The Computer and I

My apologies to Betty McDonald, but this is the best title for this article. My experience with these nasty machines goes all the way back to 1957. That is more than 50 years ago. You would think I would have had enough of them by now, but no. I'm still into them up to my neck. Some people never learn.

Never say "never"

My first exposure to the world of computing was a class in digital computers that I took in 1957 from a professor whose name was Bill Varner. I was fascinated by the subject. This was one class in which I never had a day-dream. It was too interesting for that. One of our assignments for that class was to write a short program to be run on the IBM 650 at the National Bureau of Standards, which was in the same city as the university—Boulder, Colorado. More about that computer later.

One day in class I asked Professor Varner a question. "If transistors are so reliable," I asked, "then why don't they use them in digital computers?" His answer floored me. "They're too unstable," the good professor replied. "They will never be used in digital computers." It would not be very long before I found out how wrong he was.

The primitive IBM 650

After I graduated from the University of Colorado—which was immediately after the end of the course in digital computers mentioned above—I took a job in the Radio Meteorology Section of the National Bureau of Standards. My job title was "Section Computer Programmer." As such, I began writing programs for the IBM 650 mainframe. This was a primitive machine compared with the computers we have nowadays. I don't recall what the speed of the CPU was, but it wasn't very fast. The whole machine was built out of vacuum tubes, nary a transistor in the mix. So far, Varner was doing all right in his prediction.

The 650 stored its programs and data on a magnetic drum, which spun at a fairly high rate of speed. One of the problems in programming it was in placing the instructions, which were all in machine language—actually in assembly language, which is just using alphabetic symbols to represent binary machine codes. You would put each instruction at an assigned address on the magnetic drum. The problem was in the timing. You had to know how long it took for each instruction to execute, and then place the following instruction at a place on the drum where the read head could pick it up right after executing the previous instruction. Needless to say, this was a difficult and time-consuming task for a programmer who was more interested in getting results than playing around with the timing of a rotating magnetic drum. We used an IBM assembly programming language called SOAP I. "SOAP" stood for "symbolic optimizing assembly program." Except it didn't do much optimizing.

All of us programmers were relieved when IBM came out with a new assembler called SOAP II. SOAP II took over the task of placing the instructions in the optimum places on the magnetic drum so the programmer no longer had to do this tedious job himself. There was one weird guy named Herbert Howe who insisted in doing all of his programming in binary and never used an assembler in his life. He looked like somebody you wouldn't trust to empty a wastepaper basket by himself, but he was actually a rather intelligent man—even though some of his choices in tasks like programming seemed weird to the rest of us.

I should also mention the program and data input for the 650. It was all done with punched cards that were fed into the machine automatically. These cards used Hollerith codes, which were designed to be self-correcting in case of a misread by the card reader. Sadly, they did not compensate for errors on the part of the coder. Bugs have always been the bane of the computer programmer.

Output from this odd machine was either to more punched cards, or to a huge pin-feed printer, which used that wide, green-barred paper that was once so ubiquitous. This printer was the size of most desks. The CPU unit was about seven feet tall, three feet wide, and perhaps twelve feet deep. It required an air conditioner of several tons capacity to keep it from overheating.

I still have some printed output from that primitive IBM 650 mainframe. It is a series of pages from a ray-tracing program that I wrote, which traced rays at different elevations angles from the surface to the top of the troposphere, using as data a set of punched cards giving the radio refractive index of the atmosphere at various elevations above the ground (these could be arbitrary profiles and were not limited to data at specified altitudes above the ground). This program was the workhorse of our section for several years, and went through numerous incarnations in different programming languages, as will become evident.

Never comes a bit early

Sometime in 1960 NBS Boulder took delivery of a Control Data Corporation (CDC) 1604 computer, which was a 48-bit transistorized computer, based on a redesign of an earlier machine called the ERA 1103. This machine was designed by Seymour Cray, who became famous for developing the fastest supercomputers in the world after he formed his own company, Cray Computers. Notice that it was only about three years from the time Professor Varner pronounced transistors as unusable in digital computers until the time when the first fully-transistorized computer was on the market. Actually, the 1604 was released in 1959, but the first delivery was to the U.S. Navy in 1960.

