PROGRAMMING METHODOLOGY
(PHƯƠNG PHÁP LẬP TRÌNH)

UNIT 7: Testing and Debugging

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Mr. Aaron Tan Tuck Choy for kindly sharing these materials.

Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Recording of modifications

- Currently, there are no modification on these contents.

Unit 7: Testing and Debugging

Objectives:

- Learning how to test your codes and what to look out for
- Learning techniques to debug your codes

Reference:

- Lesson 1.9 Basic Debugging

Unit 7: Testing and Debugging

1. Famous Programming Errors
2. Testing and Debugging
3. Incremental Coding
4. Using the gdb Debugger

1. Famous Programming Errors

- **Mariner Bugs Out (1962)**
  - **Cost:** $18.5 million
  - **Disaster:** The Mariner 1 rocket with a space probe headed for Venus diverted from its intended flight path shortly after launch. Mission Control destroyed the rocket 293 seconds after liftoff.
  - **Cause:** A programmer incorrectly transcribed a handwritten formula into computer code, missing a single superscript bar. Without the smoothing function indicated by the bar, the software treated normal variations of velocity as if they were serious, causing faulty corrections that sent the rocket off course.

- **Mars Climate Crasher (1998)**

  - **Cost:** $125 million
  - **Disaster:** After a 286-day journey from Earth, the Mars Climate Orbiter fired its engines to push into orbit around Mars. The engines fired, but the spacecraft fell too far into the planet's atmosphere, likely causing it to crash on Mars.
  - **Cause:** The software that controlled the Orbiter thrusters used imperial units (pounds of force), rather than metric units (Newtons) as specified by NASA.

2. Testing and Debugging (1/8)

- Test your programs with your own data
  - Do NOT rely on CodeCrunch to test your programs!
- We have discussed this in unit 5. That is the error in this code?

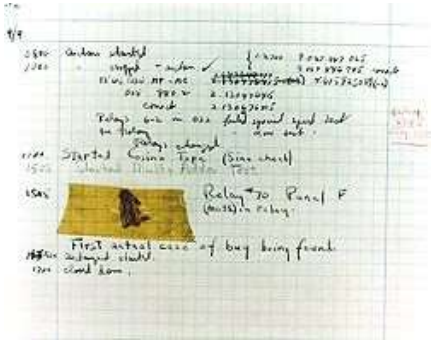
```
// To find the maximum among 3 integer
// values in variables num1, num2, num3.
int max = 0;
if ((num1 > num2) && (num1 > num3))
  max = num1;
if ((num2 > num1) && (num2 > num3))
  max = num2;
if ((num3 > num1) && (num3 > num2))
  max = num3;
```

  - It works fine if it is tested on some sets of data: <3,5,9>, <12,1,6>, <2,7,4>, etc.
  - But what test data are missing?

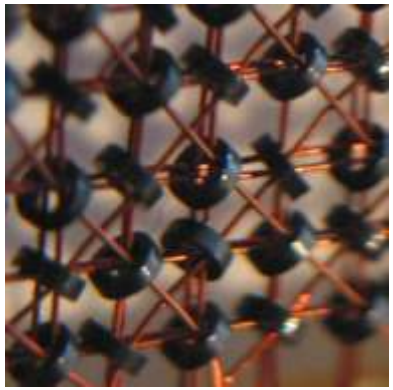
2. Testing and Debugging (2/8)

- The example in the previous slide shows the importance of testing.
- Where does the term "debugging" come from?
  - Very early computers used mechanical relays for switching currents. However, most likely the term bug existed before the "moth-in-a-relay" story.

Notebook with moth from Mark II computer:

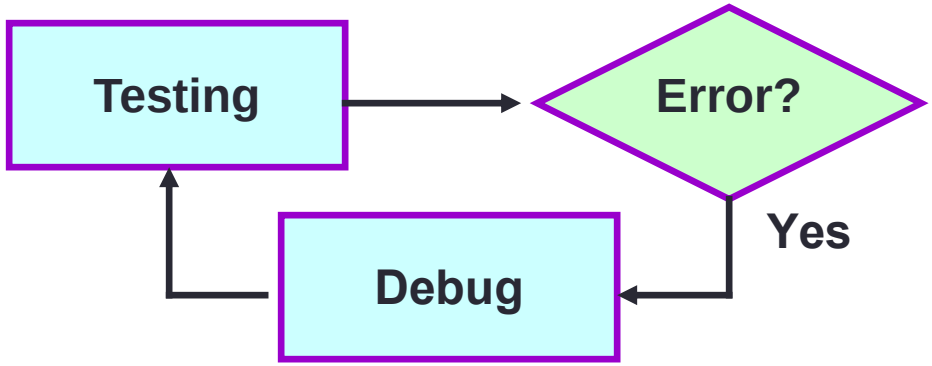


  - Why does a program "core dump"?
    - Early computers used tiny magnetic cores (rings) as memory cells to hold 0 and 1 values.



