1) Introduction to the Course

Welcome to the world of computer programming. I will teach you how to write useful applications for your daily work. I will also show you how to create simulations to test your ideas. The computer is a tool which is quite useful for expressing your thoughts. I will present many solved problems with details on how the problem was solved and the tools used. I won't be teaching you the specifics of C or C++, but I will use those languages to solve problems. I will provide many links to sites which do teach the various commands of C or C++. If you wish, you can consult Kernighan & Ritchie's classic book on the subject or dive into Bjarne Stroustrup's book on C++.

We will work from the initial concept to its set of data and from there determine which data structures and algorithms best fit the data. I will describe the problem, give a list of data, provide a few ideas on which data structure to use, and the algorithms which best fit the domain. I then will provide my solution with all of the code and the makefile necessary to create a working application. I will encourage you to run the code and modify it to see how you can change it to fit problems you want to solve. I will give you assignments to make the code more familiar to you. Those assignments are to help you make the code *yours*. Once you have completed an assignment you will understand how the code works and why I chose the route to the solution I did. You will start to see ways to improve the tools for your own purposes.

I will provide many links for you to dig deeper. You can use them to find different ways to solve the problem than the solutions I provide. This is an excellent way to learn. In fact it is how I learned how to program: by examples of working code in a variety of languages. I had to learn how to translate FORTRAN or BASIC into Forth or Pascal or C. Learning one language helps you learn another and then yet another. But the best way to learn programming is to solve problems with the language you are using at the moment. I expect you to learn other languages in the future, but I will have provided you with the tools to help you solve problems.

Problems solved using computer programs start with defining what data you have available to you and what you want to do with that data. My favorite studies are graphic simulations of natural systems. Currently I am working on convection cells in a room. I found the equations on air density as it relates to temperature, along with the radiation and conduction equations to start my thought process. Once you have a parcel of air which changes density according to how hot it is all you need to add is the gravity field. Those few steps are enough to create a convection cell. Yes, I do need to show you how to perform animation but that will be covered along the way. The basics of the solution are those few equations recalculated over time.

During the first section of lessons, we will stick with command-line tools and a command-line result. Don't worry, you can learn a lot this way and it prepares you for using a Graphic User Interface (GUI) like Windows, Mac, Gnome, or KDE. Some of the programs in the graphic section will use only the graphics window while others have a terminal window open as well. I will try to introduce as many of the other parts of a GUI as I can fit in. Menus, icons, mouse clicks, sounds, buttons, timers, dialog boxes, check boxes, radio buttons, etc. You also will see how to multithread code so the user interface (UI) is separate from the problem-solving code. This is how you achieve a quick response to your input while the problem solving continues. It may take a moment for your input to affect the working code but the input from the user is rapidly handled.

The code will become more involved when we solve more complex, and hopefully more interesting, problems. Each lesson will have assignments for you to explore modifying that lesson's code to do something else. You will find you are learning the idioms used to craft code so that, instead of writing simple sentences, you are building paragraphs' worth of a solution. You will find those patterns throughout all the code you examine, just stated a little differently as per the language used for that particular solution. Learning C and C++ is just a way to open the door. There is a large world of code out there to read and modify for your own use.

I like to have a set of tools I have either developed myself or modified so many times they may as well be my own code. You will see how I apply those same tools to solve a variety of problems. I think of them as tool kits for graphics code and animation; tool kits for lists, tables, and trees; and tool kits for report writing or scaling or transforming a variable's value into which color it is on a palette.

Once you have a variety of tools with which you are comfortable, you can start thinking in broader terms. How do I use this tool to solve that problem? Which data structures fit best with which algorithms? Then you can work backward from the output you want, to what form it should be, to what data you need, and then how to gather it. In other cases, you will work from front to back. But any time I start working on a solution to a new problem, I find I am building a tool kit to solve similar problems.

