Errors, Exceptions and Testing

9

In Sect. 1.6 we introduced the concept of an assert statement. This is a way of forcing your program to terminate execution, should something unexpected happen. The program which motivated the use of assertion in Sect. 1.6 was one which calculated the square root of a number entered at the command-line. Here is a version of that program where the assertion has been removed by turning it into a comment.

```
1  #include <iostream>
2  #include <cassert>
3  #include <cmath>
4
5  int main(int argc, char* argv[])
6  {
7     double a;
8     std::cout << "Enter a non-negative number\n";
9     std::cin >> a;
10    //Run without assertion: assert(a >= 0.0);
11    std::cout << "The square root of "<< a;
12    std::cout << " is " << sqrt(a) << "\n";
13    return 0;
14 }
```

What happens when a user ignores the request and enters a *negative* number at the command line? Without the assert statement on line 10 it is likely that the program will complete without error. This is because the computer's floating point unit renders the result of some calculations such as `sqrt(-1.0)` as "not a number" or `nan` for short.

J. Pitt-Francis and J. Whiteley, *Guide to Scientific Computing in C++*, Undergraduate Topics in Computer Science,
https://doi.org/10.1007/978-3-319-73132-2_9

```
Enter a non-negative number
-1
The square root of -1 is -nan
```

Other examples of floating point operations which produce the answer `nan` include `0.0/0.0` and `log(0.0)`. Some calculations such as `1.0/0.0` will resolve to a floating point representation of infinity (`inf`). In a scientific program, once one variable has been set to `nan` or `inf` then this value is likely to propagate to later parts of the calculation. It is normally best to check for this sort of error at the earliest possible stage so that computation is not wasted. In this context, it would be prudent to check in any piece of code that uses division, square root, logarithms etc. that the values of all the arguments are in a sensible range. As we have already seen, assertions are one method of checking such arguments. In this chapter, we will see that exceptions are another method of checking that are more flexible in some ways. We will also introduce techniques for testing software, to allow software to be developed in a sustainable manner with as few errors as possible.

9.1 Preconditions

Every section of a program (where a "section" could be a function, method, block, `for`-loop iteration body etc.) can be thought of as having the task to produce a *postcondition* when given a valid *precondition*. The postcondition of the program above (the thing which it is tasked to do) is that it prints the square root of a given number. It does this subject to the precondition that the number is nonnegative.

Consider a method which finds all the roots of a function $f(x)$ in the half-open range $x_{\min} \leq x < x_{\max}$. This method might need to assume as a precondition that the function f is continuous and differentiable over the same range $x_{\min} \leq x < x_{\max}$. More trivially, it might also need to assume that $x_{\min} < x_{\max}$. What should happen if $x_{\min} > x_{\max}$ or $x_{\min} = x_{\max}$? If the precondition for correct functionality is not met then what should happen? Before we answer this question, we will first consider a specific case.

9.1.1 Example: Two Implementations of a Graphics Function

In a particular graphics library, there is a function for rendering a 2-D annulus. This function takes four input arguments: the inner radius, the outer radius and the number of radial and axial segments. The specification of the library says that the outer radius must be bigger than the inner radius and both should be nonnegative. It also says that the segment numbers must be strictly positive. The specification further says that it is valid to give the inner radius as zero, in which case the annulus will be rendered as a 2-D disk with no hole.

There is a cautionary story about a professor who wrote a program for his students which used this graphics function to draw disks. He misread the specification and set the radius values the wrong way round so that the outer value was 0.0 and the inner value was 1.0. Without realising his mistake, he distributed the program source code to his students, some of whom began to complain that it would not run.

The problem was that the students whose code would not run were using a different implementation of the library. The two different implementations of the same specification were dealing with errors in different ways. The implementation of this function in the graphics library as used by the professor contained a check for his type of error which silently fixed the problem by interchanging values in a manner similar to the code given below.

```
#include <cassert>
void RenderAnnulus(double innerRadius, double outerRadius,
                   int slices, int segments)
{
   //A "helpful" implementation fixes the input
   //so RenderAnnulus(1.0, 0.0, 30, 3); will work
   if (innerRadius > outerRadius)
   {
      //The arguments are the wrong way round
      //Swap them
      double temp = innerRadius;
      innerRadius = outerRadius;
      outerRadius = temp;
   }
   //...then render the annulus
}
```

Meanwhile, the students who complained that the program was not running properly were using a library implementation in which the annulus function terminated on reaching this type of error. The listing below shows that this termination behaviour can easily be implemented by checking the precondition with an assertion.

```
#include <cassert>
void RenderAnnulus(double innerRadius, double outerRadius,
                   int slices, int segments)
{
   //Another implementation only checks the input
   //so RenderAnnulus(1.0, 0.0, 30, 3); trips an assertion
   assert (innerRadius < outerRadius);
   //...then render the annulus
}
```

The “helpful” implementation, as used by the professor, was in reality making a bug in his code invisible—only for it to become embarrassingly visible in the other implementation. Both implementations are *correct* in the sense that they follow the

specification and perform the correct operations provided that the preconditions are met. Unfortunately, the library specification left the handling this kind of error open to interpretation.

