CS 211 : Thurs 01/25 (lecture 07)

Prof. Hummel
(he/him)

- **<u>Topics</u>: testing, unit testing, code coverage**

Northwestern University

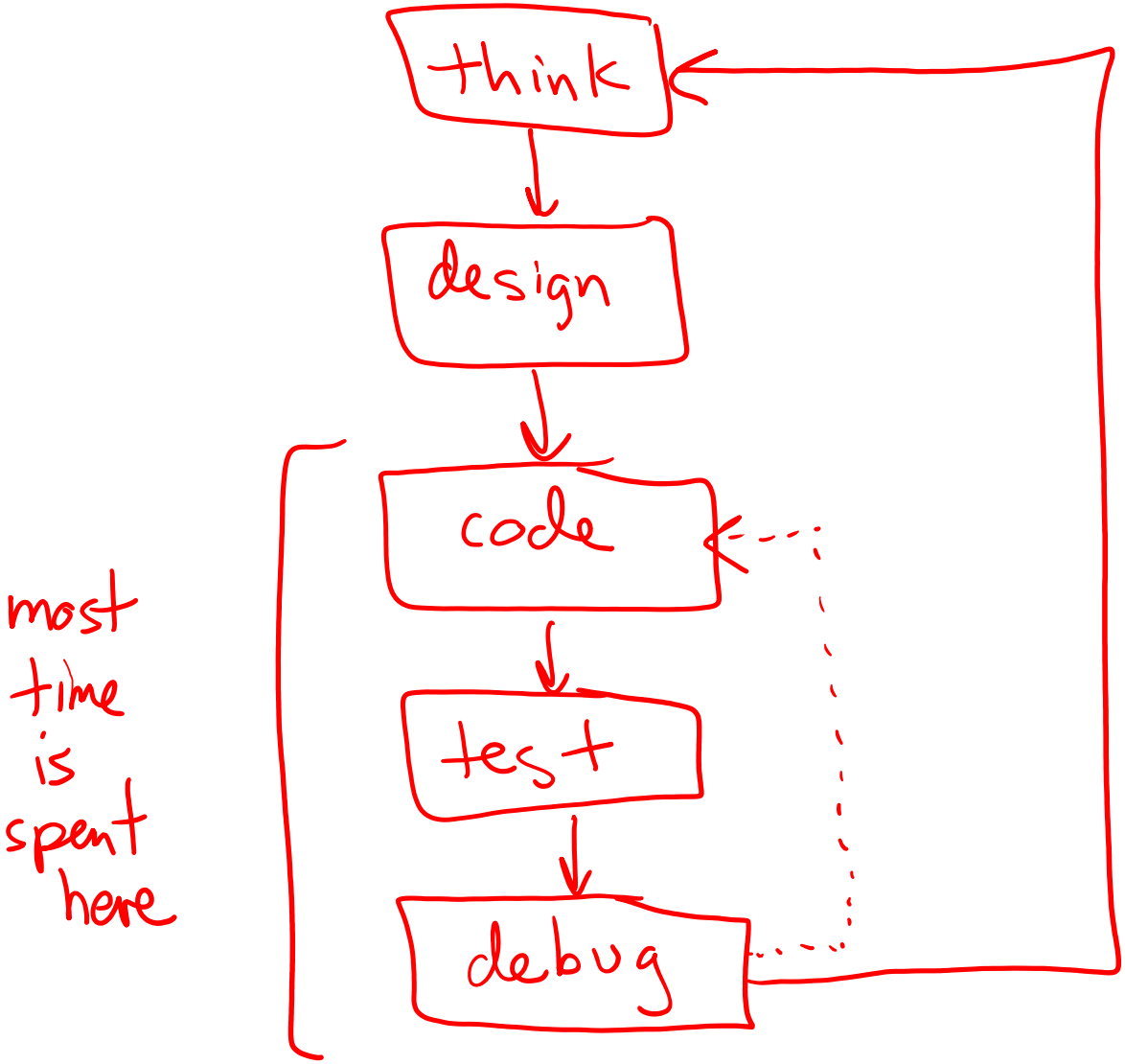
Notes:

- *Lecture slides available on Canvas*
- ***Project 03** due Friday @ 11:59pm, may be submitted up to 48 hours late. Gradescope is open for submissions.*
- *There will be a **HW 04** due next Tuesday*
- *Attendance, HW and Project scores are posting to Canvas – please check for accuracy*

Software development

- **Software development is hard**

Software development



(1) What would happen if…

- **What do you think would happen if we did not provide access to Gradescope, but instead we ran Gradescope after the deadline to grade your work?**

A) I would still earn 80/80 based on my own testing beforehand

B) I would probably get closer to 60/80 as my own testing would not be sufficient to catch all the edge cases

C) I don't read the handout very closely, so I would probably not do very well (20/80?)

