CHAPTER 12

Testing and debugging

Introduction

Two important aspects of software development are testing and debugging. The purpose of testing is to identify any problems before the software is shipped to customers. Software testing is a major aspect of producing quality software. For large software projects, testing and debugging are 40 to 50 percent of the overall project costs. For major software projects, a large software company might employ as many as one or two testers for every development programmer. Debugging is the process of locating and repairing a problem identified by testing or reported by a user. Debugging is a two step process—you identify the problem; and then you fix it. Like testing, debugging, if not done properly, can consume significant time and resources. In this chapter, we discuss the basics of testing software and strategies for debugging once a problem has been recognized.

Key Concepts

- black-box testing
- white-box testing
- inspections
- unit testing
- integration testing
- system testing
- statement coverage
- equivalence partitioning
- regression test
- boundary conditions
- code reviews
- test harness
- path coverage

12.1 TESTING

We have all encountered bugs or problems in programs we have used. If you have ever had your word processor crash after entering a particularly long passage of text, you know how irritating bugs can be. Some bugs are so costly that they make the newspaper headlines. The crash of the Mars Polar Lander on the surface of Mars in 1999 is a recent example. In this incident, a 165 million dollar mission was lost because of a problem that went undetected despite extensive testing. The story of why the Polar Lander crashed illustrates the difficulties of thorough testing.

The landing was supposed to go like this. As the lander entered the atmosphere of Mars, a parachute would deploy to slow the lander's descent. As it neared the surface, the parachute would be discarded, the lander's three legs would snap into position for landing, and the lander's 12 engines would fire to slow the craft to a speed where it could land safely. Each of the lander's three legs had a sensor that would send a signal to the onboard computer to turn off the spacecraft's landing engines when at least one of the legs touched the surface.

Using a similar lander, investigators determined that when the legs were deployed for landing, vibrations could have caused the leg sensors to send spurious signals. In this scenario, the engines would shut down when the craft was about 130 feet high, and the lander would hit the surface at 50 miles per hour.

So how did this problem go undetected? Various postcrash investigations showed that tests of individual systems would not have exposed the problem. One full-scale system test was conducted that should have revealed the presence of the problem. However, the sensors were improperly wired for that test, and the problem went undetected. After the wiring was corrected, a full-scale test was not repeated because of budgetary constraints and time pressures.

The Mars Polar Lander illustrates why thorough testing is so difficult to do. While all the components work correctly when tested individually (unit testing), the system may not work correctly when tested as a whole (system testing). Thus, one must do both thorough unit testing as well as thorough system testing. Because of budgetary constraints and deadlines, however, all too often we convince ourselves that, even though some aspect of the system has changed, further testing is not warranted. Careful programmers retest before delivering software even after the most trivial changes.

After you have written a program, how do you convince yourself that the program works correctly? The not-so careful programmer runs the program with a few test inputs and then checks to see if the answers are correct. For the simplest programs, this may be enough, but this approach is hardly adequate even for a program of moderate complexity, and it surely will not be sufficient for a complex program of more than a 1,000 lines.

In this chapter we will discuss some strategies for testing the software that you design and implement. Unfortunately, a thorough discussion of testing is beyond the scope of this book. There are many excellent texts devoted just to the theory, science, and art of testing software. Section 12.4 lists a few of the

texts that we have found to contain helpful information about testing strategies and procedures.

We should note before proceeding that testing is not a panacea for producing high-quality software. There is a well-known quote by the computer scientist Edgar Dijkstra that gets to the heart of the problem. He observed that "Program testing can be used to show the presence of bugs, but never to show their absence." High-quality software can only be achieved by applying testing along with a number of other software engineering techniques. Informal and formal reviews are necessary. These include formal and informal reviews of the software specification, the proposed design or architecture of the system, as well as the actual code. Indeed, software engineering studies have shown that a disciplined, systematic review process is more effective at avoiding bugs in shipped software than testing. Another important element is the ability to effectively manage and track evolving software. Source-code control systems and software for tracking bugs are commonly used to help automate these tasks. These subjects are typically dealt with in depth in software engineering courses.

12.1.1 Testing—an example

The first thing to realize about bugs and testing is that the earlier problems are found, the better. Software engineering studies have shown that the costs of finding and fixing a problem grow logarithmically with time. For example, a bug found early during the specification phase may cost little or nothing to repair. For the sake of argument, let's say it costs a dollar to fix. That same bug, if discovered during the final testing of the software, may cost hundreds or thousands of dollars to fix.

This means that we should test code as we write it. This approach makes sense from a number of standpoints. Let's say we are designing and coding one particular function that is part of a larger system we are working on. At this point in time, we are enmeshed in the details of the problem. This is a good point to test this module. If a problem is discovered, because of our immediate familiarity of the code, we can most likely fix it quickly. On the other hand, if the problem crops up months later, we will need to refamiliarize ourselves with the code before we can diagnose and fix the problem. Furthermore, it's likely the function or module has grown over time, which again will make finding the bug harder. The process of testing a single module or function is known as *unit testing*.

To illustrate the process of testing, and unit testing in particular, let's begin development of an EzWindow LED timer for displaying elapsed time. LED clocks are used in many consumer electronic devices such as microwave ovens, digital watches, clock radios, and VCRs to display numbers and time (time of day, elapsed time, etc.). An LED timer could be a handy class for building games that have a time limit, developing a computerized scoreboard, and so forth.

Our initial task is to develop the `LED` class that we will need to build the clock. As we will see, unit testing of our `LED` class will help us find any

problems before we tackle larger problems, but it will also help us refine the interface to the object early: before it is used in a larger project and modification becomes more costly.

Following our object-oriented design approach, we must determine the attributes and behaviors of an LED. Obviously we need to control the value displayed. Since we will be using class LED to construct other objects such as timers and clocks, we need to set an LED object's position in the display window. We also must specify the window in which an LED object will be displayed. Our approach for realizing an LED is to use bitmaps of digits to display a number. For example, the image of the bitmap for the number three looks like this:

The CD-ROM accompanying the book contains bitmaps for the numerals zero through nine.

For our initial cut at designing class LED, we have the following attributes.

- MyDigits—an array of bitmaps to hold the images of the digits 0 through 9.
- MyValue—the symbol to display when the LED is displayed.
- MyPosition—the position of the LED in the window.
- MyWindow—the EzWindow where the LED is displayed.

For the public interface, we will need inspectors and mutators for each attribute. In addition we will need a facilitator that displays the LED. Our initial declaration for class LED is

```
class LED {
   public:
      LED();
      // Inspectors
      Position GetPosition() const;
      int GetValue() const;
      // Mutators
      void SetPosition(const Position &p);
      void SetValue(int d);
      void SetWindow(SimpleWindow *W);
      // Facilitators
      void Show();
   private:
      SimpleWindow *MyWindow;
      BitMap MyDigits[MaxElements];
      Position MyPosition;
      int MyValue;
};
```

Now that we have a preliminary class declaration, we can do an implementation. The implementation is straightforward and is given in Listing 12.1. Now we have a choice. We could continue to develop the code for the clock, or we could stop and test class LED to make sure it works properly by doing unit

Listing 12.1
Implementation of class LED

```
#include "led.h"
const int MaxDigits = 10;
char *DigitNames[MaxDigits] = {
   "digit0.bmp",
   "digit1.bmp",
   "digit2.bmp",
   "digit3.bmp",
   "digit4.bmp",
   "digit5.bmp",
   "digit6.bmp",
   "digit7.bmp",
   "digit8.bmp",
   "digit9.bmp",
};

// LED() -- read the bitmaps for [0-9]
LED::LED() {
   for (int D = 0; D < MaxDigits; ++D)
      MyDigits[D].Load(DigitNames[D]);
}

// GetPosition() -- return current position of the LED
Position LED::GetPosition() const {
  return MyPosition;
}

// GetValue() -- return the current value of the LED
int LED::GetValue() const {
  return MyValue;
}

// SetWindow() -- set the window to display the LED
void LED::SetWindow(SimpleWindow *w) {
   MyWindow = w;
}

// SetPosition() -- set the position of the LED
void LED::SetPosition(const Position &p) {
  MyPosition = p;
}

// SetValue() -- set the value of the LED
void LED::SetValue(int v) {
  MyValue = v;
}

// Show() -- display the LED in the window
void LED::Show() {
   MyDigits[MyValue].SetWindow(*MyWindow);
   MyDigits[MyValue].SetPosition(MyPosition);
   MyDigits[MyValue].Draw();
}
```

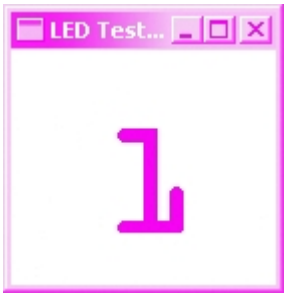
testing. As we mentioned earlier, it's much easier to test and find problems now, while we are familiar with the code, rather than waiting for problems to crop up down the road. However, how do we test our object given that we haven't written the program yet? We must provide a *test harness* or *test stub* to exercise our code. A test harness is a small piece of code written to test or exercise the code being developed.