Although the main computing language for the CDC 1604 was JOVIAL (Jules own version of the international algorithmic language [ALGOL]), ours used a high-level language called FORTRAN (an acronym for "formula translator"). This was a fairly good computer programming language, but it was not as good as the C programming language that was developed by the Bell Labs several years later. The big problem with the FORTRAN on the 1604 was that Seymour Cray wrote the compiler himself; he was something of a control freak. He was much better at designing computers than he was at writing compilers. The result was that the FORTRAN compiler was buggy and didn't really work well for the first couple of years after we got the 1604.

At least in the early years, the 1604 used punched card input just like the old 650, but it had magnetic tape drives, and we generated a lot of tapes with program output data. As an amusing aside, the number 1604 was rumored to have been the sum of the street address of Control Data Corporation (501 Park Avenue) and the number of its predecessor, the 1103. This was not true. The number actually stood for the design goal memory, 16K (16,384) 48-bit numbers, and the number of tape drives, 4. But when the 1604 was actually produced, it had 32K of 48-bit magnetic core memory. It was widely considered the first of the supercomputers.

Timeshare computers arrive on the scene

About 1966 the NBS bought one of the first Scientific Data Systems (SDS) 940 computers, which was the first computer designed for time-sharing. Input to this system was done by means of teletype keyboards, which were distributed throughout several buildings occupied by both NBS and the ESSA (Environmental Sciences Services Administration) Research Labs, into which our section had been put circa 1964-65. Programming for the 940 was done in BASIC (beginner's all-purpose symbolic instruction code). This was even worse than FORTRAN, but we had to use the tools that were available to us. When SDS was acquired by Xerox in 1969, the computer was renamed the XDS 940. It was still the same old beast. I never really liked the 940 timeshare machine.

The dawn of the microcomputer age

Meanwhile, the NBS computer center had acquired a CDC 6400, a downgraded version of the CDC 6600, which was considered the first true supercomputer. So many of the programmers in both NBS and ESSA complained about this machine that they eventually upgraded it to a true 6600. I ran some ray tracings on the 6600, and it was more than 20 times faster than the 1604. But I wasn't doing much programming at that time. I was doing more research work than programming. Yet I never lost touch with the programming side of things.

In 1977 I was forced to retire on disability because of Reiter's Syndrome, which had given me a severe allergy to cigarette smoke. In those days, they could not guarantee me a smoke-free environment, so I retired on disability and moved to Las Cruces, New Mexico. In 1980 I built a single-board computer from a kit. Then in early 1981 I answered an ad in the paper from the Septor Corporation in El Paso, Texas, which is about 45 miles south of Las Cruces. Septor turned out to be the brainchild of a man named Roger Lovrenich (pronounced "love-**wren**-itch"). They were working on a computer system to automate the machines that worked on automobile assembly lines. They had me doing some programming for them, and they supplied me with a Heath-Zenith Z89 computer. This was my first microcomputer. It had the enormous memory capacity of 48 K (back then most micros had only 16 K of RAM) and used two floppy disk drives, holding 360 K of data each, for long-term storage. The Z89 was equipped with the Microsoft BASIC compiler called "BASCOM" and ran on the CP/M disk operating system (DOS), a product of Digital Research Corporation. "CP/M" stood for "control program for microcomputers."

Gary Kildall, the head of Digital Research, developed CP/M virtually single-handedly. It ran on Intel 8080 or 8088 processors and on Zilog Z80 CPUs, which were used in the Heath-Zenith machines; the Z80 was code-compatible with the 8088. The Z80 in my Septor machine ran at the amazing speed of two megahertz. It took 12.7 seconds for the Z89 to run 10 iterations of the sieve of Erastosthenes benchmark program. With a version compiled in C, it took the Z89 37 seconds to run the benchmark. In those days I could time it with a stopwatch.

In 1980 IBM looked to Digital Research for an operating system for their upcoming personal computer. But Gary Kildall's arrogance combined with his refusal to sign a non-disclosure agreement but the kibosh on the deal. IBM turned to an upstart Harvard drop-out named Bill Gates, who had bought a rudimentary operating system from a little company called Seattle Computer Products. He renamed it as MS-DOS—IBM called it PC DOS—and the rest is history.