You will learn to think in chunks of code, not just how to write a line. At that point you will be able to sit down with paper and a pencil and rough out your design. When you get to a keyboard, you can write the pseudo code and any real code you may have written while away from the computer. By thinking in larger parts you won't get lost in the line by line detail. Those details get filled in along the way once you have established your path. Or you can test ideas on small parts of code and then put those parts together in a later application. Often, I will rough out the code and leave much of it as comments; having just enough code to compile and run. Then I build each function and test it in the harness I have just crafted. Write code, compile it, and test it all in one session. Then write another function, compile it, and test it. This is called top down design. You have the major chunks of code in your head so you can better think about the logic. Once you have those chunks figured out and typed in as comment lines, you use bottom-up design to work from the basic C language to your function.

You will see "here I need a loop" so you can type

```
for (i=0; i<N; i++)
   {
   }
```

for a loop where i equals 0 to N for each successive iteration. In other cases a

```
do {
   }  while (x);
```

may be more appropriate or

```
while (x)
   {
   }
```

could be the best loop. You can think about each of them and figure out which is more appropriate.

Other times you'll have

```
if (x)
   {
   …
   }
else if (y)
   {
   …
   }
else
   {
   …
   }
```

The other possibility is:

```
switch (Z)
   {
   case x:
       …
      break;

   case y:
      …
      break;

   default:
      …
   }
```

These are only a few types of expressions written in C. However, these forms and others are available in any language you wish to learn. Their syntax will be slightly different of course but the ideas of each form is the same.

What I advocate is to think in large brush strokes and not worry about the detail. In fact, if you write a page of pseudo code it does not have to be aimed at any specific language. When you read academic papers in computer science or computer engineering, you won’t find any specific language is used. The article will contain algorithms written in all pseudo code. It is up to you, the student, to be able to translate from those broad brush strokes of pseudo code into your chosen language. It just so happens these lessons are written in C. That is because I have a lot of experience in it and can translate from pseudo code into C most rapidly. It would be just as easy to use FORTRAN or Forth or Pascal or Perl or Python or … the list is very long and getting longer. It seems there is a new language every six months or so. But trust me, a good grounding in C is not a bad thing. You may want to know Python but it is only a small step from C to Python. What is important is learning those broad strokes. You can paint in oil, watercolors, or in acrylics. The techniques are different but creating the image is similar.

For the first part of the course all of the code will compile and run under either Windows or Linux. Remember if you compile it for Linux it won’t run under Windows. The reverse is also true, a Windows compiled program will not work under Linux. A Windows application file has the telltale .exe on the end of it while a Linux app does not. Later we’ll be working with graphics applications where you will compile under Windows only. The GUI code is very different for the two. You can run these Windows apps under Linux if you use an application called Wine. Wine lets you use graphics code using the Win32 application programmer interface (API); at least everything I have written in the last few years and everything in this course. I wanted to make sure you were not blocked from anything by the choice of operating system (OS) you use. I am fairly agnostic when it comes to operating systems. This is one of the main reasons I chose MinGW for use under Windows. It opens the door to this inter-operability. It also has some handy tools which are simple and quick.

The lessons begin with setting up your computer to use MinGW, pick an editor, and learn your file system. MinGW is a minimal set of tools to allow your Windows computer act as if it is a Unix box. Just a few tools to make life simpler such as **ls** to list your directories, **cd** to change directories, **mkdir** to make a directory, etc. It is not a full blown Unix environment but enough to allow you to use **gcc** and **g++** to compile your code under a Windows environment. The gcc and g++ compilers are very powerful (and free) while the Windows win32 libraries are a great way to write comprehensive windowing applications.

While gcc will work for C code if it is written exactly according to specification (such as no variable declarations inside your code or using /* */ for commenting) g++ lets you use features of the C++ language specification without using any object orientation whatsoever. If you read a makefile and find me using g++ instead of gcc it is probably because of something simple, it does not mean I am truly writing in C++ using objected oriented techniques. There are only a few examples in these series of lessons which are full blown C++ apps. Ecology is the one where extensive use of the language allowed me to do what I wanted to do. The inheritance tree is longer than normal. But I am getting ahead of myself. Suffice it so say the first lesson will get your work environment up and running, the associated app is trivial.