Writing test harnesses is a standard subtask of unit testing. It is tempting to discard these code fragments after testing is completed, but the modest time and effort it takes to save these stubs is an investment that will pay off later when a bug is uncovered. You will already have a set of test harnesses available to help locate the bug and then ensure that the fix did not break something else. We often create a test directory where the various test harnesses we have written for a project are kept.

The following code is our test harness for our LED class.

```
#include "led.h"
SimpleWindow *W;
LED L;
int ApiMain() {
   W = new SimpleWindow("LED Test Window", 4.0f, 4.0f);
   Position p(1.0, 0.5);
   W->Open();
   L.SetWindow(W);
   L.SetValue(1);
   L.SetPosition(p);
   L.Show();
   return 0;
}
```

When we compile and run this code, the following window appears.



Our code worked! Depending on what we are planning to do, this amount of testing might be enough. However, for code used in a commercial application, we are not even close to being done testing. To thoroughly test this code, we need to think about how the code will be used and what could possibly go wrong. What we need is a systematic approach to unit testing.

First off, we need test cases that demonstrate that the code satisfies its requirements. What are the requirements of class LED? Unfortunately, we did not formally write these down. In a real software project, the first step is to write down formal specifications of what a class is supposed to do. However, we do have an informal idea of the capabilities class LED is supposed to provide since it is to be used to develop clock type objects.

- Class LED should be capable of displaying the digits 0 through 9 in an EzWindow.
- Class LED should be able to be positioned appropriately in an EzWindow.

Using this informal specification, we can design a more comprehensive set of tests. Basically we need to make sure that the class satisfies the stated

requirements. Our test cases should make sure that we can display each digit correctly, and we should also include tests to make sure the positioning of the digits works. With a little work, we can write one test harness that does all this in a single run. After some trial and error to get the positions correct, we have the following test harness that displays each digit.

```
#include "led.h"
SimpleWindow *W;
LED L;
int ApiMain() {
   W = new SimpleWindow("LED Test Window", 8.0f, 4.0f);
   W->Open();

   Position p(1.0, 0.5);
   L.SetWindow(W);
   int i;
   for (i = 0; i < 5; ++i) {
      L.SetValue(i);
      L.SetPosition(p);
      L.Show();
      p = p + Position(1.0, 0.0);
   }
   p = Position(1.0, 2.5);
   for (i = 5; i < 10; ++i) {
      L.SetValue(i);
      L.SetPosition(p);
      L.Show();
      p = p +Position(1.0, 0.0);
   }
   return 0;
}
```

Running the test harness produces the following display.

The output shows that class LED can display all the digits and that positioning of the bitmaps works correctly.

The not-so-careful programmer would be done at this point. However, the careful programmer also tests for what happens when the object is used in a way not permitted by the requirements. One of the benefits of this process is that as you think about unit test cases, bugs are often discovered before you even run the test cases. For example, as we think about test cases to ensure the

object behaves appropriately when illegal values are passed, we immediately realize we have not handled this case at all. Class LED will crash if a value less than 0 and greater than 9 is passed to it. We had better fix that. Again, it's much easier to fix it now rather than later.

To address this issue, we modify member function SetValue() to assert an error if the value passed to it is not in the valid range. The revised member function is:

```
void LED::SetValue(int v) {
  assert(v >= 0 && v <= 9);
  MyValue = v;
}
```

We generate a couple of test cases to make sure that the error detection code works. One test case passes a value less than 0; the other test case passes a value greater than 9. When we test, we run all the tests to make sure that all the test cases still pass.

So far, so good. However, we are still not done. Class LED will be used to build clocks. To test whether class LED will work in this application, we can do a quick prototype of a clock display. Basically, we will produce a static clock face—the time will not change. This activity makes us realize that class LED is missing some features. To display a clock time like 12:30 P or 11:00 A, class LED needs the ability to display a colon and the letters *A* and *P* to denote antimeridian and postmeridian time, respectively.

Again, we revise the implementation of class LED to include these features. As we write the code to prototype a clock face, we discover that it would be convenient in laying out aggregate objects if we could get the length and width of an LED object. This is simple to implement as we can get the length and width of the bitmap used to represent the LED by calling the corresponding BitMap function. This is an example of finding a bug or problem in the interface. Finding these bugs or problems is just as important, if not more important, as finding program errors. Listing 12.2 gives the code for the test harness for the clock prototype.

Listing 12.2
Clock prototype test harness

```
#include "led.h"
SimpleWindow *W;
LED Clock[6];
int ApiMain() {
   W = new SimpleWindow("LED Test Window", 8.0f, 2.5f);
   Position p(0.5, 0.5);
   W->Open();
   int i;
   for (i = 0; i < 6; ++i) {
      Clock[i].SetWindow(W);
      Clock[i].SetPosition(p);
      p = p + Position(Clock[i].GetWidth(), 0.0);
   }
   // Display the time 12:58A
   Clock[0].SetValue(1);
   Clock[1].SetValue(2);
   Clock[2].SetValue(Colon);
   Clock[3].SetValue(5);
   Clock[4].SetValue(8);
```

```
   Clock[5].SetValue(AMIndicator);
   for (i = 0; i < 6; ++i) {
      Clock[i].Show();
   }
   return 0;
}
```

The display produced by the test harness is shown below.

The prototype unit test process makes us realize we have another problem with class LED. How do we display a time like 1:30 P? We need a way to display a blank (i.e., an unlit LED). Again, adding this new capability is easy at this stage as we are very familiar with class LED (maybe too familiar!). After we add the new capability, we add another test program to our growing suite of programs that test the functioning of the new capability. Our latest addition to our suite of test programs is:

```
#include "led.h"
SimpleWindow *W;
LED Clock[6];
int ApiMain() {
   W = new SimpleWindow("LED Test Window", 8.0f, 2.5f);
   Position p(0.5, 0.5);
   W->Open();
   int i;
   for (i = 0; i < 6; ++i) {
      Clock[i].SetWindow(W);
      Clock[i].SetPosition(p);
      p = p + Position(Clock[i].GetWidth(), 0.0);
   }
   // Display the time 2:58A
   Clock[0].SetValue(Space);
   Clock[1].SetValue(2);
   Clock[2].SetValue(Colon);
   Clock[3].SetValue(5);
   Clock[4].SetValue(8);
   Clock[5].SetValue(AMIndicator);
   for (i = 0; i < 6; ++i) {
      Clock[i].Show();
   }
   return 0;
}
```

This test program produces the following display, which verifies that we can produce a blank space.

We should emphasize that each time we make a change to class LED we rerun the entire suite of test programs. This avoids introducing a bug that is discovered only after a series of changes have been made. It is much easier to understand what went wrong when only one set of changes is involved.

After our exercise with testing class LED, you can see why testing is such a time-consuming and expensive part of software development. Running a comprehensive test suite after each change is time-consuming, but there are some things we can do to make it less painful. Programmers typically write "scripts" that automatically run the test suite and report any errors. Using scripts means running the tests is as simple as invoking a command. As new test programs are written, the script is modified so the new test program is included. These scripts are included with the test programs so that in the future other developers know how to run the test programs. Thus, the script serves as documentation for future developers and testers.

Self-check Questions

1. Typically, what percentage of a project is devoted to testing and debugging?

2. Explain the difference between testing and debugging.

3. What is a unit test?

4. What is a test harness?

5. Devise a test harness and test cases for function `CheckWord()` given in Listing 9.8.

6. Revise the test harness that displayed all the numerals to include the blank, the A, the P, and the colon.

12.1.2 Testing fundamentals

As we mentioned earlier, testing is a serious discipline that is a key to producing high-quality, robust software. Thus it should come as no surprise that testing is a well-studied area with its own terminology and research results. As beginning programmers, devoting some time to understanding the fundamentals of testing will pay handsome dividends in the years to come.

The purpose of testing is to find bugs as early as possible in the development process and to make sure they get fixed before the software is shipped. Exactly what is a bug? Certainly, when a program crashes (e.g., blue-screen of death), that's a bug. However, there are many other types of bugs that are just as serious and not so obvious. For example, suppose the user manual for a document editor says that the way to set a word in boldface type is to underline it and then click the bold button on the toolbar. Suppose you do this, and the selected word remains unchanged. Is this a bug? The program didn't crash, it just did not perform as advertised. This is also a bug.