My computer work expands

After my work with Septor was finished, I bought myself a Sanyo 550 computer. It had a 2 megahertz 8088 CPU and two 700K floppy disk drives. It was like moving up from a Chevrolet to a Cadillac. With an upgrade to a 7.16-MHz V20 CPU, an improved version of the 8088, the Sanyo ran the assembly-code benchmark in 2.6 seconds. Later I moved up to a Kaypro computer with a 10-MHz 80286 CPU. It ran the assembly benchmark in 0.886 seconds. I was now using a subroutine that timed things from the real time clock (RTC) in the computer itself.

In the late 1980s I was doing a lot of intensive programming in the C language. I had a skeleton database management system that I could—and did—easily adapt to almost any operation needed. I did real-time programming for Data Check International, using the QNX operating system (from an outfit in Canada), a real-time, multi-user operating system that was so compact it fit on one floppy disk (1.2 MB "stiffy" disks). I programmed a database system for MLS real estate systems, and another for automobile dealers who didn't want to shell out $25K or so for commercial F&I software.

In 1991 I worked for the Ramada Inn Corporation for about six months on a multi-user system called Roomfinder III, which ran on QNX. From what I heard later, it was a success. Using it, people working at one of their properties (read: motels—a dirty word in that business, it seems) could make reservations through the Ramada mainframe computer located in Phoenix, Arizona—which was the place where I worked on this software. I rented an apartment for that gig.

The advent of Microsoft Windows® and escalating CPU speeds

But all of this fun came to a grinding halt when Windows with its GUI (graphical user interface) began to take over the computer industry. Programming in a GUI system is incredibly complex, and the software necessary to do it is expensive, so I began doing more in graphical processing—such as Photoshop—and genealogy. I gradually got out of the software business, and changed the name of my business from RapidSoft to RapidSoft Press. I was now in the desktop publishing business (DTP). In 2012 I self-published a family history in two volumes totaling about 1,400 pages.

Meanwhile, the computers kept on escalating in speed and power. A 33-MHz 80386 machine ran my sieve benchmark in 0.189 seconds. A 67-MHz 80486 ran it in 0.039 seconds. A 133-MHz Pentium machine ran the same thing in 0.00854 seconds. By then I was running the sieve program for 100 iterations and dividing the result by ten to get a more accurate reading; the system clock on IBM clones ticks once every 838 nanoseconds. A 350-MHz Pentium machine dropped the reading to 0.0036 seconds. An 800-MHz Pentium III machine dropped the benchmark timing to 0.00158 seconds. Then a 3-GHz Pentium IV machine cut the number to 0.0007 seconds in compiled C; the assembly version was now slower at 0.00096 seconds. I began to call it "the incredible shrinking benchmark." In 2007, I ran this benchmark compiled in Microsoft C 7.0 on my 2.4-GHz Core 2 Duo, which cut the timing to a mere 0.00044 seconds (440 microseconds).

I now have a machine with an Intel i7 CPU running at 3.8 GHz, but it runs on 64-bit Windows 7. I can't run any 16-bit programs on it at all. So I downloaded the free Microsoft Visual Studio Express 2012. After finally getting a revised version of the sieve benchmark compiled in Visual Studio C, I got a best time of 150 microseconds. I'm unsure how well optimized this code is (Visual Studio workings are a bit murky), but in any event this time is considerably faster than the Core 2 Duo machine.

Summing all of this up, the Intel i7 machine running Windows 7 ran the C-compiled benchmark 246,700 times faster than the Z80 machine did in 1982. That's an increase in speed of 56% per year for the 28 years from 1982 to 2010, a factor of 1.9 for every 18 months. This agrees closely with Moore's law, which states that the speed of microprocessors will double every 18 months.

My i7 computer has 16 GB of high-speed RAM and two 10,000-rpm hard drives running in RAID 1 mode (the second disk is a mirror image of the first one). When I got my first hard disk in 1985, it was a Seagate ST-225 that held 20 MB of files. I sat there looking at the screen and wondering how I would ever fill up 20 megabytes of storage space. My latest computer has 500-GB hard disks, which now hold about 175 GB of files. Each disk can hold files totaling 25,600 times as much data as that ST-225. Have personal computers changed in the last 30 years or what?

