The University of Sydney

Identifying and improving the learning of borderline students in SOFT2004

Technical Report Number 545

March 2004

Alan Fekete, Bob Kummerfeld,

Judy Kay and Tony Greening

ISBN 1 86487 627 1

School of Information Technologies

University of Sydney NSW 2006

Identifying and improving the learning of borderline students in SOFT2004

Alan Fekete, Bob Kummerfeld, Judy Kay and Tony Greening

2003

Executive summary

In attempting to identify “at risk” students in second-year programming units some consideration was given to their performance in the first year unit, COMP1002; however, this analysis did not prove useful. In response, we decided to examine the assessment performance of students in the 2003 enrollment for SOFT2004. A quantitative study included the assessment results for all students in the unit. Students who achieved total raw marks from 30 to 49 (inclusive) formed a cohort that were regarded as both “at risk” and most likely to benefit from intervention. This decision was made on the basis that students in this “borderline” band exhibited mean assessment results at both pass and fail levels; students above this band showed mean results that were clear passes only, and students below this band showed mean assessment outcomes that were clear fail results only.

Students in the borderline band showed greater variance in performance for most assessment tasks, and (quite unexpectedly) a negative correlation between programming assignments (for which the group achieved high mean results) and the final examination. This may be an issue with the level of engagement in some of the assessment tasks for students in this group that distinguishes them from other groups.

Other issues were also identified as needing general attention, such as the need for greater competence in UNIX usage prior to the introduction of concepts that expose the elegance of UNIX as a programming environment. Many subtleties in UNIX programming map neatly to an understanding of the UNIX command shell where such scaffolding already exists. The desirability of such scaffolding is evident.

Four bundles of examinations (80 students) were examined in greater detail. In a quantitative study of those papers it was found that deep understanding of the memory model corresponded to performance in certain programming tasks, including the most cognitively demanding question in the paper. This is of interest because even marginal performance gains in this particular question were a strong indicator of overall success in the exam.

A qualitative analysis of the examination papers for the 80 students revealed a list of errors, many of which were identified as being potentially generated by specific misconceptions. These misconceptions form the basis for a set of self-assessment resources that are being developed for inclusion in the self-assessment web site. Additionally, those with the greatest profile may be developed into lab activities that will confront any underlying misconceptions and enable the tutor to address these issues more broadly.

Tutors were consulted in order to gain some benefit from their close association with students in the labs; it was noted that many of their comments reflected the outcomes derived from the analyses described above.

Currently, a large number of self-test and quiz questions have been developed for core concepts of the unit. In addition the structure of lab sessions in weeks 2-4 (inclusive) has been modified to facilitate the early identification of “at risk” students so they can be targeted for remedial assistance.

Introduction

The assessment results for students in a second-year software development unit (SOFT2004) are explored in order to provide a feedback mechanism into the further teaching of that unit. As well as generic software development skills, the unit introduces students to *C* programming in a UNIX environment; almost universally, students enrolled in this unit had completed two first-year programming units introducing object-orientated programming via the *Java* programming language.

Particular emphasis is given to students with borderline pass/fail results in the belief that this cohort may benefit most from insights into improving their learning.

The discussion is divided into four sections:

A. *General assessment:* All assessment items for all students enrolled in the unit are included in generating an overview of the teaching and learning issues for the unit.

B. *The exam:* Exam papers for 80 students were extracted in order to gain a more detailed understanding of the possible learning issues. The discussion assumes both quantitative and qualitative dimensions in order to identify as many issues as possible.

C. *Tutor feedback:* Tutors were consulted in order to incorporate their insights into the review process.

D. *Summary and recommendations:* The most relevant issues from the earlier exploration are identified and these are used to generate some recommendations for teaching the next iteration of the unit.

Section A: General Assessment

Overall assessment

The breakdown by assessment item is as follows:

	Weighting	*Mean mark*	*Mean mark (%)*
Assignment 1	10%	6.7	67
Assignment 2	10%	7.0	70
Homework	6%	3.6	59
Quizzes	4%	1.8	44
Exam	70%	33.9	48

The mean raw total for all assessment items (weighted as above) was 49.2%.

Looking at correlation between assessment items shows high levels of positive correlation between individual performance in homework and quiz results (0.711); this is reasonable, given that both are focussed on specific learning items from the lecture material. However, it may also reflect the fact that serious attempts at the homework result in learning that is reflected in better quiz performance (and *vice versa*!). Less obvious is a reasonably high positive correlation between performances in the quiz and the exam (0.651); this may simply reflect some ability (or lack thereof!) to perform under exam conditions. Alternatively, it may simply identify student learning.

It is worth noting that the 3 unsupervised forms of assessment (2 assignments and homework) have clear pass results, whereas the examination components (formal exam and weekly quizzes) are considerably weaker.

To explore this observation further the class was divided into three groups based on total raw assessment:

- *Clear fail:* Results less than 30 (86 students, only 31 of which sat for the final exam).
- *Borderline:* Results from 30 to 49 inclusive (113 students).
- *Clear pass:* Results greater than 49 (285 students).

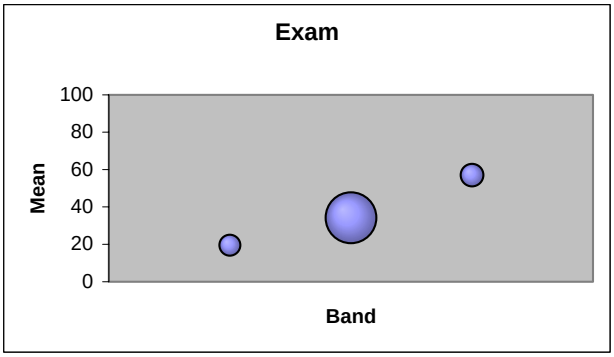
The following pages present a summary of centrality and dispersion for all assessment items for each of the three groups by use of "bubble diagrams"[*] and bar graphs. Note that the graphs express means as percentages.

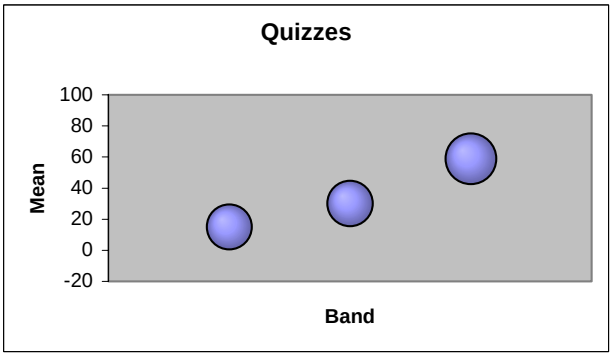
Percentage means are also presented in the following graph.

	Mean mark (%)		
	Clear Fail	*Borderline*	*Clear Pass*
Assignment 1	23.3	61.1	81.6
Assignment 2	24.2	66.3	84.1
Homework	17.5	45.1	77.5
Quizzes	15.0	30.2	58.8
Exam	19.6	34.3	57.1

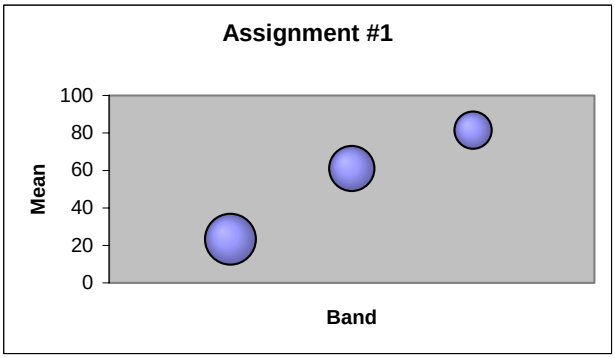
[*] Each group's performance is represented by a circle, the diameter of which enables a relative comparison of standard deviation, while the position of the centre on the Y-axis represents the mean. The groups ("bands") from left to right are "clear fail", "borderline", and "clear pass", respectively.

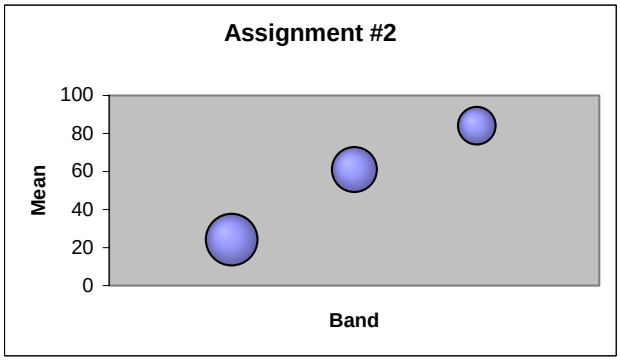
Supervised assessment tasks

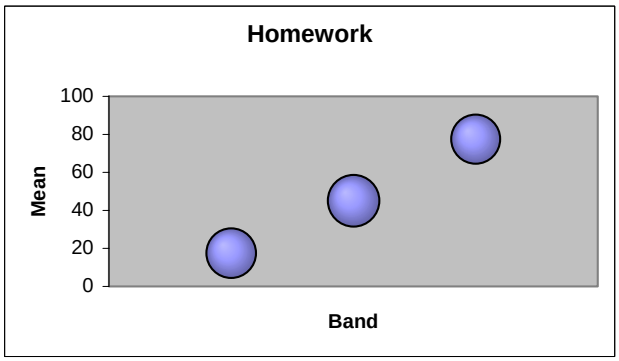




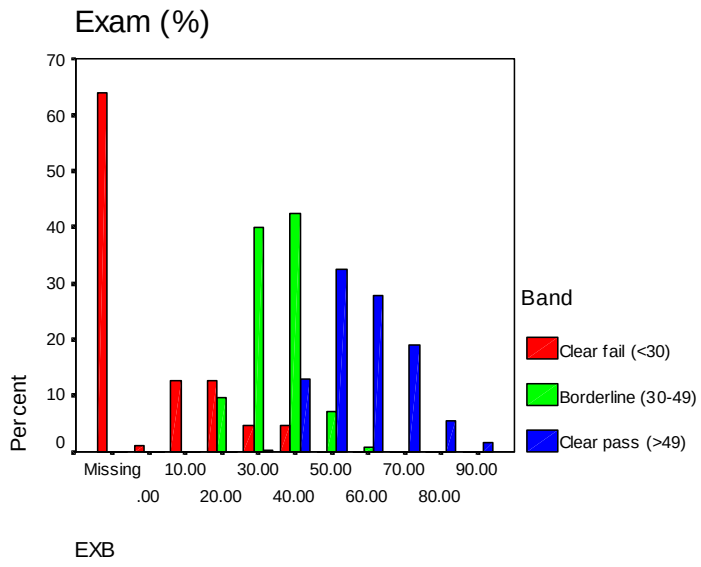
Unsupervised assessment tasks

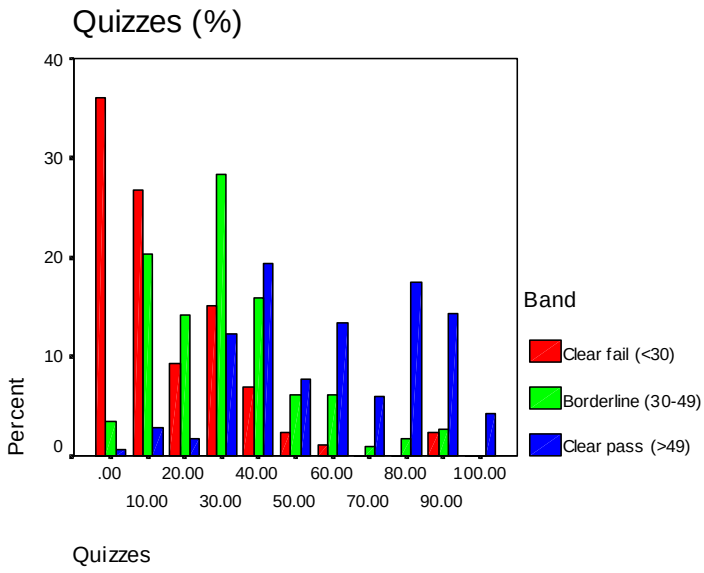




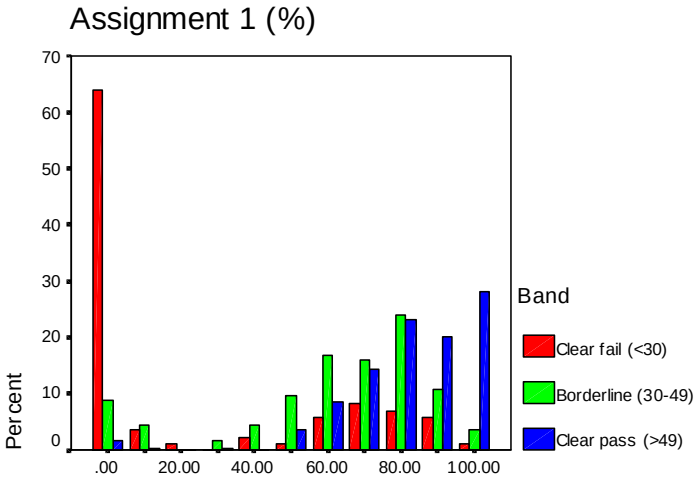


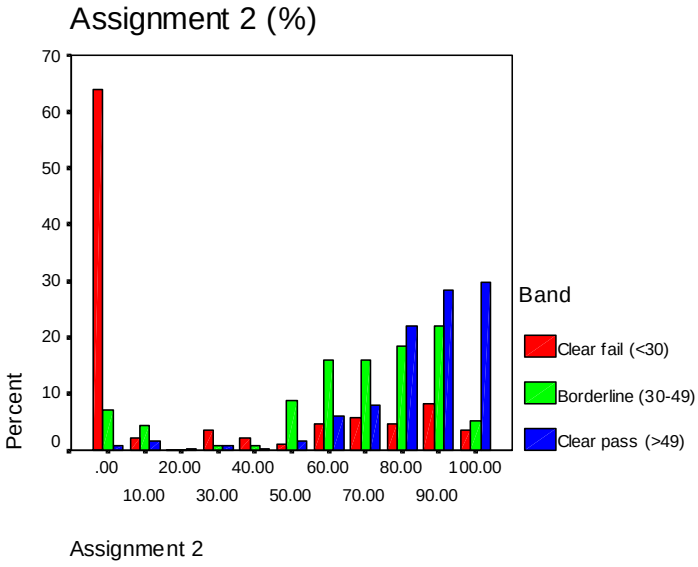
Supervised assessment tasks

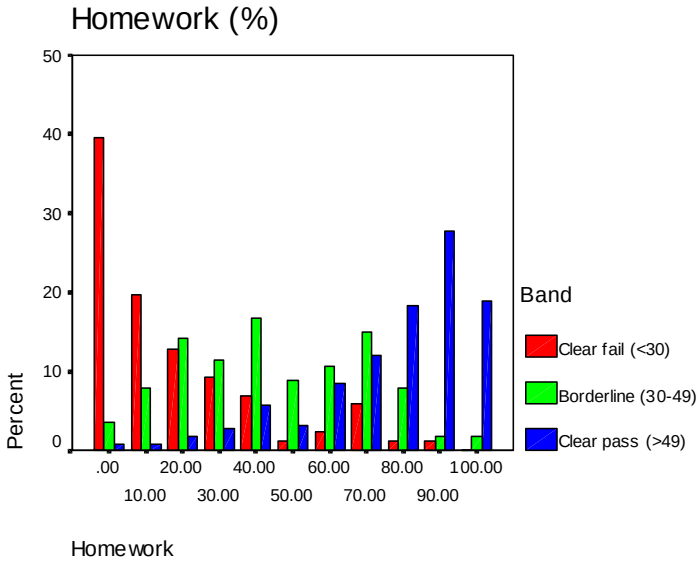




Unsupervised assessment tasks







The following observations are extrapolated from the graphs and tables presented above.