If we practice sound software engineering techniques, then we will write a complete and detailed specification of what the software is supposed to do, how it will operate, the features it will and will not support, and its performance requirements. In general, a bug is when the program does not meet the specification. However, testers and most programmers classify bugs into the following four broad categories:

- software crashes or data corruption,
- does not meet or satisfy the specification,
- poor or unacceptable performance, and
- hard or difficult to use.

A software crash is when a program fails in a noticeable way. Examples of software crashes include when the program exits unexpectedly or it stops responding to commands and has to be manually killed via operating system commands.

Data corruption occurs when a program writes bad data to a file. Suppose you were editing a file with your favorite word processor or editor, and after you saved the file you discovered the file contained gibberish. This is an example of a data corruption bug. Data corruption bugs are particularly insidious because they can easily go undetected. The error can propagate to other data files, and if the error goes undetected for a long period of time, it can be difficult to restore the corrupted files to a correct state.

An important component of a software specification is the features the system will provide. A list of features helps everybody, including the customer, know when the system is functionally complete. For example, the specification for a calculator program may state that the program should provide operations for converting between various number bases. If this feature gets left out or is incomplete (e.g., you can only convert to binary), then there is a bug.

A performance bug is present when the program fails to meet the performance requirements contained in the specification, or the program performs so poorly that it, in effect, does not satisfy the specification. As an example of a

performance bug, suppose your E-mail system includes a feature for searching your archive of saved messages. While the feature works, a search takes so long that you never use the feature. In effect, it is as if the E-mail system did not provide the feature.

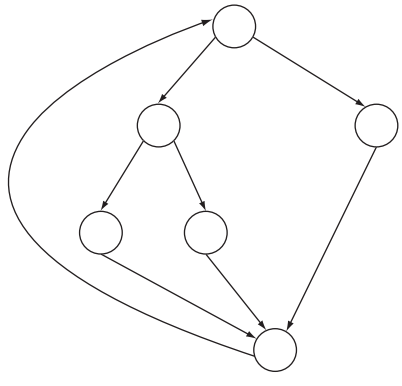
Modern user interfaces have made software easier to use than ever. However, we also expect software to do more and more. Designing software that is easy for people of various skill levels to use is a very difficult task. Indeed, the area of user interfaces and usability are subdisciplines of computer science. If the design of a program makes accomplishing a task overly difficult, this too is a bug—a bug in the design of the user interface. As a side note, this kind of bug can be very expensive to fix as a project is nearing completion. Consequently, it is very important to do early testing of the user interface to ensure that the program is easy to use or, as some people like to say, user friendly.

Interestingly, when you are testing a program, crashes and data corruption are the best kind of bugs to encounter. The behavior is clearly an error, and it is often pretty obvious what has gone wrong. On the other hand, bugs where the program has an odd quirk, is slow, or is difficult to use are much more subjective. A developer may argue that the quirk is not a bug, but a feature; that the program is not that slow; or that the program is really not that hard to use.

It is important to understand the limitations of testing. As we mentioned in the introduction, testing cannot show or prove a program is bug free. It can only show or expose bugs. Furthermore, for all but the most trivial programs it is impossible to completely test a program. The problem is that the number of inputs to most programs is very large, and the number of possible paths through the program is also very large. To simplify the problem, let's just talk about the number of paths in a program. Consider the simple program whose controlflow graph is shown in Figure 12.1.

Figure 12.1

Program controlflow graph



Each circle represents a block of statements. To test this program completely, we must cause the program to execute every possible permutation of statements because any sequence could potentially fail. Disregarding the loop, there are three distinct paths through the program. If the loop executes 20

times, there are 3^{20} different sequences of execution of the statements. That is about 3 billion different test cases! Exhaustively testing all possible executions of a program is impossible.

Clearly we need some good procedures and strategies for testing so that as many bugs as possible are found before the software is delivered, yet testing is done efficiently.

12.1.3 Reviews and inspections

Recall our earlier comment that the sooner you find bugs the better. The goal of design and code reviews is to find bugs even before the code is run. Reviews of a design or code can run the spectrum from an informal meeting where one programmer explains the design or code to another programmer to a rigorous formal process. Whether the review is informal or formal the goal is the same—to identify bugs or problems early in the development process. However, reviews, if done and managed properly, can have other benefits.

A review can be a learning process for all involved. During a review, you may see a particularly elegant or effective design or technique for solving a problem. Similarly, you may see bad design or a poor implementation identified along with an explanation of why it is bad. We can learn from successes as well as mistakes.

Reviews can help entry-level programmers learn the expectations and coding standards of the company. When working on a large project, it is vital that each programmer adheres to the coding standard that has been chosen. A coding standard specifies how the program is to be laid out (i.e., the indentation to use for the various language constructs, expectations for the form and content of comments, constructs that are not permitted, etc.). Uniform use of a coding standard produces code that is more reliable and easier to maintain.

Reviews are also useful for project management. Reviews help managers assess the skills of the project members. These assessments can be used to determine the assignment of tasks to project members and to form effective teams. Reviews can also help management assess the progress of the project. This information can be used to take corrective action such as shifting resources, adding more resources, or allocating more time to certain aspects of the project.

Software engineering studies have shown that reviews are very effective at bug detection—more so than other kinds of testing. A study of a large software organization showed that reviews led to a 14 percent increase in productivity and a 90 percent decrease in defects. Another study found that reviews are at least twice as effective as unit testing.

A review process that has been shown to be very effective is the *inspection*. An inspection is a formal process where the personnel involved are assigned specific roles. Inspections were first employed by IBM in 1976. This pioneering work showed that design and code inspections typically remove 60 percent of the bugs in a product.

One characteristic that distinguishes an inspection from other types of reviews is that an inspection is highly structured. Each person involved in the

inspection is assigned one of four roles. Furthermore, each participant receives training on how inspections are carried out and what his or her duties are. A participant in an inspection serves in one of four roles: moderator, inspector, author, or scribe.

Moderator. The moderator is in charge of running the inspection. The moderator's most important job is ensuring that the inspection proceeds at a reasonable pace. The inspection should be thorough so that as many problems as possible are identified, but it should not drag out so as to be unproductive. A very important aspect of being moderator is making sure that the inspection participants treat each other with respect and courtesy. In addition to running the inspection, the moderator is also responsible for distributing the code or design being reviewed to the inspection participants, scheduling the time and place of the inspection, reporting the inspection results, and making sure any action items generated as a result of the inspection are completed.

Inspector. An inspector, or reviewer, is someone other than the author who has some interest in the design or code (e.g., using the code to build a component, implementing the design, testing, etc.). The job of the inspector is to carefully scrutinize the design or code to find any potential problems. The inspection of the code is done prior to the inspection meeting.

Author. The author of the code or the design plays a minor role in the inspection. If the author has done his or her job, the code will be well-documented, easy to understand, and bug free. If the reviewers detect problems or the code is unclear in certain areas, an action item is generated directing the author to remedy the situation. Sometimes what may be perceived as an error by an inspector might not be an error. In this situation, the author can explain why the code is correct.

Scribe. The role of the scribe is to record all the errors that are detected and keep a list of action items generated.

Interestingly, managers are excluded from inspections. Software inspections are technical reviews with the goal of finding as many problems as possible, as early as possible. The presence of management personnel can change the tenor of the inspection. Inspection participants may become more defensive or less likely to speak up if they feel they are being evaluated.

Another characteristic that distinguishes an inspection from other types of reviews is that it consists of five well-defined phases or steps.

Planning. During the planning step, the portion of code to be inspected is chosen, and the moderator assigns tasks to the inspectors. Inspectors may be assigned different parts of the code to review, or they may be asked to review the code from a certain perspective (e.g., testability, extensibility, performance, etc.). Checklists are created to focus the inspectors' attention on certain areas that are known to be critical or that have caused problems in past projects. The moderator also chooses one of the inspectors to be the presenter. During the inspection step, the presenter will walk through the code, presenting it to the inspection team.

Overview. At the overview the author describes any high-level aspects of the project that may have affected the design or code being reviewed. If all of

the project participants are familiar with these aspects of the project, the overview can be skipped.

Preparation. Working alone, each inspector carefully reviews the code using the supplied checklists as a guide. The inspectors note any problems or deficiencies in the code and come to the inspection meeting prepared to present their results. The inspector chosen as presenter uses the preparation plans to present the code or design during the inspection meeting. Studies of the inspection process have shown that this phase should last no more than a couple of hours. Reading code is hard work, and after two hours inspectors become tired and errors can go undetected.

