A Comparison of Tool-Based and Paper-Based Software Inspection

F. Macdonald and J. Miller *
ISERN-98-17

April 1997

Abstract

Software inspection is an effective method of defect detection. Recent research activity has considered the development of tool support to further increase the efficiency and effectiveness of inspection, resulting in a number of prototype tools being developed. However, no comprehensive evaluations of these tools have been carried out to determine their effectiveness in comparison with traditional paper-based inspection. This issue must be addressed if tool-supported inspection is to become an accepted alternative to, or even replace, paper-based inspection.

This paper describes a controlled experiment comparing the effectiveness of tool-supported software inspection with paper-based inspection, using a new prototype software inspection tool known as ASSIST (Asynchronous/Synchronous Software Inspection Support Tool). 43 students used ASSIST and paper-based inspection to inspect two C++ programs of approximately 150 lines. The subjects performed both individual inspection and a group collection meeting, representing a typical inspection process. It was found that subjects performed equally well with tool-based inspection as with paper-based, measured in terms of the number of defects found, the number of false positives reported, and meeting gains and losses.

Keywords: Software inspection, tool support, controlled experiment

1 Introduction

Software inspection, originally described by Michael Fagan in 1976 [11], is well-known as an effective defect finding technique. Experience reports extolling the virtues of inspection are easily found. For example, Gilb and Graham [12] present a number of success stories from a variety of projects. More quantitative evidence of the effectiveness of inspections has also been reported: Doolan [8] reports industrial experience indicating a 30 times return on investment for every hour spent inspecting software requirement specifications. Russell [26] reports a similar return of 33 hours of maintenance saved for every hour of inspection invested.

Despite the benefits, inspection remains an expensive process. This expense mainly derives from the number of people who are involved (usually at least three), and the amount of time which the team must spend both individually and meeting as a group. Furthermore, inspection is most effective when the process is applied rigorously. If rigour is lacking, feedback from the process cannot be used to improve both the inspection process and the overall software development process. Yet rigour can be difficult to achieve due to incomplete or insufficiently detailed descriptions of the process [26], such that actual practice varies depending on the interpretation. The issue is compounded by the variety of processes which have been proposed, some of which are contradictory.

*Empirical Foundations of Computer Science (EFoCS), Department of Computer Science, University of Strathclyde, Livingstone Tower, Richmond Street, Glasgow, G1 1XH, U.K., Tel: +44 (0)141 552 4400, Fax: +44 (0)141 552 5330, fraser@cs.strath.ac.uk

Computer support has been suggested as a means to reduce costs and improve rigour, and to this end a number of prototype tools have been developed. A comprehensive review of these can be found in [19, 20]. Generally, these tools allow the inspection team to browse and annotate the product (the document under inspection) on-line, and may support discussion of these annotations during meetings.

Although existing systems present innovative approaches to supporting software inspection, in general they suffer from a number of shortcomings. Primarily, they support only a single, usually proprietary, inspection process. They also only support inspection of plain text documents, while today's software development environments produce a number of different document types, from plain text to postscript and other graphics formats. The move to an electronic medium provides an opportunity to explore new defect detection techniques, yet such work remains sparse. Finally, although collection and analysis of inspection data is deemed to be desirable for process improvement, existing tools perform little such analysis.

Given the limitations of existing tool support, the authors have been working to design and implement a second generation inspection support tool which embodies the important lessons learned from first generation tools, as well as tackling perceived weaknesses. This system is known as ASSIST (Asynchronous/Synchronous Software Inspection Support Tool). ASSIST has also been developed with the goal of comparing tool-based inspection with paper-based. In the next section, the evaluations of existing tools are described. Some issues pertaining to tool support for software inspection are then discussed, followed by an overview of ASSIST.

1.1 Existing Support Tool Evaluations

While there have been a number of attempts at implementing tool support for software inspection, the quality of evaluation of each tool varies enormously. In the case of ICICLE [5], the only published evaluation comes in the form of 'lessons learned'. In the case of Scrutiny, in addition to lessons learned [14], the authors also claim that tool-based inspection is as effective as paper-based, but there is no quantifiable evidence to support this claim [13].

Knight and Myers [17] describe two experiments involving their InspeQ inspection tool, designed to support their phased inspection method. The first simply provided information on the feasibility of their method, and on the usability of the associated toolset. The second experiment involved inspection of C code, but provided no comparison with paper-based inspection. Mashayekhi [22] reports on the implementation of three prototype tools, with the aim of investigating distribution and asynchrony in software engineering. Again, no comparisons with paper-based inspection are made, except in the case of group meetings, where comparable meeting losses are found using both the tool and paper-based methods.