- On average, students banded in the "clear fail" or "clear pass" categories achieve results consistent with those results on all assessment tasks (failing or passing all tasks, respectively). However, students banded in the "borderline" category exhibit both pass and fail results. This lends credibility to the categorisation, especially given the intention to identify areas of concern for currently borderline students. These are students who are doing well in some tasks, but are at risk of failing due to bad performance in others.

- Students in the "borderline" category exhibit clear pass results in the assignments, borderline results for homework, and unsatisfactory performance in both the exam and the quizzes. Thus, the aforementioned difference between performance in supervised versus unsupervised tasks is confined to this section of the population when divided into bands. Furthermore, the mean proximity of "borderline" students is noticeably closer to that of "clear fail" students than "clear pass" students for the exam and the quizzes; the opposite is true for assignments 1 and 2 when they are closer to the "clear pass" students.

- The dispersion of results for "borderline" students is considerably larger for the exam than other students.

Examining the correlation between performance outcomes for each of the three bands produces some interesting results[*]. These are summarised by band:

- *Clear pass:* Students in this band exhibited significant positive correlation between all assessment tasks. It may be speculated that this might be due to a high level of engagement with the material.

- *Clear fail:* Students in this band exhibited significant positive correlation between all non-examination assessment tasks. That is, exam performance did not correlate to any other task (including quizzes) but performance in all *other* tasks showed significant positive correlation to each other. Given that results were uniformly poor for each task this correlation is not particularly meaningful. The lack of correlation between the exam and any assessment tasks may be more meaningful for "clear fail" students. A negligible level of engagement may be responsible for this "signature".

- *Borderline:* Students in this band showed an unusual correlation of performance across assessment tasks. First, performance showed much less correlation between tasks than other bands. The only significant positive correlation was between the homework and the quiz (both of which were designed to be related tasks and also were associated with recent lectures). The second point of interest is the presence of significant negative correlation (which did not occur in other bands). Performance in both of the assignments was negatively correlated with performance in the exam! The final point of interest is that there was no positive correlation between the two assignment tasks (something found in the other two bands, and something that would have been expected regardless). It is difficult to offer an explanation for these anomalies without drawing on the possibility that many students in this group did not engage with the assignment tasks (a crucial part of the learning experience for the unit). Given strong results for the assignments by students in this group, one is drawn to consider the prospect that students may have resorted to some levels of sharing of solutions without engagement. This possibility would be reduced were the exam questions quite different from the assignment material; however, as programming assignments they are likely to be quite influential on understanding the deeper questions presented in the exam (something that the correlation of performance for "clear pass" students demonstrates). The issue of learner engagement was one that had been raised during semester, albeit founded on anecdotal evidence. Anomalous performance results, such as those witnessed here, may be a manifestation of those concerns.

---

[*] Significance defined at the 0.95 level.

Exam

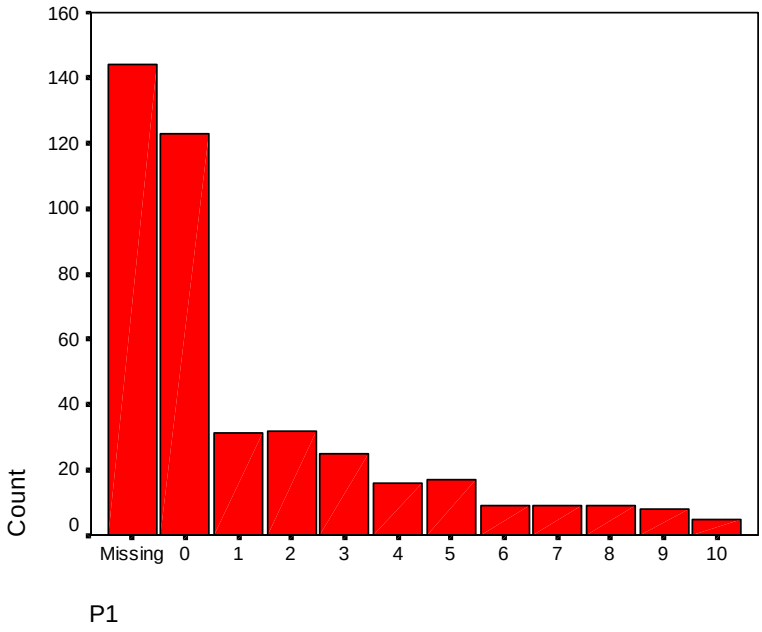
428 students participated in the exam. Nine exam marks were submitted to the marking software. Mean marks, standard deviations and participation rates are summarised in the following table:

Question	*Weighting (%)*	*Mean (%)*	*Standard deviation*	*Non-participants*
1	15	63	3.5	0
2	10	85	2.2	0
3	10	48	2.5	0
4(a)	10	83	2.3	0
4(b)	15	52	3.4	11
5	10	48	2.4	0
6	10	12	2.0	6
7(a)	10	27	2.8	33
7(b)	10	22	2.8	144

Note that the mean results reported here exclude non-attempts (but include attempts that were assessed as 0).

Questions 6 and 7 showed very low levels of success, and questions 3, 4(b) and 5 showed only moderate levels of success. These questions presumably identify issues that have the potential to prevent students from passing the course; more on this later. Note also that questions 1 and 4(b) exhibited the highest levels of variation in marks; it may be worth looking also at the nature of question 1 with respect to failure rates.

Question 7 (especially part b) also had high-levels of non-participation. In exploring its role as a discriminator question the following graph shows the spread of marks for question 7(b):



The number of failed attempts (0 marks) is almost equivalent to the number of non-attempts (labelled as . "missing"). However, looking at the overall exam performance of students who made "serious" attempts (marks > 0) for question 7(b) we find *over 95%* of them achieved a raw mark of 40 or more in the exam. The following table summarises the overall performance of those students:

Question	*Mean (%)*	*Standard deviation*
1	75	2.4
2	92	1.6
3	60	2.5
4(a)	92	1.2
4(b)	63	3.2
5	60	2.2
6	20	2.6
7(a)	43	2.6
7(b)	39	2.6

In all cases mean performance shows strong improvement. Note that question 7(b) is one which asks students to demonstrate considerable proficiency in programming and a more integrated sense of understanding than most other parts of the paper.

When looking at the correlation between final exam mark and performance in individual questions there are a few possible surprises. Question 4(b) has the strongest correlation (.761), followed by questions 1 and 7(a) (.72 each), question 7(b) (.711) and question 5 (0.709).

The following table shows means student performance in the exam by groups. Note that groups are defined using the same ranges, but based on *total exam marks* rather than total assessment marks.

	Mean (%)		
	Clear Fail	*Borderline*	*Clear Pass*
1	33.3	54.7	78.2
2	49.6	83.4	96.2
3	20.9	37.4	63.7
4(a)	42.5	81.3	94.1
4(b)	15.1	44.8	64.8
5	19.0	38.8	63.2
6	1.9	5.6	20.2
7(a)	0.6	9.8	44.1
7(b)	0.4	2.5	29.3

It is clear that the only particularly strong results for “borderline” students are questions 2 (which involves tracing some basic pointer code and drawing an associated memory model) and 4a (writing some basic list traversal code). It is not apparent as to why these questions rate so well, although it is worth mentioning that students were allowed to bring in a single sheet of handwritten notes into the exam, and sample list traversal code might have conceivably been well represented in those notes.

Section B: The Exam

This discussion is based on a sample of 80 students' exams; access to these exams enabled a qualitative exploration to be conducted (revealing common errors and potential misconceptions that may generate them), as well as a finer granularity to be applied to the quantitative analysis.

Performance correlation between questions

The following table shows significant correlation of performance for exam questions; *all correlation was positive*. Solid circles represent questions with a strong correlation (0.01 level of significance); open circles represent questions with a correlation at the 0.05 level. Tests for significance are 2-tailed.

	1a	1b	2a	2b	3a	3b	3c	4a	4b	5a	5b	5c	6	7a	7b
1a	-							●				●			
1b		-	○			●		○		○	○	○	○	●	●
2a		○	-			○			○			○			
2b				-		●		●	●			●		○	○
3a					-		○	●	●		○				
3b		●	○	●		-		●	●			●	○	●	○
3c					○		-				○				○
4a	●	○		●	●	●		-	●		○	●		●	
4b			○	●	●	●		●	-			○	●	●	
5a		○								-	○	●			
5b		○			○		○	○		○	-	○			○
5c	●	○	○	●		●		●	○	●	○	-		○	
6		○				○			●				-	●	
7a		●		○		●		●	●			○	●	-	●
7b		●		○		○	○				○			●	-

Relevant sections of this table are reproduced in the discussion below for clarity.

While it is difficult to impose a consistent reduction of the information conveyed above, a couple of points are worth noting:

- As one might expect, there was a strong positive correlation between many of the code writing questions (questions 1b, 4a, 4b, 6, 7a and 7b).

	1b	4a	4b	6	7a	7b
1b	-	○		○	●	●
4a	○	-	●		●	
4b		●	-	●	●	
6	○		●	-	●	
7a	●	●	●	●	-	●
7b	●				●	-

- Less expected was a *lack* of significant correlation between the code reading/tracing questions (questions 1a, 2a and 3a). Examining these questions confirms that they may be qualitatively different (successive questions involving increased levels of subtlety).

	1a	2a	3a
1a	-		
2a		-	
3a			-

- Questions relating specifically to the memory model (questions 2b, 3b and 3c) only showed significant correlation questions 2b and 3b. Question 3c does appear to test a deeper understanding of the memory model.

	2b	3b	3c
2b	-	●	
3b	●	-	
3c			-

- There is significant correlation between questions 3a and 3c. The former is a code tracing exercise; the latter tests understanding of the memory model applied to the same code segment given in 3a. Although it is purely speculative this raises the possibility of a relationship between deep understanding of the code (as evidenced by success in answering the memory model question) and the ability to trace code, especially that involving pointers. The argument is supported by noting the correlation between the "strong" memory model question (3c) and the most involved of the programming questions (7b). To the extent that this is true, it is also unsurprising; that "complex" programming tasks are better performed by students who are capable of visualising the underlying memory model appeals to intuition. Memory was given emphasis in this unit for exactly this reason (and may be especially important given that students came from a Java background where the importance of this conceptual model is largely hidden).

	3a	**3c**	**7b**
3a	-	○	
3c	○	-	○
7b		○	-

- Not surprisingly, performance in all three UNIX questions (5a, 5b and 5c) exhibited significant levels of positive correlation.

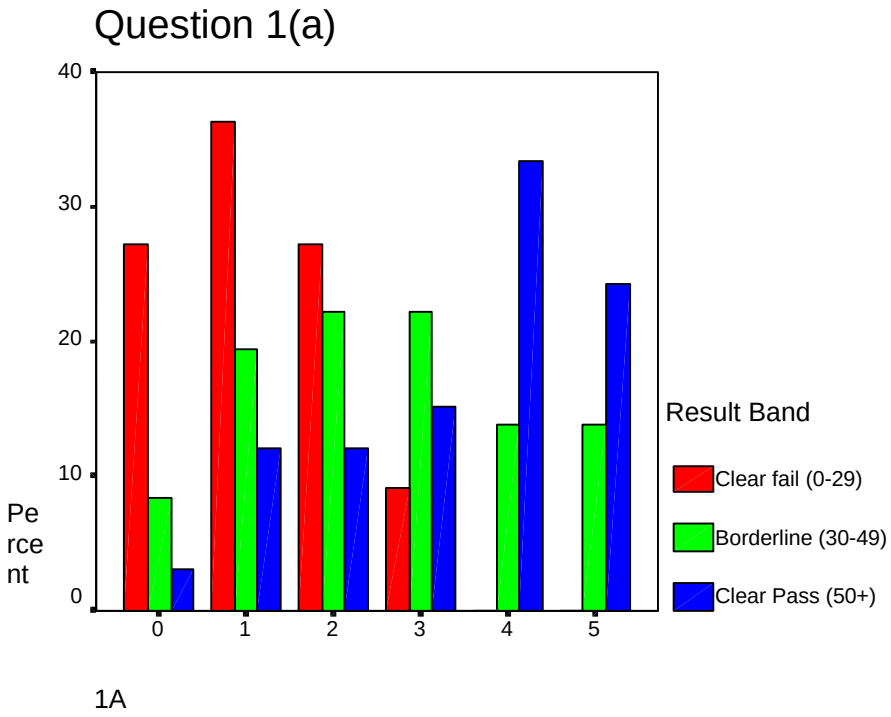
	5a	**5b**	**5c**
5a	-	○	•
5b	○	-	○
5c	•	○	-

Exam questions

This section discusses each of the exam questions in turn. Given special interest in improving the learning outcomes for students that are currently borderline pass/fail students, the quantitative results are expressed in three performance "bands", based on final exam mark: "clear fail" (0-29), "borderline pass/fail" (30-49) and "clear pass" (50+).

In those questions with a qualitative discussion, numbers in bold indicate the count of instances of a particular error. The use of the term "error" is used to represent a visible artifact within the answers; the word "misconception" is used to represent an interpretation of those errors in order to suggest problems within the conceptual domain that might lead to the errors witnessed. This raises the possibility of confronting likely underlying misconceptions within the teaching, rather than simply drawing awareness to singular error-prone aspects of programming.

Question 1(a)



Band	Mean	Std. Dev.
Clear Fail	1.2	1.0
Borderline	2.6	1.5
Clear Pass	3.4	1.4

This was a code-tracing question (worth 5 marks) that involved understanding recursion and integer division, and complicated by the need to recognise the absence of a termination condition for full marks. The "borderline" group was distributed across the marks range.

There were two (related) components to this question: (a) reporting the first five lines of output, and (b) describing the number of lines of output that would be produced. The complicating factors are the use of integer division and the use of recursion; the recursion does not include a terminating condition so the output is "infinite" (limited only by resources).

The two components reveal different issues in the answers, so are dealt with separately.

Errors and misconceptions (Part 1: First 5 lines of output)

- *Treating the post-increment operator as a pre-increment operator* ***(22)***: This was a common error. Two possible misconceptions suggest themselves. (a) It may simply be the case that students do not

recognise the difference between the effect of the pre-increment operator and the post-increment operator. This misconception may be derived from the observation of common idiomatic cases, in which pre- and post- increment operations may appear isomorphic. (b) Students recognise the difference between pre- and post- increment operators, but are confused by the context (use within a *printf* function), interpreting it outside of the function (which, again, suggests that both forms are isomorphic *in this context*).

