Problem Solving

You need to create a report for the next meeting. You know the format of the report and the data it requires to fill all of the fields. Your job is to gather all that data, process it, then build the report. At other times you gather a great deal of data then ask yourself how you should write a report of the most relevant data. In the first case, you work backwards from the report to the data. In the second case, you work from your raw data to determine what the output should look like.

When I solve a new problem I write down the steps I would take to solve it. I look for patterns which resemble other problems I have solved. I think about the kit of tools I've crafted over the years, to determine how any of them can help me solve this new problem. Over the course of these lessons, I have given you glimpses of my toolkit and how I used those tools to solve problems. Write your problem description in detail. These notes will help you determine what data to gather for your input, then determine which data structure to use. Once you are sure of the data set, and its data structure representation, you can choose which algorithms to employ to write your solution. Compare your problem to any of those I have presented to you in these lessons. Once you find a good fit contrast your problem with your choice. What will you need to change to make the solution you've chosen, resemble the problem you are working on? Are those data structures and algorithms the best for this problem, or should you find new ones? Once you get a good idea of how you want to proceed, it is best to search the net for examples of how others have solved the problem. I find reading code is the best way to learn new ideas. Implement their code in your own terms to test how it fits your problem's needs.

Once I have find a good solution, I collect all the necessary tools from my toolkit and toss them into one file. I create a makefile and write a main function so I can edit, compile, and run the code as I create each function. I like to have working code as soon as possible, so I can test each function as I write it. I use the main function as my scaffolding, writing test routines right alongside the functions which will be included in the final application. Sometimes, I write a separate application to test ideas, but normally everything is done inside one application.

Working from my notes I write out all of the steps necessary to solve the problem as **pseudo code** comments. As I progress I translate my pseudo code into actual code for each function. My pseudo code is a list of instructions to myself. Sometimes it is as simple as "print data", other times it is more extensive such as:

```
   // a) scootch when balls overlap
```

"Scootch" became this routine in the working code:

```
if (dist < 2*radius)   // for overlapping balls
   {                   // scootch out the vectors
   x1P = x2 + (x1 - x2) * nudge;   // affine transform
   x2P = x1 + (x2 - x1) * nudge;
```

```
y1P = y2 + (y1 - y2) * nudge;
y2P = y1 + (y2 - y1) * nudge;

ptr1->setX(x1P);
ptr2->setX(x2P);
ptr1->setY(y1P);
ptr2->setY(y2P);
}
```

When I started work on **Timer.**exe I began with my design parameters – small memory footprint, very low CPU usage, small screen size, always available, with a simple UI. I wanted a countdown timer for cooking, and for keeping track of my computer use to schedule breaks, for stretching or walking. I chose nine buttons in a keypad arrangement. Each button is labeled with its time interval. Press the button, a timer starts, and the button turns grey. There was no need for any time display, this was enough to do the job. Later I decided I needed a way to turn off any button. I added a few lines of code and captured a right click which resets the timer to its ready state. All nine timers can be used simultaneously, while using very few CPU cycles. It is so convenient I've set it to run on startup. I use it quite often for cooking and baking.

I was able to make a very simple tool by choosing my design goals, and knowing what I wanted the timer to accomplish. It does its job so well because there are no extra modes; there are no hidden menus. It does just what I designed it to do, and is small enough to leave on the screen all the time. One reason it gets used so often, is because it is available at only a flick of the mouse, and a click on the timer of choice. No thinking. No finding. Just do the job. By choosing my goals to fit how I would use a timer, I found I had created something which I actually use; not a toy which gets saved on a drive somewhere, soon to be forgotten. Thinking about what I wanted, checking for apps I could download to test, and keeping my goals simple gave me a plan to start writing pseudo code.

Most of the notes I typed up were about my design goals, since the pseudo code described exactly how I wanted to use and write the code. Then I started a search on Microsoft's site for which button I wanted to use, and the form of the window the keypad would take. I dug in my box of tools for a simple Win32 graphics application where I could rip out the innards and replace them with my timer notes. But, it wasn't as simple as that, because I had few apps which used so many left and right mouse click routines. Nor did I have any code which had more than one button in it. So I got to learn something new, while I was crafting a tool I find useful. Two birds with one stone :) If you have not guessed by now I enjoy learning. I enjoy using a computer to learn about things by writing simulations. So learning while I am building is always fun for me. In this case I expanded my knowledge of timers, mouse click regions, and how to use the sound output routines. Part of the fun of this timer app is finding the best sounds for any given timer alarm. The larger time periods have longer alarm sound sequences. I made sure to keep each distinctive so I could recognize it throughout my house.

