ANTONY HOARE

Made enduring contributions to programming language design and definition

A.M. TURING AWARD 1980
Emeritus Professor of Computing at the University of Oxford and is now a senior researcher at Microsoft Research in Cambridge, England. He received the 1980 ACM Turing Award for “his fundamental contributions to the definition and design of programming languages.” Knighted by the Queen of England in 2000.
When Brunel’s ship the SS Great Britain was launched into the River Thames, it made such a splash that several spectators on the opposite bank were drowned. Nowadays, engineers reduce the force of entry into the water by rope tethers which are designed to break at carefully calculated intervals.

When the first computer came into operation in the Mathematish Centrum in Amsterdam, one of the first tasks was to calculate the appropriate intervals and breaking strains of these tethers. In order to ensure the correctness of the program which did the calculations, the programmers were invited to watch the launching from the first row of the ceremonial viewing stand set up on the opposite bank. They accepted and they survived.
I therefore suggest that we should explore an additional method, which promises to increase the reliability of programs. The same method has assisted the reliability of designs in other branches of engineering, namely the use of mathematics to calculate the parameters and check, the soundness of a design before passing it for construction and installation.

Most people can learn to write simple programs. Few people can learn to write correct programs.

In September 1999, NASA’s Mars Climate Orbiter crashed into Mars instead of going into orbit. Preliminary findings indicated that one team of programmers used English units while the other used metric units for a key spacecraft operation. Total mission cost of $327.6 million.
Programming Without Thought

If I let my fingers wander idly over the keys of a typewriter it might happen that my screed made an intelligible sentence. If an army of monkeys were strumming on typewriters they might write all the books in the British Museum.

Levels of Correctness

1. legal according to syntax rules
2. ... fine print
3. good style
4. makes sense
5. no runtime errors
6. works on some data (tested)
7. formal correctness with respect to some specification
8. *is correct*
The opening levels of correctness are more familiar since they are obvious parts of the development cycle. It is the later levels which are more subtle and difficult. One reason that state of software in the world is disappointing is that programmers tend to pay less attention to the later levels because

1. they are unwilling to put in the effort,
2. they are pressed for time, or
3. not enough emphasis is placed on correctness.

For this reason it is important to examine correctness.
The Java compiler checks to see if the input file conforms to a precise set of syntax rules. Adherence to syntax rules is necessary for the program to successfully communicate its actions to the computer and to any human reader. Advanced Java programmers know more constructs enabling them to do write more complex programs easier than novice programs. For example, `?`, for-each loops, exception handling, interfaces, generics, wildcards, and so on. The compiler may check that the program uses the constructs properly, but it does not teach the programmer how to use them.
In addition to the syntax checks, the compiler checks lots of obvious and not so obvious rules. For example, the programmer is free to choose any syntactically correct identifier to refer to objects in the program, but it is a rule that the identifier must be declared somewhere. So, if the programmer misspells a name, this will lead to a semantic error.
• good identifier names
• indent consistently, use white space judiciously
• use exception handling
• do not use gotos or multiple return
• avoid `import .*`
• localize declarations
• avoid side effects: use functions, avoid non-local variables
• modularize
Localize Scope

- Declare variables right before you need them. (Localyze scope.)
- Initialize the variable with the value you need.
- Don’t reuse the variable with a different value or purpose.
- Mark the variable as `final`.
Some languages do not check for runtime errors. This leads to something worse than a runtime error—the absence of a runtime error. For example, by not checking array indexes or doing illegal operations on pointers, C and C++ allow the program to write over useful parts of memory. This is called a buffer overflow and is the cause of much malware. This is not possible in Java, Ada (checks can be suppressed), C#, Python, Modula-3, Haskell, or any recently designed high-level language.
• `java.lang.ArithmeticException` (e.g., division by zero)
• `java.lang.NullPointerException` ("billion dollar mistake")
• `java.lang.ArrayIndexOutOfBoundsException`

These represent a certain type of logical error in the program. Java was designed so that compiler would detect at compile-time many things which would be runtime errors in other languages.

No language can eliminate runtime errors altogether; no language can eliminate logical errors.

*Billion Dollar Mistake* — presentation by Hoare in 2009.
Hoare in his 1981 Turing award lecture decried the lack of runtime range checking. The problem persists today.

*In any respectable branch of engineering, failure to observe such elementary precautions would have long been against the law.*
Figure 1. The number of software vulnerabilities cataloged by the NIST National Vulnerability Database skyrocketed in 2017, and the fraction of vulnerabilities involving buffers (either categorized as “buffer error” or containing the keyword “buffer”) kept pace.
The whole economic boom in cybersecurity seems largely to be a consequence of poor engineering. We have allowed ourselves to become dependent on an infrastructure with the characteristics of a medieval firetrap—a maze of twisty little streets and passages bordered by buildings highly vulnerable to arson.
To a disturbing extent the kinds of underlying flaws exploited by attackers have not changed very much. . . . One of the most widespread vulnerabilities found recently, the so-called Heartbleed flaw in OpenSSL, was apparently overlooked . . . for more than two years. What was the flaw? Failure to apply adequate bounds-checking to a memory buffer.