9.2 Three Levels of Errors

Some of the most important decisions that a programmer has to make are about how errors should be treated. What should happen if the user misreads a prompt and enters some invalid input? What should happen if the application writer accidentally permutes the input arguments of a library function? What should happen if some numerical scheme has generated `inf` or `nan` because of divergence?

The answer to all these questions is the same: "It depends". It's good to treat errors differently depending on their severity, both in terms of how likely they are to happen and in terms of how easy it might be to fix the problem and carry on. The difficult balance of knowing how severe an error might be is illustrated by the `RenderAnnulus` story in Sect. 9.1.1 where the programmers of different library implementations chose to deal with the same error in completely different ways. One set of programmers decided the error was trivial to fix, while the other set decided to abort the program.

We propose a strategy for handling errors which is built on a framework of three levels of errors.

1. If the error can be fixed safely, then *fix* it. If need be, warn the user.
2. If the error could be caused by some reasonable user input then throw an *exception* up to the calling code, since the calling code should have enough context to fix the problem.
3. If the error should not happen under normal circumstances then trip an *assertion*.

These three basic levels could be further refined. You may distinguish between errors that trip assertions (which are normally removed in optimised code) and errors that should halt the program under all circumstances. At the other end of the scale, you might distinguish between error fixes which are silent and those which should warn the user that something has been changed.

The *exception* level of error is a compromise between patching the problem to carry on, and stopping completely. It is used in circumstances where the caller of a function may have enough information to be able to deal with the error. For example, a nonlinear Newton root finder may diverge and hence signal an error, but the programmer may know that the original task in question can still be solved by calling the same function with a different initial guess, or by calling it with a damping factor, or by calling a bisection root finder. The logic would be to first try the Newton solver, but if that function signalled an error then to find the root using a more expensive bisection routine.

9.3 Introducing the Exception

An exception in C++ is a way of interrupting the normal flow of control of a program and *throwing* a bundle of information back to the calling code. This bundle of information is encapsulated inside an object. We define in this section a class called `Exception`, but objects of any class may be thrown between functions to signal an error.

The use of exceptions requires the keywords `try`, `throw` and `catch`.

- `try` is used in the calling code and tells the program to execute some statements in the knowledge that some error might happen.
- `throw` is used when the error is identified. The function called will encapsulate information about the error into an `Exception` object and throw it back to the caller.
- `catch` is used in the calling code to show how to attempt to fix the error. Every block of code that has the `try` keyword must be matched by a `catch` block.
- Exceptions which are not caught by the calling code may cause the program to halt.

When an error occurs we want the code to "throw" two pieces of information: a one-word summary of the problem type and a more lengthy description of the error. We write a class `Exception` (shown below) to store these two pieces of information, and with the ability to print this information when required.

Listing 9.1 Exception.hpp

```
#ifndef EXCEPTIONDEF
#define EXCEPTIONDEF

#include <string>

class Exception
{
private:
   std::string mTag, mProblem;
public:
   Exception(std::string tagString, std::string probString);
   void PrintDebug() const;
};
#endif //EXCEPTIONDEF
```

Listing 9.2 Exception.cpp

```
#include <iostream>
#include "Exception.hpp"
//Constructor
Exception::Exception(std::string tagString,
                     std::string probString)
```

```
{
   mTag = tagString;
   mProblem = probString;
}

void Exception::PrintDebug() const
{
   std::cerr << "**  Error ("<<mTag<<") **\n";
   std::cerr << "Problem: " << mProblem << "\n\n";
}
```

