COSC345
Software Engineering

Instructors and Support People

- Dr. Andrew TrotmanOffice: 123A, Owheo
  - Email: andrew@cs.otago.ac.nz
  - No office hours, just “drop in”

- •Department Kaiāwhina(Māori and Pacific Support):
  - Steven Mills (Office: 245, Owheo)

- •Department Disability Support:
  - Contact the main office.

Learning Outcomes

- This paper will give students practice in:
  - Designing a software system
  - Planning the development
  - Carrying out the development using appropriate tools
  - Evaluating their work
- Students will learn to use "good practices" including:
  - Version control
  - Static checking
  - Testing
  - Following industry or platform standards

Academic Integrity and Academic Misconduct

- Academic integrity means being honest in your studying and assessments. It is the basis for ethical decision-making and behaviour in an academic context. Academic integrity is informed by the values of honesty, trust, responsibility, fairness, respect and courage. Students are expected to be aware of, and act in accordance with, the University's Academic Integrity Policy.
- Academic Misconduct, such as plagiarism or cheating, is a breach of Academic Integrity and is taken very seriously by the University. Types of misconduct include plagiarism, copying, unauthorised collaboration, taking unauthorised material into a test or exam, impersonation, and assisting someone else's misconduct. A more extensive list of the types of academic misconduct and associated processes and penalties is available in the University's Student Academic Misconduct Procedures.
- It is your responsibility to be aware of and use acceptable academic practices when completing your assessments. To access the information in the Academic Integrity Policy and learn more, please visit the University's Academic Integrity website at www.otago.ac.nz/study/academicintegrity or ask at the Student Learning Centre or Library. If you have any questions, ask your lecturer.
- Academic Integrity Policy
- Student Academic Misconduct Procedures

What is Software Engineering?

- Building and maintain large scale software projects
  - A branch of
    - Computer Science
    - Engineering
    - Management
- How to manage
  - People, processes, products
- Origins
  - Successes and (too often) failures
    - MARS Lander disaster
    - Examples from previous lectures

Course Outline

- 26 weeks
  - Semester 1:
    - Lecture (Tuesday, St David B, 4pm-5pm)
    - Tutorial (Friday, Quad 1, 9am-10am)
  - Semester 2 is not finalized, but probably the same

- 18 point paper
  - So 180 hours throughout the year
  - 90 hours per semester
    - 2 hours per week for 13 weeks are class time.
    - So how many hours of your time should you spend on the paper?

Course Outline

- Assessment
  - 40% internal assessment
  - 60% exam
- 4 assignments (10% each)
  - Group engineering project
  - Semester 1 dates are:
    - 9 April, 29 May
  - Semester 2 dates are probably:
    - 31 July, 25 September
  - Form a group of 3 or 4 (***not*** less than 3 and ***not*** more than 4)
  - Come to Friday's Tutorial with ideas
    - We ***will*** be using C/C++
  - Short presentations in last week of year

Resources

- Textbook
  - S. McConnell, *Code Complete*, Microsoft Press, 1991
  - Recommended Reading:
    - F. Brooks, *Mythical Man Month*
    - J. Bentley, *Programming Pearls*
    - J. Bentley, *More Programming Pearls*
    - S. Maguire, *Writing Solid Code*
    - I. Sommerville, *Software Engineering*
- Resources
  - COSC345 Website
  - Internet
  - Science Library
  - Email (check this regularly)

Course Outline

- Semester 1
  - Introduction and little languages (today)
  - Project Planning
  - Team Organization
  - Requirements
  - Communications
  - Waterfall
  - Agile
  - Literate Programming
  - Make
  - Version Control
  - Continuous Integration
  - Code Review

Course Outline

- Semester 2
  - Debugging
  - Testing
  - Optimisation
  - Computer architecture
  - The Stack
  - The Heap
  - Memory Managers
  - Memory Checkers
  - Libraries
  - Components
  - Portability
  - Ethics

Little Languages

What are Little Languages?

- Elegant interfaces to control programs
- Simple problem
  - Complex to code in a ‘regular’ language (e.g. Java)
  - Easily described in a ‘limited’ language
- Interactive problem
  - Texturally control a program
- Typical problems
  - Repetitive descriptions
  - Tricky manipulations
  - Control of hardware
- Just like methods and routines they move repetitive or complex tasks to one place

Why Little Languages?