Inspection meeting. At the inspection meeting, the presenter walks through the code line by line, explaining what the code does. As the presenter reads and explains the code, any problems for that portion of code are identified and discussed. The scribe records all the errors detected and the action items associated with them. The moderator makes sure that the inspection proceeds at a reasonable pace and the inspection stays focused. For example, it is tempting to discuss how a problem might be fixed. This is not the goal of an inspection. Like the preparation phase, the inspection meeting should not last longer than a couple of hours.

Inspection report. After an inspection meeting, the moderator prepares a written report that identifies the work that needs to be done and who is responsible for each task. Depending on the magnitude of changes, an inspection of the revised code may be scheduled. The inspection report may suggest additions or changes to the checklist based on the results of the inspection. This information can improve the effectiveness of subsequent inspections.

Inspections are effective because they provide a structured environment for having the code read and understood. Left to their own devices, most people find reading code rather boring. As you read code, it is easy to slip into a mode in which you are just skimming the code and not fully understanding what the code is doing. Inspections help people read code in a focused, productive way. Inspections are also effective because they provide feedback about common problems. Integrating this information into checklists for future inspections improves the effectiveness of subsequent inspections.

12.1.4 Black-box and white-box testing

Two other testing strategies for delivering robust, high-quality software are block-box and white-box testing. The testing of class `LED` at the beginning of this chapter is an example of white-box testing. The term *white-box testing* indicates that we can "see" or examine the code as we devise our test cases. The term *black-box testing* indicates that we cannot examine the code as we devise test cases. The code is hidden in a black-box we cannot see through.

How can you test code when you cannot see it? Why would you want to test code that you cannot see? There are good answers to both these questions. With black-box testing, although you do not know how the code works, the specification tells you what the code is supposed to do. You can create inputs, get output, and check the results for correctness without having access to the

source code. The answer to the second question is that white-box testing can bias the testing toward finding errors in the code. If the code does not implement the specification, white-box testing is unlikely to expose that type of bug. The advantage to white-box testing is that knowledge of how the code works can help you test more effectively by avoiding redundant test cases.

Because black-box and white-box testing are complementary, both are used on large software projects. Since this text is about programming, we will focus our discussion on white-box testing. However, the techniques we discuss apply to black-box testing as well.

The key to successful, efficient testing is producing good test cases—test cases that are most likely to expose bugs. This task is hard because the input possibilities accepted by a nontrivial program are, for all practical purposes, infinite. Thus we must find a way to reduce the number of possible test cases into a smaller, more manageable set that is still effective at exposing any potential bugs. The process of weeding out unnecessary or redundant test cases is called *equivalence partitioning*.

The basic idea behind equivalence partitioning is that if two inputs test the same portions of code, you only need one of the inputs in your test set. From a testing standpoint, the two inputs are equivalent. For example, suppose you are developing a calculator program. You have just implemented the addition operation and you are developing test cases to make sure addition works properly before implementing other operations. You try the test cases 1 + 2, 2 + 1, 0 + 3, and 0 + 0. The calculator produces the correct sums for these test cases. Do you think it will be useful to add the test case 1 + 3? No, because test case 1 + 3 is in the same equivalence class as 1 + 2. If test case 1 + 2 worked, test case 1 + 3 will work. A good test case to add would be −1 + 3. This test case is in a new equivalence class because the first operand is negative. The self-check exercises ask you to develop additional test cases for the calculator program that are in new equivalence classes.

There are several strategies programmers and testers use to generate effective test cases. One of the most common strategies is boundary testing. The motivation for boundary testing is that program bugs occur most often at boundaries. Furthermore, if the code works properly at the boundaries, it probably works correctly elsewhere. One analogy sometimes used is that if you can walk along an edge of a cliff on a plateau without falling off, you can probably walk in the middle of the plateau. There are several types of boundaries depending on the code. There are loop boundaries—does the loop do the right thing at the beginning and the end; data boundaries—does the code do the right thing when handling data that is at the boundary of allowable values; and capacity boundaries—does the code correctly handle the situation when the array is full and empty.

With white-box testing, we can examine the code to look for boundary conditions. For example, Listing 12.3 contains function `BinarySearch()` introduced in Chapter 9. Initial inspection of this code suggests that the boundary conditions are when the key value being searched for is located at the beginning of the array or at the end of the array. If the code works for those

Listing 12.3
Function BinarySearch()

```
// BinarySearch(): examine sorted list A for Key
int BinarySearch(vector<char> &A, char Key) {
   int left = 0;
   int right = A.size() - 1;
   while (left <= right) {
      int mid = (left + right)/2;
      if (A[mid] == Key)
         return mid;
      else if (A[mid] < Key)
         left = mid + 1;
      else
         right = mid - 1;
   }
   return A.size();
}
```

situations, it will likely work when the key is located somewhere in the middle of vector A.

An example of a capacity boundary condition is whether the code works when the size of vector A is one or zero. We should include tests for these two cases. Test cases like these are testing degenerate situations. Degenerate situations are ones that would not arise in typical use of the code, but if they occur the program should work. Because the loop is controlled by the size of the vector, these tests also serve as loop boundary tests. If vector A is empty, the loop will not be executed at all. Does this code work when vector A has size one? What about when its size is zero?

As another example of boundary testing, consider the code in Listing 12.4 from the stock charting utility of Chapter 6. Examination of the code shows that the code checks (using function `Valid()`) that the low stock price is greater than or equal to zero and that the high price is greater than or equal to the low price. This immediately suggests the following data boundary test cases.

Low Stock Price	High Stock Price
0	0
0	10
0	−1
−1	3
10	8

The first test case checks whether a zero stock price is allowed for both the low and the high price. The second test case checks whether the program handles the case where the low stock price is zero and the high stock price is nonzero. The program should produce a graph for these two test cases. The next three test cases should cause the program to produce an error message. The third test case checks whether the program issues an error message when the low stock price is zero and the high stock price is negative. The fourth test case checks the lower boundary of the low stock price. This test should generate an

Listing 12.4
Code from stock charting program

```
// Valid(): are weekly stock prices sensible
bool Valid(float low, float high) {
   return (0 <= low) && (low <= high);
}
// ReadStockInterval(): read weekly low and high for stock
bool ReadStockInterval(istream &fin,
 const string &FileName, int &Low, int &High, int Week) {
   if (fin == cin)
      cout << "Enter the low and high stock price";
   fin >> Low >> High;
   // if no more data return false
   if (! fin)
      return false;
   // check for valid data
   if (! Valid(Low, High)) {
      cerr << FileName << ": Bad data for week "
       << Week + 1 << endl;
      exit(1);
   }
   return true;
}
```

error. The final test case checks whether the program handles the case where the low stock price is higher than the high stock price.

The test cases can be partitioned into two equivalence classes. The first two test cases are legal input, while the last three are illegal and should cause the program to generate an error message. If the program does not generate an error message, then we have exposed a bug. In general, it is a good idea to classify all test cases as to whether they are valid inputs and the program should produce valid output, or whether they are invalid inputs and the program should generate an error message.

Another approach for generating test cases is to produce a set of test cases that cause each statement in the program to be executed at least once. This is known as *statement* or *code coverage testing*. The basic idea is that unless you have executed every line of code at least once you have not thoroughly tested the code. Of course, statement coverage testing can miss bugs because you may not execute a particular sequence of statements that exposes a bug. Also, for complicated programs, the number of test cases needed to guarantee complete code coverage can be quite high. Nonetheless, code coverage is a technique that is sometimes used. To support code coverage testing, there are software tools available that instrument the code and produce reports that show which program statements have been executed.

There are other techniques for test set generation. One approach is to generate a test set that causes each edge of the program's controlflow graph to be executed. This technique is called *path coverage* or *path testing*. To illustrate path testing, consider the following code fragment.