D) I don't follow instructions well, and computers are tedious, so I would probably get 0/80

E) Other

Testing

- **Testing is hard**
- **You have to create scenarios that execute *all* possible paths through the code**
- **You have to repeat it over and over again…**

Project 03

- **Project 03 has 4 data types (int, real, string, boolean), 12 operators, 4 functions (print, input, int, float), 4 statement types, and various semantic errors.**

  Think about just the data types and the operators. What is the minimal number of unique tests needed? Don't just guess, think about it…

Enter your answer…

Testing commercial software

- **An internship has you working on a "small" program of 100,000 lines of code with 5,000 functions spread across 1,000 C source files.** *[For comparison, Microsoft office suite is 30 million lines of C++.]*

  You are asked to make a change, which requires modifying a few functions.

  How do you test it to make sure (a) you made the change correctly, and (b) you did not break something else?

We need a <u>process</u> for testing, you can't just try a few things and hope it works.

Rule #1

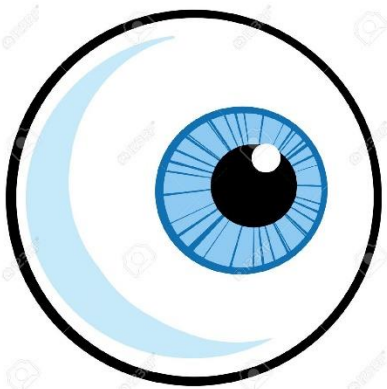
- **You have to design software so it can be tested**

- **Example:**
  - *The nuPython project supports interactive input for easier testing*

```
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = 123
y = x ** 3.1
z = "a string" + 1
$
**no syntax errors...
**building program graph...
**executing...
**SEMANTIC ERROR: invalid operand types (line 3)
```

Interactive Testing

- **Interactive testing is tedious, and doesn't scale…**



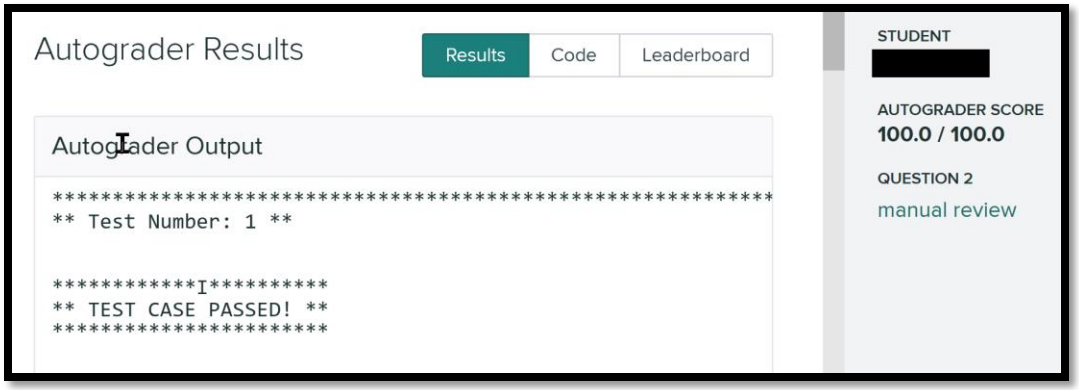
```
hummel> ./a.out
nuPython input (enter $ when you're done)>
x = 3.14159
y = x - 6.2
z = 1 + y
s = "start of string"
s2 = s + " end of string"
a = x
b = x ** 2.2
print(x)
print(y)
print(z)
print(s2)
print(b)
$
**no syntax errors...
**building program graph...
**executing...
3.141590
-3.058410
-2.058410
start of string end of string
12.408775
**done
**MEMORY PRINT**
Capacity: 8
Num values: 7
Contents:
 0: x, real, 3.141590
 1: y, real, -3.058410
 2: z, real, -2.058410
 3: s, str, 'start of string'
 4: s2, str, 'start of string end of string'
 5: a, real, 3.141590
 6: b, real, 12.408775
**END PRINT**
hummel>
```

Is the output correct? Do I really have to type this over and over again?

