Where we are; where we’re going

- What we’ve done last quarter
  - Basic programming C++
  - Input and output via stdin and stdout
  - Basic command-line operations
- What we’re about to do this quarter
  - Low-level features: *pointers and dynamic data structures*
  - Improved methods: *test, code management, debug*
- This week
  - Testing and version control
  - Intro to pointers

Assessments and Testing

Portfolio : how to assess?

- Possible methods of assessment
  1. Manual inspection : reading each submission
  2. Manual execution : typing in commands for each ex.

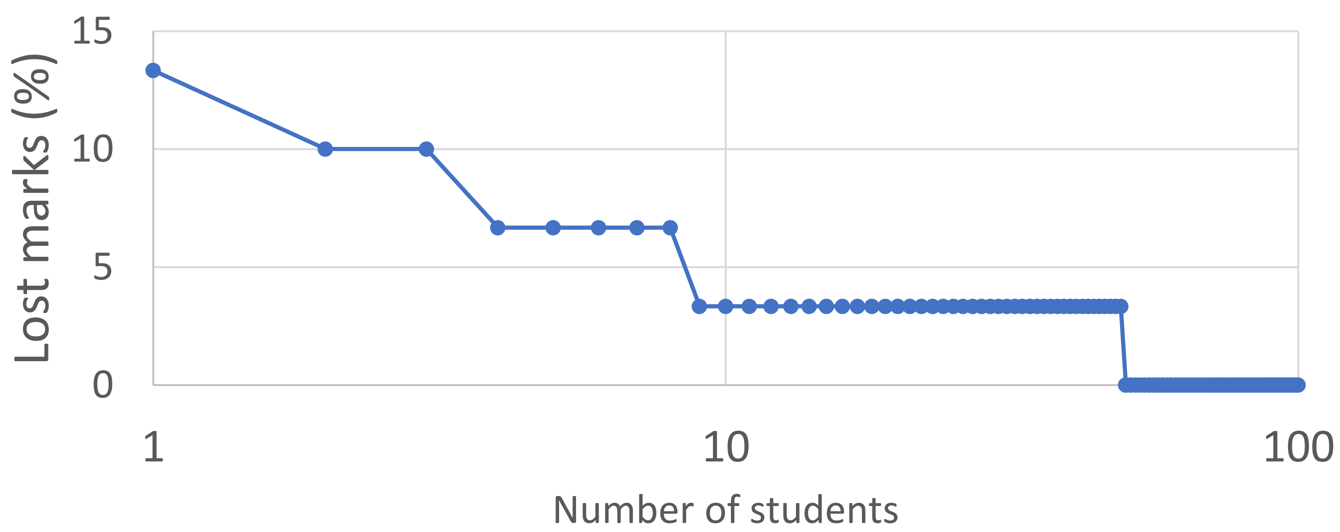
- Potential problems
  - Accuracy : are the exercise requirements assessed?
  - Precision : is each submission assessed fairly?
  - Time : how long does it take to assess?

Accuracy: is each exercise correct?

- We have:
  - 200 submissions
  - 30 exercises per submission
  - ~6000 “things” to test
- Exercises require different numbers of checks
  - “Delete this directory” – Is the directory still there?
  - “Implement this poly.” – Need to check multiple inputs
- On average we need about three checks per exercise
  - ~18000 “checks” enough for decent accuracy

Precision: is assessment fair?

- We have about 18000 checks to perform
  - Let's assume a human is 99% accurate at checks
  - In 1% of cases we get a false negative
    - Human claims it fails, when it actually succeeds
- In total we get ~180 errors across all cases



Time : when will it be finished?

- We have 18000 checks to do
  - Assume one person takes 10 seconds per check
  - Comes to ~180000 person-seconds or £1000
    - One person: 50 hours full-time (1.25 weeks)
    - One day: seven people working for 1 full day
- Assume we automate the checks
  - Assume each check takes 1 second (compilation is slow)
  - Assume 10 seconds setup per submission
  - Comes to ~20000 CPU-seconds
    - Sequential : 5.6 hours on one CPU
    - Parallel : 40 minutes on eight CPUs

Portfolio : how to assess?

- Possible methods of assessment
  1. ~~Manual inspection : reading each submission~~
  2. ~~Manual execution : typing in commands for each ex.~~
  3. Automated testing : program to test each submission
  4. Batch testing: program to test all submissions
- Potential problems
  - Accuracy : are the exercise requirements assessed?
  - Precision : is each submission assessed fairly?
  - Time : how long does it take to assess?