2. Testing and Debugging (3/8)

- Testing
  - To determine if a code contains errors.
- Debugging
  - To locate the errors and them.
- Documentation
  - To improve maintainability of the code.
  - Include sensible comments, good coding style and clear logic.



2. Testing and Debugging (4/8)

- Philosophical notes on program design, debugging and testing
  - A good design is important
    - A good design results in a high quality program.
    - A low quality design cannot be “debugged into” high quality.
  - Do not optimize for speed too early
    - It is possible to make a correct program run faster.
    - It is much more difficult to make a fast (but wrong) program run correctly.

2. Testing and Debugging (5/8)

- A program should be tested with various input values to make sure that it performs correctly across all inputs.
- A program should make as few assumptions about the input as possible.
  - E.g.: Your program assumes that the user will type a number. But she types a string → crash!
  - However, in CS1010 we assume that input follows the specification. We do this to focus on the basics first. Writing robust programs is not trivial.
  - Still, like the example in slide 5, we should not make wrong assumption, such as assuming that all the input integers are distinct.
- Many of today's methods to hack into computers work by feeding programs with unexpected inputs. This results in crashes, buffer overflows, etc.

2. Testing and Debugging (6/8)

How to test?

- By user/programmer:
  - Trace program by hand multiple times.
- By test program:
  - Write a little test program that runs the program to be tested with different inputs.
- By test environments:
  - Large-scale test suites that generate test cases, run them, compare the expected output and provide a pass/fail assessment.

2. Testing and Debugging (7/8)

- Manual walkthroughs
  - Tracing with pencil-and-paper
  - Verbal walkthroughs
- "Wolf fencing" with printf()
  - Easy to add
  - Provide information:
    - Which functions have been called
    - The value of parameters
    - The order in which functions have been called
    - The values of local variables and fields at strategic points
  - Disadvantages
    - Not practical to add printf() statements in every function
    - Too many printf() statements lead to information overload
    - Removal of printf() statements tedious

Wolf fence:
"There's one wolf in Alaska, how do you find it? First build a fence down the middle of the state, wait for the wolf to howl, determine which side of the fence it is on. Repeat process on that side only, until you get to the point where you can see the wolf. In other words, put in a few 'print' statements until you find the statement that is failing (then maybe work backwords from the 'tracks' to find out where the wolf/bug comes from)."

2. Testing and Debugging (8/8)

- Tips and Techniques
  - Start off with a working algorithm
  - Incremental coding/test early/fix bugs as you find them
  - Simplify the problem
  - Explain the bug to someone else
  - Recognize common errors (such as using '=' instead of '==', did not initialise, etc.)
  - Recompile everything (referring to a suite of programs)
  - Test boundaries
    - Eg: For primality test (Week 4 Exercise #3), did you test your program with the value 1? 2?
  - Test exceptional conditions
  - Take a break!

3. Incremental Coding (1/2)

- Incremental coding: Implementing a well-designed algorithm part by part systematically
  - You must design the algorithm first!
- How?
  - Choose a basic unit in your algorithm
  - Implement (code) it, then compile and test your program until it is correct
  - Proceed to the next basic unit in your algorithm
- Consequences
  - Your program is compilable and executable at all time
  - With every completion of a basic unit, the functionality grows
  - Bugs can be more easily detected and cleared
  - Boost morale

3. Incremental Coding (2/2)

- Basic unit
  - A part of the program that is self-contained and well defined
  - Eg: the part that handles user's inputs; the part the prints the outputs in the required format; the function that computes some result.
- Case study
  - We will illustrate the idea of incremental coding using the Taxi Fare exercise you have seen in week 3

Taxi Fare (1/4)

- The taxi fare structure in Singapore must be one of the most complex in the world! See http://www.taxisingapore.com/taxi-fare/
- Write a program TaxiFare.c that reads the following input data (all are of int type) from the user, and computes the taxi fare:
  - dayType: 0 represents weekends and public holidays (PH for short);
    1 represents weekdays and non-PH
  - boardHour, boardMin: the hour and minute the passengers board the taxi (eg: 14 27 if the passengers board the taxi at 2:27 PM)
  - distance: the distance of the journey, in metres
- Your program should have a function

```
float computeFare(int dayType, int boardTime, int distance)
```

  - The parameter boardTime is converted from the input data boardHour and boardMin. It is the number of minutes since 0:00hr.
    - Eg: If boardHour and boardMin are 14 and 27 respectively, then boardTime is 867.