Finally, CSRS (Collaborative Software Review System) has been used to compare the cost effectiveness of a group-based review method with that of an individual-based review method [27]. Again, since both methods are tool-based, there is no indication of the relative merits of tool-based and paper-based inspection.

Although the evaluations described above attempt to measure, in various ways, the effectiveness of tool support, the fundamental question "Is tool-based software inspection as effective as paper-based inspection?" remains unanswered. Tool-supported inspection will only become an accepted practice if it can be demonstrated that it does not detract from the main goal of software inspections: finding defects.

1.2 Issues in Tool Support for Software Inspection

When moving from paper-based to even the simplest tool-based inspection, there are a number of advantages which can be identified, from both existing tool support research and intuition. To begin with, all documents used in the inspection are usually presented in electronic form. This is natural since virtually all documents will be prepared electronically in the first instance. The tool can

therefore automatically ensure that the most up-to-date version of the document is used, especially if integrated with any version control system in use. It is also possible to provide cross-referencing facilities for documents, from simple search facilities to sophisticated code browsers. Use of electronic documents also avoids the cost of printing multiple copies of documents for each inspector, along with the associated environmental factors.

The crux of inspection is the creation of lists of defects and comments about the product. A fundamental feature of any inspection support tool, therefore, is on-line storage of these lists, and this provides another set of advantages. On-line storage allows defects to be linked to the position in which they occur in the document, allowing them to be accessed easily, obviating the need to add line numbers to documents and minimising errors due to inaccurate noting of the position of the defect. On-line storage also removes any problems caused by unintelligible handwriting, which can be problematic in paper-based inspection. When a synchronous group meeting is held, defects can be easily shared between inspectors, since the tool simply shows everyone the same text. The defect can then be discussed, and any required changes made on-line by the scribe. The updated defect can then be propagated to all participants. The burden on the scribe is therefore reduced, as the whole defect does not have to be written from scratch on the master defect list. This also precludes any misunderstanding that may arise during a traditional meeting when the scribe is transcribing the defect. Finally, the on-line defect list allows a voting mechanism to be used to determine the validity of a defect. Judicious use of this can greatly speed the discussion process.

On the other hand, the move to a computer-based process does create several drawbacks. One concerns the time required to train an inspector in the use of the tool, especially if the inspector is learning the process of inspection at the same time. If the tool is complex, this may detract from the effectiveness of the inspection, even in experienced inspectors. Also, many people are far slower at typing than handwriting, which may slow down the process of noting down defects. This problem is especially acute if the tool is heavily mouse-based, since the inspector has to constantly move between mouse and keyboard.

There may also be two problems concerning the move from paper to screen. One concern is the limited amount of screen space. If an inspector is required to examine the product, along with a source document and checklist, then three windows are required to be on-screen simultaneously. However, most common displays are not capable of displaying three such windows (of sufficient size to be useful) at the same time. The screen may also be cluttered with other windows necessary for operation of the tool, such as has been described in Scrutiny [14]. Contrast this with paper-based inspection, where inspectors are free to find as large a workspace as required and to spread all the documents around in a manner comfortable to their working method.

The second problem concerns reading text from a screen. There have been a number of studies comparing reading from screen versus reading from paper, and Dillon [7] provides a good review of these. He suggests that evidence points to a 20–30% reduction in speed when reading from screen compared to reading from paper, although the different experimental factors make it difficult to reach a certain conclusion. In terms of accuracy, there appears to be no difference for simple spelling checks, but reading from screen may have an adverse effect for more visually- or cognitively-demanding tasks. On the other hand, comprehension appears not to be affected, and may even be improved. This finding is tempered by the difficulty of defining comprehension measures. Reading from screen has also traditionally been thought to be more tiring than reading from paper, with increased eyestrain. The balance of studies performed in fact suggest it is not intrinsically fatiguing, but that average quality displays may limit the length of use. Finally, paper documents have been in use far longer than computers, and there is a natural tendency for users to prefer paper. Studies have shown that this is still the case, although preference is highly dependent on the quality of both presentation mediums.

Ergonomic issues may also have a part to play in reading text from a screen compared with paper. Computers display text in a (mostly) fixed vertical orientation, while paper can easily be oriented to suit the user exactly. Line lengths are usually greater on VDUs than on paper, forcing users to adjust their technique. Displays are wider than they are higher, while the opposite is true

for paper. The dynamic properties of a screen, in terms of update speed of filling and scrolling, may also be a factor, while flicker may occur with low refresh rates. Image polarity is another issue, where text may be presented as dark characters on a light background or vice-versa. There is evidence to suggest that the latter may be preferable. There are issues in terms of text size, font and line spacing. The quality of the text may vary, linked with the quality of the display. User experience is another factor which may produce variations. When the amount of text is greater than the available screen area other factors become relevant. The facilities available for moving through the text may affect its usability. Use of multiple windows to present a document may improve readability, while search facilities can enhance cross-referencing. There is little conclusive evidence on how all of these factors affect the presentation of text.