Why I think programmers are dirt lazy

If you've ever bought anything on the internet I'm sure you have run into that annoying business about credit card numbers. They want digits only, no dashes or spaces. There is a good reason that the credit card companies separate the numbers into groups of four digits. They are much easier to read that way, so if you're typing them in you can easily see if you made a mistake. Not so if you leave out the spaces or dashes.

So why do websites do this to you? Is it difficult to parse out the number from a string of characters that contains spaces or dashes? No, it's as easy as falling off a log. The programmers are just plain lazy. Disgustingly lazy. It's easier for them if they get a number that is all digits. The joker in the deck is that it's just about as hard to figure out if a number string is all digits as it is to make it that way.

Now, I haven't done any programming in about twenty years. Also, I've never written any code in Javascript, which is the programming language used on websites. Yet I was able to come up with this little Javascript function that parses credit card numbers input by the user and generates a number that is all digits. The variable "ccStr" is a string—a series of ASCII characters—containing the credit card number input by the user. The prompt function would likely not be the way that the number is actually read on a website, but it will do for a demonstration function.

```
ccStr = ( prompt ("Credit Card Number: ", "...") );
ccNum = "";                        /* initialize ccNum to a null (empty) string */
for ( i = 0; i < ccStr.length; i++ )  /* do this once for every char in ccStr      */
  { ccChr = ccStr[ i ];            /* ccChr is the ith character in ccStr ...    */
     if ( isDigit( ccChr ) )       /* … if it's a numerical digit …              */
       ccNum += ccChr;             /* ... append it to the string ccNum          */
  };
function isDigit ( ccChr )         /* this function will not execute unless called */
  {
    return ( ( 0 <= ccChr ) && ( ccChr <= 9 ) );   /* return true if ccChr is a digit  */
  }
print (ccNum);                     /* display the resulting credit card number */
```

The string "ccNum" will hold the parsed, all-digits, credit card number; it's initialized to a null string. The "for" loop runs through values of "i" (a number) from 0 through the length of the input string (length is a built-in property of strings in Javascript). If i is less than the length of the input string, the for loop increments the value of i and continues. In this case, it assigns a one-character string named "ccChr" (credit card Character) the value of the iTH character in the input string. It calls the isDigit function—defined below the main function—giving it ccChr as input. If ccChr is between 0 and 9 (the && is a Boolean AND operator; and <= means "equal to or greater than") isDigit returns true (number 1); if not, it returns false (0). If the return value is true, the statement ccNum += ccChr is executed, which adds the new character, ccChr, to the end of the ccNum string. The print statement at the bottom displays the parsed credit card number to see if the algorithm works. It does work. How hard is this? Any web spinner worth his salt could write this code in his sleep. So why don't they?

A short lesson in good programming practices

If you are an astute reader, you are probably wondering why I bothered to declare a separate function called "isDigit" when I could just as easily have embedded its code in the main function, like this:

```
ccStr = ( prompt ("Credit Card Number: ", "...") );
ccNum = "";
for ( i = 0; i < ccStr.length; i++ )
 { ccChr = ccStr[ i ];
   if ( ( 0 <= ccChr ) && ( ccChr <= 9) )
     ccNum += ccChr;
 };
print (ccNum);
```

Surely this is better; it's so much more compact. Yes, that is true. But it ignores one of the basic rules of good programming. By declaring "isDigit" as a stand-alone function, I can now use it anywhere in the Javascript code for my website. The alternative is to write out the code for isDigit every time it's needed. For example, if I want to parse the numbers of a user-input telephone number like 123-456-7890, I can use the isDigit function to see if the characters in the user-input string are digits or not. Just as I did with the credit card number, I can easily end up with an all-digit phone number that I can use to dial out directly (after all, when you call someone you don't put in the dashes in the telephone number, you just dial the digits—or these days, the phone does it for you).

This is one of those little programming secrets that one learns either from books or by trial and error—in fact, writing a computer program that works can be defined as an exercise in trial and error. Before long, things like this become second nature to a programmer. You simply don't think anything of it; you just do it. Most people think computer programming is a dark art, but it isn't. It's simply a matter of learning to think logically while keeping in mind what a computer can do and what it can't do (mainly, what it can't do is think for itself—something programmers are wont to forget every once in a while, to their ultimate chagrin).