Carl Landwehr, CACM, 2015, pages 24–25
Runtime Guarantees Better Than Nothing

DESIGN & IMPLEMENTATION

Stack smashing

The lack of bounds checking on array subscripts and pointer arithmetic is a major source of bugs and security problems in C. Many of the most infamous Internet viruses have propagated by means of stack smashing, a particularly nasty form of buffer overflow attack. Consider a (very naive) routine designed to read a number from an input stream:

```c
int get_acct_num(FILE *fp) {
    char buf[100];
    char *p = buf;
    do {
        /* read from stream s: */
        *p = getc(s);
    } while (*p++ != '\n');
    *p = '\0';
    /* convert ascii to int: */
    return atoi(buf);
}
```

If the stream provides more than 100 characters without a newline ('\n'), those characters will overwrite memory beyond the confines of `buf`, as shown by the large white arrow in the figure. A careful attacker may be able to invent a string whose bits include both a sequence of valid machine instructions and a replacement value for the subroutine's return address. When the routine attempts to return, it will jump into the attacker's instructions instead.

Stack smashing can be prevented by manually checking array bounds in C, or by configuring the hardware to prevent the execution of instructions in the stack (see the sidebar on page 179). It would never have been a problem in the first place; however, if C had been designed for automatic bounds checks.
Testing

- code walk-throughs, code inspection
- unit testing
- integration testing
- bottom-up testing, drivers
- top-down testing, stubs
- regression testing
- black-box testing
- white-/clear-/glass-box testing
- statement and path coverage
**Assertions**

*Assertion.* An assertion is a boolean-valued expression relating the program variables, e.g., $x>0$, or $x+y>0$.

*Precondition.* A precondition is an assertion that must be true, *if* the following statement or block of code in the program is to work correctly. To make use of code correctly, one must insure that the preconditions are met.

*Postcondition.* A postcondition is an assertion that is true, after a statement or block of code has been executed, *if* the preconditions are met. The author of the code promises that the postconditions will be achieved.
Figure 5.5
Annotating an algorithm with invariants.
Assert statement

Examples in Java ...
A property that remains true about an object after every method call on the object. E.e., “Account balance is always non-negative.”
It is not possible to understand a program completely by mentally executing it. The number of states is too large to comprehend. Well-constructed programs have meaningful relationships among the program variables, and even though the values of the variables change the relationships do not. It is through these unchanging relationships that it is possible to understand and to write correct programs.

Loop invariant. A loop invariant is an assertion relating the values of the program variables which is true before and after every execution of the body of a loop. Invariants are a bridge from the mutable world of the machine to the timeless truths of mathematics.
The following game illustrates the power of an invariant to understand dynamic systems.
Red and Blue take alternating turns, with Red going first. Red takes a turn by drawing a red line segment, either horizontal or vertical, connecting any two adjacent points on the grid that are not yet connected by any line segment. Blue takes a turn by doing the same thing, except that the line segment drawn is blue. Red’s goal is to form a closed curve (i.e., a sequence of (four or more) distinct line segments starting at some point and returning to that point) comprised entirely of red line segments. Blue’s goal is to prevent Red from doing so. The game ends when either Red has formed a closed curve or there are no more line segments to draw.

```
6 . . . . . . . .
5 . . . . . . . .
4 . . . . . . . .
3 . . . . . . . .
2 . . . . . . . .
1 . . . . . . . .
```

```
1 2 3 4 5 6 7 8
```
```java
public static int sum(int a[]) {
    int s = 0;
    for (int i = 0; i < a.length; i++) {
        // s is the sum of the first i array elements
        // s == a[0] + .. + a[i-1]
        s = s + a[i];
    }
    return s;
}
```
```java
public static int quotient(int n, int d) {
    int q = 0, r = n;
    assert n=q*d + r;
    while (r >= d) {
        r = r - d;
        q = q + 1;
        assert n=q*d + r;
    }
    return q;
}
```
public static int power(int x, int n) {
    int p = 1, i = 0;
    assert p == Math.pow(x, i);
    while (i < n) {
        p = p * x;
        i = i + 1;
        assert p == Math.pow(x, i);
    }
    return p;
}
public static int power(int x, int n) {
    int p = 1, i = 0;
    while (i < n) {
        p = p * x;
        i = i + 1;
    }
    return p;
}
Performance Requirements

Some program requirements do not pertain to the output, but to the performance. Compare the following two ways to compute $\sum_{i=1}^{n} i$.

Algorithm 1 – $O(n)$

```
final int n = Integer.parseInt (args[0]);
int sum = 0;
for (int count=1; count<=n; i++) {
    sum += count;
}
```

Algorithm 2 – $O(1)$

```
final int n = Integer.parseInt (args[0]);
int sum = (n*(n+1))/2;
```
Figure 7.4
Growth rates of some functions.
Figure 7.6
The sphere of algorithmic problems: Version I.
Figure 8.3
The sphere of algorithmic problems: Version II.