9.4 Using Exceptions

In Listing 3.4, we read from a named file `Output.dat`. We assumed that this file existed and tripped an assertion if it did not. In the code below, we present a more sophisticated program for opening a file which uses exceptions to attempt to fix the problem. If the file cannot be opened by the `ReadFile` function, an exception is thrown. This is caught by code that prompts the user to enter an alternative file name. Note that `ReadFile` takes the name of the file as a C++ string which is converted to a C string on line 8 (using `c_str` which was introduced in Sect. 1.4.8).

```
1  #include <iostream>
2  #include <fstream>
3  #include "Exception.hpp"
4
5  void ReadFile(const std::string& fileName, double x[],
6                double y[])
7  {
8     std::ifstream read_file(fileName.c_str());
9     if (read_file.is_open() == false)
10    {
11       throw (Exception("FILE", "File can't be opened"));
12    }
13    for (int i=0; i<6; i++)
14    {
15       read_file >> x[i] >> y[i];
16    }
17    read_file.close();
18
19    std::cout << fileName <<" read successfully\n";
20 }
21
22 int main(int argc, char* argv[])
23 {
24    double x[6], y[6];
25    try
```

```
    {
        ReadFile("Output.dat", x, y);
    }
    catch (Exception& error)
    {
        error.PrintDebug();
        std::cout << "Couldn't open Output.dat\n";
        std::cout << "Give alternative location\n";
        std::string file_name;
        std::cin >> file_name;
        ReadFile(file_name, x, y);
    }
}
```

9.5 Testing Software

It is often the case that you need to take a program which has been developed in the past and seek to extend its functionality, perhaps to address some new research question. Assuming that you are able to understand the working of the original code because it is well-documented (as suggested in the tips given in Sect. 5.10) and has a literate coding style (as suggested in the tips given in Sect. 6.6), there is still a potential pitfall. Suppose you add the new functionality, use it to solve your new research problem, but later discover that the original functionality of the code has changed. Perhaps you are no longer able to reproduce the results which are needed for a publication. This pitfall may have been avoided had an appropriate software testing strategy been used for the original code.

For reasons including those given above, it is universally accepted that software should always be tested to give confidence in the output when a code is executed. There is, however, less agreement on how much effort should be put into testing, and on the methodology to be used for testing software. One reason for the absence of a unified view is that the rigour required depends on many characteristics of the software which we now explain with the aid of examples.

Suppose we have a file that contains many 2×2 matrices that are believed to represent rotations, that has been generated from a piece of software. If $\mathbf{Q}$ is one of these matrices, then $\mathbf{Q}$ must be an orthogonal matrix and so we must have $\mathbf{Q}\mathbf{Q}^{\top} = \mathbf{I}$, where $\mathbf{I}$ is the 2×2 identity matrix, and $\det(\mathbf{Q}) = 1$. Suppose further that a colleague wants to use this file, provided he or she can be reasonably certain that the matrices are indeed orthogonal. If we were to allow this colleague to use this file of matrices then we should first check that the matrices really are orthogonal. We may check this by writing a short program that reads these matrices in and checks that they are orthogonal (subject to rounding errors) by printing to screen any warnings that a matrix isn't orthogonal. In this case it can be argued that this rudimentary method for testing the software is appropriate, as we are checking that the file that we share with our colleague does indeed contain matrices that represent rotations.

Nevertheless we should be aware of the limitations of testing software in such an unsophisticated manner. This method does not ensure that the original software is error free; all we have done is to confirm that the given file does indeed contain orthogonal matrices. For example, if the file is believed to represent 1000 distinct matrices we have not checked that there really are 1000 matrices, or that they are distinct—we may only have 500 distinct matrices, or we may have one matrix that has been printed 1000 times. It is also possible that an error exists in the software used to generate the matrices, and that a subsequent execution of the software generates some matrices that are not orthogonal. Many other potential sources of error also exist.

Consider, by contrast, a piece of software containing many lines of code that controls a mechanical ventilator in the clinical setting. It is clearly of critical importance that as many errors as possible are eradicated from the software before it is used, and so much more rigorous testing of the code is required. Furthermore, it is likely that the software may be updated for future generations of ventilators. It is surprisingly easy to break the original functionality of software when making what appears to be a small extension. It is therefore extremely useful to be able to test the whole code after making even a small modification to this code. The basic technique of testing software described above for the file of matrices is not appropriate in this case, and more sophisticated techniques should be used. The testing of *safety-critical software systems* is a research topic in its own right.

The two examples above illustrate that the effort that should be dedicated to testing software depends on many factors. The first, simpler case required nothing more than a short, disposable C++ program that may easily be written by a competent programmer and requires no more discussion. The second case requires far more attention to the testing strategy. We will now describe some common testing strategies.

9.5.1 Unit Testing

An effective technique for testing software, that is particularly useful for software that may be extended in the future, is known as *unit testing*. When using this technique, a collection of tests are written, known as *unit tests*. Each unit test is designed to test a particular section of the code, for example a single method of a class. Each test should then be executed when new functionality is added; should a test fail then it is clear that the new functionality has broken an existing part of the original functionality.

Unit testing is particularly effective when: (i) each unit test covers only a very small number of lines of the original code; and (ii) each line of the original code is covered by at least one test. Whenever we add a small amount of new functionality we can then re-run each test. If we have broken any existing functionality at least one test would hopefully fail (as each line of code is covered by at least one test). Furthermore, as each test covers only a few lines of code, knowing which tests had failed should help us pinpoint the lines of code where the original functionality had been broken.