Rule #2

- **We need a system that is**
  - *Automated*
  - *Repeatable*
  - *Easy to determine pass/fail*

- **Example:**
  - ***Gradescope***



Unit testing

- **Industry standard approach**
- **Idea:**
  - *Break software into "units"*
    - e.g. database, parser, analyzer, resultset
  - *Lots of tests (think **thousands**)*
  - *Automated by a testing framework*
    - CATCH, Google Test, JUnit
  - *Run tests nightly / after any change*

```
Test01()
{ ... }
```

```
Test02()
{ ... }
```

```
Test03()
{ ... }
```

```
Test04()
{ ... }
```

⋮

Example: automated testing

```
void Test01()
{
  FILE* output = fopen("test.py", "w");
  fprintf(output, "x = 100\ny = x + 1\n");
  fclose(output);

  FILE* input = fopen("test.py", "r");

  parser_init();
  struct TokenQueue* tokens = parser_parse(input);
  struct STMT* program = programgraph_build(tokens);

  struct RAM* memory = ram_init();

  execute(program, memory);

  assert(memory->num_values == 2);

  assert(strcmp(memory->cells[0].identifier, "x") == 0);
  assert(memory->cells[0].ram_cell_type == RAM_TYPE_INT);
  assert(memory->cells[0].types.i == 100);

  assert(strcmp(memory->cells[1].identifier, "y") == 0);
  assert(memory->cells[1].ram_cell_type == RAM_TYPE_INT);
  assert(memory->cells[1].types.i == 101);

  printf("Test01 passed!\n");
}
```

```
x = 100
y = x + 1
```

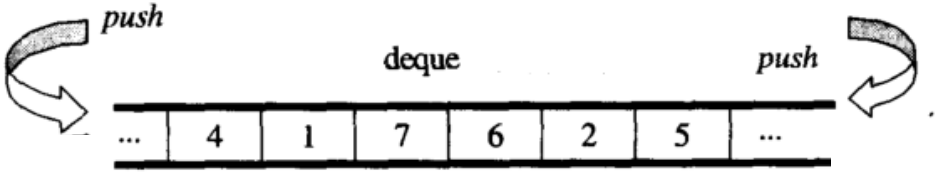
our program

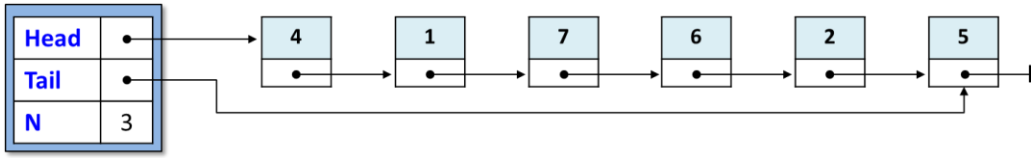
```
nuPython input (enter $ when you're done)>
x = 100
y = x + 1
$
**parsing successful
**building AST...
**starting execution...
**MEMORY PRINT**
Size: 4
Num values: 2
Contents:
 0: x, int, 100
 1: y, int, 101
**END PRINT**
```

You have to write code to test code…

A better example

- **Deque ("deck")**
  - *From C++ library*
  - *An **abstract data type** that allows insert @ front and back*
  - *Implemented using a **linked-list** data structure*






```
struct IntDeque {
  struct IntNode* head; // 1st element
  struct IntNode* tail; // last element
  int             N;    // # of values
};
```

```
struct IntNode {
  int             value;
  struct IntNode* next;
};
```

Demo – unit testing

- Login to replit.com
- Open team…
- Open project "Lecture 7 – Unit testing"

```
struct IntDeque* intdeque_create(void);

void intdeque_destroy(struct IntDeque* dq);

void intdeque_push_front(struct IntDeque* dq, int value);

void intdeque_push_back(struct IntDeque* dq, int value);

int  intdeque_size(struct IntDeque* dq);  // # of elements

int  intdeque_get(struct IntDeque* dq, int position /*1..N*/);

void intdeque_print(struct IntDeque* dq);
```

Google Test ("gtest")