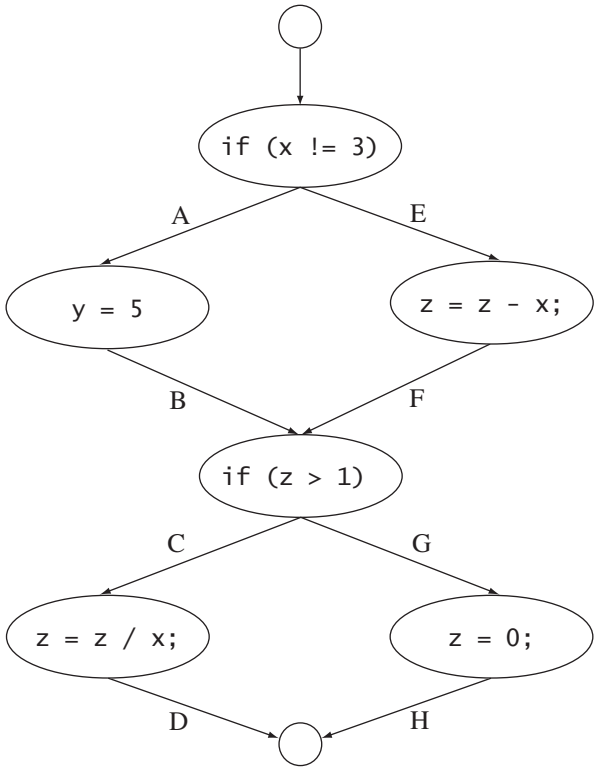
```
if (x != y)
   y = 5;
else
   z = z - z;
```

```
if (x > 1)
   z = z / x;
else
   z = 0;
```

The controlflow graph of this program is shown in Figure 12.2. A set of tests that cause each edge to be traversed is <x = 0, z = 1> and <x = 3, z = 3>. The first test case causes paths A, B, G, H to be executed. The second test case causes paths E, F, C, D to be executed. The problem with this test set is that an important case has been missed. What happens when test case <x = 0, z = 3> is executed? To address this problem, we would need to test every possible path, which we have seen is infeasible.

Figure 12.2

Controlflow graph of two if-else statements



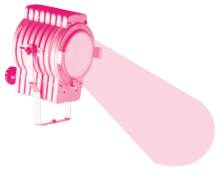
Regardless of the amount of testing and the strategy used to produce the test cases, an important component of testing is automation. As the software is developed, tests will need to be rerun periodically. Thus it is important to set up an automated procedure for running the tests, capturing the output, and comparing the actual output to the expected output. This promotes running *regression tests* periodically. A regression test compares the operation of the new version of the software to the operation of a previous version. The idea is that the behavior of the program should not change in unanticipated ways. If

something that once worked no longer works, you have a regression. Regression testing ensures that you do not introduce new bugs or resurrect old ones!

The other reason to automate testing is that the testing procedure will serve to document how to run the tests. This is helpful when a bug is reported and fixed several years after the software is released. Without an automated testing procedure, you would have to remember how to run all the tests and which were supposed to pass and which were supposed to fail.

In summary, testing software is an integral and key component of the software development process. The size, complexity, and importance of today's software systems demand the application of effective testing techniques.

Programming Tip



Testing tips

The following are some general tips for effective testing.

Test early. The sooner you find bugs, the easier they are to fix. Also, it is easier and more effective to generate tests cases as you develop the code. Unit tests are an effective way to find bugs early in individual components or modules.

Use inspections. Inspections are extremely effective at exposing bugs and other deficiencies in the software. Even if a formal inspection process is not practical, sitting down and explaining your code to another programmer can be productive.

Test boundaries. Look for boundary conditions in your code and make sure you have test cases that test on the boundary and around the boundary. Off-by-one errors are fairly common, so it pays to test for them specifically.

Test exceptional conditions. Try to think of situations that shouldn't happen, and then add tests for them. Typical situations include empty files, no data entered, invalid data, too little data, and too much data. A robust program should handle all these cases.

Make testing repeatable. Set up an automated procedure for running your tests and comparing the actual output to the expected output. Scripting languages and shell languages are useful tools for this purpose.

12.1.5 Integration and system testing

Unit testing focuses on a single function, module, or component. Testing done as the pieces of the software are put together is called *integration testing*. *System testing* is testing done when the whole system is put together. Good unit testing simplifies integration and system testing. Since we are confident that the pieces work, we can focus our efforts on testing the interfaces between the pieces or components. Furthermore since we are confident the individual pieces work, when a test fails we can focus our effort to find the problem on the interfaces between the components.

The guidelines for developing good unit tests apply to integration and system testing. The difference is the focus. With integration testing the focus is on testing the interaction between the software components. Thus, the tests inputs

you develop should focus on exercising that aspect of the system. Similarly with system testing, the test inputs should look to test overall system behavior, not the behavior of an individual component. That was accomplished by unit testing.

Such a modular approach to testing is necessary because as components are assembled to build larger components and the final system, testing the entire system becomes infeasible.

Self-check Questions

7. Name the four roles used in an inspection.

8. With many inspection methodologies, the presenter is someone other than the author of the code. Why is this a good idea?

9. Explain the difference between black-box testing and white-box testing.

10. Devise three new equivalence classes of tests for testing the addition operation of the calculator.

11. What is statement coverage testing?

12. What is path coverage testing?

13. Set up a test harness for the binary search function, and test it thoroughly. Report any errors you found.

14. Devise test cases that cause each statement in the following program to be executed at least once.

```
int Euclid(int x, int y) {
    while (x != y) {
        if (x > y)
            x = x - y;
        else
            x = y - x;
    }
    return 0;
}
```

12.2 DEBUGGING

Testing is the processing of detecting the existence of a bug. *Debugging* is the processing of revealing what the bug is and removing it. Sometimes when a test case exposes a bug, the reason for the bug is obvious. Those are the easy bugs. Other times discovering why a program does not work correctly can be a tedious and time-consuming task—especially if an undisciplined approach is used. In Section 12.2.1, we describe an approach to debugging based on the

scientific method. In Section 12.2.2 we give other advice and tips about debugging that experienced programmers use.

12.2.1 The scientific method

Finding that last elusive bug before an assignment is due or the software is shipped can be a frustrating and stressful experience, sometimes so much so that otherwise bright programmers resort to making random changes to their code in hopes that insight into the problem will emerge. This is not a very productive approach. In this section we introduce the notion of applying the scientific method to debugging.

The scientific method is a systematic way of reaching a conclusion based on inductive logic. The scientific method uses the following steps.

Gather data. Observe facts and look for patterns in the data.

Develop a hypothesis. Formulate a plausible explanation that accounts for or explains the observed facts. This is the hypothesis.

Predict new facts. Using the hypothesis, predict new facts or new behaviors that have not yet been observed.

Perform experiments. Design experiments to observe the new facts. Run the experiments and collect data.

Prove or disprove the hypothesis. If the predicted facts are observed, the hypothesis is assumed to be true. If observations do not support the hypothesis, the process is repeated by developing an alternative hypothesis. Additional data may need to be collected to formulate an alternative hypothesis.

Here's a simple example to illustrate the application of the scientific method to debugging. A program is throwing an exception because of a division by zero in an arithmetic statement. You observe that the value used as the divisor in the offending statement is computed by a loop that counts the number of nonzero values in an array. Based on this observation, you hypothesize that the array must not contain any nonzero values. Using this hypothesis, you predict that if you insert code to print the array right before the loop the output will contain all zeros. Running the modified code is the experiment. If the output shows the array contained only zeros, your hypothesis is true. If the output has nonzero values, the result of the experiment did not support the original hypothesis, and you need to formulate another hypothesis to explain why the divisor is zero.

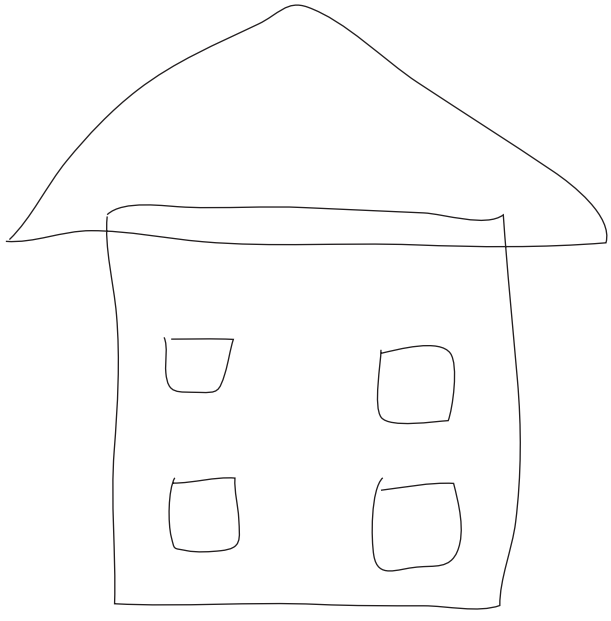
This process sounds time-consuming, but it is really not. Often, the experiment to test the hypothesis can be carried out using a debugger. In the previous example, instead of inserting code and recompiling the program, you could have used the debugger to set a breakpoint before the loop and then print the array.

The important point is that we need to reason about the code and not go willy-nilly changing statements without some clear idea of what we hope to discover.