In the previous paragraph we explained that, when using unit testing, should existing functionality be broken then at least one test will "hopefully" fail. The

reader may expect that, rather than one test hopefully failing, at least one test would *definitely* fail. Unfortunately this assumes that we fully understand the algorithm being used by the software, and have written our unit tests to cover every possible cause of this algorithm failing. For effective unit testing, all possible scenarios must be tested. Suppose, for example, we are writing a graphics application. As part of this application we may want to know where two lines in the (x, y)-plane intersect. This can easily be done by solving two simultaneous equations to calculate the coordinates of the points where the lines meet. We should obviously write a test to check that these coordinates are accurately calculated for two example lines with a unique point of intersection. Despite having written a test that has passed in the example case, there are possibilities where this method does not behave as expected. First, suppose the two lines are identical. They will then intersect at every point. A method written to calculate the intersection of these lines will either fail, or will return one point on the line. A second case is when the lines are parallel, but don't intersect. Any method used to calculate the intersection of these lines would not be able to give a correct answer. To fully test this code we should write tests that cover all possibilities highlighted here. If we don't do this then it is possible that the errors described here may occur, and will propagate into other parts of the code. This may cause other tests to fail, identifying that a problem exists. However the cause of the failing test will not be as clearly located, and may require many tedious and frustrating hours of debugging to pinpoint. We therefore encourage programmers to write tests that cover all possible scenarios.

One highly recommended strategy for writing unit tests is to write the tests for new functionality *before* adding this new functionality. This test will clearly fail initially. All tests—including the new test—are then run when the new functionality has been added, ensuring that both the new functionality has been correctly implemented and that the existing software has not been broken. This method of software development is known as *test driven development*.

Several C++ testing framework libraries exist, such as `CxxTest`, `Boost.Test` and `googletest`. These are designed to help you structure your testing, and we recommend using one of these libraries when writing a suite of tests.

9.5.2 Extending Software

It is very rare that a software package is written from scratch. It is more common for existing software to be extended. For example, you may be expected to extend the functionality of software written by a colleague. Alternatively you may develop software that is underpinned by libraries from external sources. Even if the existing software is believed to be reliable, the user should at least test their own implementation of the functionality offered. This can be done by simply testing all functionality of the software, without understanding the implementation of the functionality—this is known as *black box testing*. This is appropriate for well-supported, mature libraries, that are widely accepted to be robust and reliable. There are, however, potential pitfalls associated with black box testing. Suppose we are using some externally written

software that contains the functionality to solve a linear system. When using black box testing, we would simply check that this functionality works for a given linear system. However, if we had taken a course in linear algebra, we would know that there is no solution to some linear systems, and a non-unique solution to other linear systems. To limit errors from the externally written software propagating into the code we develop, we may want to know how the software handles these cases; this will depend on the implementation of the functionality for these special cases. In these cases we would deliberately test the externally written software by choosing one example linear system with no solution, and one example linear system with a non-unique solution. In this case we may also investigate the algorithm that underpins the functionality of the system, allowing us to understand how the given software handles these systems of equations. This variety of testing is known as *white box testing*.

We now explain how both black box testing, white box testing and test driven development may be carried out. We illustrate the concepts discussed above using the `CxxTest` library, applied to the class of complex numbers developed in Sect. 6.4. We focus on the principles of testing, thus allowing the reader to apply these principles to other testing libraries. As such, we do not focus heavily on the details of using `CxxTest`; a user guide for this library may be found at http://www.cxxtest.com.

9.5.3 Black Box Testing

We illustrate black box testing using the class of complex numbers developed in Sect. 6.4. As explained earlier, when using black box testing we check that the functionality works correctly without inspecting the implementation. In Listing 9.3 we have written a suite of tests for some of the public methods contained in the header file for this class (given in Listing 6.9); we leave the remainder of the black box testing of these public methods as an exercise. As explained earlier, we use the C++ testing framework library `CxxTest` for writing these tests. We reiterate that we are focusing on how suitable tests may be written, rather than explaining how to use the `CxxTest` library. Nevertheless, a few comments on this library are necessary to allow the reader to understand the tests written. First, lines 5–6 and 78 may be considered to be a wrapper that allows us to use the functionality of this library (after it has been installed). Within this wrapper we have written a collection of unit tests: `TestDefaultConstructor` (line 8); `TestCustomisedConstructor` (line 17); `TestCalculatePower` (line 36); and `TestAgainstStdLibrary` (line 61). Within these tests, we test that a floating point variable resulting from a calculation is equal to the true value, subject to ignoring the effects of rounding errors as described in Sect. 2.6.5. If, for example, we were using assertions to check that two double precision variables `x` and `y` differed by less than some value `epsilon`, we would write

```
assert( x-y < epsilon && x-y > -epsilon);
```

