to the functional units in your plan. Then test those functions to make sure they do what you want them to. Write simple main programs that call the individual functions to make sure the processing and return values are correct.

Put comments in your code as you write. Translate your planning logic into descriptions in the code. This may seem tedious and time-consuming but is extremely helpful. Doing so allows you to review your thoughts and clarify, refine, and revise them. If the comments do not make sense as you write them, you may not have a clear picture of what you are trying to do or how to do it. Coding without a clear understanding usually leads to tricky to find/fix bugs. It takes a lot of patience to code as if your plans might be fuzzy, but this is a skill that saves a lot of frustration and wasted time later.

COMPILE

Compile frequently with -Wall to catch syntax errors and odd usage. Go back to the program code to fix syntax errors and clean up odd warnings. This second part includes correct use of parentheses, header file includes, and unused variables.

TESTING

Test your program with the sample data you created during the problem definition stage. If the program does not produce results you want, do not go back and fiddle with the code unless the fix is simple. For most logic errors, go back to the planning outline and trace through the logic to see why your procedure is producing incorrect or unexpected results. Once you revise the outline, you can translate those changes into changes in the code. Doing so keeps your documents up to date.