- *Failure to recognise / understand integer division **(16)***: A large number of students neglected to treat the division operation as an *integer* one, fractional components in their answers. Students either did not recognise that such a conceptual issue exists (which could only suggest insufficient C programming experience) or they neglected to give sufficient attention to the type of the variable '*value*'. The former might be placed in the "unforgivable" category, suggestive of low levels of engagement. The latter may be regarded as somewhat more reasonable, given exam pressures and the fact that the difference is dependent on variable type and not operator type.

- *Early termination **(7)***: A number of students reported less than five lines of output (the actual number depends on how legitimately they treat the post-increment operator). This early-termination error may be "triggered" at the moment that '*value*' is an odd number in the call to '*drab(value / 2);*'. It involves recognition that the function *drab( )* takes an integer value, but confusion over the nature of integer division leads to a situation where *drab( )* is passed a floating-point variable; as this is illegal, the program terminates (one answer mentioned an 'abnormal termination' which provided the clue to the thinking that leads to this error condition). Although there is a clear misconception involving the nature of integer division (as referred to above), this error suggests another area of misconception involving the *typing* of variables. Variable typing is clearly relevant to detection of integer division except that in the previous context we surmised its importance whereas in this context we can see its influence more directly.

Errors and misconceptions (Part 2: Number of lines of output)

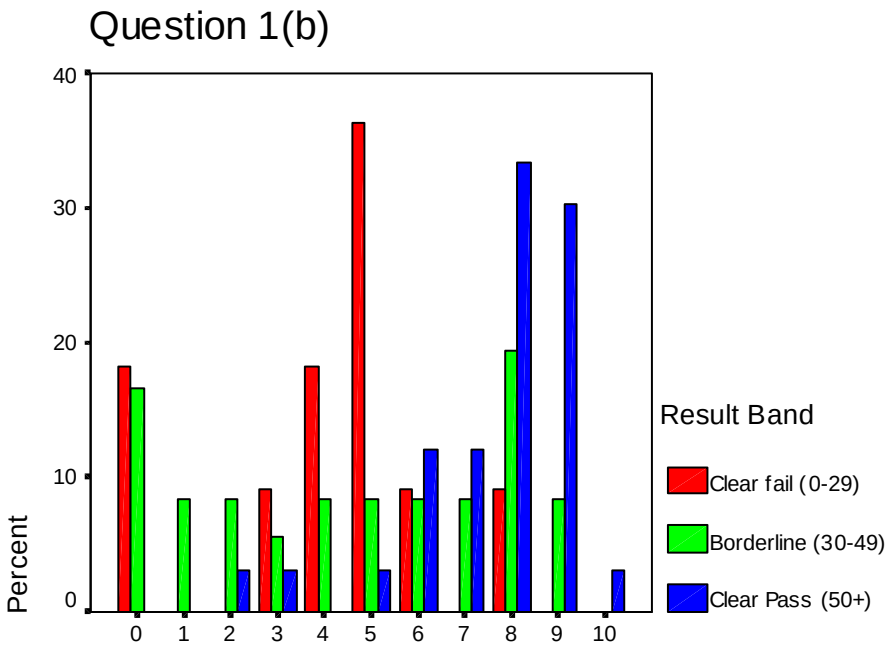
- *4 iterations **(13)***: What is immediately interesting about the fact that 13 students stating only 4 lines of output were produced is that only 7 students suggested early termination in the first component of the same question! This was evidence of clear contradiction in understanding the same code segment despite being asked fundamentally similar questions. The 4-iteration error seems based on the misconception that the print conditional '*if (value < 5)*' somehow represents a termination condition, and that the code therefore terminates after the "*drab too small to print*" output (in one case this was explicitly stated, providing the insight for misconception behind this error).

- *5 iterations **(1):*** This report will not typically report on "one-off" errors (only 1 student made this error). However, this is another (if slightly even less legitimate) version of the above misconception. In this case it seems that the student was imagining the '*value < 5*' conditional to suggest some kind of loop counter!

- *Infinite due to flawed logic **(2)***: Two students stated that the infinite nature of the code was due to failure of a termination condition. For example, in one case it was stated that this was because integer division meant that ' 3 / 2 = 2' (!) and that the code therefore never terminated; presumably, they somehow defaulted to thinking that if '*value*' ever reached zero then termination would be effected.

All of the above errors reflect the same misconception: failure to recognise lack of termination condition. In the last error, only two students were identified. However, it is an important observation despite the small numbers. The error was only detected because they added additional information (as the answer "infinite output" was a sufficient answer). Therefore, it is entirely possible that this misconception is more widely held than the 16 cases identified above, even among those students who received full marks for the second component of the question.

Question 1(b)



1B

Band	Mean	Std. Dev.
Clear Fail	4.1	2.4
Borderline	4.6	3.2
Clear Pass	7.6	1.7

This question (worth 10 marks) involved writing a short C program that would read lines from standard input and print every second line to standard output. There was little coherence to the performance of the "borderline" group, and the mean result resembled that of the "clear fail" group. This was a relatively simple piece of code, so one would anticipate that students performing badly in this question would be in serious trouble so far as passing the unit is concerned. Furthermore, some questions would have to be raised about the level of engagement by such students (especially given the likelihood of an "engagement" issue raised previously in relation to "borderline" students).

Before identifying the errors associated with this question, it is interesting to note the "algorithms" used by students to achieve the toggling effect (considering only those cases that were correct or "almost" correct):

1. The vast majority of answers used the "modulo 2 approach", incrementing line numbers and printing the current line if '*(line % 2)*'.

2. The "boolean approach", explicitly toggling between even and odd lines; 8 students used this approach. Most simply declared a boolean-like variable (not present in C) and treated it as such using

the *not* operator, '*bool = !bool;*'. A couple used the '*if … else…*' control structure to effect the toggle. One used a 'multiplication by minus 1' approach.

3. The "read twice, print once" approach (7 students), such as '*while (gets(line)) { gets(line); printf("%s\n, line);}*'. Typically such answers did not re-check for EOF on the second read.

Errors and misconceptions

- *Lack of main( ) function* ***(21)***. This may be a little pedantic, but was also the most common error (over ¼ of the sample), given that students were asked to write a program and not a code segment. This is probably more likely to be founded on oversight than misconception, although with a background in Java it is *possible* that low-engagement students may be confused about what constitutes a minimalist program.

- *Unable to handle arbitrarily long input* ***(14)***: This was usually highly visible due to the forced use of a *for* loop to handle a specified number of lines.

- *Use of strings without <u>any</u> memory allocation* ***(12)***: Typically this involved the use of pointers to characters without any attempt to allocate storage. This may well be an oversight, rather than a misconception, but the use of strings may well be motivated by the desire to make use of the convenience of *scanf("%s"…)* (or other string-reading functions) to read entire strings instead of characters.

- *Reasonable-sized, but off-topic answers* ***(11)***: Many of these were quite bizarre solutions, as they were C programs but deliberately not addressing the question asked! In fact, 3 of them were identical copies of the provided code for question 1(a)!!! One of them was code for a linked-list traversal (probably relating to a later question on linked lists). The only explanation I can muster is that, although the question does not state it, students recognised that there were significant marks for programming style associated with this question (which was true). The guess about this behaviour is that students were hoping for these stylistic marks even though the question was entirely avoided! It is hard to imagine any other basis for this behaviour other than strategic exploitation of the marking system.

- *Use of strings with arbitrarily fixed memory allocation* ***(10)***: Once again, this is probably motivated by the convenience of reading entire strings as above (except that in this case an attempt was made to allocate memory). This "question modification" is quite visible throughout code-writing aspects of the paper more generally.

- *Failure to increment 'count'* ***(10)****:* This is probably oversight, and certainly readily corrected (by experienced students) at a terminal. In cases where students opted for the "modulo 2" approach, this error is to simply forget to increment the line counter, which would result in either all lines being output or none (depending on treatment of the line counter).

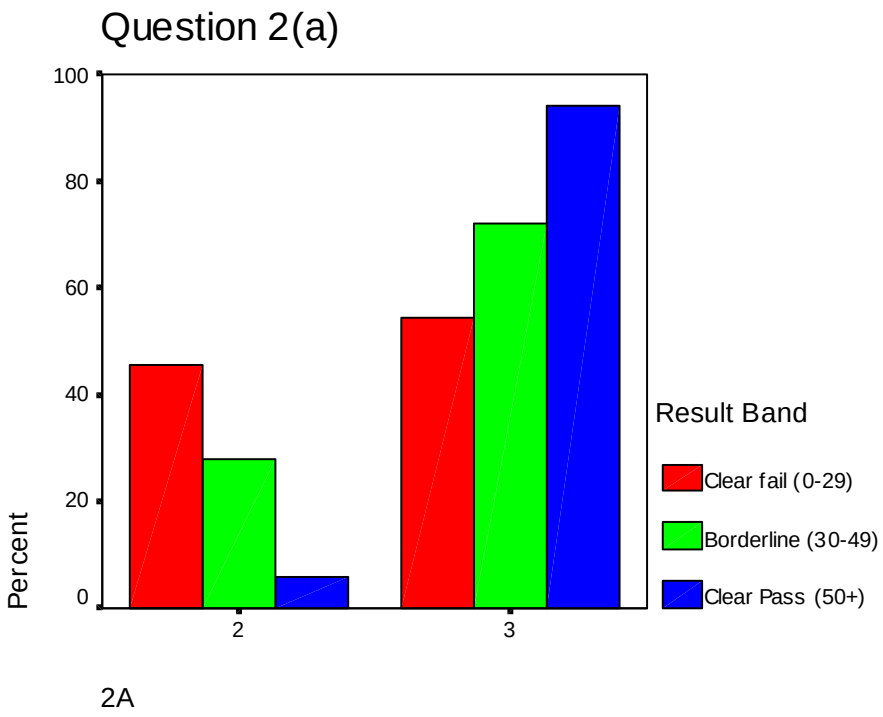
- *Confusion as to source of input* ***(10)****:* Quite a lot of students interpreted the question to require input from other than *stdin*. Five students attempted solutions that read from regular files, and five students attempted solutions that read from the command-line. In the case of the latter, this *may* possibly be driven by a desire to simplify the question (as entire strings are readily available via '***argv*'). Otherwise, this may simply be a matter of confused thinking under exam pressure.

- *<u>No attempt</u> to toggle output lines* ***(9)***: A number of answers simply avoided even making the effort to only output every second line, although other aspects of the code were reasonable. A number of these students still achieved high marks (of 8 or 9 marks out of 10) for the question.

- *Failure to even attempt to recognise the EOF condition* ***(8)***: In those solutions that used *stdin* (as required by the question) 8 attempts avoided detection of the EOF.

- *Use of division operator instead of modulus operator **(5)***: A number of students who used the “modulo 2” approach made the mistake of using integer division instead of integer remainder (modulus).

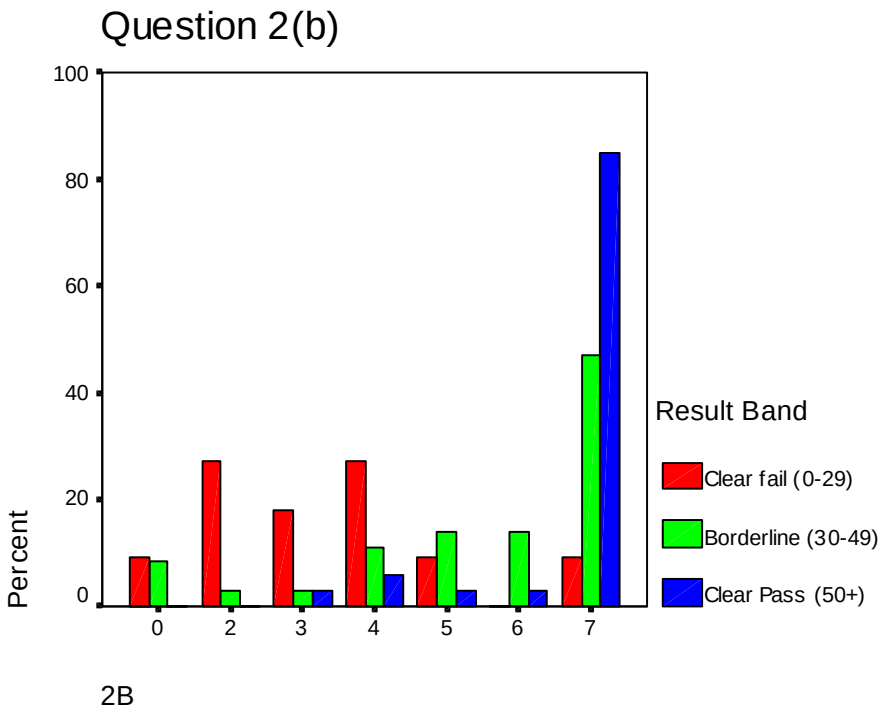
Question 2(a)



Band	Mean	Std. Dev.
Clear Fail	2.5	0.5
Borderline	2.7	0.4
Clear Pass	2.9	0.2

This is a three mark question that tests ability to trace simple pointer code. The ability to use pointers was an important learning outcome in the unit, and one that traditionally frustrates students of C. It was therefore satisfying to see that although this question received a minor marks allocation it was well done by all students. The point needs to be made that even students who have here been placed in the “clear fail” category are exhibiting clear understanding of the fundamentals of the use of pointers.

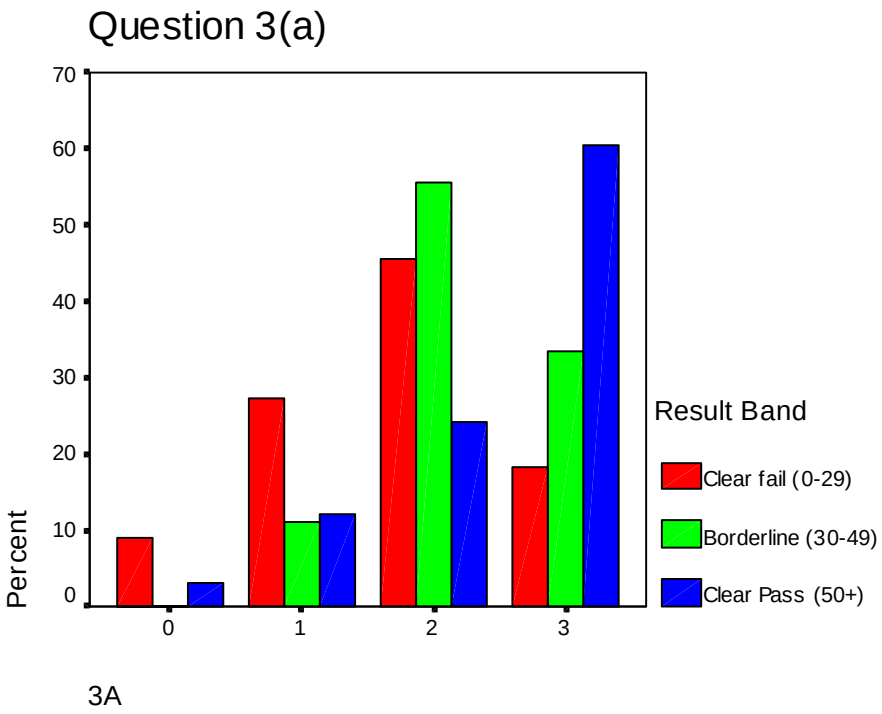
Question 2(b)



Band	Mean	Std. Dev.
Clear Fail	3.3	1.8
Borderline	5.4	2.1
Clear Pass	6.6	1.0

This (7 mark) question extended the previous question on tracing some basic pointer code by asking students to draw and label memory associated with that code at the time of its completion. Again, this was a fundamental learning outcome for the unit and it was pleasing to see that students were able to demonstrate that they had understood these concepts.