or, slightly more compactly

```
assert(fabs(x-y) < epsilon);
```

To write this as a test using the `CxxTest` library, rather than an assert statement, we would use the specially defined `CxxTest` assertion

```
TS_ASSERT_DELTA(x, y, epsilon);
```

which, rather than acting as an assertion, would simply report a failure if `x` and `y` differ by at least some value `epsilon`. Many other test assertions are offered by the `CxxTest` library. When the tests have been written, the library may then be used to generate a test runner that may be compiled so that the tests may be executed. This executable would then report which tests had passed, and which tests had failed. Further details on the features available, and instructions on how to install and use these libraries may be found at http://www.cxxtest.com. We now explain why the tests given in Listing 9.3 are suitable for black box testing of both constructors, and the members `CalculateModulus`, `CalculateArgument` and `CalculatePower`.

We begin by testing the default constructor. This constructor was written with the intention that both the real part and the imaginary part of a complex number created using this constructor should be initialised to zero. A suitable test for this constructor is to check that an instance of a complex number created using this constructor has zero modulus. Clearly this assumes that the method `CalculateModulus` correctly calculates the modulus of this complex number. As such, this test may be considered to also test the method `CalculateModulus`, albeit with a particularly simple input. This test may be found in lines 8–15 of the listing. Line 8 defines a test called `TestDefaultConstructor`. Line 12 then defines an instance of a complex number that is created using the default constructor, and line 13 calculates the modulus of this complex number. Finally, in line 14, we use the function `TS_ASSERT_DELTA` to test that the calculated modulus really is within 10^{-16} of the true value of zero remembering, as discussed in Sect. 2.6.5, that two floating point numbers that should (mathematically) be equal may differ slightly due to rounding errors.

The test between lines 17 and 34 is intended to test the customised constructor. This constructor allows an instance of a complex number to be generated initialising the real and imaginary parts to specified values (lines 21–23). As the real and imaginary parts of the complex number are private members with no methods that allow us to access these members, we may only confirm the real and imaginary parts of the complex number are correctly initialised by confirming that both the modulus (lines 26–28) and the argument (lines 31–33) of the complex number are correct. We note that this test also allows testing of the members `CalculateModulus` and

`CalculateArgument`. It is also worth noting that, as we are treating the class as a black box, we have not copied code from the original class and we are instead calculating the modulus and argument via independent means.

Our next test is to test the member `CalculatePower` (lines 36–60). In this test we use the customised constructor to create a complex number with non-zero real and imaginary parts (lines 40–42), and calculate the modulus and argument of this number (lines 43 and 44). We test `CalculatePower` by raising the original complex number to the power of 2, and calculating the modulus and argument of this squared complex number (lines 48–50). We then use properties of complex numbers to check that the modulus of the squared complex number is correct (lines 54 and 55) and that the argument of the squared complex number is correct (line 59).

Our final test in this section is to test some of our functionality against a trusted complex number class `std::complex` (lines 61–77). The C++ library version of $3 - 4i$, `std_z`, is initialised on line 64. Note that the `std::complex` is templated with a floating point number type in angle brackets. Here we use `double`, to match the type of the private data in our own class, but the class also allows for complex numbers with are stored as `float`. Notice that the syntax of the functions on `std_z` is completely different to our own. Despite this, the mathematical specification is the same and, consequently, we may perform the same tests on them in tandem.