Brainstorming – I like to think about a problem for a while before I even start taking notes. Sleep on it, take a shower to think about it, and then finally jot a few notes. Then let them sit for a while. Once your subconscious has had its design meetings, it will tell you to dig out the notes and start typing them up. Take my timer as an example. I used timers I'd located on the net and found them not quite right. The alarms were obtrusive, the UI too complex, or they took too long to boot. But, they gave me a better idea of what I really wanted. All of this brainstorming was done internally. When I had had a chance to type up my design goals, I searched the net for code which did what I wanted. I never found it, so I wrote my own. I found I used my app almost daily. This is a good way of knowing whether you wrote something good: do you use it yourself? Every day? Then it gets tested and improved if necessary. If you plan on brainstorming in a group, then work up to the typing notes stage on your own. Go to the brainstorming meeting with your design roughed out on paper. It may get shot down, but it may trigger someone to create an even better solution.

Divide and conquer – most computer code is designed through a divide and conquer scheme. You know what the overall app should do, you just need to break it into ever smaller parts. Most code is written for the input – process – output cycle. But, how do you input your data? Is it from the user as typed data entry? Or is it from others' typed input, saved in computer files? Is it collected directly from the net with an app? Or is it a mixture of all three? This will determine how you divide up the input portion of your task. The process and output modules will similarly be broken down, into ever smaller parts until you have rather explicit pseudo code. At that point you can start working up from the language of choice, to write the functions for each portion of pseudo code. The process is top-down design followed by bottom-up programming. This is my normal style of design and application creation.

Reduction – transforms old solutions into new applications. When you write code to solve a problem in one area, it is easy to rewrite the code to solve another problem in the same area. Then I examine the tools I reused. How can I generalize them for use elsewhere? Once I reuse them a few times I think of them as tools in my toolbox. I periodically go through my box to generalize the solutions there for easier reuse. When I am busy writing the code, I tend to modify the existing tools as little as possible to make them work. Then, once the problem has been solved, and the program is working, I clean up the interfaces between the tools and the app. This cleaning step allows me to generalize my toolkit. There are a number of ways to solve most problems. By testing a variety of paths I find my code becomes more streamlined. During development, things are in a state of chaos, so refining your code after you get to your goal is valuable. You make your toolkit more robust, and you learn more about each tool. Cleaning up the interfaces makes your code less error prone too. Reusing tested code speeds your way toward the next solution.

Research – if I cannot find the tools I need in my toolkit, or in the code I've written, I go to my textbooks and to the net. Finding out how other people solve a problem gives me insight into how I solve a problem. I try their ideas, and if they fit the way I write code, I adopt them for my own use. Searching the literature helps find the best solutions. Best is defined by quickest, smallest, or some other metric. Maybe you don't need a fancy, complex solution. Maybe your problem does not use

large data sets, but smaller, more limited sets. Searching and sorting can be accomplished by a number of algorithms. Some are better for small data sets, some are best for very large data sets, and then some are best for moderately sized data sets. Does your computer have a lot of memory? Do you have a large disk drive for storage? Is your CPU the fastest one on the block, or do you have a more modest machine? All of these questions help you determine the best design for your application in this environment. You can write the code to check its environment, then choose which algorithms will be the best fit. Makes for a larger program, one which will take longer to write and debug, but it will be the best fit for many more computer environments.

Invisibility - If you are writing code for office use, or for another company, it is best to observe the environment, talk to the people who use the computers, and orient your solution more directly to their needs. Their managers and their supervisors have less working knowledge, and may hand you a design which will not suit the worker's needs, causing you to provide the company less optimum code. Make sure you design the code so it is easy to use. So easy that the worker does not notice they are using your application; they are just getting their job done more quickly, more easily, make fewer mistakes. Work should not be about the tools. Work should be about getting the job done in the best way possible, with the least amount of effort. Your application should be invisible to the user. They get home at the end of the day feeling better. Don't get in your user's way. They will remember how your app treated them and call you names. What you really want is for them to talk to a friend and say, "This new app I use works great. You should try it." The best code gets used, and remembered.