What’s going on with the portfolio

- *Done*: initial filtering
  - Minor fix-ups to slightly incorrect submissions (with penalty)
- *Now*: testing the testbench
  - Running the tests against all submission and look for bugs
  - Investigate any common failures – are the tests wrong?
  - Add diagnostics to explain any common problems
- *Soon*: Summary results probably ready about Thursday
- *Also*: some people have extra time due to illness
  - Full results not released till they are done as the results of tests might give an advantage

From assessment to testing

Assessment is a special-case of testing

Assessment	Testing
Instructor	Client
Student	Developer
Exam / coursework	Requirements
Answers / solutions	Deliverables[*]

[*] Deliverables: the thing(s) delivered to the client

- Software
- Documentation
- Test-cases

Testing : did we do it right?

- All software should be tested in some way
  - Simple program: a *few manual sanity tests*
  - Complex program: *thousands of automated tests*
- EIE2 are currently writing CPU simulators

  a lot of effort goes into test: about 50-200 test-cases
- Google has about 2 billion lines of code

  ... and about 4 million tests[1]

- To test you must first have *requirements*
  - What did the client want it do?

[1] https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html

Requirements : what do we want?

- Requirements are needed to ***engineer*** a system
  - Software: *what are the input and output formats?*
  - Digital: *how many logic gates can we use?*
  - Signals: *what band-pass frequency is needed?*
  - Control: *what is the maximum allowed acceleration?*
  - Systems*: how much power can it consume?*

- Requirements define and constrain the problem
- If you don't have requirements you are hacking
  - *Disclaimer*: academics often do this
  - Engineering *research* is a bit different to engineering

Requirements: FRs and NFRs

Requirements are often split into two groups

- **F**unctional **R**equirements
- **N**on-**F**unctional **R**equirements

Functional: what should a system ***do***?

- Multiply two matrices
- Transform an image using a filter

Non-functional: what should a system ***be***?

- User-friendly
- Fast
- Power efficient
- …

Assessments : mainly FRs

- In the first year most assessments are functional
  - We need clearly defined goals and specifications
    - Create a program which performs function x
    - Designed a circuit which resonates at frequency y
  - Non-functional requirements are often implicit
    - e.g. your program shouldn’t take 1 hour and 64GB to run
- Non-functional aspects will come up later
  - Execution time constraints on software
  - Qualitative targets in the project
- Later years introduce more non-functional aspects
  - Most difficult engineering is about FR vs NFR tradeoffs

Testing: verification and validation

- ***Verification***: does the system meet requirements?
- ***Validation***: does the system meet user needs?
- A system can pass verification but fail validation

  Client: “*build me a system that multiplies large matrices*”

  Developer: “*Ok. So it needs to calculate A = B C. Fine.*”

  …

  Developer: “*Here is your system. Functions perfectly.*”

  Client: “*Yes… but it takes 3 days?*”

  Developer: “*That wasn’t in the requirements.*”

Assessments : mainly verification

- Academics prefer verification (assessment & research)
  - Verification can often be reduced to some form of maths
  - Academics love maths!
  - Verification can be made accurate, fair, and fast(-ish)
- Don’t mistake assessment for engineering
  - Maths is used to support and enable engineering
  - Engineering is not maths
- Validation is often harder than verification
  - Involves humans and the real-world
  - Slower, messier, subjective, expensive, and difficult

Practical testing

(Meta-comment)

- The following mixes concepts and application
  - Fundamental concepts about programs and testing
  - Applied knowledge about getting stuff done in the shell
- The "how" may not make much sense till you do it
  - Some of the commands and syntax are weird
- The "why" may take a while to become clear
  - For simple programs you may not see the point
  - The true value becomes clearer with larger projects
  - Some people never truly grok[1] it

[1] grok (verb) : to understand in a deep/profound way; programming slang

Zero testing

A terrible development strategy:

1. Read the specification
2. Modify code
3. Accept

Testing by compilation

A weak development strategy[1]:

1. Read the specification
2. Modify code
3. Compile it
4. Accept

[1] : In some languages this does work due to the Curry-Howard isomorphism : types <=> proofs
- Haskell : successful compilation *often* means the program is correct
- Coq : successful compilation **definitely** means the program is correct

Manual testing

A manual testing strategy

1. Read the specification
2. Modify code
3. Compile it
4. Run program and enter test input
5. Check output
6. Accept

Manual testing

A manual testing strategy

1. Read the specification
2. Query the specification
3. Modify code
4. Compile it
5. Run program and enter test input
6. Check output
7. Accept

Automated testing

A manual testing strategy

1. Read the specification
2. Query the specification
3. Modify code
4. Compile it
5. Run program and enter test input
6. Check output
7. Accept

Automated testing

A manual testing strategy

1. Read the specification
2. Query the specification
3. Modify code
4. Compile it
5. For each test case:
    1. Run program and enter test input
    2. Check output
6. Accept