Listing 9.3 Black box testing of the class of complex numbers

```
1  #include <cmath>
2  #include <cxxtest/TestSuite.h>
3  #include "ComplexNumber.hpp"
4
5  class ComplexNumberTestSuite : public CxxTest::TestSuite
6  {
7  public:
8     void TestDefaultConstructor(void)
9     {
10       // Test default constructor sets complex
11       // number to zero
12       ComplexNumber z;
13       double mod_z = z.CalculateModulus();
14       TS_ASSERT_DELTA(mod_z, 0.0, 1.0e-16);
15    }
16
17    void TestCustomisedConstructor(void)
18    {
19       // Use constructor that allows us to specify
20       // real and imaginary parts of a complex number
21       double real = 4.0;
22       double imaginary = -3.0;
23       ComplexNumber z(real, imaginary);
24
25       // Test that modulus is correct
26       double modulus = z.CalculateModulus();
27       double true_modulus = 5.0; // (3,4,5) triangle
```

```
28          TS_ASSERT_DELTA(modulus, true_modulus, 1.0e-8);
29
30          // Test argument is correct via different function
31          double argument = z.CalculateArgument();
32          double true_argument = -asin(3.0/5.0);
33          TS_ASSERT_DELTA(argument, true_argument, 1.0e-8);
34      }
35
36      void TestCalculatePower(void)
37      {
38          // Specify a complex number, z, and calculate the
39          // modulus and argument
40          double real = 4.0;
41          double imaginary = -3.0;
42          ComplexNumber z(real, imaginary);
43          double modulus_z = z.CalculateModulus();
44          double argument_z = z.CalculateArgument();
45
46          // Calculate z*z and calculate the modulus and
47          // argument of z*z
48          ComplexNumber z_squared = z.CalculatePower(2.0);
49          double mod_z_squared = z_squared.CalculateModulus();
50          double arg_z_squared = z_squared.CalculateArgument();
51
52          // Test that:
53          //  modulus of z*z = (modulus of z)*(modulus of z)
54          TS_ASSERT_DELTA(mod_z_squared, modulus_z*modulus_z,
55                          1.0e-8);
56
57          // Test that:
58          //  argument of z*z = 2*(argument of z)
59          TS_ASSERT_DELTA(arg_z_squared, 2.0*argument_z, 1.0e-8);
60      }
61      void TestAgainstStdLibrary()
62      {
63          ComplexNumber z(4.0, -3.0);
64          std::complex<double> std_z(4.0, -3.0);
65          TS_ASSERT_DELTA(z.CalculateArgument(),
66                          arg(std_z), 1e-8);
67          TS_ASSERT_DELTA(z.CalculateModulus(),
68                          abs(std_z), 1e-8);
69          // Raise both numbers to power 5
70          ComplexNumber z_5=z.CalculatePower(5.0);
71          std::complex<double> std_z_5 = pow(std_z, 5.0);
72          // Check they are the same
73          TS_ASSERT_DELTA(z_5.CalculateArgument(),
74                          arg(std_z_5), 1e-8);
75          TS_ASSERT_DELTA(z_5.CalculateModulus(),
76                          abs(std_z_5), 1e-8);
77      }
78  };
```

Using the black box testing above has given us some confidence that the members of the class of complex numbers that have been tested have been implemented correctly. Note, however, that we have only used arbitrary choices to test these members. Were we to consider the implementation of these members we may discover some cases that could give unexpected results. We now discuss such an instance when describing white box testing.

9.5.4 White Box Testing

In the class of complex numbers, the real part and the imaginary part of an instance of a complex number are both private members of this class. This made it difficult to black box test the customised constructor of this class in Sect. 9.5.3, where we create an instance of the class of complex numbers and simultaneously initialise both the real part and imaginary part to specified values. The difficulty arose because, within the test we wrote, we were unable to access the private members of the class, and were therefore unable to test directly that these had been set to the correct values. Instead, we tested these values were correct indirectly by testing that the modulus and the argument of the complex number were correct. This, however, relies on the public methods used to calculate the modulus and the argument of a complex number being correct. Should the test of the customised constructor fail, we would not know whether the test failed because of an error in the customised constructor, or in one of the methods used to calculate the modulus and the argument of a complex number. This may be avoided by white box testing where, in contrast to black box testing, we inspect the implementation of the functionality offered by the class of complex numbers. We simply make the test suite in Listing 9.3 (which is a class) a friend of the class of complex numbers, allowing us to access—and test for correctness—the real and imaginary parts of a complex number. In Listing 9.4 we have given an example white box style test of the default constructor. This test may be used to replace the original test (lines 8–15 in Listing 9.3) provided that the test suite itself is given access to the private members of the complex number class via "`friend class ComplexNumberTestSuite;`".

Listing 9.4 Extract from white box testing of the class of complex numbers

```
8   void TestDefaultConstructorWhiteBox(void)
9   {
10      // Test default constructor sets to zero.
11      // Add to ComplexNumber.hpp :
12      // friend class ComplexNumberTestSuite;
13      ComplexNumber z;
14      TS_ASSERT_DELTA(z.mRealPart, 0.0, 1.0e-16);
15      TS_ASSERT_DELTA(z.mImaginaryPart, 0.0, 1.0e-16);
16  }
```

A second use of white box testing may be illustrated by creating an instance of the class of complex numbers using the default constructor. This default constructor will set both the real and the imaginary part of this complex number to zero. The modulus of this complex number is clearly zero. However, the argument of this complex number is given by `atan2(0.0,0.0)`. Mathematically, this is $\arctan(0/0)$. As $0/0$ is not defined, it is not immediately clear what the result of `atan2(0.0,0.0)` is. To find out, we visit the C++ reference page at http://www.cplusplus.com/reference/cmath/atan2/, where we discover that a *domain error* occurs.[1] This is to be avoided, and so we should update the method `CalculateArgument` given in Listing 6.10 to take account of this special case. An appropriate course of action, that is followed by the scientific computing environment MATLAB, is to set the argument of the complex number zero to 0. We leave the implementation, and testing, of this as an exercise.

