CS 6015: Software Engineering

Spring 2024

Lecture 7: Defensive programming

Last Week

- How to design a program in C++
- Debugging
- Lab 2 ?? (did you read how/where to submit?)

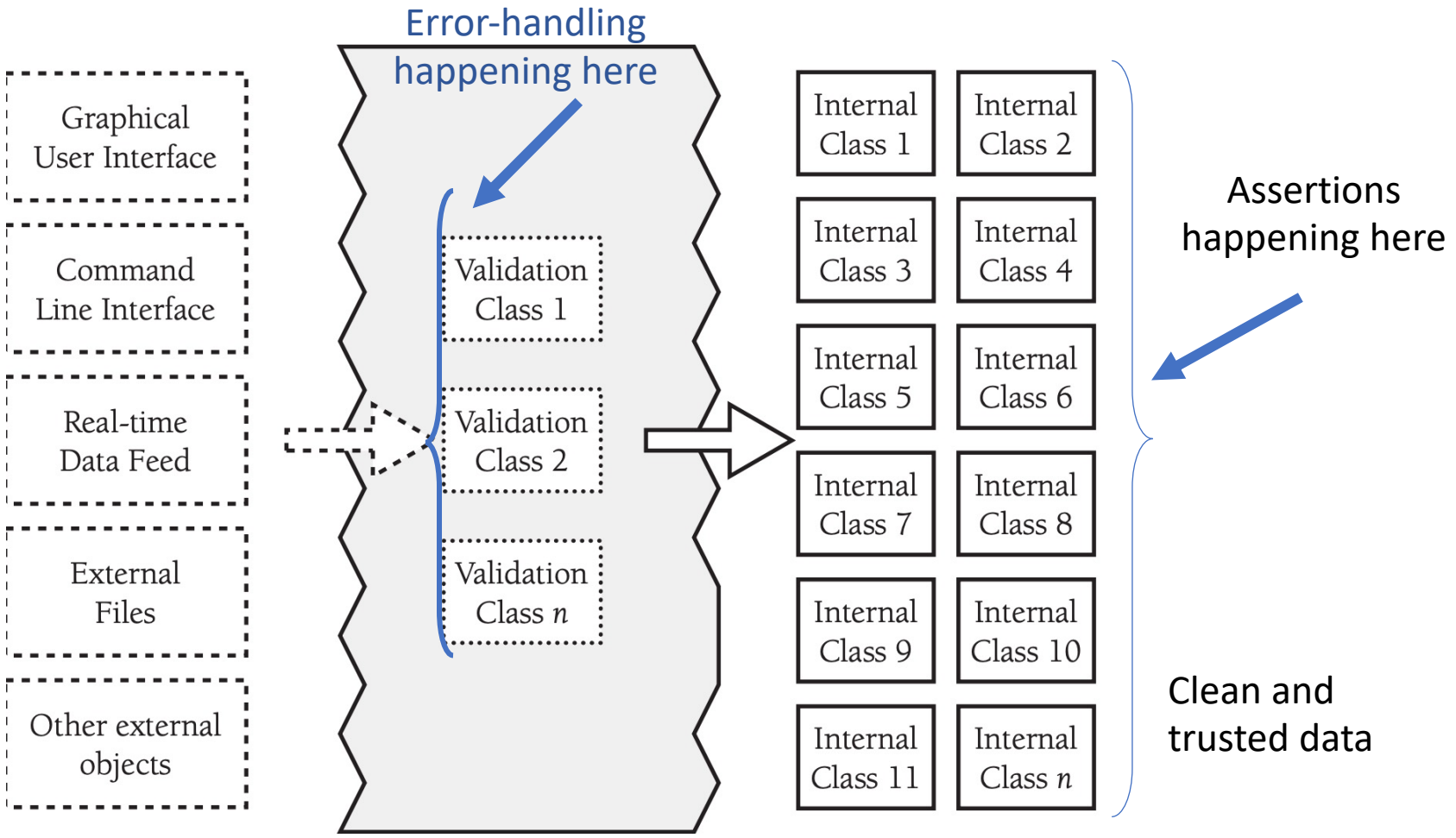
This Week

- Defensive programming
- Test/Code Coverage (Lab 3)
- Code review today
- Assignment 4 released – due next Tuesday

How to protect your program from invalid inputs?

- Take some Prevention steps
- Check the values of all data from different resources
  - File
  - Network
  - Another routine
- How to handle bad input?
    - **Assertions**
    - **Error handling**

Assertions vs Error-Handling



Classes responsible for cleaning the data making up the barricade.