Question 3(a)



Band	Mean	Std. Dev.
Clear Fail	1.7	0.9
Borderline	2.2	0.6
Clear Pass	2.4	0.8

This three-mark question tested students' abilities to trace a segment of code that made heavy use of pointers including a simple string traversal loop. In general, the majority of students were able to demonstrate that they possessed the basic skills required to perform such a task.

There were many erroneous variations on the output of this code which, when considered in terms of the thinking required to generate such errors, suggest a number of misconceptions. Each of the following listings represents errors made by individuals (whereas in the listings above individuals may make multiples of the errors noted).

Each error listed below is mapped to a misconception that is internally consistent with the error, and may be responsible for the misinterpretation. Detection of such misconceptions is extremely valuable as they are typically invisible but play havoc with students' abilities to read or write code. In this question, almost 50% of students (39/80) in the sample demonstrated evidence of conceptual issues associated with pointers. On one occasion (mentioned below) the suspected misconception was actually made explicit in an explanation by a student. However, this question lends itself to fairly reliable "guesstimates" about student thinking in all cases, as there is negligible code between print statements.

- *Lines 1 and 3 OK, but line 2 contains the address of 'A'* ***(15)***: We must be careful in believing that this represents a misconception about the behaviour of pointers, as the same students correctly predicted the output of line 3 which requires greater understanding of typical pointer operations. Thus, it is anomalous to state that the conceptual difficulty hinges on the assignment statement '*pA = A;*' (the same operation must be understood in order to trace line 3) even though this is the only statement since the previous *printf*. However, line 2 (of output) is the only line that uses *printf* to print a string argument referenced by an "explicit" pointer (a variable declared using pointer notation, '*char *pA;*'). It is potentially easy for students to form the idea that *printf* does not use *pointers* to strings as arguments (which is a serious misunderstanding of how string variables are referenced, and how arrays and strings are related[*]). For example, the output of lines 1 and 3 use strings declared as arrays (despite pointer operations on the contents of the array in the case of line 3). Thus, students generating this error seem comfortable with the use of *printf* in situations such as:

  ```
  char A = "Fred";
  char B[80] = "Freda";
  .
  .
  .
  printf("%s and %s\n", A, B);
  ```

  However, the same students are having problems with:

  ```
  char A = "Fred";
  char *B;
  .
  .
  .
  B = A;
  printf("%s\n", B);
  ```

  The misconception that *printf* does not take "pointer" arguments (really meaning pointers that "look like" pointers) is essentially a misconception about the naming of array variables, in which the variable '*A*' (above) is actually a shorthand notation for '*&A[0]*' (i.e., that names of arrays and strings *are* pointers). In other words, there is a possibility that students are left confused by notational conventions. Perhaps, according to this world-view, the use of a pointer as an argument causes *printf* to output the address of the string, based on the knowledge that pointers (unlike "array variables"!) contain addresses. Clearly, there is *also* a misconception about the compiler's response to specification of the format string as "%s" when *printf* is given an address argument.

- *Lines 1 and 3 OK, but line 2 contains the single character 'S'* ***(6)***: This error suggests that the assignment '*pA = A;*' results in a copy operation of the first character of the string '*A*' to the (pointer) variable '*pA*'. This requires a suspension of understanding of the nature of pointers (treated as a *char* rather than as a *pointer to char*). It is difficult to cast this in terms of a more internally consistent framework except to state that it may possibly be a similar misconception to the one expressed above, with a different effect from *printf* when it is passed a pointer argument. Assuming this is the case, over 25% of the sample have potentially been affected by this sort of misconception. Some indication of the confusion involved comes from one student who justified their response by stating: "Can't print as *pA* pointed to first element in string (i.e., 'S') and we're trying to print a string". In this case, the comment would seem to refer less to the fundamental nature of pointers (to store the address of a variable) than

[*] I wonder if prior exposure to *Java* makes a difference here; as an object-oriented language it becomes easier to visualise a string as a unit rather than a contiguous array of characters. C may require a "change of focus" in order to properly recognise the "microscopic" issues that are hidden in a language such as *Java*.

to the ***use*** of pointers, again showing misconceptions about how *printf* would handle the address of a character when asked to print a string!

- *Lines 1 and 2 OK, but line 3 explicitly denoted as being 'spaces' or 'garbage' or '?'* ***(6)***: This error may be generated by a misconception about the nature of the '*' operator to access content. The second line of output is correct, so we know that students are not having problem with the assignment statement '*pB = B;*' (the second line requires an understanding of this statement). However, not understanding that pointer operations within the *while* loop result in content being copied into the address space of '*B*' leaves that variable (declared as '*char B[80];*') full of undefined content. Some students may regard this uninitialised content as defaulting to "spaces", indicating a further possible misconception about the behaviour of C with respect to uninitialised variables (Java takes a different approach to C by using default values for uninitialised variables).

- *Lines 1 and 2 OK, but line 3 missing* ***(4)***: One possible cause of this error is a failure to properly account for the effect of the increment operator on the left-hand side of the assignment statement in the *while* loop; the effect would be to create an empty string. Otherwise, there may be a conceptual issue along the lines of the previous error.

- *Line 1 OK, but lines 2 and 3 missing* ***(3)***: It is possible that the lack of output for lines 2 and 3 is simply the result of uncertainty about what happens past the first line of output. That is, an understanding of array notation for strings but confusion when dealing with pointer representation.

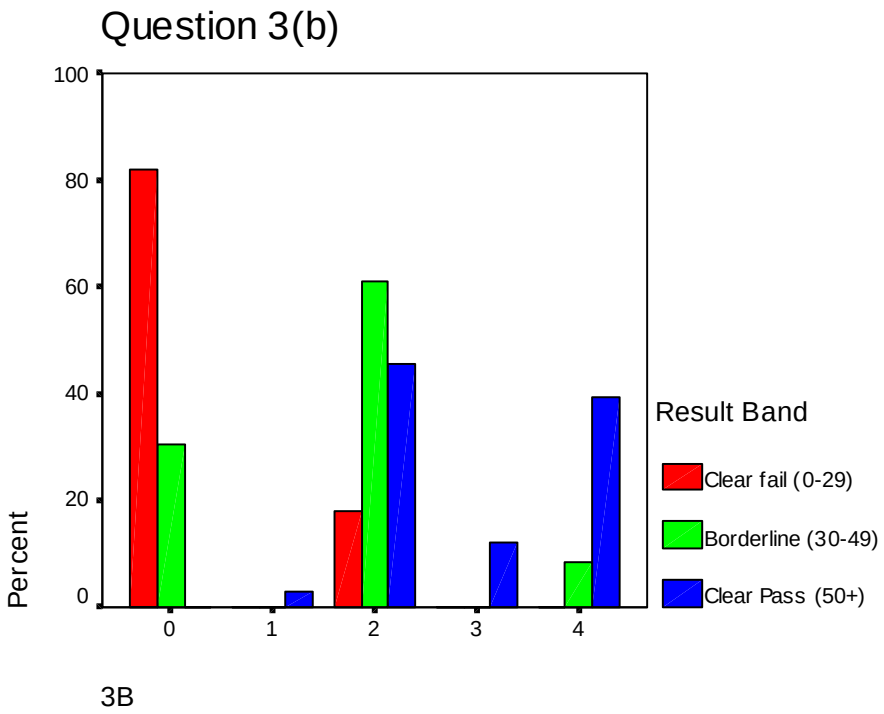
- *Lines 1 and 2 OK, but line 3 includes a space character* ***(2)***: One of these answers showed a space prepended to the string initially represented by 'A' ("<space>Soft2004…"), whereas the other showed a space that replaced the first character of that string ("<space>oft2004…"). The misconception here seems to center on some confusion about the synchronisation (order of operation) of post-increment operators when used in the following idiomatic expression: '**pB++ = *pA++;*'

- *The single letter 'S' on each of three lines* ***(1)***: This error may be generated by failure to recognise how *printf* handles the "%s" format string by iterating over multiple characters until a terminating character is detected, or by simply misinterpreting the format string to be "%c".

- *Lines 1 and 3 OK, but line 2 contains the single character 'A'* ***(1)***: May be motivated by the same confusion over pointer arguments to *printf* as mentioned in the first (most common) error, although in this case it is accompanied by a (bizarre) interpretation of the preceding assignment operation.

- *Line 1 OK, but lines 2 and three contain addresses of variables* ***(1)***: This may be generated by the same misconception as the one associated with the first (most common) error, except that the student also interpreted the *while* loop as copying the contents of '*pA*' to '*pB*'. This results in a propagation of the error to output line 3.

Question 3(b)

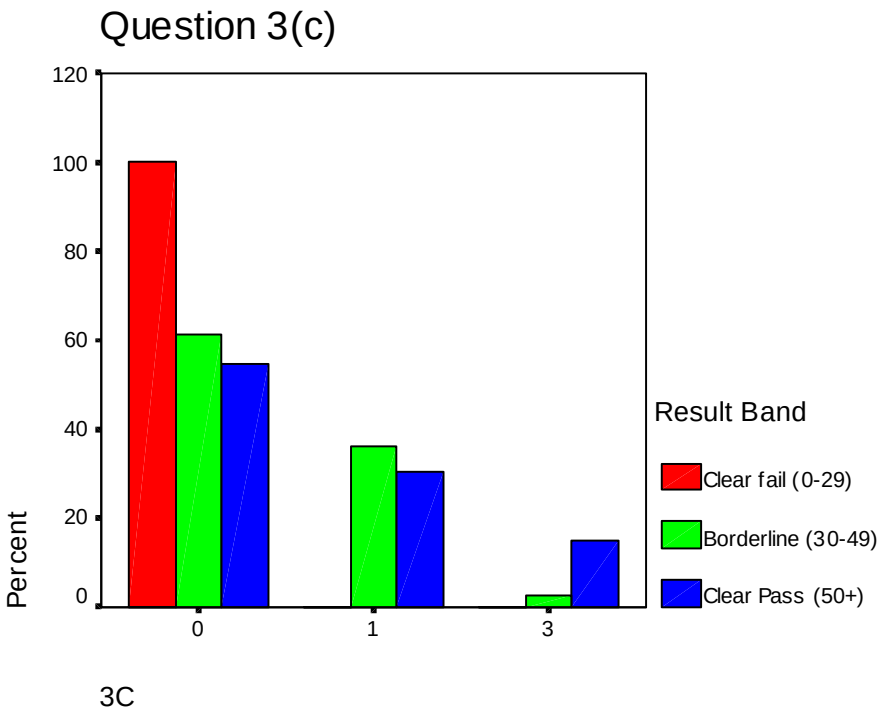


Band	Mean	Std. Dev.
Clear Fail	0.4	0.8
Borderline	1.6	1.2
Clear Pass	2.9	1.0

This four-mark question asked students to produce a memory diagram of the previous code, and was reasonably well done when it was seriously attempted. There was some evidence of confusion about heap memory and stack memory, reflected by apparent guesswork or (more commonly) simply avoiding the task by neglecting to label the memory.

However, there was generally very good evidence of the ability to correctly locate pointers to appropriate memory locations.

Question 3(c)



Band	Mean	Std. Dev.
Clear Fail	0	0
Borderline	0.4	0.6
Clear Pass	0.8	1.1

This three-mark question explored more subtle points of the memory model, requiring students to identify a buffer over-run problem in order to achieve full marks. This additional level of subtlety was reflected in the performance separation between the three groups.

An interesting aspect of the answers to this question is that only 9 mentioned the memory over-run problem which (presumably) the question was designed to test.

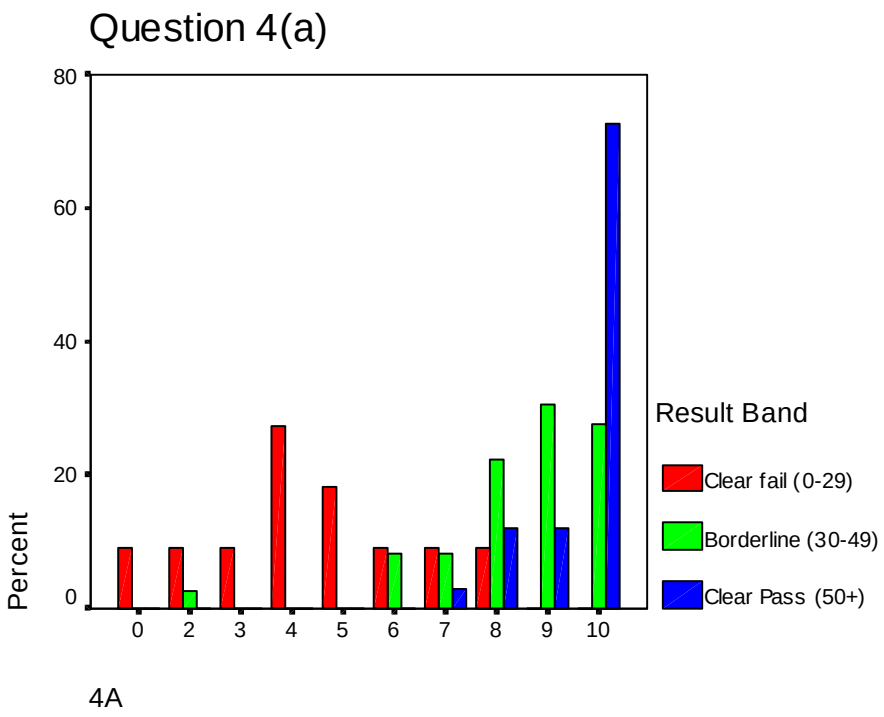
- *No effect **(29)***: That is, those answers that show the contents of '*B*' as "*Q3 string*". The strong showing of this error was initially quite puzzling. On reflection, however, it may suggest an ambiguity in the exam question. The question states "*…Now redraw your diagram in the space below…*", which in the flow of context from question3(b) tends to indicate that this should illustrate the state of things *after* program execution. However, there is a possibility that many students interpreted the question to mean draw the memory model *prior to* code execution. While this may not seem realistic, given the context stated in question 3(b) and given that this misinterpretation trivialises the question, it is difficult to explain the large presence of this error without this diagnosis.

