



Testing

“I have only proved the code correct, not tried it.”

—Donald Knuth

This chapter covers testing and design for correctness. These are huge topics, so we can only scratch their surfaces. The emphasis is on giving some practical ideas and techniques for testing units, such as functions and classes, of a program. We discuss the use of interfaces and the selection of tests to run against them. We emphasize the importance of designing systems to simplify testing and the use of testing from the earliest stages of development. Proving programs correct and dealing with performance problems are also briefly considered.

26.1 What we want	26.4 Design for testing
26.1.1 Caveat	26.5 Debugging
26.2 Proofs	26.6 Performance
26.3 Testing	26.6.1 Timing
26.3.1 Regression tests	26.7 References
26.3.2 Unit tests	
26.3.3 Algorithms and non-algorithms	
26.3.4 System tests	
26.3.5 Finding assumptions that do not hold	

26.1 What we want

Let's try a simple experiment. Write a binary search. Do it now. Don't wait until the end of the chapter. Don't wait until after the next section. It's important that you try. Now! A binary search is a search in a sorted sequence that starts at the middle:

- If the middle element is equal to what we are searching for, we are finished.
- If the middle element is less than what we are searching for, we look at the right-hand half, doing a binary search on that.
- If the middle element is greater than what we are searching for, we look at the left-hand half, doing a binary search on that.
- The result is an indicator of whether the search was successful and something that allows us to modify the element, if found, such as an index, a pointer, or an iterator.

Use less than ($<$) as the comparison (sorting) criterion. Feel free to use any data structure you like, any calling conventions you like, and any way of returning the result that you like, but do write the search code yourself. In this rare case, using someone else's function is counterproductive, even with proper acknowledgment. In particular, don't use the standard library algorithm (`binary_search` or `equal_range`) that would have been your first choice in most situations. Take as much time as you like.

So now you have written your binary search function. If not, go back to the previous paragraph. How do you know that your search function is correct? If you haven't already, write down why you are convinced that this code is correct. How confident are you about your reasoning? Are there parts of your argument that might be weak?

That was a trivially simple piece of code. It implemented a very regular and well-known algorithm. Your compiler is on the order of 200K lines of code, your operating system is 10M to 50M lines of code, and the safety-critical code in the airplane you'll fly on for your next vacation or conference is 500K to 2M lines of code. Does that make you feel comfortable? How do the techniques you used for your binary search function scale to real-world software sizes?

Curiously, given all that complex code, most software works correctly most of the time. We do not count anything running on a game-infested consumer PC as "critical." Even more importantly, safety-critical software works correctly just about all of the time. We cannot recall an example of a plane or a car crashing because of a software failure over the last decade. Stories about bank software getting seriously confused by a check for \$0.00 are now very old; such things essentially don't happen anymore. Yet software is written by people like you. You know that you make mistakes; we all do, so how do "they" get it right?

The most fundamental answer is that "we" have figured out how to build reliable systems out of unreliable parts. We try hard to make every program, every class, and every function correct, but we typically fail our first attempt at that. Then we debug, test, and redesign to find and remove as many errors as possible. However, in any nontrivial system, some bugs will still be hiding. We know that, but we can't find them – or rather, we can't find them all with the time and effort we are able and willing to expend. Then, we redesign the system yet again to recover from unexpected and "impossible" events. The result can be systems that are spectacularly reliable. Note that such reliable systems may still harbor errors – they usually do – and still occasionally work less well than we would like. However, they don't crash and always deliver minimally acceptable service. For example, a phone system may not manage to connect every call when demand is exceptionally high, but it never fails to connect many calls.

Now, we could be philosophical and discuss whether an unexpected error that we have conjectured and catered for is really an error, but let's not. It is more profitable and productive for systems builders "just" to figure how to make our systems more reliable.

26.1.1 Caveat

Testing is a huge topic. There are several schools of thought about how testing should be done, and different industries and application areas have different traditions and standards for testing. That's natural – you really don't need the same reliability standard for video games and avionics software – but it leads to confusing differences in terminology and tools. Treat this chapter as a source of ideas for your personal projects and as a source of ideals if you encounter testing of major systems. The testing of major systems involves a variety of combinations of tools and organizational structures that it would make little sense to try to describe here.

26.2 Proofs

Wait a minute! Why don't we just prove that our programs are correct, rather than fussing around with tests? As Edsger Dijkstra succinctly pointed out, "Testing can reveal the presence of errors, not their absence." This leads to an obvious desire to prove programs correct "much as mathematicians prove theorems."

Unfortunately, proving nontrivial programs correct is beyond the state of the art (outside very constrained applications domains), the proofs themselves can contain errors (as can the ones mathematicians produce), and the whole field of program proving is an advanced topic. So, we try as hard as we can to structure our programs so that we can reason about them and convince ourselves that they are correct. However, we also test (§26.3) and try to organize our code to be resilient against remaining errors (§26.4).

26.3 Testing

In §5.11, we described testing as "a systematic way to search for errors." Let's look at techniques for doing that.

People distinguish between *unit testing* and *system testing*. A "unit" is something like a function or a class that is a part of a complete program. If we test such units in isolation, we know where to look for the cause of problems when we find an error; any error will be in the unit that we are testing (or in the code we use to conduct the tests). This contrasts with system testing, where we test a complete system and all we know is that an error is "somewhere in the system." Typically, errors found in system testing – once we have done a good job at unit testing – relate to undesirable interactions between units. They are harder to find than errors within individual units and often more expensive to fix.

Obviously, a unit (say, a class) can be composed of other units (say, functions and other classes), and systems (say, an electronic commerce system) can be composed of other systems (say, a database, a GUI, a networking system, and an order validation system), so the distinction between unit testing and systems testing isn't as clear as you might have thought, but the general idea is that by testing our units well, we save ourselves work – and our end users pain.

One way of looking at testing is that any nontrivial system is built out of units, and these units are themselves built out of smaller units. So, we start testing the smallest units, then we test the units composed from those, and we work our way up until we have tested the whole system; that is, "the system" is just the largest unit (until we use that as a unit for some yet larger system).

So, let's first consider how to test a unit (such as a function, a class, a class hierarchy, or a template). Testers distinguish between white-box testing (where you can look at the detailed implementation of what you are testing) and black-box testing

(where you can look only at the interface of what you are testing). We will not make a big deal of this distinction; by all means read the implementation of what you test. But remember that someone might later come and rewrite that implementation, so try not to depend on anything that is not guaranteed in the interface. In fact, when testing anything, the basic idea is to throw anything we can at its interface to see if it responds reasonably.