- Simplify the problem
  - Reduce the chance of making errors
- Reduce repetition
  - Reduce the chance of making errors
- Easy to write
  - Compilers and interpreters easy to write
  - Programs in little languages are easy to write
- Often line-based languages
- Easy to extend
- Reusable in many related projects

Example Little Languages

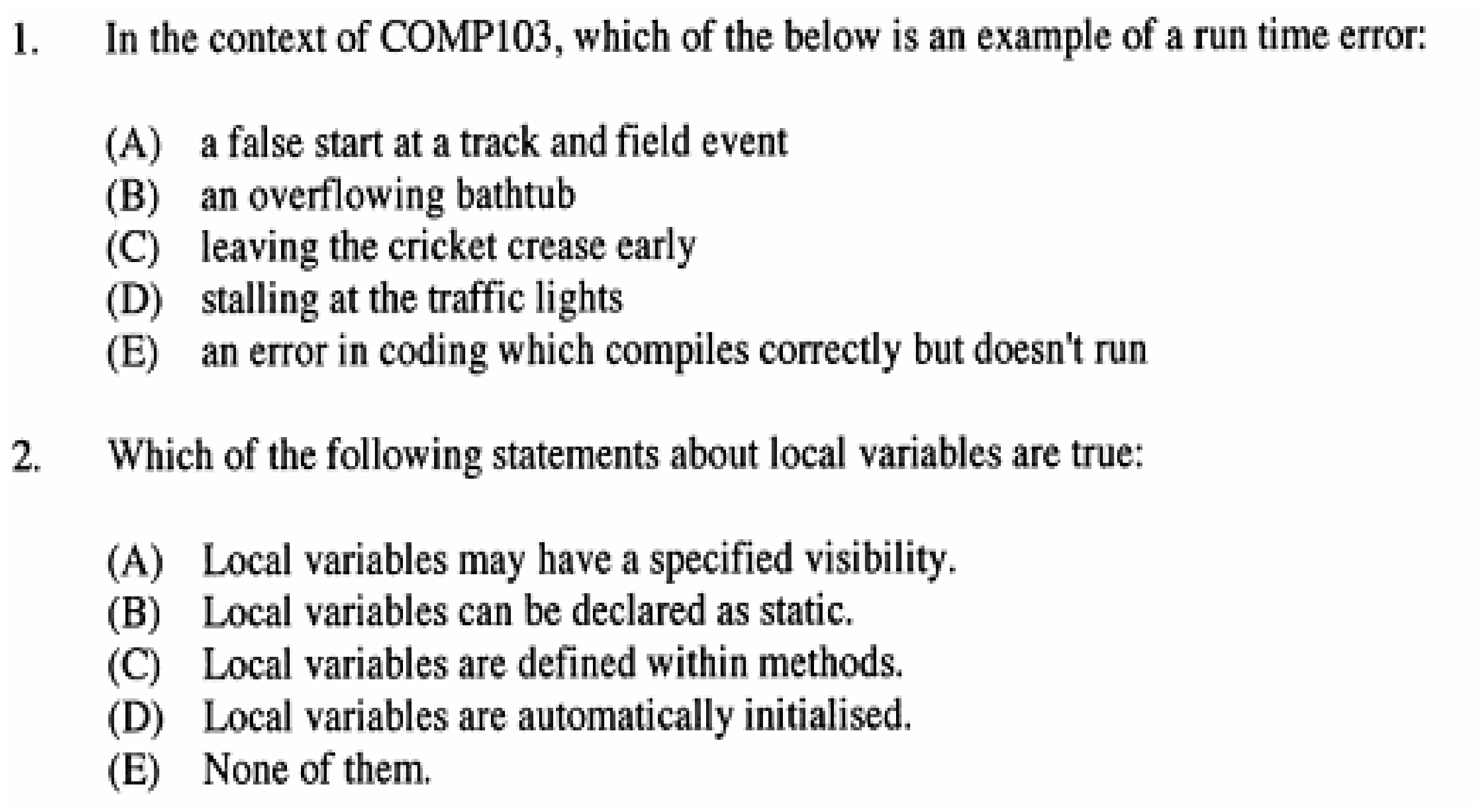
- ***Make***
- ***Unix shell*** / DOS shell
- Lex / Yacc
- Python
- Perl
- ***LaTeX***
- Awk

- You have probable used those marked in ***italics***

Example: Multi-choice Program

- You have been approached by a lecturer about writing a program that will interactively conduct a multi-choice exam for the fictitious COMP103. The questions have just been written and delivered to you. Your task now is to write a program that conducts the exam.