9.5.5 Test Driven Development

We have already recommended using test driven development to extend software. When using this technique we first write the tests that are required to test the new functionality, forcing us to be very clear about what we expect our modified software to achieve. These new tests will clearly fail initially, as the new functionality does not yet exist. The new functionality will usually first require some refactoring of existing code, for example modifying an existing constructor to take account of extra data that is now associated with a class to implement the new functionality. If you have a well written and maintained suite of tests you can then run these tests to ensure that you haven't broken any existing functionality. The new functionality is then added, and the tests originally written are run to ensure that the new functionality behaves as expected.

For some applications of complex numbers—for example: the calculation of powers of complex numbers; investigation of the stability of a numerical method for solving initial value ordinary differential equations; and integration of complex numbers around poles—it is convenient to have access to the modulus and argument of a complex number. Rather than calculate these quantities every time they are used by using the methods `CalculateModulus` and `CalculateArgument` that already exist within the class of complex numbers, we could modify the class so that the class contains the private members `mModulus` and `mArgument` to represent these quantities. Should we do this, we would then have to decide whether to introduce the members `mModulus` and `mArgument` instead of the existing members `mRealPart` and `mImaginaryPart`, or in addition to these existing members.

If we modify the class of complex numbers so that we include the private members `mRealPart`, `mImaginaryPart`, `mModulus` and `mArgument` we will have to

[1] If you were to use the C version of the trigonometry functions, rather than the C++ one, then you will find that `atan2(0.0,0.0)` gives no error and is defined to be 0.

modify other methods in the class so that all of these members are specified whenever an operation is performed on an instance of the class. If we decide to only include the members `mModulus` and `mArgument` we will have to modify other methods in the class to specify these members, rather than `mRealPart` and `mImaginaryPart`, whenever an operation is performed on an instance of the class. Whatever choice is made, much of the existing functionality of the class will need to be altered. That is, we will have to refactor the code. This illustrates the importance of having a collection of well written unit tests that each cover a small fraction of the whole functionality. Should any of the existing functionality be broken when the code is refactored, at least one test should fail. The location of the error(s) should then be highlighted.

We leave the implementation of this new functionality as an exercise.

9.6 Tips: Writing Appropriate Tests

In this chapter we have attempted to convince you that an appropriate a collection of unit tests will increase the reliability and longevity of your software. These unit tests should each test a very small part of your code, and each line of software should be covered by at least one test. This testing strategy is, however, underpinned by the assumption that the tests are suitable. The following tips may help you to write appropriate tests, and to get the most out of this technique.

1. Use a C++ testing framework library, such as `CxxTest`, `Boost.Test` or `googletest`. This will help you structure your tests.
2. Add one or more tests for every new piece of functionality, no matter how small the added functionality is.
3. Make tests definitive—they should either pass or fail. However, beware of floating point tolerances and allow for rounding errors in calculations.
4. Remember to write tests for *corner cases*. These are test inputs which may be rare, but might cause problems—collinear triangles, singular matrices, the complex number $0 + 0i$ etc.
5. Rather than spreading test input parameters randomly or evenly, it is more efficient to concentrate on the boundary between types of input. For example, if a test input `p` is supposed to be a probability ($0 \leq \mathtt{p} \leq 1$) then check that $\mathtt{p} = 1$ gives the correct answer, but that $\mathtt{p} = 1.0001$ gives an error.
6. Review your tests from time to time. Add new tests as necessary and remove only those which you know to be redundant.
7. Automate your testing, so that you do not have to remember to run the tests or remember to check the results.

9.7 Exercises

9.1 Extend the `Exception` class given in Listings 9.1 and 9.2 by creating two inherited classes `OutOfRangeException` and `FileNotOpenException`. Each of these two new inherited classes will derive from the `Exception` class in a similar manner to the way the `Ebook` class derived from the `Book` class in Sect. 7.1. The constructors for each of the two classes should take only the `probString` argument to set the `mProblem` member. Each constructor should ensure that the `mTag` member is automatically set in a similar manner to the way the `format` member was set in the constructor of the `Ebook` class. Write a catch block which is able to catch a generic exception but can also differentiate between these two types of error.