Mentioning that someone (maybe yourself) might change the code after you tested it brings us to regression testing. Basically, whenever you make a change, you have to retest to make sure that you have not broken anything. So when you have improved a unit, you rerun its unit tests, and before you give the complete system to someone else (or use it for something real yourself), you run the complete system test.

Running such complete tests of a system is often called *regression testing* because it usually includes running tests that have previously found errors to see if these errors are still fixed. If not, the program has “regressed” and needs to be fixed again.

26.3.1 Regression tests

Building up a large collection of tests that have been useful for finding errors in the past is a major part of building an effective test suite for a system. Assume that you have users; they will send you bugs. Never throw away a bug report! Professionals use bug-tracking systems to ensure that. Anyway, a bug report demonstrates either an error in the system or an error in a user’s understanding of the system. Either way it is useful.

Usually, a bug report contains far too much extraneous information, and the first task of dealing with it is to produce the smallest program that exhibits the reported problem. This often involves cutting away most of the code submitted: in particular, we try to eliminate the use of libraries and application code that does not affect the error. Finding that minimal test program often helps us localize the bug in the system’s code, and that minimal program is what is added to the regression test suite. The way we find that minimal program is to keep removing code until the error disappears – and then reinsert the last bit of code we removed. This we do until we run out of candidates for removal.

Just running hundreds (or tens of thousands) of tests produced from old bug reports may not seem very systematic, but what we are really doing here is to systematically use the experience of users and developers. The regression test suite is a major part of a developer group’s institutional memory. For a large system, we simply can’t rely on having the original developers available to explain details of the design and implementation. The regression suite is what keeps a system from mutating away from what the developers and users have agreed to be its proper behavior.

26.3.2 Unit tests

OK. Enough words for now! Let's try a concrete example: let's test a binary search. Here is the specification from the ISO standard (§25.3.3.4):

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value);
```

```
template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                  const T& value, Compare comp);
```

Requires: The elements e of $[first, last)$ are partitioned with respect to the expressions $e < value$ and $!(value < e)$ or $comp(e, value)$ and $!comp(value, e)$. Also, for all elements e of $[first, last)$, $e < value$ implies $!(value < e)$ or $comp(e, value)$ implies $!comp(value, e)$.

Returns: true if there is an iterator i in the range $[first, last)$ that satisfies the corresponding conditions: $!(*i < value) \ \&\& \ !(value < *i)$ or $comp(*i, value) == false \ \&\& \ comp(value, *i) == false$.

Complexity: At most $\log(last - first) + 2$ comparisons.

Nobody said that a formal specification (well, semiformal) was easy to read for the uninitiated. However, if you actually did the exercise of designing and implementing a binary search that we strongly suggested at the beginning of the chapter, you have a pretty good idea of what a binary search does and how to test it. This (standard) version takes a pair of forward iterators (§20.10.1) and a value as arguments and returns **true** if the value is in the range defined by the iterators. The iterators must define a sorted sequence. The comparison (sorting) criterion is $<$. We'll leave the second version of **binary_search** that takes a comparison criterion as an extra argument as an exercise.

Here, we will deal only with errors that are not caught by the compiler, so examples like these are somebody else's problem:

```
binary_search(1,4,5);           // error: an int is not a forward iterator
vector<int> v(10);
binary_search(v.begin(),v.end(),"7"); // error: can't search for a string
// in a vector of ints
binary_search(v.begin(),v.end()); // error: forgot the value
```

How can we *systematically* test **binary_search()**? Obviously we can't just try every possible argument for it, because every possible argument would be every possible sequence of every possible type of value – that would be an infinite number of



tests! So, we must choose tests and to choose, we need some principles for making a choice:

- Test for *likely mistakes* (find the most errors).
- Test for *bad mistakes* (find the errors with the worst potential consequences).

By “bad,” we mean errors that would have the direst consequences. In general, that’s a fuzzy notion, but it can be made precise for a specific program. For example, for a binary search considered in isolation, all errors are about equally bad, but if we used that `binary_search()` in a program where all answers were carefully double-checked, getting a wrong answer from `binary_search()` might be far more acceptable than having it not return because it went into an infinite loop. In that case, we would spend greater effort tricking `binary_search()` into an infinite (or very long) loop than we would trying to trick it into giving a wrong answer. Note our use of “tricking” here. Testing is – among other things – an exercise in applying creative thinking to the problem of “How can we get this code to misbehave?” The best testers are not just systematic, but also quite devious (in a good cause, of course).

26.3.2.1 Testing strategy

How do we go about breaking `binary_search()`? We start by looking at `binary_search()`’s requirements, that is, what it assumes about its inputs. Unfortunately, from our perspective as testers, it is clearly stated that `[first,last)` must be a sorted sequence; that is, it is the caller’s job to ensure that, so we can’t fairly try to break `binary_search()` by giving it unsorted input or a `[first,last)` where `last < first`. Note that the requirements for `binary_search()` do not say what it will do if we give it input that doesn’t meet its requirements. Elsewhere in the standard, it says that it may throw an exception in that case, but it is not required to. These facts are good to remember for when we test uses of `binary_search()`, though, because a caller failing to establish the requirements of a function, such as `binary_search()`, is a likely source of errors.

We can imagine the following kinds of errors for `binary_search()`:

- Never returned (e.g., infinite loop)
- Crash (e.g., bad dereference, infinite recursion)
- Value not found even though it was in the sequence
- Value found even though it wasn’t in the sequence

In addition, we remember the following “opportunities” for user errors:

- The sequence is not sorted (e.g., `{2,1,5,-7,2,10}`).
- The sequence is not a valid sequence (e.g., `binary_search(&a[100], &a[50],77)`).

How might an implementer have made a mistake (for testers to find) for a simple call `binary_search(p1,p2,v)`? Errors often occur for “special cases.” In particular, when considering sequences (of any sort), we always look for the beginning and the end. In particular, the empty sequence should always be tested. So, let’s consider a few arrays of integers that are properly ordered as required:

```

{ 1,2,3,5,8,13,21 } // an “ordinary sequence”
{ } // the empty sequence
{ 1 } // just one element
{ 1,2,3,4 } // even number of elements
{ 1,2,3,4,5 } // odd number of elements
{ 1, 1, 1, 1, 1, 1, 1 } // all elements equal
{ 0,1,1,1,1,1,1,1,1,1,1 } // different element at beginning
{ 0,0,0,0,0,0,0,0,0,0,0,1 } // different element at end

```

Some test sequences are best generated by a program:

- `vector<int> v;`
`for (int i=0; i<100000000; ++i) v.push_back(i); // a very large sequence`
- Some sequences with a random number of elements
- Some sequences with random elements (but still ordered)

This is not as systematic as we’d have liked. After all, we “just picked” some sequences. However, we used some fairly general rules of thumb that often are useful when dealing with sets of values; consider:

- The empty set
- Small sets
- Large sets
- Sets with extreme distributions
- Sets where “what is of interest” happens near the end
- Sets with duplicate elements
- Sets with even and with odd numbers of elements
- Sets generated using random numbers

We use the random sequences just to see if we can get lucky (i.e., find an error) with something we didn’t think about. It’s a brute-force technique, but relatively cheap in terms of our time.

Why “odd and even”? Well, lots of algorithms partition their input sequences, e.g., into the first half and the last half, and maybe the programmer considered only the odd or the even case. More generally, when we partition a sequence,

the point where we split it becomes the end of a subsequence, and we know that errors are likely near ends of sequences.

In general, we look for

- Extreme cases (large, small, strange distributions of input, etc.)
- Boundary conditions (anything near a limit)

What that really means, depends on the particular program we are testing.

26.3.2.2 A simple test harness

We have two categories of tests: tests that should succeed (e.g., searching for a value that's in a sequence) and tests that should fail (e.g., searching for a value in an empty sequence). For each of our sequences, let's construct some succeeding and some failing tests. We will start from the simplest and most obvious and proceed to improve until we have something that's good enough for our `binary_search` example:

```
vector<int> v { 1,2,3,5,8,13,21 };
if (binary_search(v.begin(),v.end(),1) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),5) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),8) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),21) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),-7) == true) cout << "failed";
if (binary_search(v.begin(),v.end(),4) == true) cout << "failed";
if (binary_search(v.begin(),v.end(),22) == true) cout << "failed";
```

This is repetitive and tedious, but it will do for a start. In fact, many simple tests are nothing but a long list of calls like this. This naive approach has the virtue of being extremely simple. Even the newest member of the test team can add a new test to the set. However, we can usually do much better. For example, when something failed here, we are not told which test failed. That's unacceptable. Also, writing tests is no excuse for regressing to "cut and paste" programming. We need to consider the design of our testing code, just like any other code. So:

```
vector<int> v { 1,2,3,5,8,13,21 };
for (int x : {1,5,8,21,-7,2,44})
    if (binary_search(v.begin(),v.end(),x) == false) cout << x << " failed";
```

Assuming that we will eventually have dozens of tests, this will make a huge difference. For testing real-world systems, we often have many thousands of tests, so being precise about what test failed is essential.

Before going further, note another example of (semi-systematic) testing technique: we tested with correct values, choosing some from the ends of the sequence and some from “the middle.” For this sequence we could have tried all values, but typically that’s not a realistic option. For the failing values, we chose one from each end and one in the middle. Again, this is not perfectly systematic, but we begin to see a pattern that is useful whenever we deal with sequences of values or ranges of values – and that’s very common.

What’s wrong with these tests?

- We (initially) wrote the same things repeatedly.
- We (initially) numbered the tests manually.
- The output is very minimal (not very helpful).

After looking at this for a while, we decided to keep our tests as data in a file. Each test would contain an identifying label, a value to be looked up, a sequence, and an expected result. For example:

```
{ 27 7 { 1 2 3 5 8 13 21 } 0 }
```

This is test number **27**. It looks for **7** in the sequence **{ 1,2,3,5,8,13,21 }** expecting the result **0** (meaning **false**). Why do we put the test inputs in a file rather than placing them right into the text of the test program? Well, in this case we could have typed the tests straight into the program text, but having a lot of data in a source code file can be messy, and often, we use programs to generate test cases. Machine-generated test cases are typically in data files. Also, we can now write a test program that we can run with a variety of files of test cases:

```
struct Test {
    string label;
    int val;
    vector<int> seq;
    bool res;
};

istream& operator>>(istream& is, Test& t); // use the described format

int test_all(istream& is)
{
    int error_count = 0;
    for (Test t; is>>t; ) {
        bool r = binary_search(t.seq.begin(), t.seq.end(), t.val);
```

```

        if (r != t.res) {
            cout << "failure: test " << t.label
                << " binary_search: "
                << t.seq.size() << " elements, val==" << t.val
                << " -> " << t.res << "\n";
            ++error_count;
        }
    }
    return error_count;
}

int main()
{
    int errors = test_all(ifstream("my_tests.txt"));
    cout << "number of errors: " << errors << "\n";
}

```

Here is some test input using the sequences we listed above:

```

{1.1 1 { 1 2 3 5 8 13 21 } 1 }
{1.2 5 { 1 2 3 5 8 13 21 } 1 }
{1.3 8 { 1 2 3 5 8 13 21 } 1 }
{1.4 21 { 1 2 3 5 8 13 21 } 1 }
{1.5 -7 { 1 2 3 5 8 13 21 } 0 }
{1.6 4 { 1 2 3 5 8 13 21 } 0 }
{1.7 22 { 1 2 3 5 8 13 21 } 0 }

{2 1 { } 0 }

{3.1 1 { 1 } 1 }
{3.2 0 { 1 } 0 }
{3.3 2 { 1 } 0 }

```

Here we see why we used a string label rather than a number: that way we can “number” our tests using a more flexible system – here using a decimal system to indicate separate tests for the same sequence. A more sophisticated format would eliminate the need to repeat a sequence in our test data file.

26.3.2.3 Random sequences

When we choose values to be used in testing, we try to outwit the implementers (who are often ourselves) and to use values that focus on areas where we know bugs can hide (e.g., complicated sequences of conditions, the ends of sequences,



loops, etc.). However, that's also what we did when we tried to write and debug the code. So, we might repeat a logical mistake from the design when we design the tests and completely miss a problem. This is one reason it is a good idea to have someone different from the developer(s) involved with designing the tests. We have one technique that occasionally helps with that problem: just generate (a lot of) random values. For example, here is a function that writes a test description to `cout` using `randint()` from §24.7 and `std_lib_facilities.h`:

```
void make_test(const string& lab, int n, int base, int spread)
    // write a test description with the label lab to cout
    // generate a sequence of n elements starting at base
    // the average distance between elements is uniformly distributed
    // in [0:spread)
{
    cout << "{ " << lab << " " << n << " { ";
    vector<int> v;
    int elem = base;
    for (int i = 0; i<n; ++i) { // make elements
        elem+= randint(spread);
        v.push_back(elem);
    }

    int val = base+ randint(elem-base); // make search value
    bool found = false;
    for (int i = 0; i<n; ++i) { // print elements and see if val is found
        if (v[i]==val) found = true;
        cout << v[i] << " ";
    }
    cout << " } " << found << " }\n";
}
```

Note that we did not use `binary_search` to see if the random `val` was in the random sequence. We can't use what we are testing to determine the correct value of a test.

Actually, `binary_search` isn't a particularly suitable example of the brute-force random number approach to testing. We doubt that this will find any bugs that are not picked up by our "hand-crafted" tests, but often this technique is useful. Anyway, let's make a few random tests:

```
int no_of_tests = randint(100); // make about 50 tests
for (int i = 0; i<no_of_tests; ++i) {
    string lab = "rand_test_";
    make_test(lab+to_string(i), // to_string from §23.2
```

```

    randint(500),           // number of elements
    0,                     // base
    randint(50));         // spread
}

```

Generated tests based on random numbers are particularly useful when we need to test the cumulative effects of many operations where the result of an operation depends on how earlier operations were handled, that is, when a system has state; see §5.2.

The reason that random numbers are not all that useful for `binary_search` is that each search of a sequence is independent of all other searches of that sequence. That of course assumes that the implementation of `binary_search` hasn't done something terminally stupid, such as modifying its sequence. We have a better test for that (exercise 5).

26.3.3 Algorithms and non-algorithms

We have used `binary_search()` as an example. It's a proper algorithm with

- Well-specified requirements on its inputs
- A well-specified effect on its inputs (in this case, no effects)
- No dependencies on objects that are not its explicit inputs
- Without serious constraints imposed by the environment (e.g., no specified time, space, or resource-sharing requirements)


It has obvious and explicitly stated pre- and post-conditions (§5.10). In other words, it's a tester's dream. Often, we are not so lucky: we have to test messy code that (at best) is defined by a somewhat sloppy English text and a couple of diagrams.


Wait a minute! Are we indulging in sloppy logic here? How can we talk about correctness and testing when we don't have a precise specification of what the code is supposed to do? The problem is that much of what needs to be done in software is not easy to specify in perfectly clear mathematical terms. Also, in many cases where it in theory could be specified like that, the math is beyond the abilities of the programmers who write and test the code. So we are left with the ideal of perfectly precise specifications and a reality of what someone (such as us) can manage under real-world conditions and time pressures.

So, assume that you have a messy function that you have to test. By “messy” we mean:

- *Inputs*: Its requirements on its (explicit or implicit) inputs are not specified quite as well as we would like.
- *Outputs*: Its (explicit or implicit) outputs are not specified quite as well as we would like.
- *Resources*: Its use of resources (time, memory, files, etc.) is not specified quite as well as we would like.

By “explicit or implicit” we mean that we have to look not just at the formal parameters and the return value, but also at any effects on global variables, **io-streams**, files, free-store memory allocation, etc. So, what can we do? First of all, such a function is almost certainly too long – or we could have stated its requirements and effects more clearly. Maybe we are talking about a function that is five pages long or uses “helper functions” in complicated and non-obvious ways. You may think that five pages is a lot for a function. It is, but we have seen much, much longer functions than that. Unfortunately, they are not uncommon.

 If it is our code and if we had time, we would first of all try to break such a “messy function” up into smaller functions that come closer to our ideals of a well-specified function and first test those. However, here we will assume that our aim is to test the software – that is, to systematically find as many errors as possible – rather than (just) fixing bugs as we find them.

 So, what do we look for? Our job as testers is to find errors. Where are bugs likely to hide? What characterizes code that is likely to contain bugs?

- Subtle dependencies on “other code”: look for use of global variables, non-**const**-reference arguments, pointers, etc.
- Resource management: look for memory management (**new** and **delete**), file use, locks, etc.
- Look for loops: check end conditions (as for **binary_search()**).
- **if**-statements and switches (often referred to as “branching”): look for errors in their logic.

Let’s look at examples of each.

26.3.3.1 Dependencies

Consider this nonsense function:

```
int do_dependent(int a, int& b)    // messy function
    // undisciplined dependencies
{
    int val ;
    cin>>val;
    vec[val] += 10;
    cout << a;
    b++;
    return b;
}
```

To test **do_dependent()**, we can’t just synthesize sets of arguments and see what it does with them. We have to take into account that it uses the global variables **cin**,

cout, and **vec**. That's pretty obvious in this little nonsense function, but in real code this may be hidden in a larger amount of code. Fortunately, there is software that can help us find such dependencies. Unfortunately, it is not always easily available or widely used. Assuming that we don't have analysis software to help us, we go through the function line by line, listing all its dependencies.

To test **do_dependent()**, we have to consider

- Its inputs:
 - The value of **a**
 - The value of **b** and the value of the **int** referenced by **b**
 - The input from **cin** (into **val**) and the state of **cin**
 - The state of **cout**
 - The value of **vec**, in particular, the value of **vec[val]**
- Its outputs:
 - The return value
 - The value of the **int** referenced by **b** (we incremented it)
 - The state of **cin** (beware of stream state and format state)
 - The state of **cout** (beware of stream state and format state)
 - The state of **vec** (we assigned to **vec[val]**)
 - Any exceptions that **vec** might have thrown (**vec[val]** might be out of range)

This is a long list. In fact, that list is longer than the function itself. This goes a long way toward explaining our dislike of global variables and our concerns about non-**const** references (and pointers). There really is something very nice about a function that just reads its arguments and produces a result as a return value: we can easily understand and test it.

Once the inputs and outputs are identified, we are basically back to the **binary_search()** case. We simply generate tests with input values (for explicit and implicit inputs) to see if they give the desired outputs (considering both implicit and explicit outputs). With **do_dependent()**, we would probably start with a very large **val** and a negative **val**, to see what happens. It looks as if **vec** had better be a range-checked **vector** (or we can very simply generate really bad errors). We would of course check what the documentation said about all those inputs and outputs, but with a messy function like that we have little hope of the specification being complete and precise, so we will just break the functions (i.e., find errors) and start asking questions about what is correct. Often, such testing and questions should lead to a redesign.

26.3.3.2 Resource management

Consider this nonsense function:

```
void do_resources1(int a, int b, const char* s) // messy function
    // undisciplined resource use
{
    FILE* f = fopen(s,"r");           // open file (C style)
    int* p = new int[a];              // allocate some memory
    if (b<=0) throw Bad_arg();       // maybe throw an exception
    int* q = new int[b];              // allocate some more memory
    delete[] p;                       // deallocate the memory pointed to by p
}
```


To test `do_resources1()`, we have to consider whether every resource acquired has been properly disposed of, that is, whether every resource has been either released or passed to some other function.


Here, it is obvious that

- The file named `s` is not closed
- The memory allocated for `p` is leaked if `b<=0` or if the second `new` throws
- The memory for `q` is leaked if `0<b`

In addition, we should always consider the possibility that an attempt at opening a file might fail. To get this miserable result, we deliberately used a very old-fashioned programming style (`fopen()` is the standard C way of opening files). We could have made the job for testers more straightforward by writing

```
void do_resources2(int a, int b, const char* s) // less messy function
{
    ifstream is(s);                   // open file
    vector<int>v1(a);                  // create vector (owning memory)
    if (b<=0) throw Bad_arg();       // maybe throw an exception
    vector<int> v2(b);                 // create another vector (owning memory)
}
```

 Now every resource is owned by an object with a destructor that will release it. Considering how we could write a function more simply (more cleanly) is sometimes a good way to get ideas for testing. The “Resource Acquisition Is Initialization” (RAII) technique from §19.5.2 provides a general strategy for this kind of resource management problem.

 Please note that resource management is not just checking that every piece of memory allocated is deleted. Sometimes we receive resources from elsewhere (e.g.,

as an argument), and sometimes we pass resources out of a function (e.g., as a return value). It can be quite hard to determine what is right about such cases. Consider:

```
FILE* do_resources3(int a, int* p, const char* s)    // messy function
// undisciplined resource passing
{
    FILE* f = fopen(s,"r");
    delete p;
    delete var;
    var = new int[27];
    return f;
}
```

Is it right for `do_resources3()` to pass the (supposedly) opened file back as the return value? Is it right for `do_resources3()` to delete the memory passed to it as the argument `p`? We also added a really sneaky use of the global variable `var` (obviously a pointer). Basically, passing resources in and out of functions is common and useful, but to know if it is correct requires knowledge of a resource management strategy. Who owns the resource? Who is supposed to delete/release it? The documentation should clearly and simply answer those questions. (Dream on.) In either case, passing of resources is a fertile area for bugs and a tempting target for testing.

Note how we (deliberately) complicated the resource management example by using a global variable. Things can get really messy when we start to mix the sources of likely bugs. As programmers, we try to avoid that. As testers, we look for such examples as easy pickings.

26.3.3.3 Loops

We have looked at loops when we discussed `binary_search()`. Basically most errors occur at the ends:

- Is everything properly initialized when we start the loop?
- Do we correctly end with the last case (often the last element)?

Here is an example where we get it wrong:

```
int do_loop(const vector<int>& v)    // messy function
// undisciplined loop
{
    int i;
    int sum;
    while(i<=vec.size()) sum+=v[i];
    return sum;
}
```

There are three obvious errors. (What are they?) In addition, a good tester will immediately spot the opportunity for an overflow where we are adding to **sum**:



- Many loops involve data and might cause some sort of overflow when they are given large inputs.

A famous and particularly nasty loop error, the buffer overflow, falls into the category that can be caught by systematically asking the two key questions about loops:

```
char buf[MAX];      // fixed-size buffer

char* read_line()  // dangerously sloppy
{
    int i = 0;
    char ch;
    while(cin.get(ch) && ch!='\n') buf[i++] = ch;
    buf[i+1] = 0;
    return buf;
}
```

Of course, *you* wouldn't write something like that! (Why not? What's so wrong with **read_line()**?) However, it is sadly common and comes in many variations, such as

```
// dangerously sloppy:
gets(buf);          // read a line into buf
scanf("%s",buf);   // read a line into buf
```



Look up **gets()** and **scanf()** in your documentation and avoid them like the plague. By “dangerous,” we mean that such buffer overflows are a staple of “cracking” – that is, break-ins – on computers. Many implementations now warn against **gets()** and its cousins for exactly this reason.

26.3.3.4 Branching

Obviously, when we have to make a choice, we may make the wrong choice. This makes **if**-statements and **switch**-statements good targets for testers. There are two major problems to look for:



- Are all possibilities covered?
- Are the right actions associated with the right possibilities?

Consider this nonsense function:

```

void do_branch1(int x, int y)    // messy function
    // undisciplined use of if
{
    if (x<0) {
        if (y<0)
            cout << "very negative\n";
        else
            cout << "somewhat negative\n";
    }
    else if (x>0) {
        if (y<0)
            cout << "very positive\n";
        else
            cout << "somewhat positive\n";
    }
}

```

The most obvious error here is that we “forgot” the case where **x** is 0. When testing against zero (or for positive and negative values), zero is often forgotten or lumped with the wrong case (e.g., considered negative). Also, there is a more subtle (but not uncommon) error lurking here: the actions for **(x>0 && y<0)** and **(x>0 && y>=0)** have “somehow” been reversed. This happens a lot with cut-and-paste editing.

The more complicated the use of **if**-statements is, the more likely such errors become. From a tester’s point of view, we look at such code and try to make sure that every branch is tested. For **do_branch1()** the obvious test set is

```

do_branch1(-1,-1);
do_branch1(-1, 1);
do_branch1(1,-1);
do_branch1(1,1);
do_branch1(-1,0);
do_branch1(0,-1);
do_branch1(1,0);
do_branch1(0,1);
do_branch1(0,0);

```


Basically, that’s the brute-force “try all the alternatives” approach after we noticed that **do_branch1()** tested against 0 using **<** and **>**. To catch the wrong actions for positive values of **x**, we have to combine the calls with their desired output.


Dealing with **switch**-statements is fundamentally similar to dealing with **if**-statements.