Solve the Right Problem - As I am writing the pseudo code I like to check the original requirements, to see if I am getting off track. Write a module and its test apparatus at one time. Then you can compile the code and test your new work. As you are testing you constantly verify the software, determining whether you are writing the code correctly. But working with the client, and everyone who will be using the code, validates you are writing the right code. Once a few functions are debugged and working to spec, I like to let the supervisor, and one of the worker's, test my raw application. It is best to get something working, into your client's hands as soon as possible. Tell them not all of the functions will work because they haven't been written yet. But do tell them you want their feedback on the user interface, and how any functions you have implemented work for them. Constant feedback from the client is a good way to limit wasted effort. Add a few more functions and repeat. Once you know the UI is working, and you are actually solving the problem the way the user needs it to be solved, you can feel better about your work. If you keep the feedback meetings short, yet timely, your relationship with the client grows stronger and they know exactly where you are in their project. Ask them for about 30 minutes once a week along with one of the users. If you get a new user each week, all the better, because you are training them on the new application plus you are getting naive testers. Watching a new user get up and running on your code is eye opening. You will find where you made mistakes and can improve. They will find errors you never thought to trap. Take good notes, to make your UI better so the next new user you have will need less time to learn the app. This feedback is more valuable than gold. It teaches you how non-programmers think about computers. It will also show you how that company does its work, and how that user thinks. These experiences will teach you how to make your code more invisible to the user. The only errors or

difficulties they will experience will be of their own making; entering the wrong numbers, or forgetting to save the data. You can assist in avoiding the latter but not the former, that is the user's problem. Pay attention to how the user scans your menus or dialog boxes. Watch them, notice how they search for something, and where they look. This will give you a better idea of where things should be located on the screen, and how users think about solving their task. Keep good notes, to make the necessary corrections while they are still fresh in your mind.

1) Find the right problem to solve
    - At the beginning of a programming assignment it is good for all concerned to collect all of the requirements in writing, as well as a schedule of meetings, and a due date for the finished project. Make sure you have talked to all involved, from the managers, to the supervisors, to the people who use your prospective application.
    - Make sure you are solving the problem the end users require solved, not what their managers think the solution may be.
    - Create a schedule and measure the completion of each mile stone. Use the time data for future reference to craft better schedules. Repeat over your entire career.
    - Research other possible solutions. If a better solution is available advise your client of the fact. By saving your client money you will gain his respect and future business.
    - Always add time for training the end users into your initial estimate.
    - Make sure you keep a debugging schedule, as well as a coding schedule. The two will advise you for future projects.
2) Define the problem
    - Once you have your project requirements document you can start defining the problem in programming terms.
    - Examine different ways of solving the defined problem
3) Analyze the problem
    - Who knows how the application will fit into the work flow of the company? Think about the interfaces between workers, managers, and their supervisors.
    - Who cares about how the application fits the needs of the company? Supervisors and users are the first and best sources of this information.
    - Who can implement the solution in the company. Managers and their bosses. Look for the people who control the money.
    - Gain input from each of these three groups and add them to the project requirements document.
4) Develop possibilities
    - When you have your project requirements document you can start breaking the problem down into modules which are easily described.
    - Now is the time to determine your data requirements and how they should be structured.
    - Use the data requirements and the data structures to choose your algorithms. Make notes of big O analysis, CPU requirements, and size of the stored and running application.

5) Select the best solution
    - Does it do the correct job for the end user?
    - Is it an ‘invisible’ application? The gold standard.
    - Is it economically viable? How long will it take to write, test, and train the workers on its use?
    - Do you feel a personal commitment to this project? Is the concept good enough to keep you working late, or on weekends, to refine it?
6) Implement it
    - Make sure your outcome requirements are in place
    - How often will you meet with the managers and a worker for testing, evaluation, and training? Create and keep a schedule. Document times and outcomes of meetings together.
    - Do you have a detailed design document from which to craft code?
    - Remember to limit the number of workers in each group to between three and five. Then add one manager over this group of application developers.
    - Weekly builds are final by closing time Friday night. Encourage your workers to go home at the end of the week for some rest. Working later is not always wiser. A programmer’s mind is always working on the project until it is done. Just because that worker is not sitting at a box banging out code does not mean they are not programming.
    - Keep track of bug reports with time date stamps. Keep good metrics on bug killing. How long from catching the bug to debugging starts. How long does it take to eliminate a bug? These numbers help you make better schedules for future projects. They can also be used to determine if you’re on track in the midst of development.
7) Evaluate and improve solution
    - Are you prepared for bug tracking after your product has been implemented and running within the company? Timeliness in this aspect is the sign of a quality developer.
    - Make sure to get input from the users, as well as the managers above them.
    - Schedule time with the client until they are completely satisfied. Even if it is only a phone call it is a valuable way to keep the project from rough times.
    - Gauge how well you did by how the end users react to it. You are aiming for the ‘invisible’ application. If you get anywhere close to that you have won.
    - Review all the metrics of this project. Use them to improve your schedules, and how you implement your work force in the future.