Multi-choice Exam



1. In the context of COMP103, which of the below is an example of a run time error:

   (A) a false start at a track and field event
   (B) an overflowing bathtub
   (C) leaving the cricket crease early
   (D) stalling at the traffic lights
   (E) an error in coding which compiles correctly but doesn't run

2. Which of the following statements about local variables are true:

   (A) Local variables may have a specified visibility.
   (B) Local variables can be declared as static.
   (C) Local variables are defined within methods.
   (D) Local variables are automatically initialised.
   (E) None of them.

C Program

```
#include <stdio.h>
#include <ctype.h>
int main(int argc, char *argv[])
{
char buffer[10];
int correct = 0;
printf("1. In the context of COMP103, which of the below is an example of a run time
    error:\n");
printf("\n");
printf("(A) a false start at a track and field event\n");
printf("(B) an overflowing bathtub\n");
printf("(C) leaving the cricket crease early\n");
printf("(D) stalling at the traffic lights\n");
printf("(E) an error in coding which compiles correctly but doesn't run\n");
printf("Answer\n");
fgets(buffer, sizeof(buffer), stdin);
if (toupper(*buffer) == 'E')
    correct++;
printf("\n");
printf("2. Which of the following statements about local variables are true:\n");
printf("(A) Local variables may have a specified visibility\n");
printf("(B) Local variables can be declared static\n");
printf("(C) Local variables are defined within methods\n");
printf("(C) Local variables are automatically initialised\n");
printf("(E) None of them\n");
fgets(buffer, sizeof(buffer), stdin);
if (toupper(*buffer) == 'C')
    correct++;
printf("Your score:%d", correct);
return correct;
}
```

Needs

- What does this program *need* to do?
  - Print text
  - Read answers
  - Mark (match) answers
  - Compute scores

- Ideal use of a little language! Perhaps use:
  - T: text output
  - A: answer input
  - M: match answers
  - C: compute values

Input / Output / Matching

```
T:
 : 1.  In the context of COMP103, which of the
 :     below is an example of a run time error:
 : (A) a false start at a track and field event
 : (B) an overflowing bathtub
 : (C) leaving the cricket crease early
 : (D) stalling at the traffic lights
 : (E) an error in coding which compiles correctly
 :     but doesn't run
 :
A:
M: E
```

Computing scores

```
C : #correct = 0

M :E
CY: #correct = #correct + 1
CN: #wrong = #wrong + 1
```

Whole Program

```
C: #correct = 0
T:
 : 1. In the context of COMP103, which of the below is an example of a run time error:
 : (A) a false start at a track and field event
 : (B) an overflowing bathtub
 : (C) leaving the cricket crease early
 : (D) stalling at the traffic lights
 : (E) an error in coding which compiles correctly but doesn't run
 :
A:
M: E
CY:#correct = #correct + 1

T:
 : 2. Which of the following statements about local variables are true:
 : (A) Local variables may have a specified visibility
 : (B) Local variables can be declared static
 : (C) Local variables are defined within methods
 : (C) Local variables are automatically initialised
 : (E) None of them
 :
A:
M: C
CY:#correct = #correct + 1

T: Your score:#correct
```

PILOT

- Programmed Inquiry, Learning, Or Teaching language
  - First developed in 1962
  - Standardized by IEEE 1991

- A little language for teaching
  - Multi choice questions
  - Short answer questions
  - Interactive learning sessions

- Versions released by Apple (and others)

Designing Little Languages

- Study the whole problem (regardless of the problem)
  - Think about how a little language might help
- Think about:
  - Abstraction
    - Identify the key components
    - Identify the key manipulations
  - Simplicity
    - Keep any language as simple as possible
  - Linguistic structure
    - What is the natural form of expression
    - Always allow comments

Language Design

- Completeness
  - Ensure all the necessary operations exist
    - Write several programs on paper first
    - Then write a compiler or interpreter
  - Ensure all the necessary restrictions exist
    - Make sure you can’t write a program that does them
  - Add features dictated by use
- Orthogonality
  - Keep unrelated features unrelated
- Parsimony
  - Delete unneeded features

Language Design

- Generality
  - Use an operation for many purposes
- Similarity
  - Make the language suggestive
- Extensibility
  - Make it extendable
  - Allow the language to evolve
- Openness
  - Ensure related tools can be used with it
  - Use pre-existing tools to build it (Lex / Yacc / AWK)
  - Build on experience of using languages (Java, etc.)

References

- J. Bentley, *More Programming Pearls*, Chapter 9