Computer programs were a mystery to me, too, until I finally realized that they are nothing but a list of instructions that tell the computer what to do. Like a cook following a recipe, the CPU simply executes whatever command the program tells it to do, whether it makes sense or not. If the program tells it to do something dumb, the machine usually crashes. End of story.

My farewell to the Ramada Inns programming team

This is a document that I left on the hard disk of the computer I used while working on the Room-Finder III software that I mentioned earlier. Most of these comments are fathomable only by other computer programmers, any of whom will immediately understand what I mean. To those of you who are not computer programmers, I apologize for boring you with mystical rantings.

To Whom it May or May Not Concern:

For those who may be so lucky (!) as to inherit these files and their maintenance, herewith is a general summary of the coding practices I used in developing the report programs.

I, too, inherited most of these programs--specifically the first 21 in the list enumerated in rpt_proto.h (which, despite its name, is **not** a prototype file), whose origins are shrouded in antiquity, although they were heavily massaged by Richard Theobald, Esq.; and the discrepancy report (# 24), which was originally written by Jerry Grady. I have tried to bring these programs into compliance with the standards set forth in this document, but I may have missed a few tricks here and there.
===

Standards I have attempted to maintain in this code:

(1) Function declarations and definitions. I always separate the type of a function from its name by at least two blanks (rather than putting the name on a separate line, like Jerry does). I separate the parameter list with blanks, including the end spaces next to the parentheses. Parameter types are separated from parameter names by at least two blanks. A function declaration and its definition look like this:

```
int  func_name( int  parm1, char  parm2, struct parm_list *  parm3 );

int  func_name( int  parm1, char  parm2, struct parm_list *  parm3 )
{
        /* function body */
}
```

I do **not** separate the name of a function from its parentheses, as several of the other programmers hereabouts do. I have never in my life seen this done anywhere else (including all the many books on the C language that I have used) and it is a mystery to me where the devil they got this idea from. I do, however, separate the token "return" from anything that is to be returned from a function, since "return" is not a function; exit(), however, is a function, and its declaration can be found in stdlib.h, on line # 82 in the cii stdlib.h file [*"cii" was the name of the C compiler we used*]:

```
        return ( whatever_is_being_returned );
```

I almost always enclose anything being returned by "return" in parentheses, although they are certainly not required, since it makes clear the idea that it is the result of any expression (so enclosed) that is returned by the return statement.

(2) Variable declarations. I separate the type of a variable from its name by at least two blanks, just like function declarations:

```
int         count;
char        ch;
char *      pointer;

struct Input  input_fields[] = ...
```

You will notice that I also try to line up the names vertically in a list. It is much easier to read such a list than it is one where everything is staggered because the types are of different length. When a type name is very long (such as the "struct Input" above), I try to leave a blank line between declarations so that the stagger is not so visually obnoxious.

(3) Pointers. You may have noticed—if you are not half asleep by now—that I write pointer declarations in a rather odd way. For example,

```
        char *  pointer;
```

In my humble opinion, the worst single mistake made in designing the C language was to make the asterisk bind to the variable in declarations (as opposed to its binding in statements, where it obviously must bind to the variable). You should be able to write:

```
        char *  ptr1,  ptr2,  ptr3;
```

and have them all declared as pointers to type character. Unfortunately, this does not work: ptr2 and ptr3 from the above will be defined as simple character variables so far as the compiler is concerned.

That this is essentially wrong is shown by such constructions as

```
        int  func_name( char *, char *, ... );
```

which are routinely used in prototypes. If the asterisk should not really bind to the type name, what the devil is it doing in the prototype? A pointer to a particular variable type is an entirely different sort of beast from variables of that type; it is, in fact, a new type in and of itself. This deficiency was recognized by Bjarne Stroustrup when he designed the C++ language, which has a new *declaration* token, the ampersand ("&"), that defines a variable as a reference to (i.e., a pointer to) the formal type. Thus:

```
        char&  ptr1, ptr2, ptr3;
```

will define all three variables as pointers to type char. (An additional advantage of the C++ reference type is that it dereferences automatically, so that "ch = ptr1;" assigns what ptr1 is pointing at to the variable ch. This does, however, make the meaning of a statement such as "ch = ptr1++;" a bit murky.)

