CSCI 1300 – Introduction to Computing October 31, 2003
Professor Mike Mozer
Fall, 2003

Program Design

(adapted from text by Professor Clayton Lewis)

With some reluctance, I have done a lot of "hand holding" in this course. I've done this in part by specifying the assignments very precisely, and in part by providing hints that specify data structures and algorithms. In the real world, when you are faced with a problem, you will be told more-or-less what to do, but the specifics will be left up to you. You also won't have an expert to help you organize your program, that is, how to divide it up into functions, and much less about what language features to use in attacking the problems.

Very often real problems are only partially specified, and the client (who may be you) doesn't know exactly what the program should do, beyond some basic core functionality. The client relies on you to come up with something reasonable. Similarly, the client often does not care at all how you implement the solution, and can't tell you anything about how to frame your solution.

Don't be surprised if you find this kind of problem challenging at first. Very often things are pretty fully spelled out for us in school, both in terms of exactly what we are supposed to do, and in terms of how we should do it. It takes practice to write programs without these aids. Here are some suggestions for programming in this more open-ended milieu.

Learn to expect uncertainty and learn how to manage it. Nobody can look at a complex problem and see right away how to solve it. Rather, everyone uses some trial and error. You try an approach, say a particular way to represent important data, without being sure it will work out. You try to complete the solution, realizing that you may run into a problem and have to try something else. If you have to abandon or modify your approach, you accept that as part of life and not as a mistake you made. This behavior is called backtracking, and it's normal and unavoidable.

Divide and conquer. Don't expect to solve a complex problem all at once. Rather, pick some part of the overall problem and work on just that part. When you have a workable solution to that part, work on another part. Also work on fitting the solutions for the parts together.

Insist on simple parts. The key to divide and conquer is choosing parts small enough that you can make progress on them. Usually you will write a function to do the work in each part, and these functions should be small (a dozen lines or so) and should do a job that can be described clearly in a sentence or two.

Use sketches to develop your design. Make a storyboard, a sketch like a comic strip, to show what the user will see and do when they use your program. Make structure charts (showing which functions call which other functions) and flowcharts or pseudocode (showing the basic algorithms). Identify the key functions that will be created to do the work the program requires, the order in which they will be used, and what information goes into and comes out of each one. Don't start writing code until you are confident that your design is workable: it's a lot cheaper to go back and change the sketch than to throw away a lot of code.

Work on the hardest things first (mostly). There's no point in working up all the easy parts only to find that you have to use a different approach when you finally tackle the hard stuff. Try first to crack the hard parts by breaking them up into simple parts that you can see how to do. On the other hand, sometimes it helps in attacking the hard parts to warm up on some easy parts. Try this if the hard parts keep putting you off.

Test as you write. If your functions do simple jobs, it will be simple to test them and make sure they are working. Test each function as you write it, rather than writing a whole nest of functions and testing them all together. Doing this will greatly simplify your overall development because it will make it much easier to find and fix bugs. Only if you are very confident about what you are doing does it make sense to defer testing.

Provide functions early to create and display the things your program works with. If your program processes representations of widgets, testing the functions will be much easier if you have given yourself a way to create widgets and display them out in some intelligible way. Even if the ultimate user never needs to see the widgets, you should provide these functions for your own use in testing your code.

Don't worry about efficiency (for now). The important thing to work on now is how to write programs that are simple and clear, not fast but hard to decypher. You can learn later on how to decide what few parts of your program (if any) are worth making more complex to gain speed. You'll also learn that for many problems, speed is attainable only by choosing the right algorithm, or basic method. You aren't likely to come up with good algorithms by using little tricks in your code. Rather, you need to become familiar with good algorithms for common kinds of problems, which is the subject of later courses.