```
void do_branch1(int x, int y) // messy function
    // undisciplined use of switch
{
    if (y<0 && y<=3)
        switch (x) {
            case 1:
                cout << "one\n";
                break;
            case 2:
                cout << "two\n";
            case 3:
                cout << "three\n";
        }
}
```

Here we have made four classic mistakes:

- We range checked the wrong variable (**y** instead of **x**).
- We forgot a **break** statement leading to a wrong action for **x==2**.
- We forgot a default case (thinking we had taken care of that with the **if**-statement).
- We wrote **y<0** when we meant to say **0<y**.

 As testers, we always look for unhandled cases. Please note that “just fixing the problem” is not enough. It may reappear when we are not looking. As testers, we want to write tests that systematically catch errors. If we just fixed this simple code, we may very well get our fix wrong so that it either doesn’t solve the problem or introduces new and different errors. The purpose of looking at the code is not really to spot errors (though that’s always useful), but to design a suitable set of tests that will catch all errors (or, more realistically, will catch many errors).

 Note that loops have an implicit “**if**”: they test whether we have reached the end. Thus loops are also branching statements. When we look at programs containing branching, the first question is always, “Have we covered (tested) every branch?” Surprisingly that is not always possible in real code (because in real code, a function is called as needed by other functions and not necessarily in all possible ways). Consequently, a common question for testers is, “What is your code coverage?” and the answer had better be, “We tested most branches,” followed by an explanation of why the remaining branches are hard to reach. 100% coverage is the ideal.

26.3.4 System tests

Testing any significant system is a skilled job. For example, the testing of the computers that control telephone systems takes place in specially constructed rooms with racks full of computers simulating the traffic of tens of thousands of people. Such systems cost millions and are the work of teams of very skilled engineers. After it is deployed, a main telephone switch is supposed to work continuously for 20 years with at most 20 minutes of downtime (for any reason, including power failures, flooding, and earthquakes). We will not go into detail here – it would be easier to teach a physics freshman to calculate course corrections for a Mars probe – but we’ll try to give you some ideas that could be useful for a smaller project or for understanding the testing of a larger system.

First of all, please remember that the purpose of testing is to find errors, especially potentially frequent and potentially serious errors. It is not simply to write and run the largest number of tests. This implies that some understanding of the system being tested is highly desirable. Even more than for unit testing, effective system testing relies on knowledge of the application (domain knowledge). Developing a system takes more than just knowledge of programming language issues and computer science; it requires an understanding of the application areas and of the people who use the applications. This is something we find important for motivating us to work with code: we get to see so many interesting applications and meet interesting people.

For a complete system to be tested, it has to be built out of all of its parts (units). This can take significant time, so many system tests are run just once a day (often at night while the developers are supposed to be asleep) after all unit tests have been done. Regression tests are a key component here. The areas of a program in which we are most likely to find errors are new code and areas of code where errors were found earlier. So running the collection of old tests (the regression tests) is essential; without those a large system will never become stable. We would introduce new bugs as fast as we removed old ones.

Note that we take it for granted that when we fix a few errors, we accidentally introduce a few new ones. We hope the number of new bugs is lower than the number of old ones that we removed, and that the consequences of the new ones are less severe. However, at least until we have rerun our regression tests and added new tests for our new code, we must assume that our system is broken (by our bug fixes).

26.3.5 Finding assumptions that do not hold

The specification of `binary_search` clearly stated that the sequence in which we search must be sorted. That deprived us of many opportunities for sneaky unit tests. But obviously there are opportunities for writing bad code that we have not devised tests to detect (except for the system tests). Can we use our understanding of a system’s “units” (functions, classes, etc.) to devise better tests?

Unfortunately, the simplest answer is no. As pure testers, we cannot change the code, but to detect violations of an interface's requirements (pre-conditions), someone must either check before each call or as part of the implementation of each call (see §5.5). However, if we are testing our own code, we can insert such tests. If we are testers and the people who write the code will listen to us (that's not always the case), we can tell them about the unchecked requirements and have them ensure that they are checked.

Consider again `binary_search`: we couldn't test that the input sequence `[first:last)` really was a sequence and that it was sorted (§26.3.2.2). However, we could write a function that does check:

```
template<class Iter, class T>
bool b2(Iter first, Iter last, const T& value)
{
    // check if [first:last) is a sequence:
    if (last<first) throw Bad_sequence();

    // check if the sequence is ordered:
    if (2<=last-first)
        for (Iter p = first+1; p<last; ++p)
            if (*p<*(p-1)) throw Not_ordered();

    // all's OK, call binary_search:
    return binary_search(first,last,value);
}
```

Now, there are reasons why `binary_search` isn't written with such tests, including these:

- The test for `last<first` can't be done for a forward iterator; for example, the iterator for `std::list` does not have a `<` (§B.3.2). In general, there is no really good way of testing that a pair of iterators defines a sequence (starting to iterate from `first` hoping to meet `last` is not a good idea).
- Scanning the sequence to check that the values are ordered is far more expensive than executing `binary_search` itself (the real purpose of `binary_search` is not to have to blindly walk through the sequence looking for a value the way `std::find` does).

So what could we do? We could replace `binary_search` with `b2` when we are testing (only for calls to `binary_search` with random-access iterators, though).

Alternatively, we could have the implementer of `binary_search` insert code that a tester could enable:

```

template<class Iter, class T>      // warning: contains pseudo code
bool binary_search (Iter first, Iter last, const T& value)
{
    if (test enabled) {
        if (Iter is a random access iterator) {
            // check if [first:last) is a sequence:
            if (last<first) throw Bad_sequence();
        }

        // check if the sequence is ordered:
        if (first!=last) {
            Iter prev = first;
            for (Iter p = ++first; p!=last; ++p, ++ prev)
                if (*p<*prev) throw Not_ordered();
        }
    }

