Ve 280

Programming and Introductory Data Structures

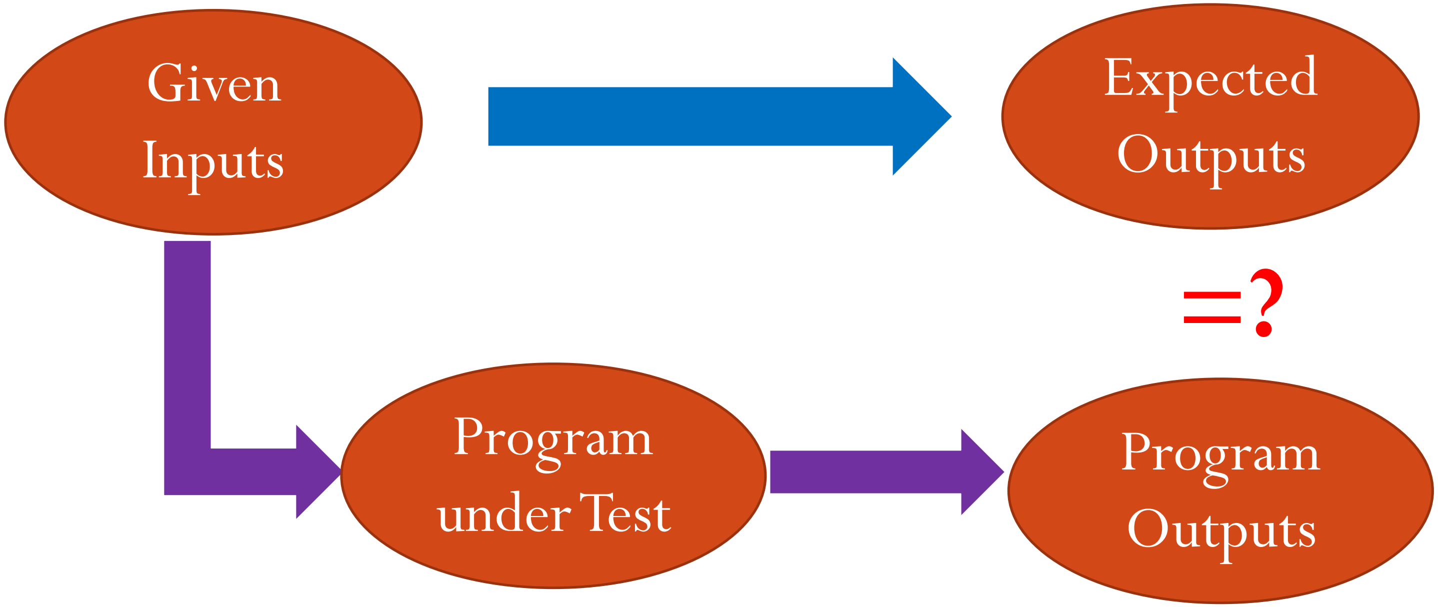
Testing; Exception

Outline

- Testing

- Exception

Testing



Given Inputs
Expected Outputs
=?
Program under Test
Program Outputs

Testing

It's important!

- Be skeptical.

- Typically, the difference between a good and bad score on a project doesn't have much to do with your talent as a programmer. **It has much more to do with your talents as a tester!**

- Testing is not the same as debugging
  - Debugging: **Fixing** something once you know it's broken.
  - Testing: **Discovering** that something is broken.

Testing

It’s important!

- Some tips and truths about being a good tester:

1. Convince yourself that the code is broken.
2. Be in an adversarial frame of mind.
3. NEVER REST and must ALWAYS BE DILIGENT, because the code is NEVER FINISHED!
4. Everyone makes mistakes, and one essential nature of a mistake is that the person who made it didn't realize it was wrong – you thought it was perfect!

Testing

End-to-end vs. incremental testing

- End-to-end testing is not a good idea
  - Errors made early tend to be pervasive and fixing them requires re-writing a large fraction of the existing program
  - Putting off testing until the program is "finished" increases your workload

- Instead, **test individual pieces of your program (such as functions) as you write them**
  - This is **incremental testing**

Incremental Testing

The better type of testing

- There are two advantages of incremental testing:

1. You are testing smaller, less complex, easier to understand units.
2. You just wrote the code, so you have a firm expectation of what it should do. If it's broken, it is fresh in your mind, so you can more easily fix it.

- This will often require you to write extra code (**the driver program**) to test your program effectively. However, this is usually time well spent.