Decomposition, factoring, bottom up, or top down

When you are done with a project scan your code for repeated phrases. This is a sign you need to factor your code. Those repeated lines of code can easily be written as functions, and placed into libraries for your toolkit routines. You must know they work well for you, since you rewrote those phrases so often. Save some time, and a little typing, by factoring them into functions. Take them from your most tested code, and make sure you name them something descriptive, and memorable. Reuse them knowing they work.

Start looking for bigger modules in your code as you are factoring. Do they mimic how you decomposed the problem into pseudo code? You can find more functions to be honed for your toolkit, by reading your code, and comparing it to what you were trying to do (hopefully written into your comments).

Some days I write small chunks of code to test ideas or learn new algorithms. These may be useful so I try to work them into functions for my toolkit. It is good to review the tools you have stored in your kit when starting a project. Then you can do some bottom-up design to build large parts of your application. You also want to compare your prospective project to all of those you have written before. When you find one similar to the application you want to create, copy its code and makefile, into a new project folder. Then toss in copies of the tools you found on your toolkit search. With the old code as scaffolding, write pseudo code to fill in the missing parts. Clean up the project's code base as you implement your pseudo code with the tools you've collected. When you have modified the original code you copied, and placed it as functions in your new code, the old files can be deleted from the project's folder. If you have crafted any supporting test structures, save those to your toolkit. Test structures are just as important as the tools you save from each project. Remember to keep track of the time involved in building the code from your past successes, and how long it takes to debug each function. Keep good notes because they will help you create a good schedule for future projects. Knowing how long it will take to build something robust is valuable for obtaining the contract from your accurately assessed bid.

Math and Formulas and Vectors and Such

distance equals rate times time

f = m a

PV = n R T

P in atmospheres V in liters

n # of moles R in (liter atmospheres) / (moles K) T in Kelvin degrees

c = 300 * 10^6 meters per second (speed of light in a vacuum)

λ wavelength in meters

Hz frequency in $second^{-1}$ (1/second)

Using dimensional analysis we can relate these three terms correctly:

meters/second divided by meters gives $second^{-1}$

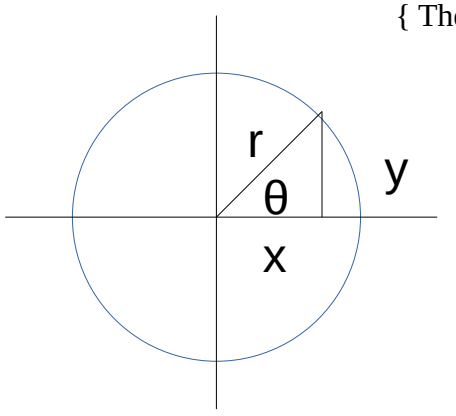
so c / λ = Hz

Dimensional analysis takes the units of each factor of the equation, and determines the correct relationship between them.

All the trigonometry you will ever need:

$\frac{\pi}{180}$ = 0.017453293

{ The classic unit circle }



$\frac{y}{r}$ = sin θ

$\frac{x}{r}$ = cos θ

$\frac{y}{x}$ = tan θ

$\sin^2 \theta + \cos^2 \theta = 1$ Pythagorean formula

$e^{i\theta} = \cos \theta + i \sin \theta$ Euler's equation

Vectors, matrices, dot product, cross product uses

Moving your object around the screen, actively or passively, is easier with vector addition. If you want to rotate an object, use a transformation matrix of the correct angle. Scale an object use a scaling matrix. Shear along any of the dimensions using another matrix. Chain these commands to create a transformation matrix for part, or all of your intended scene.

● Points as vectors

A point is located by its coordinates in two, three, or more dimensions.

It is also the vector from {0, 0, 0} to $\{x_1, y_1, z_1\}$

● Vector addition to translate objects

```
// move geodesic sphere back to where it was before you rotated it
vertex[i].x = x + 100.0;
vertex[i].y = y;
vertex[i].z = z + 100.0;
```

● Adding or subtracting vectors for use as line segments

When I wrote the scaling function I used vectors as points, with two of them defining the line segment AB (or BA). The distance between the two points is B – A (or A – B) .

● Scaling vectors

To scale the line segment I multiply that line segment by my scaling factor. So the signs are correct you need to choose which end to start with. Then you add the scaled line to the starting point.