Assertions vs Error-Handling

- Where to use assertions / where to use Error-Handling?
  - Use assertions for assumptions/conditions that should never occur
  - Use Error-handling checks for circumstances that might not occur very often

Assertions

- An assertion takes a Boolean expression
- When an assertion is true:
  - Program is operating as expected.
- When it’s false:
  - An unexpected error in the code is detected.

Assertions

- Used to detect bugs in the code:

Assertions

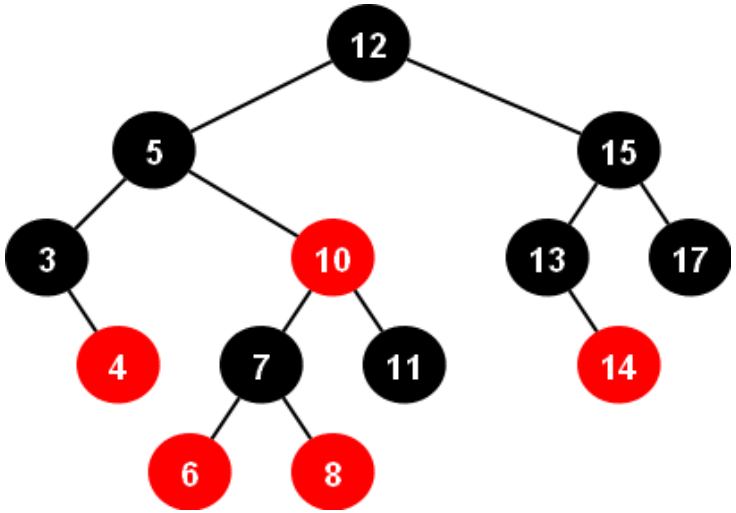
- Used to detect bugs in the code:
  - Code passes invalid parameters to another function
    - Null pointer(s)
    - An array or other container passed into a routine can containing a smaller number of elements than expected
  - Return value is not within the allowed range

Assertions: C++ Example

```
void printNumber(int* myNum) {

  assert (myNum!=nullptr); // if a nullptr is passed, then a bug.
  cout<<"myInt contains value" << " = "<<*myNum<<endl;

}

int main (){
  int myVal=5;
  int * secondPtr= nullptr;
  int * thirdPtr = nullptr;
  secondPtr=&myVal;

  printNumber (secondPtr);
  printNumber (thirdPtr);

  return 0;
}
```

Assertions:

```
void checkRep (rb_red_blk_tree *tree)
{
  /* root is black by convention */
  assert (!tree->root->left->red);
  checkRepHelper (tree->root->left, tree);
}
// returns the number of black nodes along any path through
// this subtree
int checkRepHelper (rb_red_blk_node *node, rb_red_blk_tree *t)
{

  /* both children of a red node are black */
  if (node->red) {
    assert (!node->left->red);
    assert (!node->right->red);
  }

  /* every root->leaf path has the same number of black nodes */
  int left_black_cnt = checkRepHelper (node->left, t);
  int right_black_cnt = checkRepHelper (node->right, t);
  assert (left_black_cnt == right_black_cnt);
  return left_black_cnt + (node->red ? 0 : 1);
}
```



Error types

- Errors that need to be handled:

Error types

- Type of some errors that could happen and should be handled:
  - Input values are not within some range
  - File missing / not able to open
  - Socket gets disconnected
  - Out of range
  - Division by zero

Error-handling techniques

- How to handle errors that are expected to occur?

Error-handling techniques

- How to handle errors that are expected to occur?
  - Returning a neutral value (if it is a good choice)
  - Keep reading until the next valid data
  - Log a warning message to a file
  - Return an error code (Throw an exception)
  - Display an error message (Easy but risky)
  - Shut down

Error-handling techniques

- Handle invalid parameters in consistent ways throughout the program
- The technique affects:
  - Correctness
  - Robustness

Error-Handling: Exceptions in C++

- **std::exception** - Parent class of all the standard C++ exceptions.
- **logic_error** - Exception happens in the internal logical of a program.
  - **invalid argument** - Exception due to invalid argument.
  - **out_of_range** - Exception due to out of range i.e. size requirement exceeds allocation.
  - **length_error** - Exception due to length error.
- **runtime_error** - Exception happens during runtime.
  - **range_error** - Exception due to range errors in internal computations.
  - **overflow_error** - Exception due to arithmetic overflow errors.
  - **underflow_error** - Exception due to arithmetic underflow errors
- . . . .