Five Steps in Testing

- To test some piece of code (either a component or a whole piece):

1. Understand the specification
2. Identify the required behaviors
3. Write specific tests
4. Know the answers in advance
5. Include stress tests

Five Steps in Testing

1. Understand the specification

- For an entire assignment, read through the specification very carefully, and make a note of everything it says you have to do – and stay away from the computer ☺
- Since you have to break down the solution into (smaller) constituent parts, you must write specifications for these parts.
- Sometimes your program as a whole may not work correctly, because you misunderstand the specification.

Five Steps in Testing

2. Identify the required behaviors

- For any specification, boil the specification down to a list of things that must happen.
- These are the "**required behaviors**" and a correct implementation **must exhibit all of them**.

Example: you are asked to write a command-line program called `fact` which takes one argument and calculates the factorial of the argument

Required behaviors

- If there is no argument, output "missing argument"
- If there is more than one argument, just work on the first, ignoring the remaining
- If the argument is not an integer, report "non-integral value"
- If it is a negative integer, report "negative integer"
- If it is 0, output 1
- If it is positive integer n, output n!

Five Steps in Testing

3. Write specific tests

- For each of your required behaviors, write one or more test cases that check them.

- To the extent possible, the test case should check **exactly** one behavior – no more!
  - That way, if the case fails, you know where to start looking.

Five Steps in Testing

3. Write specific tests

- There are three classes of test cases that make sense:
  - **Simple inputs**
  - **Boundary conditions**
  - **Nonsense**
- Simple cases are those that are "normal" for the problem at hand.
- "Boundary" cases are at the edges of what is expected, or formed to exploit some detail of implementation.
- "Nonsense" cases are those that are clearly unexpected.

Example: Testing Factorial Function

Assume use `cin` to get the input

- Simple inputs
  - An integer >= 1
- Boundary conditions
  - Value 0
- Nonsense
  - Negative values or non-integer values

Five Steps in Testing

3. Write specific tests: Exercise

- What are examples of the cases for testing the power number?
  - An integer is called a <u>power number</u> if it equals m^n, where $m \geq 1$ and $n \geq 2$ are both integers.

- Simple inputs:
- Boundary conditions:
- Nonsense:

Five Steps in Testing

4. Know the answers in advance

- Instead of quickly running test cases and glancing at the output:
  - First write down what you expect to be a correct answer.
- If the result differs in **any** way from what you expected, try to figure out why.
- It's possible that your **expectation** had been wrong…or your **implementation**.
- However, doing this ABSOLUTELY REQUIRES that you understand the specification.
  - If you don't, you will create an incorrect solution that satisfies your incorrect expectation!

Five Steps in Testing

5. Include stress tests

- Once you've tested each individual behavior, it's time to test all of them in concert.
- For this, you want **large** and **long running** test cases.
  - They must be **large**, to exercise resource limits in your program.
    - E.g., some web applications need to be tested under a large amount of simultaneous accesses.
  - They must be **long running**, because some errors are the result of lots of little bugs that individually don't matter much, but as they cascade produces catastrophic results.
    - E.g., the accumulation of the round-off error
    - E.g., the memory leakage

Testing

The joys of automation

- As you develop test cases for some code, it pays to write **other** programs that **automatically** test the code using those test cases.

```
for each test case ti {
  run your program on ti
  compare output with expected output
}
```

- This is important because, as the number of test cases grows (and the hour grows late) people get tired, and start to make mistakes.
- Computers, however, never get tired, so take advantage of this.

Testing

The joys of automation

- Once you have your test programs, every time you change even the smallest part of your code, you can go back and test all of the behaviors. This is also referred to as **regression testing.**

General Debugging Techniques

- Using `cout`
- Using a debugger, such as GDB
- Using the `assert` function
  - The `assert` function is a special function, defined in `<cassert>`, which takes a Boolean argument.
  - If the argument is **true**, `assert()` does nothing.
  - If the argument is **false**, `assert()` causes your program to stop, printing an **error message** to the `cerr` stream.

Using Assert Function

- `#include <cassert>`
- `assert` for the condition that should hold.
  - Example: In testing function `int min(int a, int b)`, assert that the return value is the smaller one.

```
int smaller = min(a, b);
assert(smaller <= a && smaller <= b);
```

Can you improve this?

Disable Assert

- Note that things to be asserted might be expensive.
  - `assert(very_expensive_func());`
- If it is, you can disable it, by compiling with the NDEBUG preprocessor variable.
- There are two ways to do this:
  1. Define it **before** including `<cassert>`:

     ```
     #define NDEBUG   // disable assert()
     #include <cassert>
     ```

  2. Specify it on the command line of the compiler:

     ```
     g++ -DNDEBUG ...
     ```

     -DMARCO: Define a MARCO for you code

     Same as putting "#define MARCO" in your code
- This way, you can turn it off for "production" code, but leave it in during development and testing.

