Asserts

/run/media/lijunjie/资料/Learning Video/writing-running-fixing-code/03_testing-and-debugging/01_testing/07_asserts_instructions.html

In any form of testing, it can be useful to not only check the end result, but also that the *invariants* of the system are maintained in the middle of its execution. An invariant is a property that is (or should be) always true at a given point in the program. When you know an invariant that should be true at a given point in the code, you can write an *assert statement* , which checks that it is true. *assert(expr);* checks that *expr* is true. If it is true, then nothing happens, and execution continues as normal. However, if *expr* is false, then it prints a message stating that an assertion failed, and aborts the program—terminating it immediately wherever it is.

As an example, recall the printFactors function from one of the previous practice problems. We could add some *assert* statements to express invariants of our algorithm:

```
void printFactors(int n) {
  if (n <= 1) {
    return;
  }
  int p = 2;
  while (!isPrime(n)) {
    assert(isPrime(p));      // p should always be prime
    assert(p < n);           // p should always be less than n
    if (n % p == 0) {
      printf("%d * ", p);
      n = n / p;
    }
    else {
      p = nextPrimeAfter(p); // helper function to get next prime
    }
  }
  printf("%d\n", n);
}
```

Here, we have added two *assert* statements. The first, on line 7, checks that p is prime (using our *i sPrime* function). This fact should always be true here—if it is not, our algorithm is broken (we may end up printing non-prime factors of the numbers, which is incorrect). How could such an error occur? One possibility would be a bug in *nextPrimeAfter* —the helper function we have written to find the next prime after a particular prime. Another possibility would be if we accidentally modify p in a way that we do not intend to. Of course, if our code is correct, neither of these will happen, and the *assert* will do nothing, but the point is to help us if we do make a mistake.

The second *assert* checks another invariant of our algorithm: *p* should always be less than *n* inside this loop (think about why…). As with the first *assert* , we hope that it has no effect—just checking that the expression is always true—but if something is wrong, it will help us detect the problem.

Note that *assert* statements are an example of the principle that if our program has an error we want it to *fail fast* —that is, we would rather the program crash as soon after the error occurs as possible. The longer the program runs after an error occurs, the more likely it is to give incorrect answers and the more difficult it is to debug. Ideally, when our program has an error, we will have an assert fail immediately after it, pointing out exactly what the problem is and where it lies.

In almost all cases, giving the wrong answer (due to an undetected error) is significantly worse than the program detecting the situation and crashing. To see this tradeoff, consider the case of software to compute a missile launch trajectory. If there is an error in such software, would you rather it give incorrect launching instructions, or print a message that an error occurred and abort?

Many novice and intermediate programmers worry that *asserts* will slow their program down. In general, the slowdown is negligible, especially on fast modern computers. In many situations, 1–2% performance just does not matter—do you really care if you get your answer in 0.1 seconds versus 0.11 seconds? However, there are performance critical situations where ever bit of speed matters. For these situations, you can pass the *-DNDEBUG* option to the compiler to turn off the asserts in your optimized code. For all other situations, keeping them active is generally advisable. Note that performance critical code is the domain of expert programmers who also have a deep understanding of the underlying hardware—such is well beyond the scope of this specialization.