- *Belief that an issue-free copy takes place of contents of 'A' to 'B'* ***(16)***: These answers recognised the functionality of the *while* loop to copy a string from one location to another. However, they neglected to look beyond the code at the fact the declaration '*B[]="Q3 string";*' automatically assigns memory space which is breached by the copying process. As mentioned above, only 9 students (around 12%) noticed the buffer over-run problem, so this misconception is more widely spread than suggested by this particular error. Although we are interested here in misconceptions, it is worth noting that students producing this error also lacked a strategic sense that examiners rarely test the same concepts, and under their interpretation the output was exactly that of question 3(a). No "alarm bells" went off to suggest that by the very fact that the question was being asked, this simple modification to the earlier question must hint at a problem, a problem that had been covered in the unit.

- *The reduced size of 'B' is recognised, but in the process of copying 'A' to 'B' the contents are "auto-magically" truncated at the designated length of 'B', including the appending of a string termination character* ***(4)***: A number of students recognised the memory issue, but demonstrated a misconception about how it would be handled in C. Again, this may reflect influences from *Java* exposure. In this case, they recognised the 10-character allocation (9 characters plus terminator). However, they treated the copying process as exercising some "intelligence" in preventing the buffer overflow (something that C is notorious for *not* doing). The effect of copying was that the contents of the destination string were regarded as: "*Soft2004<space><NULL>*". This is quite anomalous given the explicit nature of the code in the *while* loop and the fact that these students clearly understood its functionality. However, these students were among the few that recognised the buffer over-run threat posed by the new declaration; therefore, this misconception may have been more widely evident had more students noticed the buffer over-run problem in the first instance.

- *An issue-free copy of 'A' to 'B' takes place (as above) but the first letter of 'B' (i.e., 'Q') is strangely preserved* ***(2)***: This is essentially the same problem as the most common error for this question, compounded by the misconception (noted in question 3a) about the order of operation of statements like '**pA++ = *pB++;*'.

Question 4(a)



Band	Mean	Std. Dev.
Clear Fail	4.4	2.2
Borderline	8.4	1.6
Clear Pass	9.5	0.8

This 10-mark question asked students to write some list traversal code that also performed some basic processing. In this question, "borderline" students perform close to the level of "clear pass" students.

This question was generally well done; however, the traversal of a linked list is fairly idiomatic code (and students were permitted a page of hand-written notes to be taken into the exam), so one might expect this level of success. It is interesting, although not necessarily very meaningful (under exam conditions), that only 2 answers attempted to handle casing of letters.

- *The mysterious use of memory allocation (malloc or realloc) within the function **(5)***: Any attempt to allocate memory within the function demonstrates a clear misunderstanding about what is taking place in a list traversal. It is difficult to sense what the precise nature of such a misconception might be, but it must involve a lack of recognition that list traversal is simply moving through structures that have already been allocated memory! It should be mentioned that the code did *not* look as if students had somehow misinterpreted the question as one involving insertion. However, it is entirely possible that these students included list insertion code segments on their page of handwritten notes; in that case, we

may simply be witnessing a rather disengaged attempt at massaging a solution for one problem into a different one.

- *Performs multiple tests in one conditional **(5)***: A number of students demonstrated a perverse shorthand notation for filtering characters along the lines of '*if (current->name[0] == a || e || i || o || u)*'. Given that the exam instructions do refer to marks for correctness and style, it is hard to believe that students who knew better would risk marks for a couple of minutes of extra writing. Therefore, it is assumed that students believed that the notation would actually compile and work. This is a somewhat superficial misconception, but one that suggests low engagement in programming generally (not just this unit). One would expect that most students with programming experience would not have produced this "English-like" shorthand notation.

- *Task avoidance – the function is treated as one in which the current node pointer is passed as an argument by the calling function, neatly avoiding any list traversal operation **(4)***: This error does not represent a misconception, but is an interesting observation. It is unlikely that students believed the question required them to exercise *no* list traversal. To the extent that this is true, students producing this error *elected* to believe that the question was one of writing some code to check whether or not an element of a given structure started with a vowel.

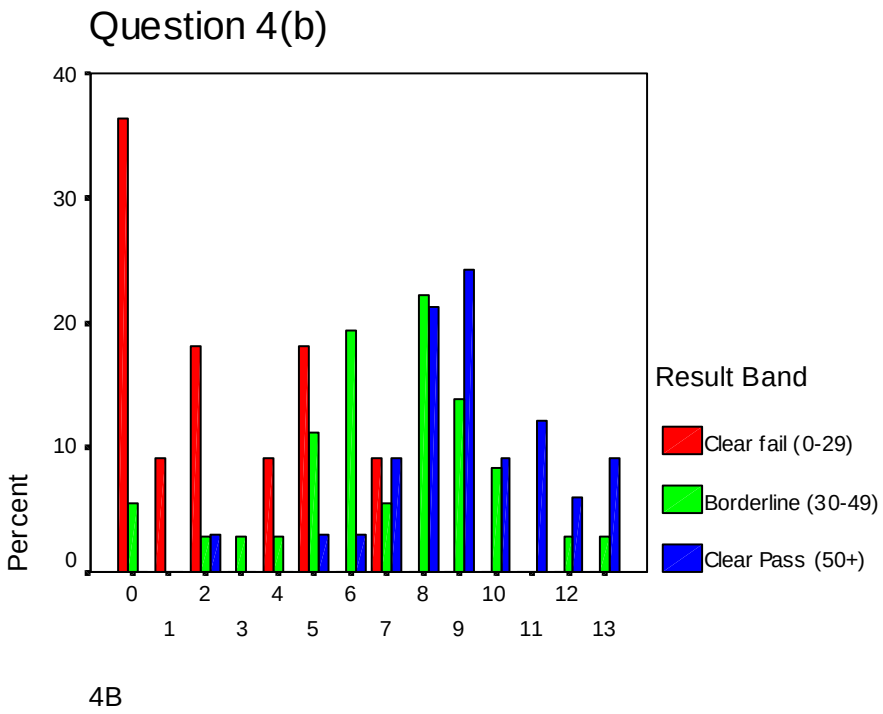
- *Tests the string instead of the first character of the string **(3)***: A number of students neglected to isolate their testing to the first character of the string, producing code similar to: '*if (start->name == 'a')*'. It suggests little exposure to string manipulation, but may simply be reducible to "exam nerves".

- *List traversal does not terminate **(2)***: Only a couple of students neglected to test for list termination. This is to be expected given the likelihood that students took at least one example of list traversal code into the exam.

- *Successfully filters for strings starting with a vowel, but does not attempt list traversal **(2)***: This is in addition to the similar "task avoidance" category above (in this case there is no attempt to justify the omission by re-interpreting the function specifications). It is likely to be simply an omission based on uncertainty about how to traverse the list.

- *Traverses the list successfully but avoids any attempt to filter for strings starting with a vowel **(1)***: Again, no evidence of misconception, although it is assumed that the student may have had trouble accessing the first element of the string.

Question 4(b)

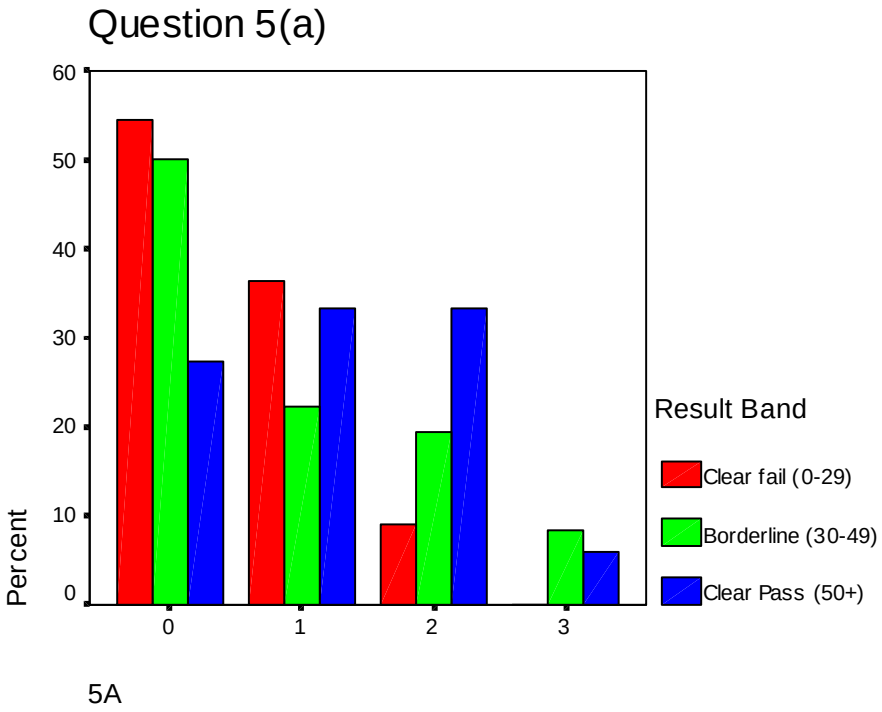


4B

Band	Mean	Std. Dev.
Clear Fail	2.4	2.5
Borderline	6.9	2.9
Clear Pass	9.0	2.4

This 15-mark question asked students to traverse a linked-list, deleting certain entries (making appropriate memory deallocation) and leaving the remaining list intact. This requires a much greater "operational" understanding of pointers in order to maintain the list (assuming that the code is written from first principals). Students in the "clear fail" group had considerable trouble with the additional manipulation required by this question over the previous one. Students in the "borderline" group, however, demonstrated a level of understanding that approached that held by the "clear pass" students.

Question 5(a)



Band	Mean	Std. Dev.
Clear Fail	0.5	0.7
Borderline	0.9	1.0
Clear Pass	1.2	0.9

There were many UNIX errors evident in answers to this question (and other UNIX questions 5b and 5c), but more involved questions would be necessary in order to determine the misconceptions associated with them. In this question, only one aspect emerged as a clear area of confusion, the means for determining access permission associated with a home directory. The question mentioned that the current working directory was the home directory. Only 8 students produced the answer that a confident UNIX user would use, '*ls -ld*'. Another common correct response was the somewhat clumsy '*ls –l ..*'.

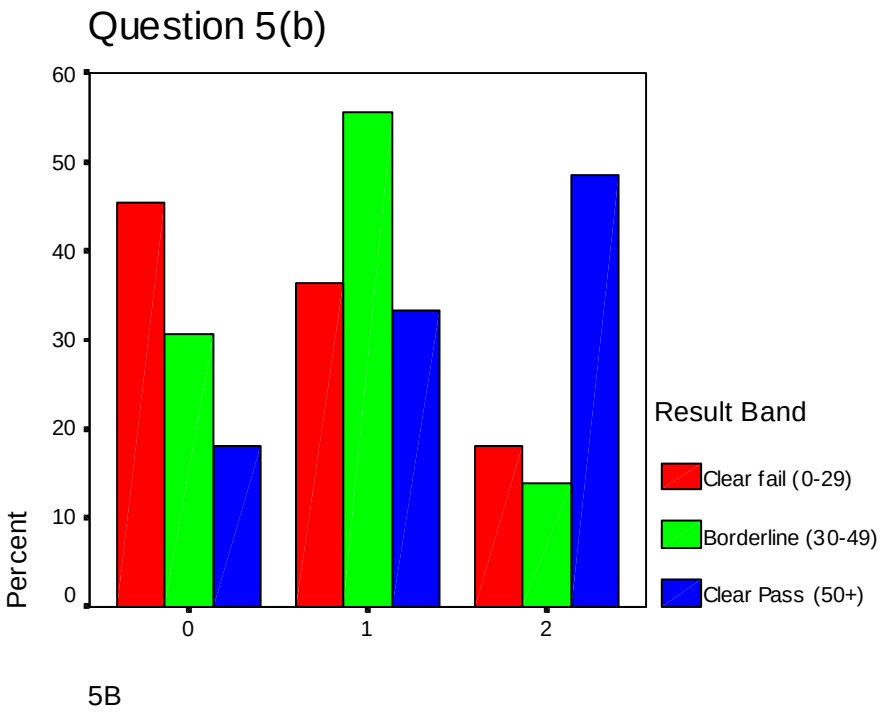
- *ls –l **(28)***: The majority of students produced this answer. It demonstrates a lack of awareness that the '*ls*' command lists the *contents* of the current working directory. It may also reflect some misunderstanding over the nature of directories in UNIX. What differentiates this answer from other erroneous responses listed below is that this issue is below the awareness threshold for the majority of students.

- The following responses are all incorrect responses, which suggests the obvious: a lack of familiarity with UNIX (also clear from other aspects of question 5). However, they are also interesting in that it many of them seem to represent an acknowledgement that '*ls -l*' will not reveal access permissions of

the current working directory (so, to this extent at least, they are more encouraging than the answer above). In the examples below, ‘fred’ is used to represent the login name.

- *ls –l fred* ***(6)***
- *ls –a* ***(4)***
- *ls ../fred* ***(3)***
- *ls –l ~* ***(1)***

Question 5(b)



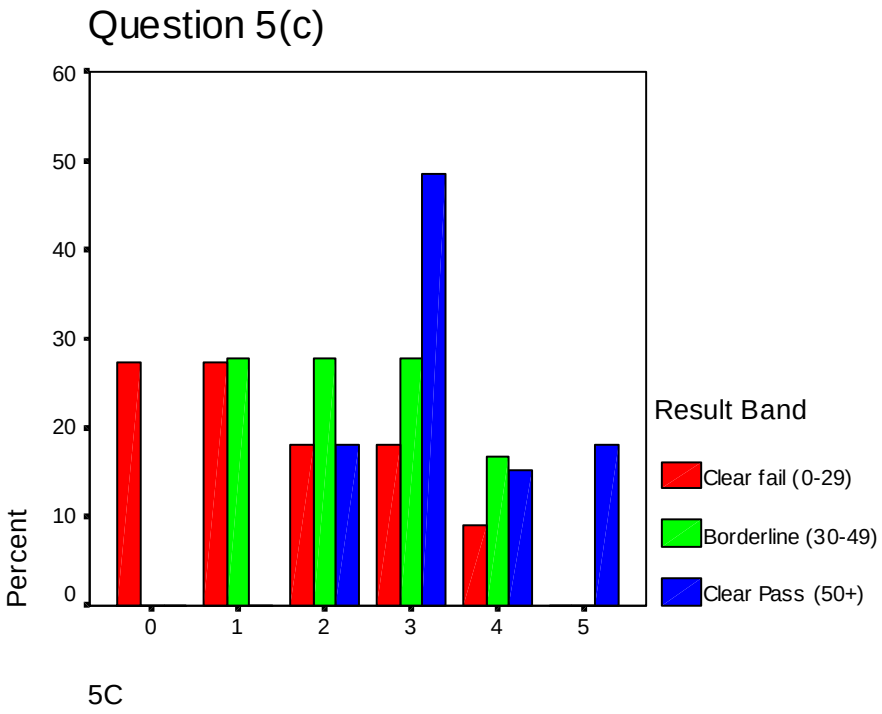
Band	Mean	Std. Dev.
Clear Fail	0.7	0.8
Borderline	0.8	0.6
Clear Pass	1.3	0.8

Again, there was insufficient coverage to form reasonable pictures of possible misconceptions with respect to UNIX questions. However, the copying of files revealed a few potential areas of confusion.