Outline

- Testing
- Exception

Exceptions
Motivation

- We want a means of **recognizing** and **handling** unusual conditions in the program at runtime
  - E.g., the program opens a file that does not exist!

- Another example: **partial functions**.
  - A function that does not produce meaningful results for **all possible** values of its input type.
  - One particular way of preventing a partial function from receiving invalid inputs: **the REQUIRES specification**.
  - However, a REQUIRES clause is just a comment and cannot **enforce** the specification…

Exceptions

Motivation

- Instead of the REQUIRES clause, there is another way of ensuring correct inputs: **runtime checking**.
  - The idea is to check the inputs **explicitly** before using them in our program.

- One nice things about REQUIRES, is that we don't have to figure out what constitutes "bad" input.

- For runtime checking, we do… ☹

Exceptions

Determining legitimate output for illegitimate input

- There are three general strategies for determining **legitimate output** for **illegitimate input**:

1. "It's my problem!"
   - Try to "fix" things and continue execution by "coercing" legitimate inputs from illegitimate ones by
     - either <u>modifying the inputs</u>
     - or <u>returning default outputs</u> that make sense in the context
   - For example, `list_rest()` could return an empty list if input is an empty list.
   - Such behavior must be explained in the specification!

Exceptions

Determining legitimate output for illegitimate input

- There are three general strategies for determining **legitimate output** for **illegitimate input**:

1. "It's my problem!"

   - However, this strategy fails whenever there is no "default" behavior for the function with the given illegal inputs.
   - For example, what is division over 0?
     - Division over 0 is simply undefined, and trying to define it changes the rules of math.

Exceptions

Determining legitimate output for illegitimate input

- There are three general strategies for determining **legitimate output** for **illegitimate input**:

2. "I Give up!"

    - Use something like `assert()`.
    - `assert(condition)` terminates the program if condition is not true.

```
list_rest (list_t l)
// REQUIRES: list is not empty
{
     assert(!list_isEmpty(l));
}
```

Exceptions

Determining legitimate output for illegitimate input

- There are three general strategies for determining **legitimate output** for **illegitimate input**:

2. "I Give up!"

  - However, it is Not Nice to terminate a program this way.
  - There are some situations where this type of "hard exit" is ok, but there is usually some more things to do before terminating.
    - For example, free the allocated memory.
  - Usually, exiting from a function deep in the call stack is not the way to do it.

Exceptions

Determining legitimate output for illegitimate input

- There are three general strategies for determining **legitimate output** for **illegitimate input**:

3. "It's <u>your</u> problem!"

   The caller of the function

   - Encode "failure" in the return values.
   - Unfortunately, you often can't encode "failure" elegantly in the return values.
   - For example, `list_first()` can return **any** integer, so no special value is available to encode "the list is empty!".
   - Compared to the other two, this is usually the strategy that you use.

Exceptions

It's your problem!

- To fully implement this strategy for runtime checking,

  - Every writer of **every function** must:

  1. Be diligent in checking for illegitimate inputs.
  2. Make sure to pass back the proper encoded "failure" return values.

  - Every writer of **every call** to one of these functions must:

  1. Be diligent in examining these returned values.
  2. Be diligent in acting on these returned values.

Exceptions

It's your problem!

- In practice, this strategy is unworkable for several reasons:

1. You get lazy.

   - You say to yourself, "This kind of error cannot **possibly** occur here, so I'll just omit this check for it."
   - Others may get lazy and not want to check for your return values.

Exceptions

It's your problem!

- In practice, this strategy is unworkable for several reasons:

2. You **forget** to check.

   - For example, if `foo` calls `bar`, `bar` calls `baz`, and `baz` returns an error; `bar` will probably notice, but `bar` has to remember to pass this to `foo`!

Exceptions

It's your problem!

- In practice, this strategy is unworkable for several reasons:

3. It gets unwieldy.

  - If you are ruthlessly diligent about it, your code becomes unmanageable.
  - You have to write too much error handling code, and it becomes hopelessly intertwined with the "normal-case" code.
  - In other words, this doesn't scale well.

- So, we need some mechanism to help deal with these runtime errors…

References

- **Problem Solving with C++ (8th Edition)**, by *Walter Savitch*, Addison Wesley Publishing (2011)
  - Chapter 16 Exception Handling