Here's a real world example to illustrate the power of the scientific method applied to debugging. Sally Code and Chuck Hacker (the names have been changed to protect the innocent) are part of a team of programmers that are cre-

ating a computerized mapping program. The program will read a file containing a list of landmarks with positions and generate a map. Sally and Chuck have been assigned an initial task of creating a house icon for representing houses on the map. Figure 12.3 shows a rough sketch of what they plan to draw. They decide to render the house by drawing a red rectangle and placing four white squares inside the rectangle to represent windows. The roof will be a green triangle. It will be implemented using the EzWindows shapes, `RectangleShape`, `SquareShape`, and `TriangleShape`.

Figure 12.3
Sketch of HouseIcon



Listing 12.5 contains the `HouseIcon` class declaration and Listing 12.6 contains the partial implementation of `HouseIcon`. The implementation is partial as Sally and Chuck plan to add more features after they get this part working. This is a sign they are good programmers—they develop their code incrementally.

Another indication that Sally and Chuck are good programmers is that after they created and implemented the initial version of `HouseIcon`; they also wrote a test harness to make sure class `HouseIcon` works correctly. If there are any problems with `HouseIcon`, debugging now will be easier than later when the implementation is more complex. Listing 12.7 gives the test harness.

Listing 12.5
Definition of HouseIcon

```
#ifndef HOUSEICON_H
#define HOUSEICON_H

#include "ezwin.h"
#include "position.h"
#include "wobject.h"
#include "square.h"
#include "rect.h"
#include "triangle.h"

class HouseIcon : public WindowObject{
   public:
      HouseIcon(SimpleWindow& w, const Position& p);
      void Draw();
   private:
      RectangleShape HouseBase;
      SquareShape Window1, Window2, Window3, Window4;
      TriangleShape Roof;
      color HouseColor;
      Position HouseBasePosition;
      Position Window1Position;
      Position Window2Position;
      Position Window3Position;
      Position Window4Position;
};

#endif
```

Listing 12.6
Partial implementation of HouseIcon

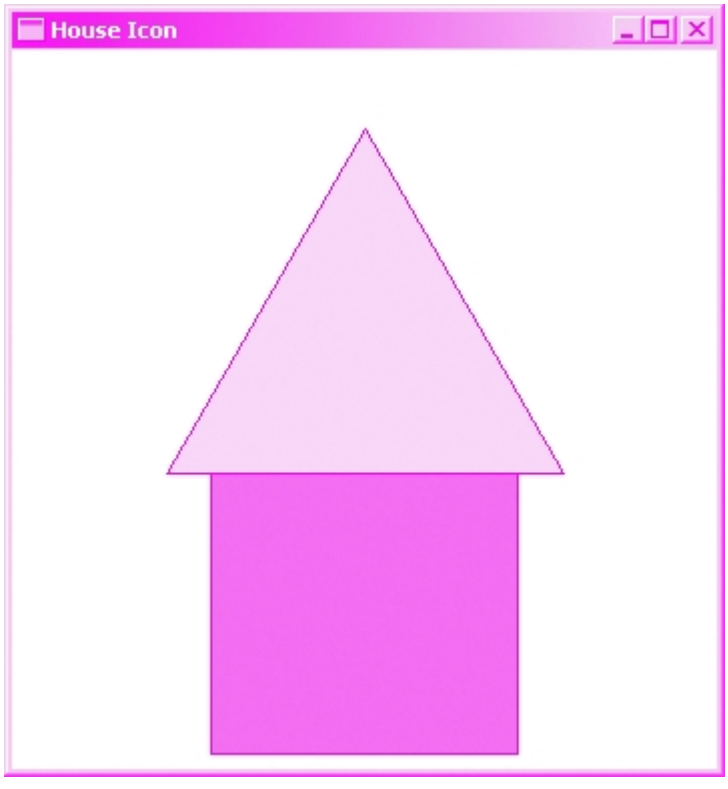
```
#include "HouseIcon.h"
HouseIcon::HouseIcon(SimpleWindow& w, const Position& p) :
   WindowObject(w, p),
   HouseBasePosition(p), HouseBase(w, p, Red, 3.5, 4.0),
   Roof(w, p + Position(0.0, -2.5), Green, 4.5f),
   Window1Position(p + Position(-0.5, -0.5)),
   Window2Position(p + Position(0.5, -0.5)),
   Window3Position(p + Position(-0.5, 0.5)),
   Window4Position(p + Position(0.5, 0.5)),
   Window1(w, Window1Position, White, 0.5),
   Window2(w, Window2Position, White, 0.5),
   Window3(w, Window3Position, White, 0.5),
   Window4(w, Window4Position, White, 0.5),
   HouseColor(Red) {
   // No code needed!
}
void HouseIcon::Draw(){
   HouseBase.Draw();
   Roof.Draw();
   Window1.Draw();
   Window2.Draw();
   Window3.Draw();
   Window4.Draw();
}
```

Listing 12.7
Test harness for HouseIcon

```
#include "HouseIcon.h"
SimpleWindow *W;
int ApiMain() {
   W = new SimpleWindow("House Icon", 10.0f, 8.0f);
   W->Open();
   Position ButtonPosition(5.0f, 6.0f);

   HouseIcon HomeButton(*W, ButtonPosition);
   HomeButton.Draw();
   return 0;
}
```

To their surprise, when Sally and Chuck run their program, they get the following output.



The windows did not appear. It must be a simple error. Indeed, Sally's and Chuck's first thought is that the windows are being drawn behind the red box that is the base of the house. However, inspection of the member function `Draw()` shows that the windows are being drawn last. Sally and Chuck decide to employ the scientific method to find the bug. Chuck comes up with the following hypothesis. He thinks that the house windows are being drawn, but they not being drawn at the right position. He thinks that the house windows are being drawn elsewhere in the display window and they just cannot be seen. He suggests changing the color of the house windows so they will be visible wherever they are drawn. Sally points out that the EzWindows objects have black

borders and that even if the house windows were drawn elsewhere in the display window, they would still be visible. Even though Sally and Chuck did not run the code, they still did an experiment. In this case, Sally and Chuck ran a "thought experiment." Running thought experiments when appropriate is much faster than setting up experiments and running code.

Sally and Chuck need more data to help formulate a plausible hypothesis. Sally suggests printing the locations of the house windows to the console, so they can see where they are supposed to be drawn. They decide to print out the location of just one of the house windows. This should be sufficient since none of the house windows are being displayed. They add the following statement to function `HouseIcon::Draw()`.

```
cout << "Window 1 position is (" << x
   << "," << y << ")" << endl;
```

When they run the program, they get the following output in the console window.

```
Window 1 position is (-1.07374e+008,-1.07374e+008)
```

The *x* and *y* coordinates are out of range. This explains why the house windows did not appear, but now the question is why are the coordinates incorrect.

The positions of the `SquareShape`s used to represent the house windows are set by the `HouseIcon`'s constructor. The code looks correct. Sally and Chuck are stymied. They are out of ideas. This is why debugging can be time-consuming. When you examined the code carefully and still do not see the problem, this usually indicates that your mental model of how the program operates is wrong. In this situation, no amount of looking at the program is likely to help. You are either looking in the wrong place, or you are looking at the right place but just not seeing the problem. There are two things programmers do in this situation. One approach is to explain the problem to someone else. Typically, that person will have a different mindset than you and will quickly point out the error. We often find ourselves saying, "Thanks, I never would have seen that in a million years!"

The other approach is to try and gather more data to expose the flaw in your mental model of the program's operation. In this situation you need detailed data even about that which you think is surely true about the program's operation. To gather this data, programmers use the debugger, or they insert diagnostic statements throughout the program to help them understand the program.

Sally and Chuck insert a `cout` statement in `HouseIcon`'s constructor to display the position of `Window1`. The coordinates are messed up at this point. That's bad news as no other code is being executed between when the coordinates are being set in the data member initialization list and the body of the constructor. At this point inexperienced programmers often jump to the conclusion that something must be wrong with the compiler. This is rarely the case. Furthermore, there's no concrete evidence that something is wrong with the compiler. Sally and Chuck have a real mystery on their hands. As the great detective Sherlock Holmes noted in *The Sign of Four*, "When you have elimi-

nated the impossible, whatever remains, however improbable, must be the truth." The data member initialization list is the only thing left. The problem must be there.

Sally and Chuck use the debugger and set breakpoints in each constructor that involves the house windows—`Position` and `SquareShape`. Perhaps this data will help them see the problem. When the program runs they notice that the constructor for `SquareShape` is called first. That's odd as they had expected the constructor for `Position` to be called first to create the object `Window1Position`. Sally smiles as she realizes what the problem is.