    // now do binary_search
}

```

Since the meaning of `test enabled` depends on how testing of code is arranged (for a specific system in a specific organization), we have left it as pseudo code: when testing your own code, you could simply have a `test_enabled` variable. We also left the `Iter is a random access iterator` test as pseudo code because we haven't explained "iterator traits." Should you really need such a test, look up *iterator traits* in a more advanced C++ textbook.


26.4 Design for testing

When we start writing a program, we know that we would like it to eventually be complete and correct. We also know that to achieve that, we must test it. Consequently, we try to design for correctness and testing from day one. In fact, many good programmers have as their slogan "Test early and often" and don't write any code before they have some idea about how they would go about testing it. Thinking about testing early helps to avoid errors in the first place (as well as helping to find them later). We subscribe to that philosophy. Some programmers even write unit tests before they implement a unit.


The example in §26.3.2.1 and the examples in §26.3.3 illustrate these key notions:

- Use well-defined interfaces so that you can write tests for the use of these interfaces.
- Have a way of representing operations as text so that they can be stored, analyzed, and replayed. This also applies to output operations.
- Embed tests of otherwise unchecked assumptions (assertions) in the calling code to catch bad arguments before system testing.
- Minimize dependencies and keep dependencies explicit.
- Have a clear resource management strategy.


Philosophically, this could be seen as enabling unit-testing techniques for subsystems and complete systems.

 If performance didn't matter, we could leave the test of the (otherwise) unchecked assumptions (requirements, pre-conditions) enabled all the time. However, there are usually reasons why they are not systematically checked. For example, we saw how checking whether a sequence is sorted is both complicated and far more expensive than using `binary_sort`. Consequently, it is a good idea to design a system that allows us to selectively enable and disable such checks. For many systems, it is a good idea to leave a fair number of the cheaper checks enabled even in the final (shipping) version: sometimes “impossible” things happen and we would prefer to know about them from a specific error message rather than from a simple crash.

26.5 Debugging

 Debugging is an issue of technique and attitude. Of these, attitude is the more important. Please revisit Chapter 5. Note how debugging and testing differ. Both catch bugs, but debugging is much more ad hoc and typically concerned with removing known bugs and implementing features. Whatever we can do to make debugging more like testing should be done. It is a slight exaggeration to say that we love testing, but we definitely hate debugging. Good early unit testing and design for testing help minimize debugging.

26.6 Performance

 Having a program correct is not enough for it to be useful. Even assuming that it has sufficient facilities to make it useful, it must also provide appropriate performance. A good program is “efficient enough”; that is, it will run in an acceptable time given the resources available. Note that absolute efficiency is uninteresting,

and an obsession with getting a program to run fast can seriously damage development by complicating code (leading to more bugs and more debugging) and making maintenance (including porting and performance tuning) more difficult and costly.

So, how can we know that a program (or a unit of a program) is “efficient enough”? In the abstract we cannot know, and for many programs the hardware is so fast that the question doesn’t arise. We have seen products shipped that were compiled in debug mode (i.e., running about 25 times slower than necessary) to enable better diagnostics for errors occurring after deployment (this can happen to even the best code when it has to coexist with code developed “elsewhere”).

Consequently, the answer to the “Is it efficient enough?” question is: “Measure how long interesting test cases take.” To do that, you obviously have to know your end users well enough to have an idea of what they would consider “interesting” and how much time such interesting uses can acceptably take. Logically, we simply clock our tests with a stopwatch and check that none consumes an unreasonable amount of time. This becomes practical when we use facilities such as `system_clock` (§26.6.1) to do the timing for us, and we can automatically compare the time taken by tests with estimates of what is reasonable. Alternatively (or additionally) we can record how long tests take and compare them to earlier test runs. This way we get a form of regression test for performance.

Some of the worst performance bugs are caused by poor algorithms and can be found by testing. One reason for testing with large sets of data is to expose inefficient algorithms. As an example, assume that an application has to make sums of the elements in rows of a matrix (using the `Matrix` library from Chapter 24). Someone supplied an appropriate function:

```
double row_sum(Matrix<double,2> m, int n); // sum of elements in m[n]
```

Now someone uses that to generate a `vector` of sums where `v[n]` is the sum of the elements of the first `n` rows:


```
double row_accum(Matrix<double,2> m, int n) // sum of elements in m[0:n]
{
    double s = 0;
    for (int i=0; i<n; ++i) s+=row_sum(m,i);
    return s;
}

// compute accumulated sums of rows of m:
vector<double> v;
for (int i = 0; i<m.dim1(); ++i) v.push_back(row_accum(m,i+1));
```

You can imagine this to be part of a unit test or executed as part of the application exercised by a system test. In either case, you will notice something strange if the matrix ever gets really large: basically, the time needed goes up with the square of the size of **m**. Why? What we did was to add all the elements of the first row, then we added all the elements in the second row (revisiting all the elements of the first row), then we added all the elements in the third row (revisiting all the elements of the first and second rows), etc.

If you think this example was bad, consider what would have happened if the `row_sum()` had had to access a database to get its data. Reading from disk is many thousands of times slower than reading from main memory.

Now, you may complain: “Nobody would write something that stupid!” Sorry, but we have seen much worse, and usually a poor algorithm (from the performance point of view) is not that easy to spot when buried in application code. Did you spot the performance problem when you first glanced at the code? A problem can be quite hard to spot unless you are specifically looking for that particular kind of problem. Here is a simple real-world example found in a server:



```
for (int i=0; i<strlen(s); ++i) {
    // . . . do something with s[i] . . .
}
```

Often, **s** was a string with about 20K characters.

Not all performance problems have to do with poor algorithms. In fact (as we pointed out in §26.3.3), much of the code we write cannot be classified as proper algorithms. Such “non-algorithmic” performance problems typically fall under the broad classification of “poor design.” They include

- Repeated recalculation of information (e.g., the row-summing problem above)
- Repeated checking of the same fact (e.g., checking that an index is in range each time it is used in a loop or checking an argument repeatedly as it is passed unchanged from function to function)
- Repeated visits to the disk (or to the web)

Note the (repeated) *repeated*. Obviously, we mean “unnecessarily repeated,” but the point is that unless you do something many times, it will not have an impact on performance. We are all for thorough checking of function arguments and loop variables, but if we do the same check a million times for the same values, those redundant checks just might hurt performance. If we – by measurement – find that performance is hurt, we will try to see if we can remove a repeated action. Don’t do that unless you are sure that performance is really a problem. Premature optimization is the source of many bugs and much wasted time.

26.6.1 Timing

How do you know if a piece of code is fast enough? How do you know how long an operation takes? Well, in many cases where it matters, you can simply look at a clock (stopwatch, wall clock, or wristwatch). That's not scientific or accurate, but if that's not feasible, you can often conclude that the program was fast enough. It is not good to be obsessed with performance.

If you need to measure smaller increments of time or if you can't sit around with a stopwatch, you need to get your computer to help you; it knows the time and can give it to you. For example, on a Unix system, simply prefixing a command with **time** will make the system print out the time taken. You might use **time** to figure out how long it takes to compile a C++ source file **x.cpp**. Normally, you compile it like this:

```
g++ x.cpp
```

To get that compilation timed, you just add **time**:

```
time g++ x.cpp
```

This will compile **x.cpp** and also print the time taken on the screen. This is a simple and effective way of timing small programs. Remember to always do several timing runs because "other activities" on your machine might interfere. If you get roughly the same answer three times, you can usually trust the result.

But what if you want to measure something that takes just milliseconds? What if you want to do your own, more detailed, measurements of a part of a program? You use standard library facilities from **<chrono>**. For example, to measure the time used by a function **do_something()** you can write code like this:

