Philosophy of design

Applications should be light, fast, and responsive to the user. Always. I paid for the machine, I will use it all, memory, storage space, multiple threads, processor cycles, and the graphics card. It must respond to mouse clicks and keystrokes immediately. The effect of the event may be delayed, but input should NEVER be blocked. Multi-threading is your friend and your enemy; come to terms with it. Use the code you write. If you don't like it make it better. Do not force the user to become an alpha tester. NEVER block access to the customer's stored data! If you sin, and break the interface between the application and older data, you MUST create a file converter. Your client spent valuable time creating that data, you should protect their investment.

I use C for a number of reasons. It is compact and close to the machine. It is fairly easy to learn. AT&Ts gift of UNIX to academia created a legacy of tools and applications. Then there is the free gcc compiler suite, which helps me get work done.

I don't use C++ unless the program I want to write lends itself to many similar objects. If there are only a few objects in the whole program it wastes time and space, while opening a chasm for bugs to invade. I don't use Python because it has a primitive graphic user interface, it is an interpreted language, and there is not a huge library of existing code for it.

Everything I am writing in these notes will not be immediately applicable. I am simply writing my thoughts of the areas of difficulties I, and others, have had with writing code. My last formal teaching experience was remedial. The ECE undergrads were being taught how to write software by the CS department. That was a mistake. They were taught C++, and object oriented code, from day one. What a confusing waste of time. They could not write a simple routine to do much of anything. My department received complaints from prospective employers at Sandia National Labs. None of the folks interviewing for a position could write anything which would fit on a microcontroller. So we got the task of reteaching them how to write code in C.

Software engineering taught me to design a program before thinking about which language to use. Some languages fit certain problems better. C is best at being close to the metal of the processor, while simultaneously allowing one to think in higher level terms. C++ doesn't allow that, Java doesn't allow that, nor do any interpreted languages. I don't like wasting processor cycles, or memory space; that comes from time spent writing assembly language routines.

Don't write fancy code! You may be required to modify it six months later, and you won't understand it. Comment like your life depends on it, it does. Don't put too many features into one app, complexity makes the tool less useful. Simply write another program to do the other stuff. What you'll find is that much of the code you've written for the first program which can be recycled into second one. Write enough code in a given problem space and you get a variety of tools to solve those problems, and add to your library.

You don't need to understand every single line of code you read, or copy. Use what others have developed, with your own modifications. Understand what the major chunks do, then figure out where

your code can fit. Ignore the rest; it works, let it be. Build your code from the application programmer's interface (APIs) of others, it will speed up your work and give you a tested base. Put a test routine into your makefile. Run a **make test** after every compile if necessary. Make sure to trap all your input code modules. That is where hackers find an opening. Close it before there is a problem.

Perfection is not possible; when is it good enough?

If your function requires more than one screen-full of code it needs to be factored. Ultimately your main function will be a series of function calls. Each function can be built from the routines you have written, or of the language itself. Being able to see your entire program in discrete chunks aids development in many ways. You can better see the flaws in your logic. You can examine the entire app easily. You can jump to the function you are debugging, work on it separately, or make a simple test harness. Working in discrete chunks helps you work and design top-down, while keeping the app compiled, and executable while you're writing. I find it lets me write working code faster. Seeing each chunk shows me where to refactor, to make a function smaller and more useful.

Employers/clients like to see working code as soon as possible. Adding all the promised features occurs on a day by day basis, while you are writing and testing each feature's functions. This also gives you a better idea of how long each new feature will take to implement. Don't let the client/employer change the design in midstream. Scream, yell, and beat on them if you have to. Don't mess with the design once you start coding!!! You should have hashed out all the features in the initial design conferences. Tell the client/employer to prioritize: get this project done on time and on budget, OR redesign the app, and let your deadline slip by months while the paychecks need to keep coming. That last argument does work occasionally :)

I am attempting to teach you how to program. Which language you choose is unimportant, they are all fundamentally equivalent. Creating an application is about defining the problem, then decomposing it into smaller and smaller parts. When those parts get too small just write them in the language of choice. But the decomposition and writing of pseudo code is where the design takes place. The problem I am having with these lessons is me trying to fit dozens of college courses, and forty years of experience into them. I need you to throttle the flow and direct me. ASK QUESTIONS! The only stupid question is one you do not ask!

I use a plain editor, instead of an integrated development environment (IDE) because of impatience (along with laziness & hubris). Using a makeFile gives me better control over the compile and link processes. It also requires very little in the way of computer resources. When using makefiles for microcontrollers, you can direct where each portion of code, or data, will go in memory. Some of it will burn to flash, and some to EEPROM. One memory area is for the program, the other is for any hard data it may need. You can also write to the EEPROM from your application. Using a makefile gives me knowledge about the processor, the memory structure, and the libraries necessary to link everything.