Taxi Fare (2/4)

- To implement the actual taxi fare could be a PE question ☺. In this exercise, we use a (grossly) simplified fare structure:
  - Basic Fare:

Flag-down (inclusive of 1st km or less)	$3.40
Every 400m thereafter or less up to 10.2km	$0.22
Every 350m thereafter or less after 10.2km	$0.22

  - Surcharge (applicable at the time of boarding):

dayType	Midnight charge (12am – 5:59am)	Peak hour charge (6am – 9:29am)	Peak hour charge (6pm – 11:59pm)
0: Weekends & PH	50% of metered fare	None	25% of metered fare
1: Weekdays and non-PH	50% of metered fare	25% of metered fare	25% of metered fare

Taxi Fare (3/4)

- You are given an incomplete program TaxiFarePartial.c. Complete the program. This exercise is mounted on CodeCrunch.
- Sample runs below for your checking.

```
Day type: 0
Boarding hour and minute: 14 27
Distance: 10950
Total taxi fare is $9.12
```

First 1km: $3.40
Next 9.2km: 23 × $0.22 = $5.06
Next 750m: 3×$0.22 = $0.66
Basic fare = $9.12
No surcharge
Total fare = $9.12

```
Day type: 1
Boarding hour and minute: 9 20
Distance: 6123
Total taxi fare is $7.83
```

First 1km: $3.40
Next 5123m: 13 × $0.22 = $2.86
Basic fare = $6.26
Surcharge = 25% × $6.26 = $1.57
Total fare = $7.83

```
Day type: 1
Boarding hour and minute: 5 59
Distance: 9000
Total taxi fare is $11.70
```

First 1km: $3.40
Next 8km: 20 × $0.22 = $4.40
Basic fare = $7.80
Surcharge = 50% × $7.80 = $3.90
Total fare = $11.70

Taxi Fare (4/4)

- Note that due to inaccuracy of floating-point number representation, depending on how you code your formula to compute the taxi fare, the result may defer slightly from the model output CodeCrunch uses. Hence, your program may fail CodeCrunch's tests.
- In this case, if the difference is very small (probably in the second decimal place), just treat your answer as correct.

3. Incremental Coding Example (1/6)

- Algorithm for Taxi Fare

1. Read inputs: dayType, boardHour, boardMin, distance
2. Compute boardTime

   boardTime ← boardHour × 60 + boardMin
3. Compute taxi fare

   float computeFare(int dayType, int boardTime, int distance)

   3.1 Compute basic fare
   basicFare = computeBasic(distance);
   … details of computeBasic() here …

   3.2 Compute surcharge
   surcharge = computeSurcharge(dayType, boardTime, basicFare);
   … details of computeSurcharge() here …

   3.3 taxiFare = basicFare + surcharge
4. Print output: taxiFare

4. Incremental Coding Example (2/6)

- How the coding proceeds… (comments in program are not shown for brevity)

Version 1

```
#include <stdio.h>
#define INCREMENT 0.22

int main(void) {
   int dayType, boardHour, boardMin, boardTime, distance;

   printf("Day type: ");
   scanf("%d", &dayType);
   ...

   boardTime = boardHour * 60 + boardMin;
   printf("Boarding time is %d minutes\n", boardTime);

   return 0;
}
```

Always print intermediate results, even if the question doesn't ask for it. Remove or comment off the printf statement before submission.

Compile and run this program, make sure it is correct before proceeding!