- **Google test** is an industry standard unit testing framework
- We'll use in project 04

```
#include <stdio.h>
#include <stdlib.h>

#include "gtest/gtest.h"

int main()
{
  ::testing::InitGoogleTest();

  //
  // run all the tests, returns 0 if all pass
  //
  int result = RUN_ALL_TESTS();

  return result;
}
```

```
TEST(deque, initialization)
{
  struct IntDeque *dq = intdeque_create();
  ASSERT_TRUE(dq != NULL);
  ASSERT_TRUE(intdeque_size(dq) == 0);
}

TEST(deque, add_to_front)
{
  struct IntDeque *dq = intdeque_create();
  ASSERT_TRUE(dq != NULL);
  ASSERT_TRUE(intdeque_size(dq) == 0);

  intdeque_push_front(dq, 123);
  ASSERT_TRUE(intdeque_size(dq) == 1);
  ASSERT_TRUE(intdeque_get(dq, 1) == 123);
}
```

```
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from deque
[ RUN      ] deque.initialization
[       OK ] deque.initialization (0 ms)
[ RUN      ] deque.add_to_front
[       OK ] deque.add_to_front (0 ms)
[----------] 2 tests from deque (0 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 2 tests.
```

Unit testing of IntDeque

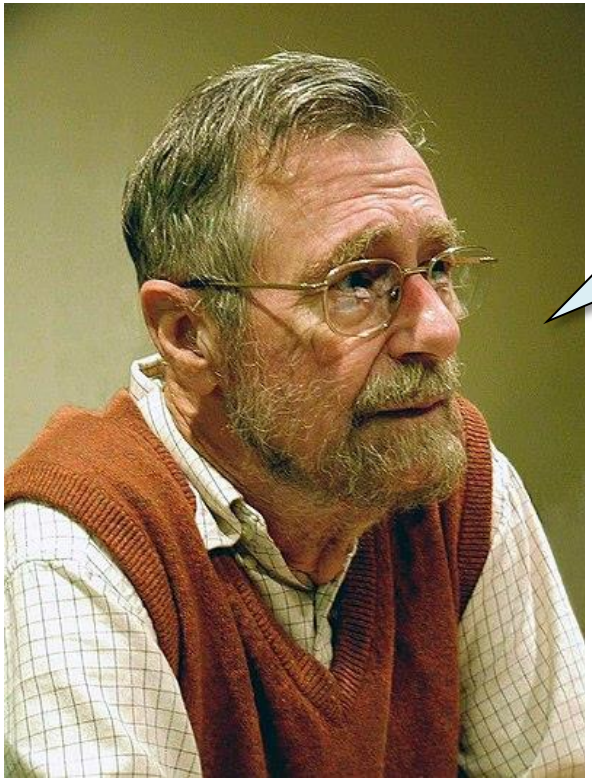
- **In "test.c", write unit tests to test the code**
  - *Call every function*
  - *Call functions in different orders*
  - *Check if functions perform correctly*

```
TEST(deque, add_to_back)
{
   struct IntDeque* dq = intdeque_create();
   ASSERT_TRUE(dq != NULL);
   ASSERT_TRUE(intdeque_size(dq) == 0);

   intdeque_push_back(dq, 456);
   ASSERT_TRUE(intdeque_size(dq) == 1);
   ASSERT_TRUE(intdeque_get(dq, 1) == 456);
}
```

- **Observations…**
  - *You have to think about ways to* ***break*** *software…*
    - Edge cases (front, back, empty, full, …)
    - Stress test (millions of values)
    - Pass invalid parameters
  - *You have to* ***design software*** *with testing in mind…*
  - *The tests become an integral part of the system* --- ***tests are code*** *that must be written and maintained*

- **How many unit tests are enough?**
- **Unit testing does *not* guarantee correctness…**



Edsger Dijkstra

"Testing shows the presence of bugs, not their absence"

[i.e. testing doesn't prove correctness…]

We really need formal methods and rigorous proofs to guarantee that software is correct. And better programming languages…

What's the standard professionally?

- **Testing effort varies by company…**
- **Not much is shared publicly…**

- **At companies that put significant effort into testing, a common expectation seems to be "5-to-1", i.e.**
  - *5 lines of testing code for every 1 line of application code*

  - ***Example**: project 03 is roughly 800 LOC ==> we should have at least 4,000 lines of test code.*

How good are my tests?

- Is there a way to measure test quality?
- Yes!
- Code coverage
- The % of code “covered” (executed) by the tests
- The goal?