Several caveats apply when considering how these types of study apply to tool-based inspection. One is their length: typically, the maximum duration of an inspection session is two hours, yet rarely do these studies last as long as this. This leaves an open question: does the sustained nature of inspection produce a greater performance deficit when reading from screen? Furthermore, the type of text being inspected should be considered. The studies described generally make use of continuous text, yet inspection can be used on source code, which may have different layouts. Inspection should also be differentiated from straightforward screen-reading tasks, since during an inspection a large proportion of the inspector's time will be devoted to thinking. Lastly, it should be noted that the quality of an individual display will affect the usability of such a tool, and that the ideal presentation will vary depending on the individual user, implying that the ability to customise the tool may be an advantage.

This summarises the advantages and disadvantages of applying the simplest tool support to inspection. While more advanced tools may provide other benefits, such as allowing distributed inspection, the main purpose of inspection is always finding defects. If it can be proven that the simplest level of support does not alter the efficiency of inspection, other, more advanced, support can be explored secure in the knowledge that the overall concept is not fundamentally flawed. In this case, an appropriate measure of efficiency is the number of defects detected in a constant period of time. More formally, the null hypothesis, H_0, can be stated as:

> There is no significant difference in performance between individuals performing tool-based inspection and those performing paper-based inspection, measured by the total number of defects found during a given time period.

The alternative hypothesis, H_1 is simply:

> There is a significant difference in performance between individuals performing tool-based inspection and those performing paper-based inspection, measured by the total number of defects found during a given time period.

Similar hypotheses can also be formed when considering the performance of inspection teams as a whole.

Practically all existing support tools provide the level of support required to test these hypotheses. Given that favourable comparison with paper-based inspection is the means by which tool-supported inspection will become acceptable, it is surprising that there is little investigation of this nature. To test these hypotheses, and to tackle perceived deficiencies in existing tools, a new prototype inspection support tool has been developed, and is described in the next section.

1.3 An Overview of ASSIST

Our research to date has concentrated on tackling the lack of flexibility of existing tools in terms of the number of inspection processes which they support. This research has produced an inspection process modelling language known as IPDL (Inspection Process Definition Language) [18], which is capable of describing any current and future inspection processes, along with the materials and personnel involved in those processes. ASSIST is capable of executing any process written in IPDL,

ensuring that the process is followed precisely and that the inspection participants are provided with the correct materials and tools at each stage of the inspection.

The first version of ASSIST provides four main facilities when performing inspections: the **execute window** and three types of browser. The execute window provides each participant with a list of the people involved in the inspection and their status, along with a list of the documents available. Double-clicking on a document name opens the appropriate browser for that document.

The **list browser** allows the user to manipulate lists of items, typically document annotations describing defects. Lists can be either read only or read-write. Read only lists may only be examined, while read-write lists can be edited. Each item within a list consists of a title, the name of the document which the item refers to, a position within that document, a classification and a free-form textual description. Items can be added, removed, edited and copied between lists. Additionally, during a group meeting the list browser allows participants to propose items from their personal lists to the whole group, allowing them to be discussed and voted on. If the item is accepted, it is copied to a master list of defects, representing the output of the group meeting. The scribe may also edit proposed items to reflect refinements suggested at the meeting.

The **text browser** allows documents to be viewed and annotated via the list browser. The browser is based on the concept of a current focus, i.e. a line of text which is currently under scrutiny. The current focus can be annotated, or existing annotations read. During a group meeting, the focus for the whole group is controlled by the reader: when the reader moves the focus to a new line, the other browsers are automatically updated. The browser indicates the current line number and the percentage of the document inspected. The view of the document can be split horizontally, allowing two separate areas of the document to be viewed simultaneously. Finally, a find facility is available, allowing the user to search for specific words or phrases in the document. This facility works in a manner similar to that found in Netscape.

Finally, the **simple browser** only allows text documents to be viewed. Facilities such as annotation and line numbering are not available, although the split window and find facilities are still available. This browser is used for all supporting documents, such as checklists and specifications, which are themselves not being inspected.

This implementation of ASSIST provides the basic facilities required to compare tool-based inspection with paper-based. It was therefore decided to investigate this question before further implementation was undertaken. Not only would this give information on the comparative effectiveness, it would provide useful feedback on the usability of ASSIST and help shape our future research. The next section describes a controlled experiment to test the hypotheses stated in Section 1.2.