Practice the principle "code one assumption in one place." This principle means that you should aim to write your programs in such a way that if you have to change some assumption, such as how many frammises there can be in a widget, you only need to change one line. You often won't always attain this ideal, but it's important to strive for it, for two reasons. First, it makes your program easier to maintain, that is, to keep in operation over time, given that what programs are supposed to do constantly changes. This is extremely important in real life, where maintenance costs for software commonly greatly exceed the cost of initial development. Second, doing this will actually make your program easier to write. Many hard to find errors creep in because as you develop your solution you have to change things, but you forget to make the changes everywhere they are needed. Striving for "one assumption, one place" gives you code in which there are fewer changes to keep track of as you work.

Make your functions check for errors. This gives a more reliable program, but also a program that's easier for you to work with. For example, if you are storing data in an array, overflowing the bounds of the array often produces truly baffling symptoms. If your functions detect this happening and tell you, it can save lots of grief.

When debugging, make your program tell you what it is doing, rather than staring at it. The psychology of programming is such that when you read your code you see what you think should be there rather than what you really wrote. For example, you can read `x=y` a thousand times and not realize it should be `x==y`: you know you meant a comparison and you read it that way even though it is not. What you have to do, after looking at the code briefly to catch obvious mistakes, is make the program display what it is doing, often enough and in enough detail that you can see what it is really doing, not what it is supposed to be doing. The debugger is one way to do this; another is to insert statements into your program that tell you what's happening. Keeping your functions small and simple helps greatly with this: you may only need to have your functions display the arguments they get and the results they return.

Don't let your code get away from you. There's a temptation to fix problems by making more or less random changes until you get what you want. This is fatal in the long run, even if you get lucky and you happen to make the correct change (which isn't likely anyway). The problem is that after a change like that you no longer know what your program is doing, and when new problems occur you can't reason about them. The code stops being your program and takes on a life of its own. Never make a change to your code if you don't feel you understand the change. (The temptation to make such changes is strongest, and hardest to resist, when you are tired. If you are getting worn out trying to fix a problem, stop and do something else for a while, like sleep. Resist the urge to get "that last bug." Once fatigue sets in, chances are you'll waste time and introduce errors.)

Develop a library of functions that you use often. Rather than rewriting different parts of your programs, re-use functions that you have written before. Good candidates to include in your library are functions for dealing with input and output files, functions for graceful exits from program, functions to initialize the graphics system. Use the `#include` command to include declarations of these functions into a program when you need them. You might also keep a collection of "standard examples," e.g., of loops using arrays, that you can refer to and imitate when developing a new program.

So how are you going to learn to do all these things? One way is by paying attention to the examples we work through in lectures and recitations. You'll get to see how we tackle problems using our collective skills. Another way is working on your assignments and rereading this advice when you get stuck. Bottom line: The more you program, the better you'll get at it.

You may benefit by keeping a log of neat ideas or tricks that you think you will be able to apply to other problems, problems you have faced in the past and your solution, and bugs that you had a hard time tracking down in your code. The most effective learning comes from a combination of doing and reflecting.

Program Design Checklist

Here are some points to keep in mind as you work on programming projects. These are also points we keep in mind as we are grading your work.

1. Does the program do what the specification asks for?
2. Is the user interface for the program reasonable? For example, are there adequate prompts or cues for input, and is output clearly labelled and presented?
3. Have you developed structure charts, flowcharts, and pseudocode for the program to provide an overview of the high-level structure?
4. Does the program follow the suggested style for indentation and layout?
5. Are variable and function names informative?
6. Are there comments included where the operation of the the program is unclear, and not where it is obvious?
7. Are appropriate data structures used, and do they organize related data in a clear way? For example, when processing collections of similar things did you use arrays rather than separate variables, and are your data structures used to group together different information about the same entity?
8. Is the logic of the program simple and clear, rather than complex and tricky?
9. Is the program organized into short functions that perform simple jobs?
10. Would the program be easy to modify if requirements change, with assumptions represented in one place?
11. Is the program free of unneeded material, such as variables that are not used, or `#includes` for libraries that are not used?
12. Does the program check for and handle errors?
13. Does the program avoid global variables? (This is a case of "do what I say, not what I do"; I occasionally use global variables in class demos to save time. That does not mean that I would like you to use global variables in your assignments.)