I write multiple pointer declarations as

```
char *  ptr1, * ptr2, * ptr3;
```

as a sort of compromise.

The only time I write a pointer as "*ptr" is when it is being explicitly dereferenced, as in

```
        ch = *ptr;
```

End of sermon.

(4) White space in general. I make liberal use of white space (especially blank lines), as I believe it makes code much more readable. I **always** leave a blank between tokens such as for, if, while, etc., and the parentheses enclosing the associated conditional expression(s); also between the sections of a for-statement. Constructions like

```
        for(i=1;i>=y;i++)g[i]=y+i+2;
```

are an abomination in my eyes. I would write the above as

```
        for ( i = 1; i >= y; ++i )
          g[ i ] = y + i + 2;
```

Which would you rather try to decipher when you're trying to understand someone else's code in a hurry?

(5) Lining up brackets and parentheses. Like many C programmers I prefer to have brackets line up vertically, both in declarations and in executable code.

```
struct parm_list
{
 int     parm1;
 char    parm2;
 char *  parm3,
      *  parm4;
};
...
 if ( x == 22 )
  {
   a = b;
   y = x;
  }
 else
  {
   y = x - 1;
  }
```

However, I carry this philosophy a step further than most C programmers. I also like parentheses to line up vertically when the code they enclose runs more than one line. This can make complicated conditionals much easier to scan quickly. For example, rather than

```
        if ( grunge_factor < grunge_limit && ( widgets > 1 ||
           gargoyles > widget_limit ) )
```

I would write

```
        if (     ( grunge_factor < grunge_limit            )
           && ( widgets > 1 || gargoyles > widget_limit )
         )
```

This makes the apposition of conditional sub-expressions easier to discern.

Also, in function calls where there are too many variables to fit on one line, I write:

```
 prt_printf( -1, 0,
                   " %s %s %c %d %s",
                   string_pointer1,
                   string_pointer2,
                   char_var,
                   int_var,
                   string_pointer3
           );
```

Lining up the opening and closing parentheses of the function call makes it more obvious where the end is. Most programmers would leave the closing parenthesis and semicolon "dangling" immediately following string_pointer3.

(6) Increment and decrement operators. It seems to be nearly universal practice with C programmers to write the unary operators ++ and -- in postfix notation unless the context requires a prefix usage. I guess C programmers must be addicted to RPN (Reverse Polish Notation), probably as a result of brain damage inflicted by overuse of Hewlett-Packard hand calculators. But when speaking, you do not say "x increment" you say "increment x." So why write it as x++ when you really mean ++x? Is it just laziness or what?

I never write x++ unless the context requires it; otherwise I write ++x. If you see x++ or x-- in my code, you may be assured that the context required a postfix notation, as in

```
        *ptr++;
```

where the intention is to dereference ptr and **then** increment it (i.e., to point to the next char in a string).

In for-statements and simple increment/decrement operations, I always write in prefix notation.

(7) Overlength lines. I believe it is cruel and unusual punishment to inflict lines over 80 characters long on programmers who are working with an 80-column video display. In a few cases this may be unavoidable (such as the godawful menu structures foisted upon those working with RoomFinder III). In general, wherever code lines would run past the end of the screen, I split them up (in whatever way works) so that one can see them without having to scroll the display horizontally.

I went so far as to split up the horribly long lines required in the struct Input definitions in grp_show2.h (in rf3_21/include), for both the grp_search_input[] and grp_acct_input[] arrays. I duplicated the title lines for each variable in the arrays so one can tell what each part of the struct does. This was a lot of work just to keep the file within 80-column limits. In retrospect, I'm not sure it was worth it.

(8) Function and variable names. I personally favor so-called "Hungarian notation" for names, rather than the venerable underbar technique. In Hungarian notation, a function call written as

```
        func_result = funky_func( passed_variable );
```

in underbar notation would look like this:

```
funcResult = FunkyFunc( passedVariable );
```

As you can see, in Hungarian notation uppercase letters substitute for the underbars formerly used. In general, function names have the first letter capitalized as well, but variable names begin with a lowercase letter. This notation is more compact than underbar notation (the underbars occupy a character space but contribute no information of their own). Hungarian notation has been a corporate standard at Microsoft for several years now, as anyone knows who has examined the APIs for Windows or OS/2. It is also standard practice in the Pascal, Modula-2, and Smalltalk languages.