Code coverage with gcov

1. Build and run program with coverage options
2. Run “gcov” to collect coverage information
3. Open .gcov file(s) to view results

```
gcc -std=c11 -g –Wall –fprofile-arcs –ftest-coverage
 main.c  intdeque.c  ...

./a.out

gcov a-intdeque.c

<< open intdeque.c.gcov in editor >>
```

```
$ gcov a-intdeque.c
File 'intdeque.c'
Lines executed:44.23% of 52
Creating 'intdeque.c.gcov'
```

% of code covered

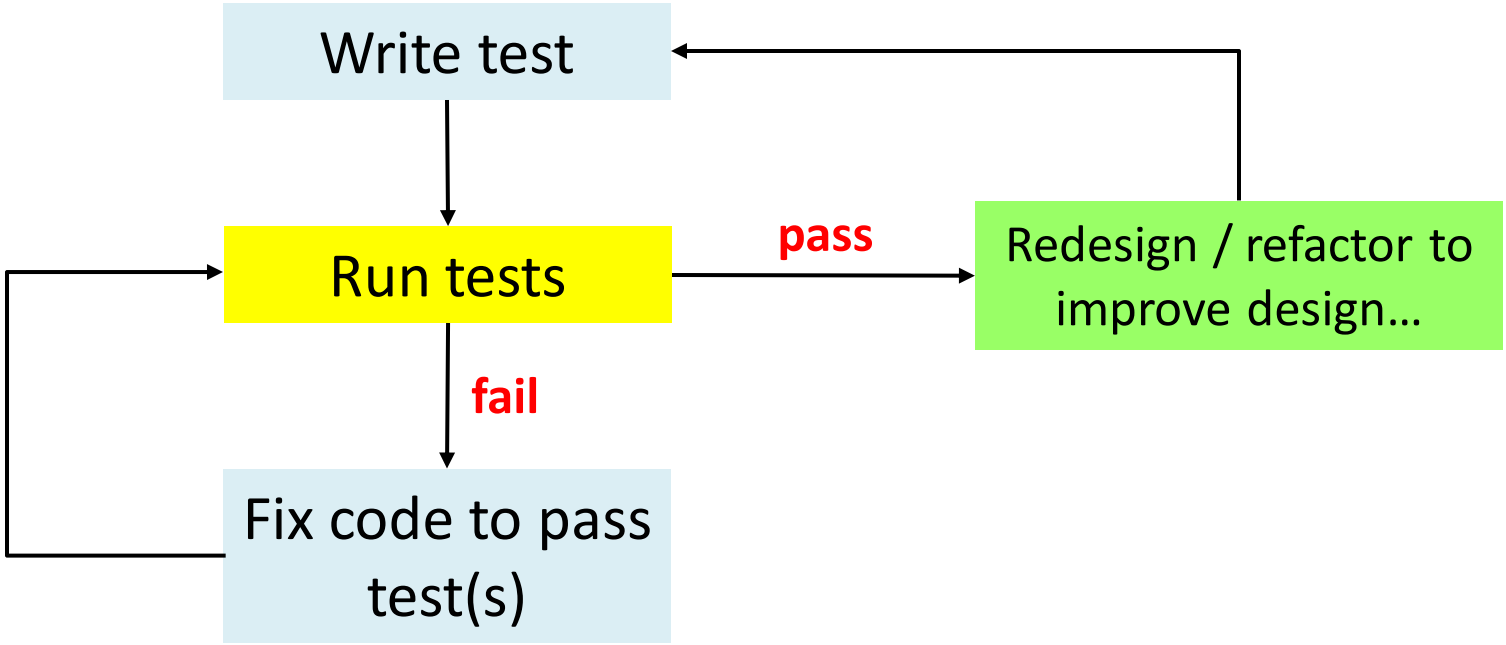
intdeque.c.gcov

```
    -:   20:
    1:   21:  dq->head = NULL;
    1:   22:  dq->tail = NULL;
    1:   23:  dq->N = 0;
    -:   24:
    1:   25:  return dq;
    -:   26:}
    -:   27:
#####:   28:void intdeque_destroy(struct IntDeque *dq) {
#####:   29:  struct IntNode *cur = dq->head;
    -:   30:
    -:   31:  //
    -:   32:  // free the nodes in the list:
    -:   33:  //
#####:   34:  while (cur != NULL) {
#####:   35:    struct IntNode *next = cur;
    -:   36:
#####:   37:    free(cur);
    -:   38:
#####:   39:    cur = next;
    -:   40:  }
    -:   41:
#####:   42:  free(dq); // now free head structure
#####:   43:}
```

Lines marked with "#####" were not executed…

TDD: Test Driven Development

- **Development approach where you write the tests FIRST**
- **Then you write the function to pass the tests**



What should I be working on?

Project 03 *is due Friday night…*

Watch for release of ***HW 04*** *and* ***Project 04*** *over the weekend…*