I used the following equations to scale the temperature in one dimension.

```
T1 = ptr1->getTemp();
T2 = ptr2->getTemp();
ptr2->setTemp(T2 + (T1 - T2) * (1-k));
ptr1->setTemp(T1 + (T2 - T1) * (1-k));
```

or how I used the following equations to ‘push’ two overlapping balls from one another. Simple vector scaling in two dimensions.

```
x1P = x2 + (x1 - x2) * nudge;   // affine transform
x2P = x1 + (x2 - x1) * nudge;
y1P = y2 + (y1 - y2) * nudge;   {equations 1, 2, 3, & 4}
y2P = y1 + (y2 - y1) * nudge;
```

or where I normalized the geodesic coordinates to fit a sphere by vector scaling in three dimensions.

```
for (j=0; j<3; j++)
   coord2[i][j] = coord2[i][j] + (1 - dist/RHO) * coord2[i][j];
```

● Finding a surface normal by using the cross product generated from its vertices.

$c_x = a_y b_z - a_z b_y$

$c_y = a_z b_x - a_x b_z$

$c_z = a_x b_y - a_y b_x$

or $a \times b = |a|\,|b| \sin(\theta)\, n$

```
if (planeequation(i,face[i].vert0,face[i].vert1,face[i].vert2) > 0)

for (i=0; i<3; i++)
   {
   if (i==2)
      j=0;
   else
      j=i+1;
   a += (vert[i].y - vert[j].y) * (vert[i].z + vert[j].z);
   b += (vert[i].z - vert[j].z) * (vert[i].x + vert[j].x);
   c += (vert[i].x - vert[j].x) * (vert[i].y + vert[j].y);
   }
d = -1.0*(a * vert[i].x + b * vert[i].y + c * vert[i].z);
face[k].distance = (a*vx + b*vy + c*vz + d) / sqrt(a*a + b*b + c*c);
```

● Using the dot product of two vectors to determine shading.

$a \cdot b = |a| \times |b| \times \cos(\theta)$

```
// This determines the shading of the face
// normalize to unit vectors to determine angle of incident light
dis = sqrt(lx*lx + ly*ly + lz*lz);
x1 = lx/dis;  // Light unit vector
y1 = ly/dis;
z1 = lz/dis;

dis=sqrt(a*a+b*b+c*c);
a1=a/dis;     // Normal unit vector
b1=b/dis;
c1=c/dis;
color = 2000.0 * (a1*x1 + b1*y1 + c1*z1);
```

or by using the matrix library

```
double shading(struct pt L, struct pt N)
   {
   struct pt Np, Lp;

   // normalize to unit vectors to determine angle of incident light
   Lp = vecDiv( L, sqrt( vecDotVec(L, L) ) );
   Np = vecDiv( N, sqrt( vecDotVec(N, N) ) );

   return vecDotVec(Lp, Np);            // cosine of lighting angle
   }
```

Solve a problem the Polya way.

1. Understand the problem
2. Make a plan
3. Follow the plan
4. Look back at your work and see if you can make it better

Heuristics

1. Analogy – can you find a similar problem and solve that one?
2. Generalization – can you find a problem more general that the one you're attempting?
3. Induction – can you generalize an answer from examples?
4. Variation – can you find a problem like yours which you have solved before?
5. Auxiliary – can you solve a part of your problem?
6. Specialization – can you find a problem more specific than yours?
7. Decompose and recombine – can you decompose the problem and recombine them differently?
8. Working backwards from the answer to the problem?
9. Can you draw a picture of the problem?
10. Can you add elements to your problem to get closer to a solution?

A sketch, or a set of sketches, helps me see things. If the problem has mathematical parts, I write down what types of numbers I'll be using. For instance – in any problem involving money you will be fine using floating point or doubles. All you need are two, or three digits beyond the decimal point. Two for dollars and cents, three (or more) for interest rates, or basis points. If you are crafting a game most likely integers will work just fine. If you are going to use a graphical representation, you will need to translate from problem space numbers, to the integers required for drawing on the computer screen. I have included transforms for converting from problem space to screen space, from screen space to problem space, and color transforms for using various color palettes.

http://www.problem-solving-techniques.com

https://en.wikipedia.org/wiki/Problem_solving

http://www.bizmove.com/skills/m8d.htm

https://the-happy-manager.com/articles/seven-step-problem-solving/

http://ee.hawaii.edu/~tep/EE160/Book/chap3/chapter2.1.html

https://farside.ph.utexas.edu/teaching/celestial/Celestial/node116.html