Lecture 18 — Testing & Error-Checking

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 27, 2016

Acknowledgments: D.W. Harder, W.D. Bishop

Programming Errors

A programming error is any kind of mistake that occurs during the development and/or maintenance of a computer program.

Even experienced developers will make mistakes. Nobody's perfect.

Errors can be classified into three broad categories:

- Compile-time errors; and
- Run-time errors
- Logic errors

Compile-Time Errors

Compile-Time Errors are:

- Detected when the program is compiled
- Programming errors that prevent compilation of the program
- An incorrect use of the programming language

Examples include:

- Forgetting a semi-colon
- Misspelling a method name
- Using an operand of an inappropriate type
- Having unmatched brackets
- Using a variable not in scope

Run-Time Errors

Run-Time Errors are:

- Programming errors that allow compilation but result in incorrect behaviour when the program is run
- Detected when the program is executing
- Not detectable at compile-time

Examples include:

- Division by zero
- An infinite loop
- Parsing a string input to integer when the user typed in something that's not a valid integer

Logic Errors

Logic Errors are:

- Programming errors that allow compilation and execution, but produce incorrect results
- Detected when the program is executing or after it is finished
- Not detectable at compile-time
- When what you wrote is valid but not what you meant to write

Examples include:

- Accessing index 1 of an array when you wanted index 0.
- Putting in the wrong mathematical formula, such as `a / b + 1` instead of `a / (b+1)`
- Using the wrong algorithm to solve a problem

Finding All Errors

If all errors were detectable at compile-time, could programmers develop error-free programs?

Possibly; an error that is detected in advance is likely to be fixed.
    Assumption: cost of fixing the error less than cost of error's impact.

This is a matter of engineering judgement...

Even in life and safety critical situations, like nuclear weapons, an assessment is done to determine the risks and how to deal with them.
    Such analysis must decide on what an "acceptable" level of risk is.

Finding All Errors

Can we even predict all errors in advance?

Unlikely. There are multiple reasons:

- User input is unpredictable at compile-time.
- Old or damaged hardware can result in strange behaviour.
- The behaviour of complex systems is difficult to model.
- ... and so on. This is not an exhaustive list.

If someone invented a programming language that allowed no possibility of error at run-time, it would probably not be useful.

Analogy: a computer that is unplugged and buried inside 2m of solid concrete is secure, but not useful...

Finding Run-Time Errors

Acknowledging that we cannot find all errors in advance, it's still worthwhile to try to find errors.

If we find errors, we can remove them, and get a little more certainty that the software does what it says it should.

One basic strategy for finding errors is: software testing.

You surely have done some testing already for your assignments.
Any time you provided input and looked at the output was a test.

Software Testing

Software testing is a complex subject that will be covered more next term in ECE 155, and then in the Software Testing course later.

Software testing, put simply, is a process in which we check if a piece of code does what it's supposed to do.

Remember that software takes some inputs and returns outputs.

Thus, testing is:

Provide inputs, examine the output, and check if it's correct.

Testing: Input & Output

Recall the `factorial()` program.

We can test it by providing 5 as an input to the function.

Then examine the output (return value, 120) to see if it's correct.

Critically, to know if the value returned is correct, we have to know what the correct value is independently of what the function returns.

When it's a mathematical problem, it's easy to know what's correct.
Other problem domains may be much more difficult...

Testing Your Code

At this stage, when you write some code, you may test it by trying out a few different inputs to see what the output is.

A pairing of inputs and outputs makes a test case.
Example: input 5, output 120

To check the code you have written, you will execute a set of test cases to check if your code does what it should.

Valid and Invalid Input

All of the preceding discussion assumes that the input is valid.

A major source of programming errors is invalid input.
This could be intentional or unintentional on the part of the user.

To reduce the possibility of negative outcomes from errors, we can implement error-checking in the code.

An implementation of error checking detects invalid input and deals with it in some way.

Why Check for Errors?

It's not strictly necessary to check for errors; we can just ignore them all and forge ahead regardless.

If we ignore an error, however, we are likely to get incorrect output.

Maybe the program keeps running; its state is corrupted and that may manifest in incorrect output or behaviour later.

Some errors cannot be ignored; they will result in the termination of the program. Example: stack overflow in a recursive function.

Ideally, check for errors in advance, such as ensuring that the divisor is not zero before a division operation.

Checking for Errors In Advance

Checking for an error in advance might allow us to deal with something before it becomes a bigger problem.

```
void printGreetings ( int n )
{
    while ( n != 0 )
    {
        cout << "Greetings!" << endl;
        n--;
    }
}
```

If `n` starts as less than 0, this will be an infinite loop.

Reporting the error is much better than the program getting stuck.

Checking the Precondition

The precondition of a function, as you recall, is whatever is assumed to be true when a function is called.

Check these assumptions at the start of the function!

If the precondition is that the value of `n` must be greater than or equal to 1, we can implement a check for that in code:

```
// Precondition: the value of n must be >= 1
int factorial ( int n )
{
     if ( n < 1 )
     {
          // Error has occurred
     }
// Rest of the implementation not shown for space reasons
}
```

Reporting the Error

Detecting a violation of the precondition is not enough on its own; we also have to alert the user that something is wrong.

Consider that in `factorial()` if the input was outside of the valid range (integers >= 1), this is an *exception* to what is expected.

The C++ concept for this is the `assertion`.

Instead of returning a value when an unexpected situation occurs, instead the code uses an `assert` statement.

Make an Assertion

An assertion statement can be very simple: `assert( n >= 1 );`.

If the function is called with a value less than 1, then program execution is immediately terminated with a terse message.

This is good for catching programming errors – but does not provide any opportunity for the program to recover from the error condition.

If we want that, then we need an additional concept: the exception.

Throwing Exceptions

Here are a few reasons why a programmer might throw an exception or make an assertion:

- To provide the programmer with detailed information on the event that caused the error to occur
- To halt program execution at the point at which the error manifests itself
- To prevent further damage to data used by the program

When a program throws an exception, it indicates that it has encountered a situation that it is not immediately ready to handle.

Throwing an Exception

When throwing an exception, we can provide some text to explain what went wrong or why we are throwing the exception (or a numeric error code)

```
// Precondition: the value of n must be >= 1
int factorial ( int n )
{
     if ( n < 1 )
     {
        throw  "n must be >= 1.";
     }
// Rest of the implementation not shown for space reasons
}
```

Exceptions

The keyword `throw` is similar to `return`, although the path back is slightly different.

If there is no code set up to catch an exception that has been thrown, the exception will terminate the program.

In the next lecture, we'll examine in more detail what happens when an exception is thrown.

We will also learn how to handle (catch) an exception, and what to do with one if we do catch it.