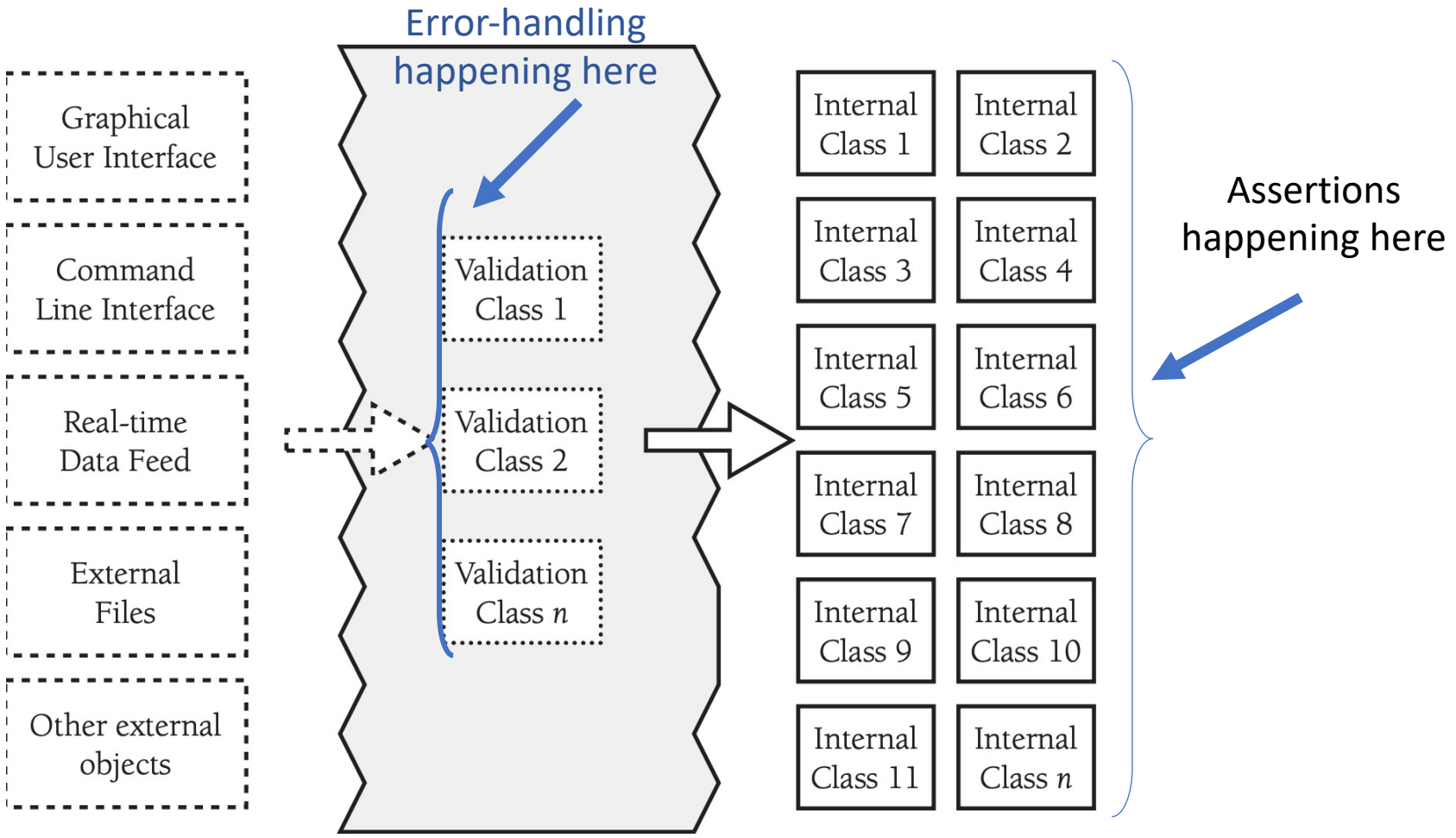
Why different exception types?

- Make code simpler and cleaner
- Easier to investigate a specific category rather than a general one
- Exceptions allow the programmer to recover from an error

Throwing an exception: highlights

- Throw an exception for conditions that are truly exceptional
- Throw an exception to notify other parts of the program about errors that should not be ignored
- Avoid throwing exceptions in constructors and destructors
- Include in the exception message reasons behind the exception
- Avoid empty catch blocks

Assertions vs Error-Handling



Classes responsible for cleaning the data making up the barricade.

Offensive programming

- Sometimes the best defense is a good offense.
- Fail hard during development so that you can fail softer during production.
- Some ways for offensive programming:
  - Make sure asserts abort the program.
  - Completely fill any memory allocated so that you can detect memory allocation errors.
  - Fill an object with junk data just before it’s deleted.
  - Set up the program to notify error log files to yourself