Automated testing

A manual testing strategy

1. Read the specification
2. Query the specification
3. Modify code

4. Run tests

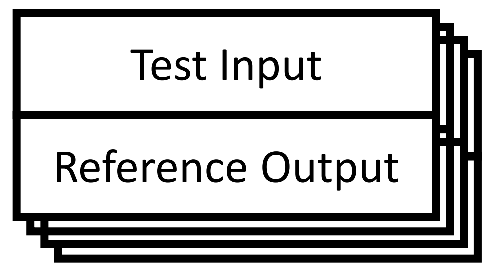
5. Accept

Automated testing

A manual testing strategy

1. Read the specification
2. Query the specification
3. Modify code

4. Run tests



5. Accept

Testing manually : comparing files

- Test Inputs
  - Ref. input : `input.txt`
  - Ref. output: `output.ref.txt`
  - Source code: `program.cpp`
- Test artefacts
  - Executable: `program`
  - Actual output: `output.got.txt`
- Test commands

```
$ g++ program.cpp -o program
$ < input.txt  ./program  > output.got.txt
$ diff output.ref.txt output.got.txt
```

artefact : something that is created as a by-product of a process or method

Putting commands into a script

- We have a manual sequence of commands

```
$ g++ program.cpp -o program
$ < input.txt  ./program  > output.got.txt
$ diff output.ref.txt output.got.txt
```

- We can put them into a script file `test.sh`

```
$ cat test.sh
g++ program.cpp -o program
< input.txt  ./program  > output.got.txt
diff output.ref.txt output.got.txt
```

- We can then run the script file `test.sh`

```
$ bash test.sh
```

A script file is “just” text

- Commands are interpreted by `bash`
  - Bash is just a program : the Bourne Again Shell
  - Typed commands go into the stdin of bash

- We can redirect the stdin of bash

  ```
  $ < script.sh bash
  ```

  - It will act as if you have typed in the contents of the file
- We can explicitly specify a script file to use as input

  ```
  $ bash script.sh
  ```

  - It will act as if you have typed in the contents of the file

Adding multiple tests

```
g++ program.cpp -o program

< input.1.txt  ./program  > output.1.got.txt
diff output.1.ref.txt output.1.got.txt

< input.2.txt  ./program  > output.2.got.txt
diff output.2.ref.txt output.2.got.txt

< input.3.txt  ./program  > output.3.got.txt
diff output.3.ref.txt output.3.got.txt
```

Programs versus functions

Complex functionality relies on composition and re-use

- Create simple units of functionality
- Re-use common pre-existing functionality
- Compose them into something more complex

	C++	Script
Unit of functionality	Function	Program
Composed using	Function calls	Running programs
Pre-existing functionality	Standard library	System programs
User-defined functionality	New functions	New programs

Scripts as programs

- The shell will treat a script file as a program if:
  1. The script starts with a “shebang”
  2. The script file has the executable permission set
- Shebang: the first line of the file should be

```
#!/bin/bash
```

- Permissions: make sure the ‘x’ bit is set

```
$ ls -la test.sh
-rw-rw-rw- 1 dt10 dt10 1 Nov 19 06:18 test.sh
$ chmod u+x test.sh
$ ls -la test.sh
-rwxrw-rw- 1 dt10 dt10 1 Nov 19 06:18 test.sh
```

Scripts as programs

- A known example: `prepare_submission.sh`

```bash
#!/bin/bash
set -euo pipefail
IFS=$'\n\t'

DATE_TIME=$(date +%Y-%m-%d--%H-%m-%S)
OUTPUT="ELEC40004-portfolio-[[USER]]-${DATE_TIME}.tar.gz"

WD=$(pwd)
>&2 echo "Capturing submission in directory $WD"

WD_BASE=$(basename $WD)
if [[ "${WD_BASE}" != "ELEC40004" ]] ; then
    >&2 echo "WARNING: expected current directory to be ELEC40004. ";
fi

LSB=$(lsb_release -a 2> /dev/null)
if ! echo "$LSB" | grep "Ubuntu 18.04" > /dev/null ; then
    >&2 echo "WARNING: is this definitely Ubuntu 18.04? "
fi

>&2 echo "Creating submission archive portfolio ${OUTPUT}"
( cd .. && tar -czf $OUTPUT ELEC40004 )
```

Programs as functions : return codes

- Every ***program*** has a return code at run-time
  - This is true of every program you have created
  - This is true of every command you have run

- The return code is an integer

- The *meaning* of return codes comes from convention

0 (zero) :	the program completed successfully
otherwise :	the program failed in some way

Programs as functions : return codes

- Standard programs have “intuitive” exit codes
  - They return 0 (success) when things have succeeded
  - They return !0 (failure) if something else happened

