programming pearls

By Jon Bentley

WRITING CORRECT PROGRAMS

In the late 1960s people were talking about the promise of programs that verify the correctness of other programs. Unfortunately, it is now the middle of the 1980s, and, with precious few exceptions, there is still little more than talk about automated verification systems. Despite unrealized expectations, however, the research on program verification has given us something far more valuable than a black box that gobbles programs and flashes "good" or "bad"—we now have a fundamental understanding of computer programming.

The purpose of this column is to show how that fundamental understanding can help programmers write correct programs. But before we get to the subject itself, we must keep it in perspective. Coding skill is just one small part of writing correct programs. The majority of the task is the subject of the three previous columns: problem definition, algorithm design, and data structure selection. If you perform those tasks well, then writing correct code is usually easy.

The Challenge of Binary Search

Even with the best of designs, every now and then a programmer has to write subtle code. This column is about one problem that requires particularly careful code: binary search. After defining the problem and sketching an algorithm to solve it, we'll use principles of program verification in several stages as we develop the program.

The problem is to determine whether the sorted array $X[1\,..\,N]$ contains the element T. Precisely, we know that $N \geq 0$ and that $X[1] \leq X[2] \leq \cdots \leq X[N]$. The types of T and the elements of X are the same; the pseudocode should work equally well for integers, reals or strings. The answer is stored in the integer P (for position); when P is zero T is not in $X[1\,..\,N]$, otherwise $1 \leq P \leq N$ and $T = X[P]$.

Binary search solves the problem by keeping track of a range within the array in which T must be if it is anywhere in the array. Initially, the range is the entire array. The range is diminished by comparing its middle element to T and discarding half the range. This process continues until T is discovered in the array or until the range in which it must lie is known to be empty. The process makes roughly $\log_2 N$ comparisons.

Most programmers think that with the above description in hand, writing the code is easy; they're wrong. The only way you'll believe this is by putting down this column right now, and writing the code yourself. Try it.

I've given this problem as an in-class assignment in courses at Bell Labs and IBM. The professional programmers had one hour (sometimes more) to convert the above description into a program in the language of their choice; a high-level pseudocode was fine. At the end of the specified time, almost all the programmers reported that they had correct code for the task. We would then take 30 minutes to examine their code, which the programmers did with test cases. In many different classes and with over a hundred programmers, the results varied little: 90 percent of the programmers found bugs in their code (and I wasn't always convinced of the correctness of the code in which no bugs were found).

I found this amazing: only about 10 percent of professional programmers were able to get this small program right. But they aren't the only ones to find this task difficult. In the history in Section 6.2.1 of his *Sorting and Searching*, Knuth points out that while the first binary search was published in 1946, the first published binary search without bugs did not appear until 1962.

Writing The Program

The key idea of binary search is that we always know that if T is anywhere in $X[1\,..\,N]$, then it must be in a certain range of X. We'll use the shorthand *MustBe*(*range*) to mean that if T is anywhere in the array, then it must be in *range*. With this notation, it's easy to convert the above description of binary search into a program sketch.

```
initialize range to designate X[1..N]
loop
    { invariant: MustBe(range) }
    if range is empty,
        return that T is nowhere in the
        array
    compute M, the middle of the range
    use M as a probe to shrink the range
        if T is found during the
        shrinking process, return its
        position
endloop
```

The crucial part of this program is the *loop invariant*, which is enclosed in {}'s. This is an *assertion* about the program state that is invariantly true at the beginning and end of each iteration of the loop (hence its name); it formalizes the intuitive notion we had above.

We'll now refine the program, making sure that all our actions respect the invariant. The first issue we must face is the representation of *range*: we'll use two indices L and U (for "lower" and "upper") to represent the range $L\,..\,U$. (There are other possible representations for a range, such as its begin-