```
#include <chrono>  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int n = 1000000;           // repeat do_something() n times  
  
    auto t1 = system_clock::now();           // begin time  
  
    for (int i = 0; i<n; i++) do_something(); // timing loop  
}
```

```

auto t2 = system_clock::now();           // end time

cout << "do_something() " << n << " times took "
      << duration_cast<milliseconds>(t2-t1).count() << "milliseconds\n";
}

```

The `system_clock` is one of the standard timers, and `system_clock::now()` returns the point of time (a `time_point`) at which it is called. Subtract two `time_point`s (here, `t2-t1`) and you get a length of time (a `duration`). We can use `auto` to save us from the details of the `duration` and `time_point` types, which are surprisingly complicated if your view of time is simply what you see on a wristwatch. In fact, the standard library's timing facilities were originally designed for advanced physics applications and are far more flexible and general than most users need.

To get a `duration` in terms of a particular unit of time, such as `seconds`, `milliseconds`, or `nanoseconds`, we convert (“cast”) it to that unit using the conversion function `duration_cast`. You need something like `duration_cast` because different systems and different clocks measure time in different units. Don't forget the `.count()`. That is what extracts the number of units (“clock ticks”) from the `duration` that contains both the clock ticks and their unit.

The `system_clock` is meant to measure intervals from a fraction of a second to a few seconds. Don't try to use it to measure hours.



Again, don't believe any time measurement that you cannot repeat with roughly the same result three times. What does “roughly the same” mean? “Within 10%” is a reasonable answer. Remember that modern computers are *fast*: 1,000,000,000 instructions per second is ordinary. This implies that you won't be able to measure anything unless you can repeat it tens of thousands of times or it does something really slow, such as writing to disk or accessing the web. In the latter case, you just have to get it to repeat a few hundred times, but you have to worry that so much is going on that you might not understand the results.

26.7 References

- Stone, Debbie, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann, 2005. ISBN 0120884364.
- Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley, 2003. ISBN 0321194330.



Drill

Get the test of `binary_search` to run:

1. Implement the input operator for `Test` from §26.3.2.2.
2. Complete a file of tests for the sequences from §26.3:
 - a. `{ 1 2 3 5 8 13 21 }` *// an "ordinary sequence"*
 - b. `{ }`
 - c. `{ 1 }`
 - d. `{ 1 2 3 4 }` *// even number of elements*
 - e. `{ 1 2 3 4 5 }` *// odd number of elements*
 - f. `{ 1 1 1 1 1 1 1 }` *// all elements equal*
 - g. `{ 0 1 1 1 1 1 1 1 1 1 1 1 }` *// different element at beginning*
 - h. `{ 0 0 0 0 0 0 0 0 0 0 0 1 }` *// different element at end*
3. Based on §26.3.1.3, complete a program that generates
 - a. A very large sequence (what would you consider very large, and why?)
 - b. Ten sequences with a random number of elements
 - c. Ten sequences with 0, 1, 2 . . . 9 random elements (but still ordered)
4. Repeat these tests for sequences of strings, such as `{ Bohr Darwin Einstein Lavoisier Newton Turing }`.

Review

1. Make a list of applications, each with a brief explanation of the worst thing that can happen if there is a bug; e.g., airplane control – crash: 231 people dead; \$500M equipment loss.
2. Why don't we just prove our programs correct?
3. What's the difference between unit testing and system testing?
4. What is regression testing and why is it important?
5. What is the purpose of testing?
6. Why doesn't `binary_search` just check its requirements?
7. If we can't check for all possible errors, what kinds of errors do we primarily look for?
8. Where are bugs most likely to occur in code manipulating a sequence of elements?
9. Why is it a good idea to test for large values?
10. Why do we often represent tests as data rather than as code?
11. Why and when would we use lots of tests based on random values?

12. Why is it hard to test a program using a GUI?
13. What is needed to test a “unit” in isolation?
14. What is the connection between testability and portability?
15. What makes testing a class harder than testing a function?
16. Why is it important that tests be repeatable?
17. What can a tester do when finding that a “unit” relies on unchecked assumptions (pre-conditions)?
18. What can a designer/implementer do to improve testing?
19. How does testing differ from debugging?
20. When does performance matter?
21. Give two (or more) examples of how to (easily) create bad performance problems.

Terms

assumptions	pre-condition	test coverage
black-box testing	proof	test harness
branching	regression	testing
design for testing	resource usage	timing
inputs	state	unit test
outputs	system_clock	white-box testing
post-condition	system test	

Exercises

1. Run your **binary_search** algorithm from §26.1 with the tests presented in §26.3.2.1.
2. Modify the testing of **binary_search** to deal with arbitrary element types. Then, test it with **string** sequences and floating-point sequences.
3. Repeat exercise 1 with the version of **binary_search** that takes a comparison criterion. Make a list of new opportunities for errors introduced by that extra argument.
4. Devise a format for test data so that you can define a sequence once and run several tests against it.
5. Add a test to the set of **binary_search** tests to try to catch the (unlikely) error of a **binary_search** modifying the sequence.
6. Modify the calculator from Chapter 7 minimally to let it take input from a file and produce output to a file (or use your operating system’s facilities for redirecting I/O). Then devise a reasonably comprehensive test for it.
7. Test the “simple text editor” from §20.6.

8. Add a text-based interface to the graphics interface library from Chapters 12–15. For example, the string `Circle(Point{0,1},15)` should generate a call `Circle(Point{0,1},15)`. Use this text interface to make a “kid’s drawing” of a two-dimensional house with a roof, two windows, and a door.
9. Add a text-based output format for the graphics interface library. For example, when a call `Circle(Point{0,1},15)` is executed, a string like `Circle(Point{0,1},15)` should be produced on an output stream.
10. Use the text-based interface from exercise 9 to write a better test for the graphical interface library.
11. Time the sum example from §26.6 with `m` being square matrices with dimensions 100, 10,000, 1,000,000, and 10,000,000. Use random element values in the range `[-10:10]`. Rewrite the calculation of `v` to use a more efficient (not $O(N^2)$) algorithm and compare the timings.
12. Write a program that generates random floating-point numbers and sort them using `std::sort()`. Measure the time used to sort 500,000 `doubles` and 5,000,000 `doubles`.
13. Repeat the experiment in the previous exercise, but with random strings of lengths in the `[0:100)` range.
14. Repeat the previous exercise, except using a `map` rather than a `vector` so that we don’t need to sort.

Postscript

As programmers, we dream about writing beautiful programs that just work – preferably the first time we try them. The reality is different: it is hard to get programs right, and it is hard to get them to stay right as we (and our colleagues) work to improve them. Testing – including design for testing – is a major way of ensuring that the systems we ship actually work. Whenever we reach the end of a day in our highly technological world, we really ought to give a kind thought to the (often forgotten) testers.