She and Chuck wrote the code based on the assumption that the various constructors were called in the order they appeared on the data member initialization list. In fact, the constructors for the data members are called in the order the objects appear in the class declaration. The object `Window1` is being constructed before `Window1Position` is created and initialized. Hence, the constructor call to create `Window1` is being passed an uninitialized `Position`. The same is true for the `Window2`, `Window3`, and `Window4`.

If Sally's hypothesis is true, one possible fix is to change the order of the private data members in the declaration of class `HouseIcon`. The four house window positions need to appear first. The revised declaration is:

```
class HouseIcon : public WindowObject{
   public:
      HouseIcon(SimpleWindow& w, const Position& p);
      void Draw();
      void MoveAbsolute(const Position& p);
      void MoveRelative(const Position& p);
   private:
      RectangleShape HouseBase;
      color HouseColor;
      TriangleShape Roof;
      Position HouseBasePosition;
      Position Window1Position;
      Position Window2Position;
      Position Window3Position;
      Position Window4Position;
      SquareShape Window1, Window2, Window3, Window4;
};
```

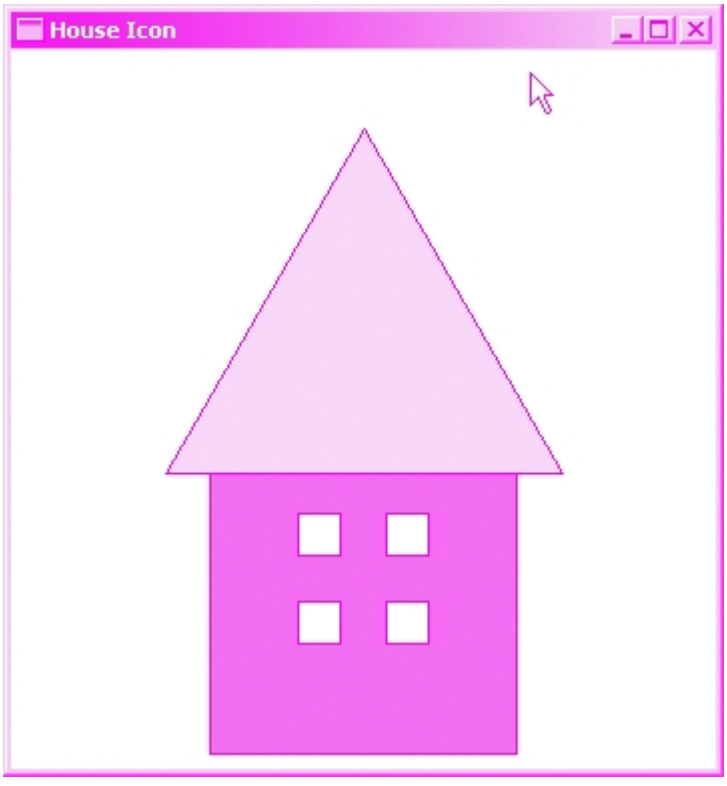
Sally and Chuck run the revised code and the program generates the image shown in Figure 12.4. Finally a house with a view!

This exercise gets Sally and Chuck to thinking about their code. Creating the house should have been simple. What went wrong? Their code depended on an obscure behavior of C++ that is easy to forget or overlook. Chuck wonders what will happen if someone modifies their code down the road. Will the maintainers realize there's a subtle dependency on the order of the data member declarations?

This brings up another important point about debugging. Once you have found a bug, especially one that's been very elusive, it is tempting to go for the quick, easy fix. Good programmers learn from their mistakes. Because they had so much trouble with the code, Sally and Chuck decide to revise `HouseIcon` so that it is less likely to break if someone extends or modifies it.

Figure 12.4

HouseIcon with windows



12.2.2 Debugging tips and techniques

The most powerful weapon in your arsenal against bugs is your ability to reason. There are also some handy techniques and tips that can reduce the time spent debugging. Many of these techniques help you apply the scientific method more efficiently.

Simplify the problem. Try to come up with the smallest amount of code that still exhibits the error. You can do this by removing calls to functions that are unnecessary and removing statements that should have no bearing on the problem. Anything you can do to reduce the complexity of the code you are debugging can be helpful. As you are removing code, you probably want to periodically run the code to make sure the bug is still there. If it goes away, you have discovered valuable information that you can use to reason about the cause of the bug. As you remove code, be sure and retain a copy of the original code.

Along these same lines, you should produce the smallest input that still causes the bug. If the program is interactive, try to find the shortest sequence of commands that cause the error to occur. Knowing the precise input that causes the problem is useful information.

Stabilize the error. Bugs that occur sporadically are some of the hardest to track down. If the bug does not happen consistently, work to make the bug appear reliably. If the bug is a hard one to track down, you will probably be running the program over and over again. In this case, you want each run to be productive.

How you make the bug appear reliably depends on the program; you will have to be resourceful. Here are some common techniques programmers use, depending on the situation.

Figure out the exact sequence of inputs or the precise conditions that cause the bug to occur. If you are using a random number generator, make sure the seed is set to the same value on each run.

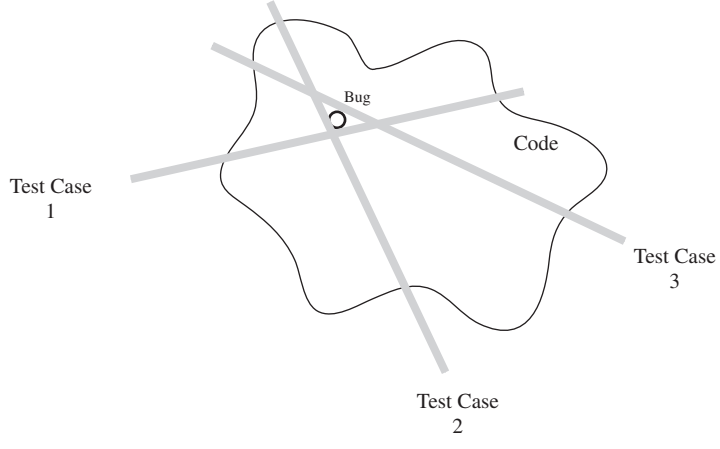
If the bug only occurs after the program has been running a long time, figure out ways to simulate that behavior. Perhaps there's a memory leak, and the bug only occurs after many dynamic memory allocations. Simulate this behavior by writing a function that allocates lots of memory. Call the function at the beginning of the program to simulate the effect of the program running a long time.

Dump the state of the program periodically. That is, print out key values and data structures. Use this last "state" information you've collected to initialize the program so it fails quickly and reliably.

Locate the error. Try to determine what function or section of code is causing the problem. To do this, print data values and observe at what point in the execution incorrect values are produced. It also helps to try different input values and observe the effect on the code. Sometimes with the right inputs you can "triangulate" the location of the error in the code. Figure 12.5 illustrates the process of triangulating the location of a bug.

Figure 12.5

Triangulating the location of a bug



Locating a bug via triangulation often requires running a handful of test cases to narrow the location of the bug down to a region of code small enough to be helpful. The problem is that each test set may identify a large region of

code. Consequently, more than two or three test cases are needed to zero in on the location of the bug.

Explain the bug to someone else. In the `HouseIcon` example, explaining the code to someone else could have caught the error. Hopefully, the person you explain the code to has a different mindset or different mental model of what's going on and will immediately see the problem that you were blind to.

If you have worked at a help desk, you have probably experienced the phenomena known as "confessional debugging." A person is explaining the problem and as they do so, it suddenly dawns on them what the problem is. The act of explaining the code to someone makes you think a little more clearly, not skip steps, and so on. Confessional debugging is surprisingly effective.

Recognize common bugs. Some common bugs have specific symptoms. Knowing the symptoms of commonly occurring bugs can help you identify them quickly.

A common bug is *oversubscripting an array.* When this happens the program writes or reads a memory location that it did not intend to access. Any time you see a program in which an object suddenly has an odd value, look at the source code near where that object was declared. Are there any arrays defined either before or after the object that is being trashed? If so, it may be that you are over- or undersubscripting one of these arrays, and it is stomping on the value in the object.

Trashing the stack is another common bug. The problem is also caused by over- or undersubscripting an array, but occurs when a local array (i.e., one that is allocated on the stack). The symptoms are different from the one described in the preceding paragraph. One symptom of stack trashing is when a function return causes a fault, or a function returns to some odd location (e.g., execution continues in some function that did not call the function that was just executing). The error is caused because a local array was over- or undersubscripted and stack locations were overwritten. One of the values stored on the stack is the address of the function to return to when the current function returns. Basically, you are returning to, as some folks like to say, "never-never land."

