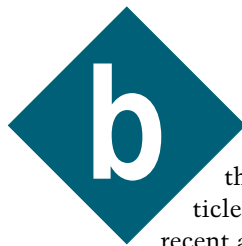


FEATURE ARTICLE

George Martin

Simple Things

This month, George looks at design problems and examines where to look for the answers. He explains that, typically, he calls on a hardware engineer for logic problems and the like, and a software engineer for help with pure software tweaking. Sometimes he's led down a blind alley, but shows us that by tapping into all of the sources available, you could be surprised to find that the answers may come from the most unlikely places.



Before I get into this month's article, let me add to a recent article I did on prime numbers (*Circuit Cellar Online*, September '00). I don't know how many of you have taken my program and gone any further looking for primes. Before I get started on this follow up to the article on prime numbers and the sieve of Eratosthenes, check out www.mersenne.org. The Great Internet Mersenne prime search site reports that, as of June 1999, $2^{6,972,593}-1$ is the largest prime number discovered. You can use your spare CPU clocks to help in the search. I am using my spare CPU time to look for ET. Check out www.setiathome.ssl.berkeley.edu.

NOW FOR THOSE SIMPLE THINGS

Sometimes it's necessary to state the obvious. The simple solution to a design problem is usually the best solution. And generally, my first attempt at solving a design problem is not the simplest (best) solution. This

comment applies equally to both my hardware and software design efforts.

I would like to present an observation that has served me well in the past. Whenever I'm designing embedded software for a customer and I get all twisted up in some logic problem or design approach, I usually seek out their resident senior hardware engineer. And nine times out of 10, that individual clears the confusion and points me in the right direction. My theory is that those seasoned hardware engineers have been trained to deal with hardware designs that are built using standard components. This standard component background and training keeps them focused on using simple components to build large complicated systems, always having a way to document and test that system.

And, whenever I have a pure software problem and ask a software engineer for help, I find that I get one of two replies. First they tell me to put the problems in a big array and sort that array in memory or build a table of pointers to the indexes I'm looking for, sort of like that useless users manual that came with your last software purchase. Or, the answer I get just leads me down a blind design alley that looks dangerous.

Neither type of reply is much help and provides little insight into the design problem. Software engineers are just as brilliant and clever as the hardware types, however, a software background does not include designing with standard components. The standard component approach leads to simpler and more uniform problem solving. Anyway, that's my theory. If you've had any similar or contrasting observations, tell me and I'll publish the interesting ones.

BEING ARDENT

What started this line of thinking and prompted me to write was the

Listing 1—

```

// Code submitted by Dave Tweed
#include <stdio.h>
#include <math.h>

typedef long int INT32;
typedef int BOOL;

typedef struct {INT32 x; INT32 y;} Point;
typedef struct {Point A; Point B; Point C;} Triangle;

double Area (Triangle *t) {
    return fabs (0.5*( ((t->B.x*t->C.y) - (t->B.y*t->C.x))
        - ((t->A.x*t->C.y) - (t->A.y*t->C.x))
        + ((t->A.x*t->B.y) - (t->A.y*t->B.x)) ));
}

BOOL Contains (Triangle *t, Point *P) {
    Triangle ABx, AxC, xBC;

    /* make three new triangles */
    ABx = *t;
    ABx.C = *P;
    AxC = *t;
    AxC.B = *P;
    xBC = *t;
    xBC.A = *P;

    return fabs (Area (t) - (Area (&ABx) + Area (&AxC) + Area
(&xBC))) < 1e-6;
}

int main (void) {
    Point P = {4, 4};
    Triangle T = {{1, 1}, {8, 2}, {5, 9}};

    printf ("Triangle T %s contain point P.\n",
        Contains (&T, &P) ? "does" : "doesn't");
}

```

article "Algorithm Tests for Point Location" by Lawrence Arendt. [1] Arendt presents a simple way to determine if a point lies inside a triangle. I was impressed because I don't usually come across software engineers who would consider this real-world geometry topic.

Most of us have probably written code for this type of testing. I've done it for rectangles and circles. I've said to myself, "This problem is trivial; you just..." and then I realized that my approach was like the second software reply from above. Lots of words, but no real substance and a long dark alley. Arendt's triangle algorithm is a clever solution.

In his article, Arendt explains that, if you have a triangle ABC that has area A, then point X lies inside triangle ABC if the sum of the three triangles formed

by point X and the three vertices equals the area A. So, if $\text{Area}(\text{ABX}) + \text{Area}(\text{AXC}) + \text{Area}(\text{XBC}) = \text{Area}(\text{ABC})$, then point X is inside triangle ABC. Draw yourself a picture and it should become clear.

Arendt used C++ to design and code his algorithm. I read that and was going to tell you how it would be much simpler to design and code in plain old C. So, I started to design and code a simpler version. To my surprise, I could do the design and coding in C, but it was not simpler.

The truth is that, in my original draft of this article, I went on to explain why C++ provided a better solution. I submitted it and then got an e-mail from *Circuit Cellar* Project Editor Dave Tweed (whose day job, ironically enough, is hardware engineering), who had proofread my manu-

script. He showed me a simple C routine that was equivalent to the original C++ routine. Needless to say, I was flabbergasted that someone could come up with a solution that I couldn't find.

Think again about seeking out the hardware experts for a good solution. I need to modify my observations and tell you that you may be surprised where you can get a clearer way of looking at your design problem. Be open to input from all sources, but be careful to weed out the signal from the noise.

AN EXAMPLE

Let's assume that the *Area()* routine is similar to Arendt's. Therefore, in C, the algorithm would look something like Listing 1.

The comparison to $1e-6$ in the example is shown because floating-point calculations can have round off problems. For example, $(1.0/3.0) \times 3.0$ might not equal 1.0, but 0.999999999999 instead. So, the typical approach is to check that the results agree to within some small error value (often called epsilon).

As you can see, it can be done with just about the same level of clarity or complication as C++, but there might be other considerations to take into account.

Does the C++ compiler blow up the code and make it unusable in embedded systems? Perhaps. Does C++ provide for simpler thinking for software designers? Perhaps (old dogs and new tricks). Should we start using C++ tomorrow? No, but it can no longer be ignored (don't become obsolete).

All right, I'm done. I'll get off the soapbox now. Next month, I'll talk about the Motorola Coldfire CPU. ☐

George started his career in the aerospace industry in 1969. After five years at a real job, he set out on his own and cofounded a design and manufacturing firm. Typical systems that George designs include servo-motion control, graphical input and output, data acquisition, and remote control. George is a charter member of the Ciarcia Design Works Team and most recently, he's been working on the people-tracking system for Bill

Gates' new house. You can reach him at george.martin@worldnet.att.net.

REFERENCE

[1] L. Arendt, "Algorithm Tests for Point Location," *EDN Magazine*, Winnipeg, Canada, August 2000, www.ednmag.com.

Circuit Cellar, the Magazine for Computer Applications. Reprinted by permission. For subscription information, call (860) 875-2199, subscribe@circuitcellar.com or www.circuitcellar.com/subscribe.htm.