- *The belief that "home" will work as a destination **(7)***: This seemed a very peculiar error indeed, until one student wrote the destination as 'HOME' (i.e., in upper case) which offered some insight into what might be happening. It is likely that students who write something like '*cp *ini home*' are really intending to use the environment variable 'HOME', so should be producing an answer such as '*cp *ini $HOME*'. The fact that they are aware of the environment variable but do not know how to use it suggests a lack of experience in writing shell scripts.

- *Intelligent destination **(5)***: A number of students simply wrote something like '*cp *ini*'. The absence of a destination may simply be an oversight, but it was noticed that these answers also performed a '*cd*' to the source directory, so may perhaps reflect some (peculiar?) belief that the home directory is implied.

Question 5(c)

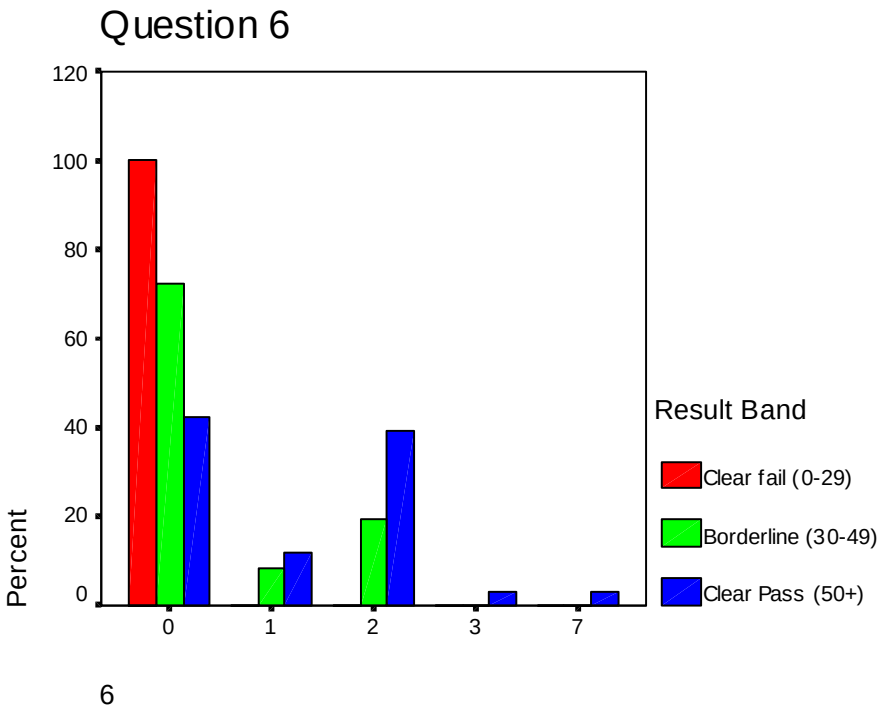


Band	Mean	Std. Dev.
Clear Fail	1.5	1.4
Borderline	2.3	1.1
Clear Pass	3.3	1.0

This 5-mark question asked students to write a script that incorporated many of the fundamental compiling and testing UNIX commands that one would expect students in this unit to be using regularly. Most students failed to place this in the context of a shell script (needed for full marks); however, students who simply understood how to compile (with the most basic options) and test (using redirection and *diff*) could achieve good marks.

The majority of students did reasonably well in this question (as would be expected). As stated, almost none tried to write the test as a script and in this sense avoided one aspect of the question. In the case of students who did badly in this question (less than 3 marks) there were too many issues to guess at the conceptual causes of them. As most of these answers appeared to be random incantations, it is assumed that they simply did not know what to do for at least one aspect of the question.

Question 6

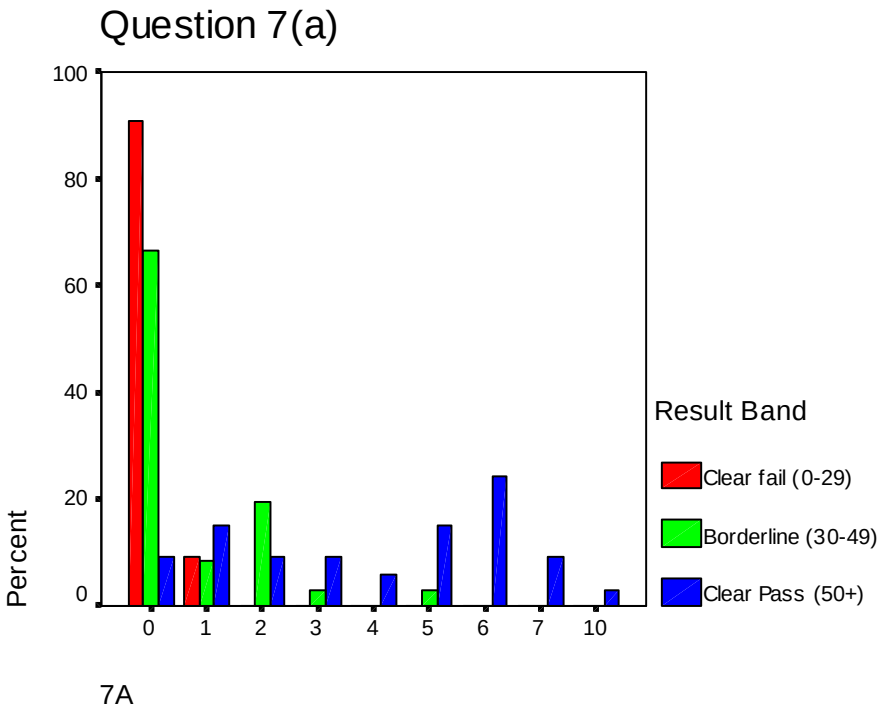


Band	Mean	Std. Dev.
Clear Fail	0	0
Borderline	0.5	0.8
Clear Pass	1.2	1.4

This 10-mark question asks students to read a synopsis of the *isprint* function and use it to write code that behaves similarly to the UNIX *strings* command.

Students in the "clear fail" category were unable to make a reasonable attempt at this question. Other students had some amounts of success. Only a small number of "clear pass" students made a strong showing on this question.

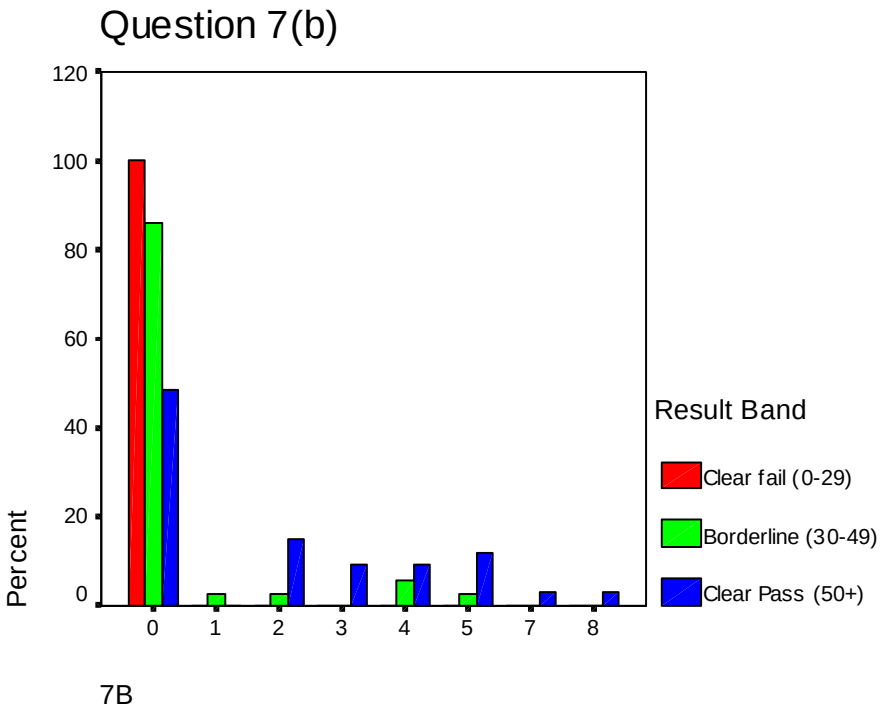
Question 7(a)



Band	Mean	Std. Dev.
Clear Fail	0.0	0.3
Borderline	0.7	1.1
Clear Pass	4.0	2.5

This 10-mark question asked students to write a C function that accessed program arguments to create a linked list. As with question 6 (and question 7b) the significance of the code takes it beyond the reach of most students. Students in the "clear pass" group were strongly differentiated from the other students.

Question 7(b)



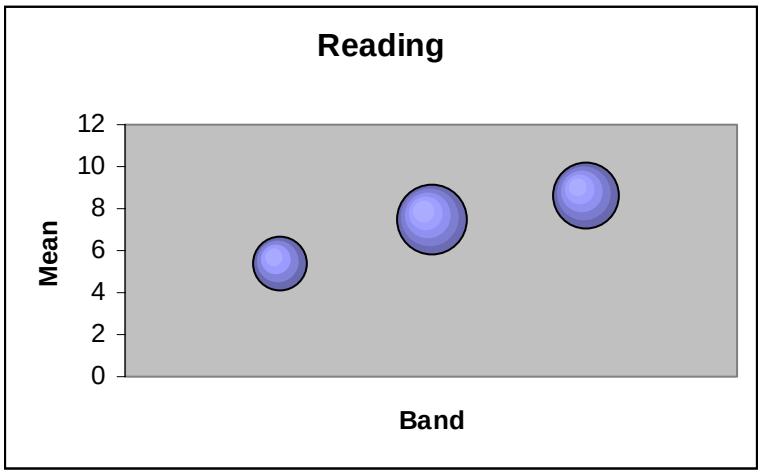
Band	Mean	Std. Dev.
Clear Fail	0	0
Borderline	0.4	1.2
Clear Pass	2.0	2.3

This 10-mark question asked students to write a more complicated piece of list manipulation code. Clearly, only a small number of students were able to achieve a strong result for this question. However, its relevance as an integrative question that acts as a strong indicator for other skills tested in the exam has already been noted. Furthermore, as programming is a fundamentally integrative task, it would be difficult to deny a role for such a question in the exam, despite (or, perhaps, because of) its strong filtering effects.

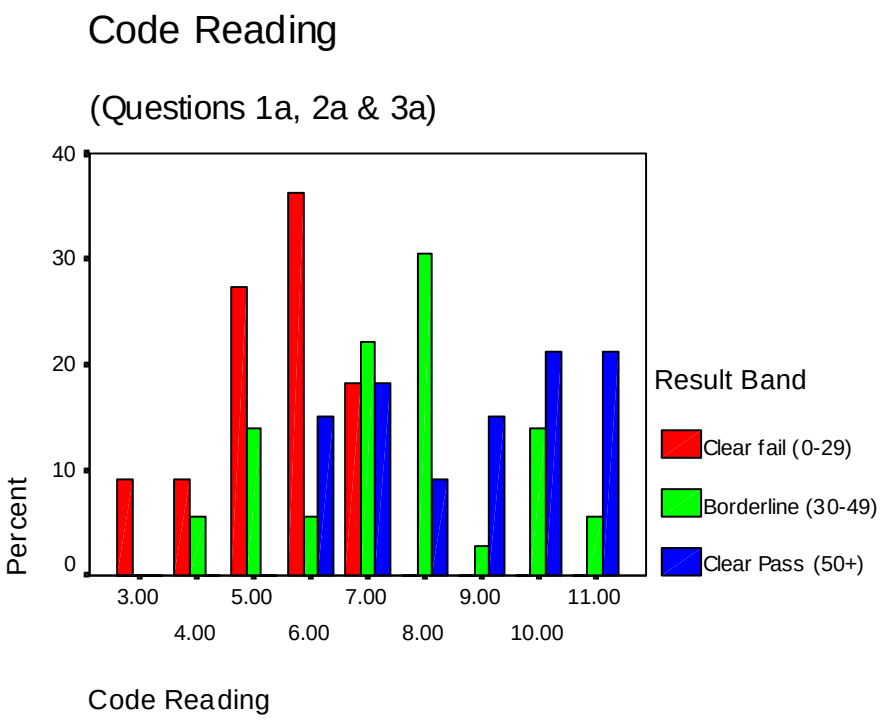
Skills

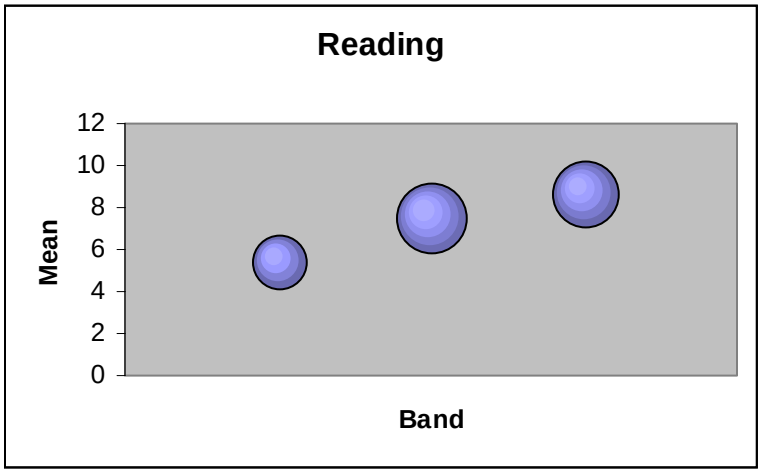
This section brings together questions according to basic skills – code reading, code writing, memory and UNIX. In the case that some questions did not exhibit strong positive correlations (see the first section)(the clustering by skills is justified by noting that many questions were qualitatively different and tested different aspects of a skill.

These discussions include a bubble diagram (such as the one below) to visually capture centrality (mean) and spread (standard deviation). Bubbles (from the left) represent bands – fail, borderline and pass respectively. The centre of the bubble represents mean result (on the Y-axis), and the diameter represents the standard deviation.