A debugger can help identify this problem. The procedure is to set a break point right before the function returns. Dump the runtime stack to verify that the return address is correct. If it is not correct, then the stack was trashed by either this function or some function that was called by this function.

Another symptom of stack trashing is that local objects get strange or incorrect values. Here, the stack was trashed, but the return address was not clobbered—local objects of the calling function were trashed. Again, the debugger can be useful in verifying that this is the problem. Set a breakpoint immediately before and after the call to the function that you suspect is trashing the stack. At the first breakpoint, verify that the values of the local objects that you believe are being trashed are correct. Continue execution. At the second breakpoint reexamine the values. Are they incorrect? If so, you probably have a classic case of stack trashing. If the values are still correct, you need to continue your investigation.

Dereferencing a null pointer is another classic bug. Fortunately, most architectures will cause an exception when a null pointer is dereferenced, and you can easily identify the offending source statement. The real issue, however, is why is the pointer null. Sometimes the pointer is legitimately null, and you just forgot to guard the statement dereferencing the pointer with an if-statement. Other times, you have a live bug that you need to remove. In these cases, using the debugger to collect data so you can use the scientific method of debugging is the best course of action.

Recompile everything. Modern IDEs are extremely useful, but sometimes they can get confused. If you have been making lots of changes to the code looking for a bug and the code is not behaving the way you think it should, or changes you have made do not seem to be having an effect, rebuild the entire project from scratch and rerun the code.

Gather more information. If you are stuck and can't seem to understand what's going on, generate more data. Some standard actions taken by experienced programmers include running different test cases, using the debugger to observe the program's flow of execution, printing out intermediate results, and dumping data structures. You need to be selective and only print as much as you can reasonably digest. Sifting through mountains of data can be like searching for a needle in a haystack.

Pay attention to the compiler. Modern compilers are very good at detecting certain types of errors. However, they tend to be conservative and sometimes produce warning messages that end up being spurious. When you are searching for a bug, it's a good time to pay attention to all those warning messages you have been ignoring. If the compiler reports an object is used before being set, then that problem is something work investigating.

Many compilers allow the programmer to control the level at which error and warning messages are issued. Setting the compiler to the strictest level can sometimes yield information that is helpful in searching for a bug.

Fix bugs as you find them. It often happens that when you are looking for a bug, you discover another bug. This bug seems unrelated to the bug you are trying to find. It is tempting to put off doing something about the bug just discovered since you are in the middle of tracking down the original bug. You don't want the trail to go cold.

Generally, it is good practice to fix bugs as you find them. The bug you just found could be related to the one you were originally trying to find. Fixing this bug, could make the other bug go away. This phenomena happens quite often. On the other hand, if you are sure the newly found bug is unrelated to the bug you were originally searching for, then it may be worthwhile to keep hunting.

Take a break. Sometimes you get nowhere with a bug despite spending hours trying different strategies to locate it. While perseverance is a valuable programmer trait, knowing when to take a break and get away from a problem is also valuable. Experienced problem solvers know how important it is to get away and do something that lets the mind wander—jog, listen to music, take a

walk, stare out the window, and so on. As you relax, the solution to the problem might just come to you.

Most experienced programmers have war stories about bug hunts. A common story is the "debugger's epiphany." The story goes something like this. Bill Geek has been hunting a particularly nasty bug for days. None of the tried-and-true techniques for finding the bug have worked. Bill decides to take a break and hit the hills for some mountain biking action. When Bill is flying down a hill with trees whizzing by, the last thing he is thinking about is the elusive bug in the web browser. After a great afternoon, Bill heads home for a shower. As he showers, he starts to think about the bug again, and it hits him—the solution to the browser problem suddenly becomes crystal clear. Bill heads to work to run an experiment to validate his hypothesis, but he's confident he's got it.

Stories like Bill's are common. Of course, the best stories always have the programmer doing something unusual when the epiphany occurs!

Think outside the box. It is very easy to get locked into a particular way of looking at or attacking a problem. If you are getting nowhere with a problem, try thinking outside the box. Try something unusual or new that you have not tried before. For example, do the opposite of what you have been doing and see what happens.

Debugging is an important part of the programming process. As you gain experience with programming, you will no doubt develop your own personal style of debugging—favorite techniques, tricks, tools, and so forth. You will also learn to use a source-level debugger. A source-level debugger integrated into a modern IDE is a powerful tool for tracking down bugs. However, the most important point to remember is that the key to effective debugging is to follow a disciplined process. Together, disciplined testing and disciplined debugging ensure that high-quality software is delivered on time and within budget.

Self-check Questions

15. Describe the steps of the scientific method.

16. What kind of logic, inductive or deductive, does the scientific method use?

17. Revise class `HouseIcon` so that it is less likely to break if it is changed.

18. Write a program that demonstrates the behavior of a program that oversubscripts a global array.

19. Write a program that demonstrates the behavior of a program that trashes the return address on the stack.

20. Use the Internet to research the Therac-25 accidents. Describe what happened.

12.3 POINTS TO REMEMBER

- ✔ Testing cannot prove that software has no bugs. It can only show the presence of bugs.
- ✔ Bugs fall into four broad categories: software crashes or data corruption, failure to meet or satisfy the specification, poor or unacceptable performance, and difficulty of use.
- ✔ Test early in the development process. It is cheaper and easier to fix bugs when they are identified early.
- ✔ It is impossible to test a program completely.
- ✔ Develop prototypes to test functional requirements.
- ✔ Inspections are an effective way to expose bugs early in the software development process.
- ✔ Testing without knowledge of how the code works is called *black-box testing*.
- ✔ Testing with knowledge of how the code works is called *white-box testing*.
- ✔ Equivalence partitioning helps develop smaller, more effective test suites.
- ✔ Good test suites include inputs that test the software's boundary conditions.
- ✔ Statement coverage testing creates test inputs so that every statement in the software is executed at least once.
- ✔ Path coverage testing creates inputs so that every controlflow edge in the software is executed at least once.
- ✔ Set up automated procedures for running and checking the results of your tests. This will make testing go faster, and it also serves to document the testing process for future developers and maintainers.
- ✔ Use the scientific method for debugging.
- ✔ Take time to fix a bug properly. Resist the temptation to apply a quick fix to get the code running. Time spent fixing a bug properly early in the software development process is an investment that will pay off in the future.
- ✔ When debugging, try to produce the simplest input that causes the problem. Be sure to add the input to your set of test cases.
- ✔ Work to make the error consistently repeatable. This will help you understand the bug, and it will speed the debugging process.
- ✔ Learn to recognize the symptoms of common bugs.

12.4 TO DELVE FURTHER

There are many excellent texts that discuss testing and debugging. The student interested in learning more about testing can begin with the following texts.

- Brian Kernighan and Rob Pike, *The Practice of Programming,* Reading, MA: Addison-Wesley, 1999.
- Steve McConnell, *Code Complete: A Practical Handbook of Software Construction*, Redmond, WA: Microsoft Press, 1993.
- Glenford Myers, *The Art of Software Testing*, New York: Wiley, 1979.
- Ron Patton, *Software Testing,* Indianapolis: Sams Publishing, 2001.

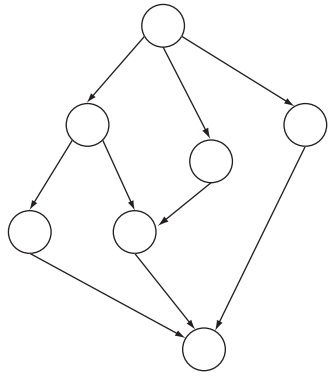
12.5 EXERCISES

12.1 The crash of the Ariane 5 was the result of poor testing practices. Using your library or Internet research facilities, find out about the Ariane 5 crash, and explain how the test procedure was flawed.

12.2 Develop test inputs to test Program 5.1. Your test inputs should achieve full statement coverage of the program. That is, the test inputs should cause every statement in the program to be executed at least once.

12.3 Test data is often partitioned into inputs that test for error conditions and inputs that test that the program works when given valid data. Examine Program 5.3. Develop one set of test inputs for error conditions and a separate set of inputs to demonstrate the program works.

12.4 How many unique paths are there in the following controlflow graph?



12.5 Develop a test harness, and test input for the class `Display` given in Listing 9.18.

12.6 With four other classmates, perform an inspection of the Red-Yellow-Green game presented in Chapter 8. Prepare an inspection report that describes any problems that the inspection discovered.

12.7 Explain why it is a good idea to have someone other than the programmer responsible for testing a program.

12.8 Another kind of testing is acceptance testing. What is acceptance testing?