I like to have my code compile and run, at least a skeleton of what I want, quite early in the process. Then I add to the framework fleshing out the details piece by piece. I test as I go so I don't need to

debug large sections of code. However, much to my teacher's and employer's dismay, I dismantle the testing framework after I know my code is running well. I could set some flags to conditionally compile in the debug code, but I find the code is much cleaner, and easier to reuse, if I remove the scaffolding as I go. Writing software is a craft, so I use ideas from the other guilds of craftsmen. Do you want to see scaffolding in the middle of your cathedral? I don't either. I want my code to read well, and display its purpose. Once the functions have been thoroughly tested, I know how they will respond. I retain debugging code in error traps for input modules; either input from the user, or input from a disk file. You need to worry about what the user is typing into your app. Are they trying to add a letter instead of a number? Are they trying to enter too many characters for a given space? You need to check their input for validity in each case. Input routines are well known avenues to hack a system.

I find the cleanest looking code, with the most comments, gets reused. If the code is not readable it won't get reused. Over time, my most reused code gets generalized to fit more use cases, and it gets commented the most liberally. I took four assembly language programming courses in school. The first one required commenting extensively; it was a major part of the grade for each assignment. I found that to be good training, and have kept up the practice ever since. I am the one who works with this code most often. I need to be able to read it quickly, and understand exactly what I was intending to do when I wrote it. So comment your code heavily. In assembly language that means almost every single line. In a higher level language, like C, you can comment a little less than that. But, any time you do something out of the ordinary, or tricky, comment as much as possible. Because, in one year or less, you will be reading that code wondering what you meant it to do. Those comments are for you, not for your instructor or your employer. You need to know what is what at all times. When I was writing code in the language Forth I learned to use the best description of what I was doing as the function name. In this way the code was self-commenting. I still try to use the best word for the function name but comment a lot more than I did then. The years have a way of making you learn.

Why C/C++ instead of some other language?

Why command line? Why make files?

There are many computer languages one can use to solve a problem. Some languages fit a problem better than others. Some languages are easier to read. Some languages use a lot of processor time, while others use much less. C is a good first language, because it teaches you the connection between a

computer's memory and how the language uses the memory. It is easier to determine how much memory is necessary to run an application, by knowing how the language uses memory. If you want to fit a large program into a small space, C does very well for a higher level language. You can use assembly language to write a program to use the least space. Then you have the fastest code but it takes a lot more time to write. However, with C you can optimize the few places in your program which use the most time, by writing them in assembly language. I have not done this in years because computers keep getting faster and memory keeps getting cheaper.

C has a simple set of basic commands, with libraries of code written by others. In fact, C is a basis for many other languages which are popular. Once you learn C it is easy to learn other languages. C and C++ are intimately connected. I will refer to them as C/C++ because when I am writing code the line between the two often gets blurred. You don't have to use all of the object oriented commands of C++. You can use a few of the C++ features while writing C code, and a modern compiler will not sound a warning.

```
for (int i=0; i<40; i++)
```

is an example of a C++ idiom within a line of code. If the C 1999 standard is used by your compiler that line of code will not give a warning. I find the idiom very useful because I know the scope of i is inside that for loop and nowhere else. Limiting the scope of a variable is a good way to prevent unforeseen interactions in your code. Use the fewest global variables you can, they are an opening for errors to enter your life which force you go bug hunting.

Memory allocation is easy in C. Scanning memory or a data file are very similar processes. You step through memory using the size of the data type you're using at the moment. If you have a line with two integers, two doubles, and a string you will be able to easily find each variable in the file or in memory without having to count bit by bit. Cutting and pasting data directly from the 'net and saving it to a text file is simple. Then writing a short app to read the data, and extract just those fields or records you want, is easy work.

Two integers are 16 bits each for a total of 32 bits. The two doubles take up 64 bits. So you can find the beginning of the string by adding 76 to the pointer to the beginning of the line. You can skip past boring bits and just read the fun parts by knowing the format of the file you are reading. You can use awk, sed, and grep to do this work in Linux, or write your own C app to read a raw data file, and parse out exactly what you need for a report. Massaging data files is a regular exercise, but it is so simple you will learn it quickly. Become familiar with scanf, sscanf, fscanf, printf, and fprintf functions. They will assist you, and occasionally bite you. Remember when you are scanning a file, or memory, you are also reading from a specific memory address. So make sure you use addresses for your destinations. For strings, that would be the name of the string array, or a pointer to the string. For integers or doubles you need to preface the variable name with an ampersand, &, to use the address of the variable, instead of the variable itself. They are two very different things, and you will wonder where your data ran off to if you use the wrong one. So remind yourself that scanf and fscanf read into a variable's address. Knowing where you are in memory, or in a file, is important in C.