Program	Success (0)	Failure (!0)
g++	Program compiled	Syntax error, …
diff	The files were the same	Different or missing
mkdir	The dir was created	Dir already existed

`man` page or `--help` tells you what is success for program

Writing test scripts: failing early

- If anything goes wrong in our test, we want to know
  1. Did compilation fail? If so, stop
  2. Did the program crash? If so, stop
  3. Was the output different? If so, stop
  4. Did ***everything*** succeed? If so, test is passed
- By default bash just carries on if a program fails
  - It has an option to make it check each return code
  - Add the following at the top of a script:

    ```
    set -e
    ```
  - *Don't set it in your terminal:* any error will exit the terminal

Programs as functions : C++

Program return codes are hidden in plain sight

```
int main()
{
    cout << "Hello";
}
```

Programs as functions : C++

`main` is the only function with an implicit `return`

```
int main()
{
    cout << "Hello";

    return 0;
}
```

Programs as functions : C++

By default C++ programs “succeed”

```
int main()
{
    cout << "Hello";

    return 0;
}
```

Programs as functions : C++

If programs crash, they “fail” with a non-zero code

```
int main()
{
    vector<int> v;
    v[1000000] = 10;
}
```

Programs as functions : C++

You can manually “fail” by returning non-zero explicitly

```
int main()
{
  int x;
  cin >> x;
  if( x < 0){
    cerr << "Expected non-negative input" << end;
    return 1; // Return non-zero error code
  }
  ...
}
```

A complete test script

```
#!/bin/bash

set -e

g++ program.cpp –o program

< input.1.txt  ./program  > output.1.got.txt
diff output.1.ref.txt output.1.got.txt

< input.2.txt  ./program  > output.2.got.txt
diff output.2.ref.txt output.2.got.txt

< input.3.txt  ./program  > output.3.got.txt
diff output.3.ref.txt output.3.got.txt

echo “Success”
```

Testing as a worldview

- There is a balance between test and implementation
  - Creating tests takes time, and may delay the solution
  - The payoff comes surprisingly fast
- Testing is as much about mentality as process
  - How you do it matters less than whether you do it
- Software (and engineering) is not just about solutions
  - The process matters a great deal
  - Processes take a long time and involves many people

Source control

- Testing is often linked with source control
- Testing is used to drive functionality
  - Tests tell you what is currently broken
  - Modifications reduce the number of faults
  - You ***incrementally change*** the system to make it better
- Source control is used to track ***incremental changes***
  - Each modification adds more functionality
  - *Some* modifications break functionality
  - We want to keep the last working version available

Source control = version control

- Incremental development results in many versions
  - Each version adds a new piece of functionality
  - Each version of the program is “better”
  - Most code lives for a long time: lifetime of years
- Version control can be done manually
  - Keep source files with different suffices:
    - `prog_v0.cpp, prog_v1.cpp, prog_v2.cpp, …`
    - Most projects rely on more than just one source
  - Keep snapshots based on date+time:
    - `prog-2019-10-01.tar.gz, prog-2019-10-02.tar.gz, …`
    - Difficult to see what has changed between snapshots
- Source control automates version management

Source control = backup + collaboration

- Most modern source control is ***distributed***
  - There are multiple copies of the projects source files
  - Copies are held on many machines in many locations
  - Copies are frequently synchronised between machines

- Most modern source control is ***concurrent***
  - Lots of people work on their own copy independently
  - Changes get merges when copies are synchronised
  - Conflicts between changes are addressed while merging

We are going to use git

- Git is now ***the*** dominant method for source control
  - Though there are a few other options out there
- Used widely across all fields of software
  - Standard for open-source
  - Very common in industry
- Also used outside software to manage files
  - Common for digital design and document control
- Supported by some well-known infrastructure
  - Github, gitlab, Microsoft, …

Basic concepts in git : repositories

- ***Repository*** : a directory representing your project
  - Files within the repository will be versioned
  - Each file has it's own history

- ***Local*** repository : the repository on your computer
- ***Remote*** repository : a repository somewhere else
  - Could be a repository on someone else's laptop
  - Could be a repository stored in github

Learning about git

- We’re going to introduce git incrementally
- This term is all single user
  - Only you will be working in a repository
- Next term will be multiple user
  - Need to deal with conflicts
- Eventually git will be used to manage submissions

Summary : testing

- Testing is critical to getting a working program
  - In your study: making sure you pass assessments
  - In industry: making sure you deliver a working system
- Testing is part of the larger software lifecycle
  1. Requirements gathering
  2. Design
  3. Implementation
  4. Testing
  5. Maintenance
- Testing goes hand-in-hand with source control