Code reading





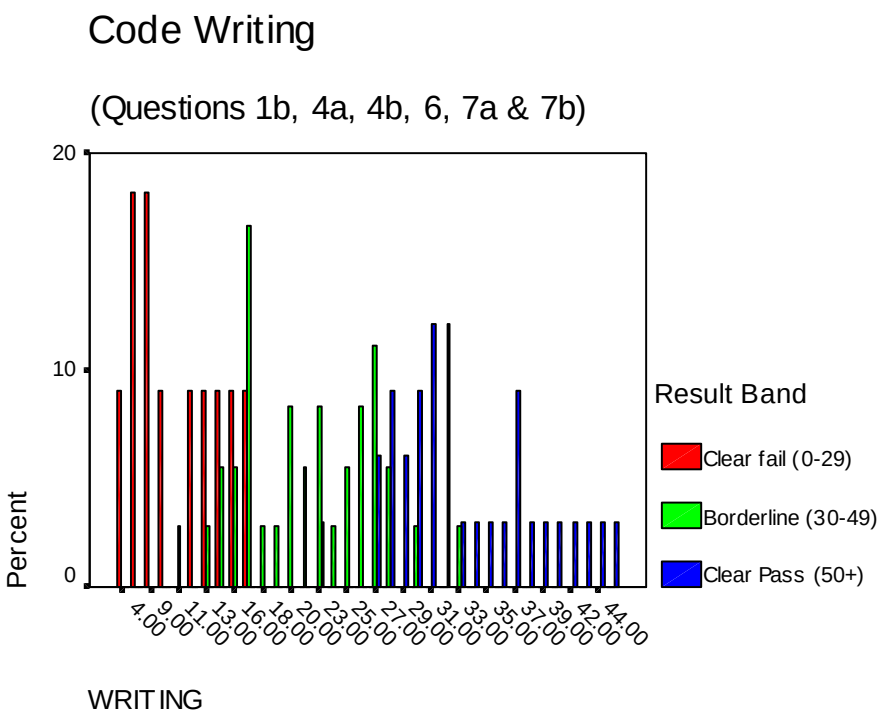
Band	Mean	Std. Dev.
Clear Fail	5.4	1.2
Borderline	7.5	1.9
Clear Pass	8.7	1.8

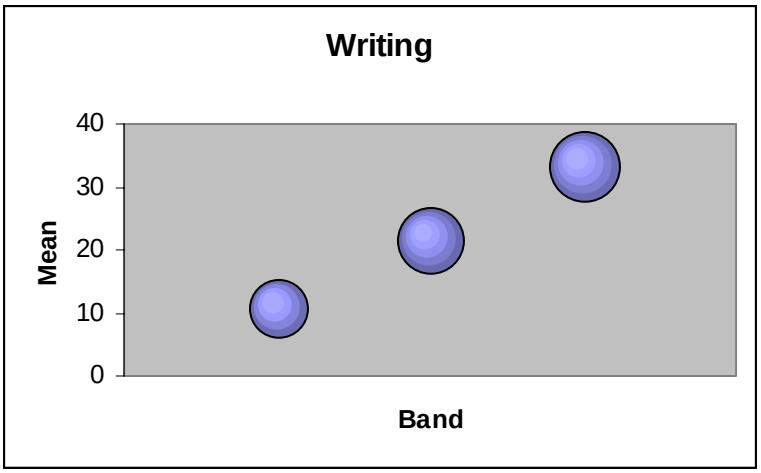
It is difficult to imagine that a student who had written the code required by the unsupervised assessment tasks in the unit would have trouble in tracing code samples. Again, the argument is speculative, but by this logic questions on code reading may provide some insight into the overall level of engagement with the

fundamental unit content. Note the earlier observation that the three code reading questions seemed to be somewhat orthogonal in terms of performance.

The “clear fail” group and the “clear pass” group almost occupy mutually exclusive parts of the results spectrum with respect to code reading; on the other hand, the “borderline” group show greater variance over the entire range of marks.

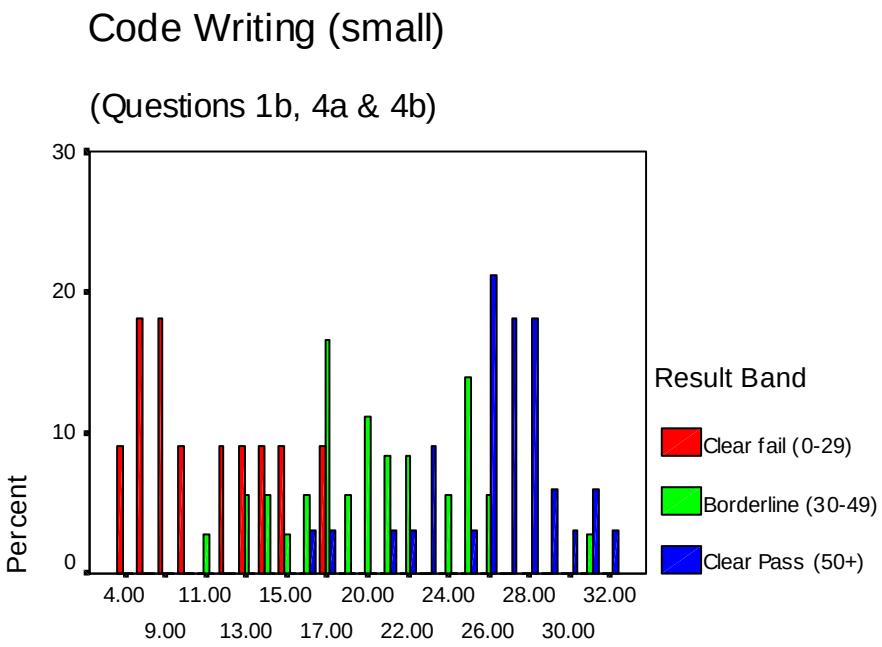
Code writing

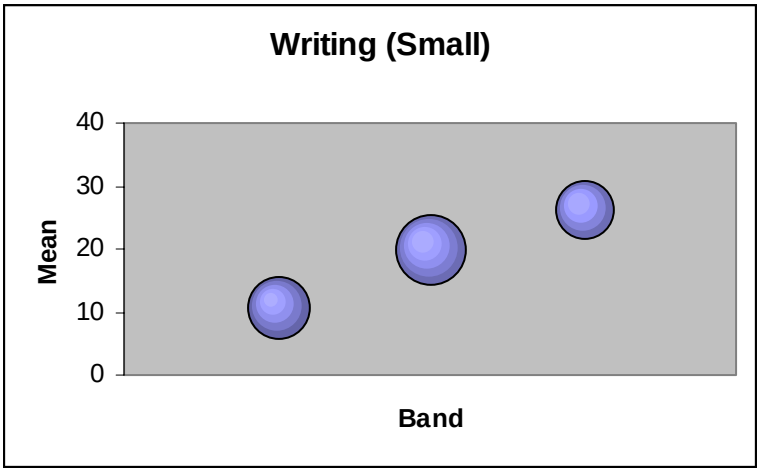




Band	Mean	Std. Dev.
Clear Fail	10.9	3.9
Borderline	21.6	5.4
Clear Pass	33.4	5.5

The “banding” effect of each of the groups is evident in the above histogram. Code writing is clearly a skill that strongly differentiates between groups. In order to explore this issue further, the code writing questions are divided between the smaller code writing questions (1b, 4a and 4b) and the larger and more integrated code writing questions (6, 7a and 7b).



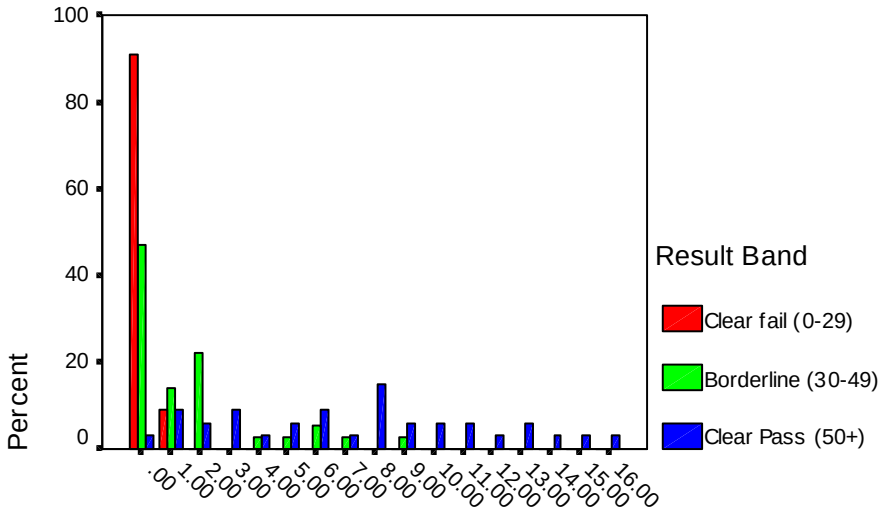


Band	Mean	Std. Dev.
Clear Fail	10.8	3.7
Borderline	19.9	4.6
Clear Pass	26.2	3.5

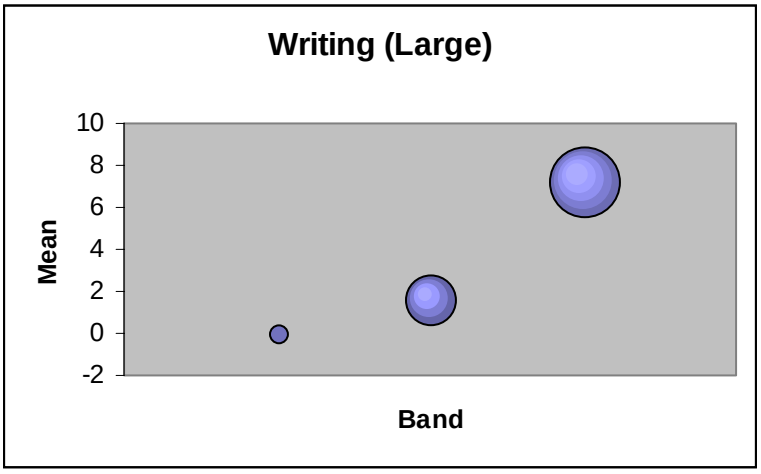
The “banding” is again evident here, with a reduced variance in the “clear pass” group.

Code Writing (larger)

(Questions 6, 7a & 7b)



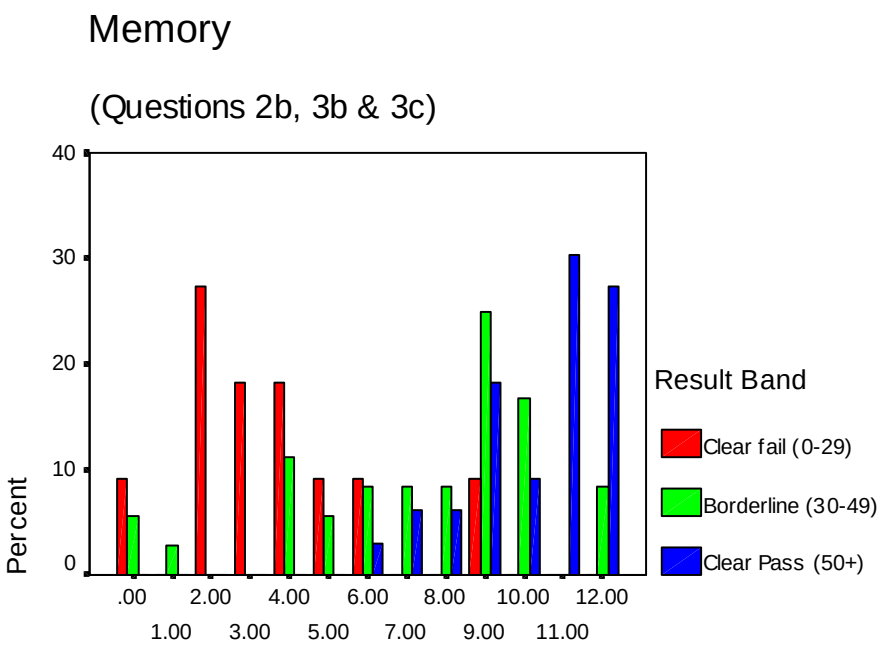
Code writing (large)

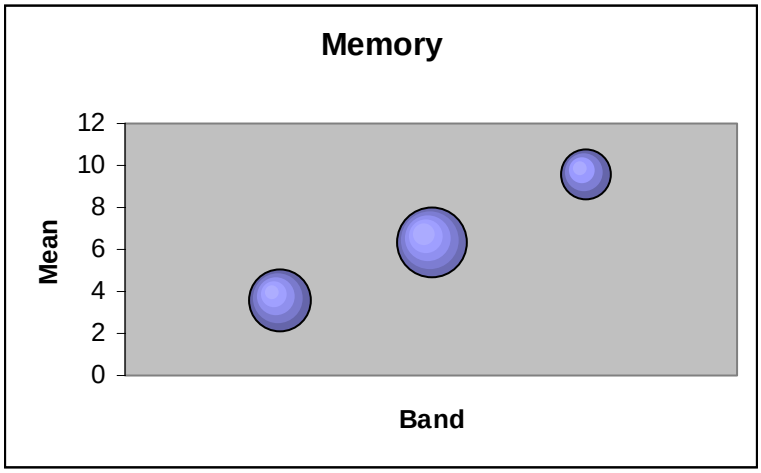


Band	Mean	Std. Dev.
Clear Fail	0.0	0.3
Borderline	1.6	2.3
Clear Pass	7.2	4.4

The more integrated code-writing questions exhibit a considerably different effect in which the “clear fail” students are uninvolved and only the “clear pass” students show a reasonable result (although with a high variance). The “borderline” students are considerably weaker at writing more integrated code.

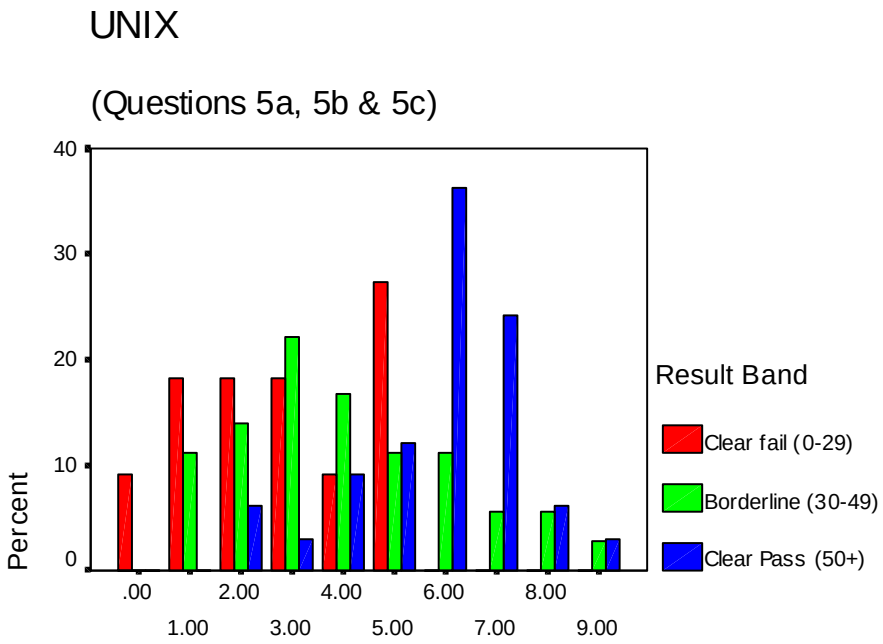
Memory

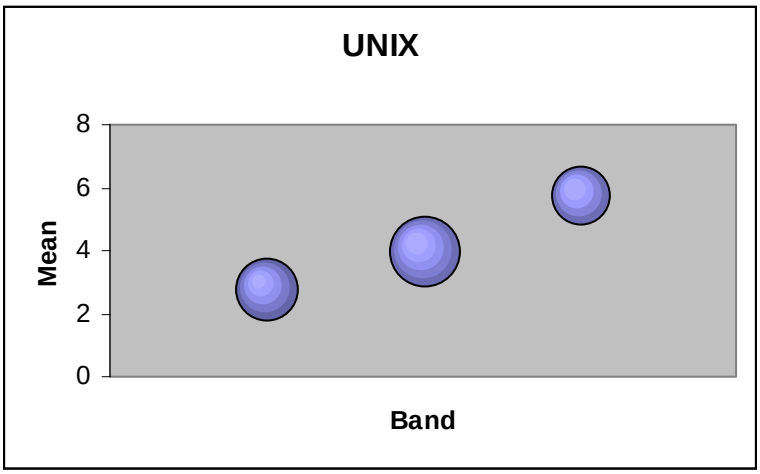




Band	Mean	Std. Dev.
Clear Fail	3.6	2.4
Borderline	6.4	3.1
Clear Pass	9.6	1.7

UNIX





Band	Mean	Std. Dev.
Clear Fail	2.8	1.8
Borderline	4.0	2.1
Clear Pass	5.8	1.6

Section C: Tutor feedback

The following points summarise the feedback from tutors regarding the issues confronting students who they perceived as “at risk”.