4. Incremental Coding Example (3/6)

- Continuing …

Version 2

```
#include <stdio.h>
#define INCREMENT 0.22
float computeFare(int, int, int);

int main(void) {
   int dayType, boardHour, boardMin, boardTime, distance;
   float taxiFare;
   ...
   boardTime = boardHour * 60 + boardMin;
   printf("Boarding time is %d minutes\n", boardTime);

   taxiFare = computeFare(dayType, boardTime, distance);
   printf("Total taxi fare is $%.2f\n", taxiFare);

   return 0;
}

// … continue on next slide
```

Newly added codes

4. Incremental Coding Example (4/6)

- Continuing …

Version 2

```
// … continue from previous slide
float computeFare(int type, int time, int dist) {

   return 123.50;
}
```

Newly added function computeFare()

- The above function is called a stub.
- It returns an arbitrary value (123.5) and not the correct answer, but doing this allows us to test whether the main() function is able to call it.
- Compile this version and make sure it works before proceeding!

4. Incremental Coding Example (5/6)

- We continue to develop the computeFare() function
  - You may choose to split the task into 2 sub-tasks: compute basic fare and compute surcharge, and write a function for each
  - You may then choose to implement one of the two sub-tasks first

Version 3

```
float computeFare(int type, int time, int dist) {

   float basicFare = computeBasic(dist);
   float surcharge = computeSurcharge(type, time, basicFare);

   return basicFare + surcharge;
}
```

4. Incremental Coding Example (6/6)

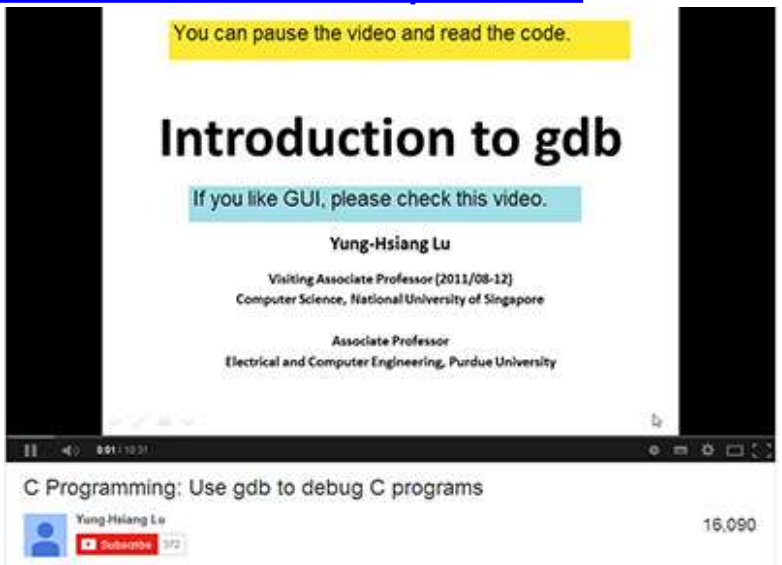
- Since computeSurcharge() seems to be easier to implement than computeBasic(), you can make the latter a stub and implement the former first.
- Compile the program and test it out before proceeding to implement computeBasic().

Version 3

```
float computeBasic(int dist) {
   return 1.35;
}

float computeSurcharge(int type, int time, float basic) {
   float surcharge;

   if (time < 360) // between 12am and 5:59am
      surcharge = 0.5 * basic;
   else if ...

   return surcharge;
}
```

4. Using the gdb Debugger (1/3)

- A debugger is a tool that assists in the detection and correction of errors in programs
- It usually provides the following
  - Stepping
  - Breakpoint
  - Watches (inspecting values of variables)
- We will illustrate these with gnu debugger gdb
  - See video http://www.youtube.com/watch?v=Z6zMxp6r4mc by A/P Lu Yung-Hsiang



4. Using the gdb Debugger (2/3)

- Step 1: Add the –g option when compiling and linking your program:
  - `gcc -g Unit4_WashersV2.c`
- Step 2: Start gdb with yur program:
  - `gdb a.out`
- Step 3: Use gdb commands to step through your program:
  - Break at a specific function: *break function_name*
    - Start with: *break main*
  - Run program: *run* (or *r*)
  - Step to the next line of code: *step* (or *s*)
  - List source code: *list* (or *l*)
  - Examine the value of a variable: *print variable_name*
  - Quit the debugger: *quit*

Explore other gdb commands on your own!

4. Using the gdb Debugger (3/3)

- Some links on gdb:
  - https://sourceware.org/gdb/current/onlinedocs/gdb/
  - http://www.cprogramming.com/gdb.html
  - http://www.thegeekstuff.com/2010/03/debug-c-program-using-gdb/
  - http://www.tutorialspoint.com/gnu_debugger/

Summary

- In this unit, you have learned about
  - How to test and debug your programs
  - How to use incremental coding to implement your code part by part systematically
  - How to use the gdb debugger

End of File