It was, however, evident from inspection of existing source code files that the underbar notation was preferred here, so I followed my number 3 rule: When in Rome, do as the Romans. (My first two rules, listed in order of importance, with #1 being the most important, are: (1) When someone is paying you for a job, anything less than the very best you are capable of is not good enough; (2) Always round off your time in favor of the client.)

(9) Globals. I'm sure it's no secret that I believe the rf3 software has an over-reliance on global variables. Richard Theobald has observed that "David doesn't like globals." That's not really true. Globals have their place. I use them in my own code. I do believe that they have been over-used in the rf3 software. I have never seen software with so many functions declared as:

```
void    func_name( void );
```

as there are in the rf3 system.

Several times, in trying to make needed or desirable modifications to the report programs, I have been frustrated by the need to recode functions so that variables can be passed instead of referred to globally. The use of globals is fine if you are sure the function will **never** need to work on anything but those globals. If you ever decide that a particular function would be just the thing for some job at hand, but the variable you want it to work on is not the global that it was originally designed to operate on, you are in for some recoding, my friend. I ended up in the rather absurd position of passing a function a global at times, so that at other times I could pass a different variable for it to work with.

One specific example: In the reports there is a global named rpt_width, which determines whether a report can print in pica (10 cpi) or must be compressed (17+ cpi) in order to fit on the paper. When I was going over the reports, I noticed a statement

```
rpt_width = 80;
```

that seemed in the wrong place. I moved it so it was with other statements of a like nature. As a result, all of the reports printed out compressed, even those with the rpt_width set to 80. This was because the rpt_init() function looks at the value of that global in order to decide how to set up the printer, and I had moved the statement setting the value of 80 beyond the call to rpt_init() [N.B: In so doing, I violated my own number 4 rule: Never make gratuitous changes to program code. Tut tut!]. This was easy enough to fix, but if the report width had been passed to rpt_init() [which, in my opinion, it should have been], as in

```
rpt_init( 80 );   or   rpt_init( rpt_width = 80 );
```

this problem would never have occurred.

Oh well, I guess that's the sort of thing that keeps us busy.

(10) Superfluous casts. I have removed a large number of unnecessary casts from the report programs. Most of these were of this type (this is a real example):

```
strcat( str, (char *) mem->pcp.hotel_city );
```

evidently there was, at some point, someone working on these programs who was not aware that the type of a value obtained with a pointer to a structure member is the same as the type of the structure member. The member hotel_city is a character array, so its name is automatically of type "pointer to char" and the cast is entirely superfluous. Unfortunately, the compiler, even at warning level 3, does not flag meaningless type casts, so whoever did this never noticed that the casts were useless. I fixed as many of these as I noticed (i.e., I may have missed a few).

===

As my parting gift to the PTU programming group, I give you what I have modestly called "Thayer's Law." It goes like this.

Thayer's Law: For every observed glitch in program behavior there are at least two bugs, either of which is sufficient to have caused the glitch.

There are two corollaries:

(1) You will only find one of these bugs at a time.
(2) The number of bugs per glitch is directly proportional to
the length of time it takes to recompile your program.

Corollary (1) derives from the optimistic nature of most programmers. When you find the first bug, you say "That's it!", fix it, and immediately recompile your program, only to discover that it still does the same damned thing. Back to the drawing board (or SID or CodeView, as the case may be)!

Corollary (2) derives from a union of Thayer's Law and Murphy's Law: the worse a problem is apt to be the more likely it is to occur.

You may think that Thayer's Law is some sort of joke. I thought so, too, when I first defined it. Trouble is, it's happened to me so many times since then that I no longer think it is a joke. I keep bug logs for my software, and in going over some of them I am amazed at the number of times an entry begins with a comment like "Thayer's Law strikes again! …".

I'm beginning to think I may have stumbled onto a basic law of nature....

—G. David Thayer
21 June 1991

Notes: Richard Theobald was one of the project programmers—a very nice British fellow—and Jerry Grady was the lead programmer for the RF3 group, also called the PTU (and I don't remember what "PTU" stood for anymore).