- *The quiz*: The weekly quiz was regarded as a motivator with respect to both attendance and preparation. As a result, tutors felt that a number of students improved their understanding of important content. This may suggest that borderline students require increased structural support; for example, they may benefit from a number of small tasks with regular deadlines rather than assignments with the more relaxed structure that comes with distant deadlines.

- *UNIX*: It was felt that UNIX skills were generally poor, and that students knew little about issues such as file permissions. While this is not strictly part of the unit content, it may have an effect on the level of comfort that borderline students have in programming in this environment. There was some suggestion that more UNIX content in quizzes may encourage engagement.

- *Memory*: It was suggested that the memory model could be given greater integration in the quizzes.

- *Message board*: Some tutors felt that a message board would be more useful to students than an evolving FAQ.

- *gdb*: Borderline students did not embrace the use of the debugger outside of lab exposure to it. In other words, they completed the lab sessions in which it was introduced, but did not use it as a tool to address generic programming problems. This was noticeably at odds with the behaviour of advanced students who adopted *gdb* as a useful tool. There may be some need to explicitly integrate such tools into later lab sessions, or perhaps simply remind tutors to suggest that problems be initially explored with the debugger. This observation has ramifications for other tools; for example, the introduction of CVS in later iterations of the unit may be designed in anticipation of the same reaction.

- *Difficulty*: Tutors mentioned that borderline students may have found the unit too difficult. They noted that some students did very little work and it is possible that this may be a result of having decided that the content was difficult to engage with. On the other hand, some tutors stated that they observed borderline students making reasonable efforts to understand the content. This highlights the need for early identification of such students. For example, it is possible that the benefits of hard work may be negated by the persistence of misconceptions.

- *Workload*: Tutors mentioned that “at risk” students might have found the early homework tasks quite daunting. Rather than establishing a “work ethic” they may have resorted to task avoidance, missing the important learning outcomes from these tasks.

Section D: Summary and Recommendations

Summary of misconceptions identified

Identification of misconceptions based on observations of errors is fundamentally speculative (in some cases more than others). However, given the apparent resilience of many misconceptions it is extremely valuable to have a "watch list" of potential problems that maybe actively sought out and confronted where they exist. The following summary is present in this spirit.

1. Occasional isomorphic behaviour of pre-decrement and post-decrement operators (1a).

2. Failure to recognise the behaviour associated with integer division, emphasising the importance of variable typing (1a).

3. Termination condition "creationism" in the case of infinite code, suggestive of the belief that all code terminates (1a).

4. (Possibly) some uncertainty about what constitutes a minimalist *C* program? (1b)

5. Possible confusion over the use of appropriate loop structures (1b).

6. Some confusion between appropriate uses for integer division and integer remainder (1b).

7. Possibly some confusion about the use of EOF in detecting end of input, and this may extend to the wider concept of the UNIX "universality" of files (so that reading from *stdin* does not trigger recognition of the need to use EOF detection) (1b).

8. Some evidence of the belief that declaration of a pointer to characters declares a string, thereby "magically" ensuring sufficient storage allocation (1b).

9. Not so much a misconception, but there appears to be some evidence of a lack of understanding about the nature of *heap* memory versus *stack* memory (3b).

10. Failure to recognise memory over-run problems (3c).

11. Possibly some belief that different variable types are similarly treated (3a).

12. Possibly some misconception about the role of memory allocation in lists, as a number of students did not recognise that list traversal was a movement through existing structures and attempted random acts of memory allocation (4a).

13. Some students tried to perform an atomic comparison operation on strings (4a). This may be a hangover from the various classes in *Java*. More importantly, it suggests an inability to see things "in the small", something that is required in *C* (and something that may be unexpected to *Java* programmers).

Other issues emerging from the exam

1. Students using the "read twice, print once" approach to handling q1b tended to do so without perceiving the need to check for EOF on the second read. This solution is quite elegant (although not very generic) but the elegance is put at risk by failure to recognise the obligation to check EOF on any read statements and not just those that occur in defining the iteration.

2. A number of students used question modification in order to simplify the answers dramatically. These are very difficult to mark, as the examiner cannot predict all permutations, and the resulting code may be reasonably impressive (in other words, the trade-off between question simplification and marks reward may work in the students' favour).

3. There was some evidence of overuse of the *for* loop (in its idiomatic form as an iterator over a known number of elements) by extending it to situations in which the number of elements is arbitrarily long. Although it is also idiomatic for programmers to '*break*' out of an infinite '*for(;;)*' loop, some students were clearly attempting to use the *for* loop to iterate over a known number of elements when the number of elements was in fact unknown.

4. Some students showed considerable uncertainty in how they interpreted a segment of code, as evidenced by contradictory answers for 2 components of q1a. This tendency to drift from a deterministic view of software behaviour (quite independent of whether or not the code tracing is *correct*) may be evidence of an underlying problem that some students have in dealing with programming generally.

Recommendations

1. It is clear that many students would benefit from a more solid grounding in UNIX from the user perspective. It is less clear where this grounding should take place. The unit is a programming unit that seeks to incorporate an understanding of UNIX from a *programmer's* perspective. This is something that is typically built on a reasonable foundation of understanding of UNIX from the *user's* perspective. Many UNIX programming concepts are readily conveyed by reference to command-line counterparts. Where the user skills are underdeveloped this mapping is denied and the learning process requires considerably more internal structure at some cost to satisfying the unit post-conditions. Ideally, a reasonable competence in UNIX command-line use would be a prerequisite for such a unit (especially given that moving to the programming environment on UNIX makes its own demands on the need for new command-line skills). Unfortunately, it is not clear where the responsibility now lies for student competence in basic UNIX skills unless it is incorporated into the prerequisite *Java* units. However, as new units emerge there will be more time to devote to the development of these essential skills. The case for the provision of specific UNIX learning outcomes is strong, as students familiar with a Windows environment need to invest significant time towards feeling comfortable in the UNIX one. This change in world-view is fundamental to appreciating the elegance of UNIX as a programming environment.

2. There is considerable evidence, albeit fragmentary, that many students are reluctant to adopt the lower-level view that programming in *C* requires. Some students (presumably with less than desirable levels of engagement with the unit content) attempt to perform higher-level operations on strings, for example, that simply do not work in *C*. Certainly, *Java* does have a lot of class support for such high-level manipulations. However, this does seem to manifest itself in weak students and may tend to reflect a weak relationship to programming in general rather than some language-related issue. In fact, the list of arguments for the use of a language such as *Java* for novice programming includes the high-level support that it offers. In any case, there may be an issue with transition between languages (from a number of perspectives) that may confuse some students. It may be possible to foster a sense of "discovery" in making the transition from *Java* to *C* by focusing on the fact that *C* is a lower-level language that requires a willingness to negotiate with many aspects of the environment that are typically hidden from the *Java* programmer.

3. To balance the last comment, it should be noted that the unit already has a strong focus on the relevance of the memory model. This should be maintained and possibly strengthened, as it is clear that it is having some success. Even many students in the "clear fail" category are able to demonstrate a basic familiarity with pointers that demonstrates a lower-level view of programming than prior experience in *Java*.

4. Many students seem to require greater exposure to writing code that requires a more integrative understanding of programming concepts. The unit does offer such exposure – in the form of two programming assignments. These assignments are "integrative" in the sense that the they require considerable thought to bring together the various elements of the task (even though they are very small in terms of the final code that a good solution should generate). However, there is real doubt about the level of active participation in these assignments. Students in the "borderline" group showed negative correlation between performance in these assignments and performance in the exam. Students in the "clear pass" group showed (an expected) positive performance correlation between these tasks. Since the assignment marks were high, one interpretation of these anomalous results for the "borderline" group is that students who did "artificially" well in the programming tasks paid the price for lack of engagement by their poor performance in integrative tasks in the exam. Perhaps feeding this perception back into the student population will assist some students to adopt different practices. However, it may be necessary to adjust the assessment mechanism to accommodate the possibility that feedback and good advice alone may not suffice. The problem is one of maintaining the presence of moderately "complex[*]" and integrative assessment tasks while making a convincing case that the learning from participating in the task is more attractive than the marks most immediately at risk. One mechanism, of course, is to reduce the marks at risk for such tasks. However, while this may work with a cognitively light task, a complex task with low marks rewards is unlikely to increase engagement. One possible compromise is to include a cognitively complex task that is highly structured; the task may be introduced in phases, each building on the previous one and gradually revealing a structured path through the problem. Such a task offers a scaffolded introduction to integrative skills and may be complemented by a less structured assignment of similar level of complexity. In terms of the discrepancy between performance in supervised and unsupervised assessment tasks, one may think of a highly structured, phased assignment as a hybrid in which the task is partially brought into the lab environment. Another option for a first scaffolded assignment might be a small group task in which a larger problem is broken down in the actual assignment specifications; thus, the assignment delivers a well-specified sub-problem to each group member in order to satisfy a larger problem. Again, the workload is reduced while maintaining the desired level of problem complexity and offering a scaffolded learning experience to students who engage with the task. If the exam measures what is regarded as valued learning, then this is a serious issue as currently only "clear pass" students show a positive performance correlation between the assignments and the exam.

5. If the ability to perform integrative programming tasks is valued, then the presence of questions such as 7b should be maintained in the exam. However, there is considerable room to adjust the overall structure of the exam to reflect a more criteria-based approach to the unit post-conditions. For example, it may be possible to ensure that students who are able to establish abilities to meet the most basic, but "pass level", outcomes pass the exam and (providing performance in other assessment tasks was "satisfactory") pass the unit as a whole. This could be achieved by introducing more questions that test fundamental skills in less integrated format, and weighting those questions sufficiently to ensure that they constitute a passing grade. However, this may occur at a cost to those questions that test higher-level skills. Therefore, it may be more appropriate to introduce the concept of a "barrier" section to the exam that covers pass-level skills. The use of the barrier concept allows for the quantity of such questions to be kept to controllable levels without the need for over-weighting the basic skills. The barrier section may, for example, only constitute 30% of raw exam marks. However, students who pass the barrier section may be seen to satisfy the pass-level requirements of the unit. Having said that, it is crucial that although the barrier questions would be characterised as ones that restrict themselves only to pass-level skills and do not require integrative abilities, they should clearly be rigorous in testing the skill set that a pass-level student should know.

---

[*] Complexity here refers simply to the multi-faceted nature of a problem, not necessarily its level of difficulty. The overall difficulty of the task may be reduced by trading off task size against complexity, ensuring that the task remains qualitatively complex while remaining highly achievable to students who invest appropriate effort.

6. The use of page of hand-written notes in the exam may need to be discontinued if a barrier component to the exam is used. Students passing the barrier (and therefore having a good chance of passing the unit as a whole) should be able to exercise those skills without reference aids. Students who engaged more fully with the unit content should be able to derive variations on list traversal code, etc., from first principles if necessary.

7. If programming style is to be assessed in the exam, it may be worth considering a penalty system for bad style rather than rewarding acceptable style. This makes it less likely that students will gain marks for inadequate answers to the question.

8. A practical exam in the early weeks (week 4 or 5) of the semester would serve a number of useful purposes, the most important of which would be identify students in crisis while the opportunity still exists to assist them. It also may be used to market the importance of engagement.

9. There was some initial suspicion that code writing and code reading appeared to be quite different skills. Tests did not verify this hypothesis. However, when the code writing skills were differentiated it was found that performance in code reading questions and smaller code writing questions exhibited significant correlation at the 0.99 level of significance, but that performance in code reading and the more integrated code writing questions did not. There may be some benefit to further exploration of this relationship.

10. The possible benefits of the emphasis on the memory model have already been mentioned. It appears that students who exhibited a strong understanding of the memory model (incorporating the subtleties in tested in question 3c) were better equipped to tackle the most difficult of the integrated programming questions (strictly, this exists as a significant positive correlation in performance, and causality may not be assumed). In this case, it may be desirable to cause students to explicitly recognise this connection during the course of the unit. A lab task may be developed that allows students who fully encounter the memory model to solve the problem readily while students who quickly turn to programming fail to fully succeed in the task.

11. Identified misconceptions may be brought into the lab tasks and the self-assessment system as a resource by which tutors and students may recognise and confront any underlying errors in thinking.

12. Exams may be tailored to readily identify misconceptions to further identify misconceptions and improve teaching in the next iteration of the unit. The current exam is well suited to this purpose in that many single issues were identifiable within the questions without being confounded by other possible issues. Some further suggestions follow.

Using exams to improve teaching

The analysis of exam errors (by both qualitative and quantitative means) could have strong benefits for teaching. From this experience a few suggestions emerge as to how to ensure that this process is easy enough to continue to be done:

1. Avoid questions with style marks as these confound the interpretation. If style is an examinable issue it might be separately tested; alternatively it might be an issue for penalisation rather than reward.

2. Design the exam to have one question per page, or to have multiple questions of similar cognitive material per page (given that page totals are recorded in the marks processing). This enables an overview to pinpoint questions (concepts) for closer scrutiny. As mentioned previously, this feature largely exists in the current exam and is mainly mentioned as an issue of maintenance.

3. Ask students to justify answers to questions. This generates richer qualitative data and provides greater insight into potential areas of conceptual problems. It also enables markers to ensure that they are not rewarding misconceptions that happened to derive a correct answer.

4. Include more questions like question 3(a) in code reading questions. This question uses print “debug” statements scattered throughout the code, with very small code segments between them. This enables quick recognition of sources of errors (and, therefore, the likely associated misconceptions).

5. Deliberately generate questions that will derive different answers for known misconceptions (derived perhaps from previous analysis of errors). This makes testing of the spread of a misconception evident. Otherwise there is the suspicion that even many correct answers are marked correct despite a possible misconception.

6. If students are allowed to take a page of notes into the exam, it should be preserved with the exam paper. This enables it to be used to add depth to the analysis. For example, it is uncertain as to whether students did well in question 4a because this represented student learning or because they had similar code at hand in their notes.

7. Misconceptions found in one exam should be specifically tested in an exam in the next iteration of the unit to determine if it persists in the face of efforts to address it.