9.2 An earlier tip in Sect. 4.3.2 showed how it was possible for bad memory allocation to terminate your program. If you want your program to continue through a memory allocation error there are two ways to cope with the exception: to turn the exception off (and check the value of the pointer) or to catch the exception. Here is some code which demonstrates how to turn off the exception message but still detect bad allocation of memory, without terminating the program.

```
1  double* p_x;
2  p_x = new (std::nothrow) double[1000000000];
3  if (p_x == NULL)
4  {
5     std::cout << "Allocation failed\n";
6  }
7  delete p_x;
```

The proper way to deal with this issue is, of course, to catch the exception. Rewrite the code fragment above so that there is a `try` block around the line of code which attempts to allocate a large vector to `p_x` and demonstrate that you can catch this exception.
[*Hint: The name of the exception class which you need to catch is not* `Exception`. *It is* `std::bad_alloc`.]

9.3 In Exercise 7.3 in Chap. 7, we developed a library for solving initial value ordinary differential equations. Let us suppose that the solution of the ordinary differential equation represents a probability of some event happening as time evolves. The true solution of this equation should therefore be nonnegative, and no greater than one. Of course, due to both rounding errors and errors induced by the numerical approximation used to calculate the numerical solution, this numerical solution may violate these restrictions slightly. In this exercise, we will suggest how to extend the library developed in Sect. 7.3 to handle these requirements in a way that is consistent with the discussion of dealing with errors given in Sect. 9.2.

We will assume that an acceptable value for the absolute error is 10^{-6}. When solving the differential equation, we therefore won't be concerned if the solution for a value of y_i in Exercise 7.3 lies in the interval $-10^{-6} < y_i < 0$. Under these circumstances, we would simply write the value 0.0 to file containing the solution at each time t_i instead of the value y_i. Similarly, if the solution lies in the interval $1 < y_i < 1 + 10^{-6}$ we would write 1.0 to file rather than the value y_i. This is an instance of an error of type #1 in the list given in Sect. 9.2.

Now suppose the value of y_i lies further outside the range of acceptable values than can be attributed to rounding error. The most likely cause of this error is a step size h that is too large. Under these circumstances, an exception should be thrown explaining this. The code that calls the library for solving initial value ordinary differential equations would then know to reduce the step size: a suitable new step size would be half of the step size currently being used. This is an instance of an error of type #2 in the list given in Sect. 9.2.

It is, of course, possible that an error has been made elsewhere in the library or in the code used to call the library. Under these circumstances persisting with making the step size smaller may not solve the problem. We therefore want to terminate the code if the step size h falls below some critical value. This is an instance of an error of type #3 in the list given in Sect. 9.2.

Incorporate the error handling procedure described above into the library for solving initial value ordinary differential equations developed in Exercise 7.3 in Chap. 7. Test this error handling using the example initial value problem

$$\frac{\mathrm{d}y}{\mathrm{d}t} = -100y,$$

with initial condition $y = 0.8$ when $t = 0$, for the time interval $0 < t < 100$. Investigate how different values of the step size h affect the error handling implemented.

9.4 In Sect. 9.5 we discussed how unit tests could be written for the class of complex numbers developed in Sect. 6.4. In this exercise we will complete the set of unit tests that we started in Sect. 9.5.3.

1. Extend the unit tests given in Listing 9.3 so that all the public methods in the class of complex numbers—listed in the header file given in Listing 6.9—are tested using black box testing.
2. The default constructor for the class of complex numbers initialises both the real and imaginary parts of an instance of a complex number to zero. We noted in Sect. 9.5.4 that the method `CalculateArgument`, as implemented in Listing 6.10, will give a domain error when applied to the complex number zero. By using white box testing, as described in Sect. 9.5.4, we suggested a suitable fix for this problem. Implement this fix, and write a test to ensure you have implemented this fix correctly.
3. Suppose we are writing a piece of software for investigating the stability of a given numerical method for solving an initial value system of ordinary differential equations. This software will require us to evaluate polynomial functions of

a given complex number, and to confirm that the modulus of a complex number is less than unity. We decide to implement this additional functionality by first modifying the class of complex numbers so that the class also contains the private members `mModulus` and `mArgument` that represent the modulus and argument of an instance of a complex number. In Sect. 9.5.5 we explained that we could add these members either in addition to the members `mRealPart` and `mImaginaryPart`, or instead of these existing members.

In this exercise you should use test-driven development to implement the new functionality. First decide whether or not to include the existing members `mRealPart` and `mImaginaryPart` in addition to the new members `mModulus` and `mArgument`, and refactor the existing code as necessary. Having done that, introduce new functionality that uses the new members `mModulus` and `mArgument` to evaluate polynomial functions of a given complex number, and to determine whether the modulus of a complex number is less than unity.