The next few lessons will introduce you to arrays, pointers, linked lists, trees, and graphs. You will learn about various types of sorting routines, a couple search routines, and how strings work. I’ve recently added a few programs from assignments I had in school. They are examples of solved problems using a number of data structures and some interesting algorithms. The last few lessons of the third section show some ideas I had and tested with the computer - a queue for a bank and a quadTree data structure I want to incorporate into orbit and bounce, two graphics programs from the fourth series of lessons.

The second section of lessons cover the Windows OS API. Again we will start with a simple example showing how to draw a variety of things in a graphics window. Then, as we work through the lessons together, I will add new parts of the windowing environment like sound, mouse tracking and clicks, timers, menus, dialog boxes, etc. I start with a simple framework and build from there.

The lessons cover a few simulations of systems, such as a queue in a credit union, Monty Hall's game show, how to determine the cheapest routes through a school district, how to use genetic algorithms to solve a problem, how heat flows in a metal plate, a geodesic sphere moving through three dimensional space with lighting and color, how the rings of Saturn are effected by its moons through orbital mechanics, an ecology of plants, herbivores, and carnivores, and how convection occurs in heated air.

I describe each problem and how it was implemented from initial concept to a working application. Stories cover the problems I had and solved along the way. I have also created a set of appendices on programming, design, problem solving, memory and variables, editors and formatting, plus the interaction between programmers, clients, and the end users of the product you are developing. I offer some history to give you perspective on why I do things the way I do and why I make the design choices I do.

There are a number of paths you can take while working through these lessons. The path you choose will be up to you but experience will allow you to skim certain parts and even skip a few of them. I want to provide as much detail as I can to allow the true novice to follow the lessons. If you are a raw beginner please work your way through the lessons more than once. Begin by studying the lessons and their associated stories. Then, if you need more insight, read the appendices and follow the links. You should work as many assignments as you can with each pass through the lessons. **The more code you write the more you will learn.** I advocate reading as much code as possible. Knowing how other people do things (not just me) is the best way to find a good path to take when you are writing your own code.

A more experienced programmer will skim through the first ten lessons fairly quickly with more time on the later sections. The first two lessons in the graphics section will also go quickly. Then things become more difficult. Hopefully, the stories associated with the lessons will help you understand what I was thinking when I wrote them. I add my own philosophy as I go because I do have feelings about how the client should be treated and how I create code from my ideas. I want you to see programming as fun; as an adventure with a powerful set of tools and your own imagination. I find writing code magnifies my imagination. I think about something then I write code to make that idea come to life. I beat out the bugs then I enjoy when it works.

But, as soon as it works the way I had intended it to, I change it and try something else. A never ending cycle of fun and frustration :) But those frustrations simply mean you are pushing your boundaries. Or maybe even pushing the boundaries of your tools. In most simulations, once you get something working you want to make it work with many more objects, or on a finer and finer grid. Each of those increases memory requirements and CPU usage. I increased the number of objects in my orbital mechanics simulation to the point where it stopped moving smoothly. Then I backed off the count until it performed to my liking. I could have taken another path, saving snapshots of every cycle of the program. Later, I would flip through those saved images to display a movie. It depends on how patient you are and how much detail you want to display. I trade off one for the other until I have found my balance point. With modern graphics cards, and the minuscule price of memory, my simulations grow more complex as the years go by. What I only dreamed about doing in 1978 I can accomplish with my current computers.

What I really want you to glean from these lessons is a sense that you can do this too. There are only three ways to put code together: step by step, branches, and loops. That is it. With those three types of paths you can write anything you like. I want you to see how much fun it is to solve a problem from a vague idea, to pseudo code, to research on the net, choosing the right data structures, using the right mathematical functions, checking your physics, to writing the code, debugging it, and then running your finished app. It is fun. It can be thrilling when it all works. It is challenging. It expands your knowledge base. It can be lucrative. It can get boring. But most of all I find writing code allows me to use the computer as an extension of me - an imagination amplifier. It is a means of doing a lot of drudge work quickly so I can write code which is fun and colorful. I imagine code in broad brush strokes as I'm taking a shower; roughing out some pseudo code ideas. Doing a little research then write the code. It is fun and quite satisfying. I enjoy writing software, I think you will too. I think of programming as exploring problem space; an adventure which could last for the rest of your life.