2 Experiment Design

The testing of our hypotheses required two groups of subjects to inspect a single document, one using a tool-based approach and the other using a paper-based approach. To ensure that any effect was not simply due to one group of subjects being of higher ability, the subjects must also inspect a second document, this time using the alternative approach. The inspection process used consisted of two stages: an individual detection phase, where each subject inspected the document for faults, and a group collection meeting, where individual lists were consolidated into a single master list for the group.

2.1 Subjects

The experiment was carried out during late 1996 as part of a team-based Software Engineering course run by Strathclyde University's Department of Computer Science for third year undergraduates. The subjects already had a firm grounding in many aspects of Computer Science. In particular, they had been taught programming in Scheme, C++ and Eiffel, and had also completed a course in the fundamentals of Software Engineering. Student motivation was high, since the practical aspects of

the experiment formed part of their continual assessment for this course, the final mark of which contributes to their overall degree class.

A total of 43 students participated in the class, split into two approximately equal sections. Section 1 had 22 subjects and Section 2 had 21 subjects. The split was achieved by ordering subjects according to their mark in a C++ programming class (C++ was chosen as the type of code to be used in the experiment). Adjacent subjects were then blocked into sets of four, with two randomly chosen subjects assigned to one section, with the remaining two subjects assigned to the other. Within the two sections the students were organised into groups of three (and a single group of four). This was done in such a way as to create equal ability groups, based on their C++ programming marks. Section 1 therefore consisted of six groups of three students and one group of four students, while Section 2 contained seven groups of three students.

2.2 Statistical Power

The existence of the phenomenon being empirically investigated does not guarantee production of a statistically significant result. Statistical power analysis is a method of increasing the probability that an effect is found in the empirical study. A high power level indicates that a statistical test has a high probability of producing a statistically significant result, i.e. if an effect exists it is highly likely that it will be found, and a Type II error is unlikely to be committed. Similarly, if an effect does not exist, the researcher has a solid statistical argument for accepting the null hypothesis, which is not the case if the study has low power.

The power of the statistical test becomes a particularly important factor when H_0 is not rejected, i.e. the effect being tested for is not found. The lower the power of the test, the less likely H_0 is accepted correctly. Consequently, when H_0 is not rejected and the statistical test is of low power, the results are ambiguous and the only conclusion that can be drawn is that the effect examined has not been demonstrated by the study. Studies with a high power, on the other hand, offer the advantage of an interpretation of the results when there is insignificance. There exists strong support for the decision not to reject the null hypothesis, something a low powered, statistically insignificant study cannot give.

In common with many software engineering experiments, this experiment has had limited control over the components of statistical power. The hypothesis requires a two-tailed decision, and the experimental plan was to use parametric testing (ANOVA). The sample size (or more correctly the harmonic mean) is defined by the size of the class, and is 21.5. The exact value of required statistical power is a topic of debate amongst statisticians, but most experts agree that a power rating of between 0.7 and 0.8 is required for a correct experimental design. If we desire a statistical power in this range, this implies that the experiment will have to assume a large effect size (approximately 0.8), and hence this defines the sensitivity of the experiment, and the minimal sensitivity where it is 'statistically safe' to accept the null hypothesis.

During the training period, all the exercises where closely monitored, allowing an estimate of the statistical variance of both populations. Combining the two variances produces a result of 0.16, allowing us to calculate the minimum normalised difference (average number of defects found divided by the maximum number of defects available to be found) for which the experimental design can be considered statistically safe— 0.13. This equates to 1.5 defects, given that both experimental programs have 12 defects. Therefore, the experimental design safely allows us to accept the null hypothesis for normalised differences above (approximately) 13%. Obviously the null hypothesis could still be accepted for differences less than 13%, but for these effect sizes the experiment has an increased risk of committing a Type II error. Indeed, by best experimental practice this risk is deemed unacceptable in this situation. Hence, at these reduced effect sizes the experiment is unable to reliably prove that no effect exists. See Miller *et al.*[23] for a fuller discussion of the role of statistical power in experimental design.

2.3 Materials

Having previous experience of running defect detection experiments [24], it was decided that the most appropriate material to inspect would be C++ code. A number of factors influenced this decision. Initially, source code was chosen as the appropriate material due to ease with which defects in code can be defined. This is in contrast with, say, English language specifications, which provide many problems of ambiguity. It is also easy for inspection of such material to degenerate into arguments over English style and usage. Intelligent seeding of defects in code avoids these problems and provides a well-defined target against which performance can be judged. Student experience was also taken into account. Inspection must be performed by personnel with experience in the type of document being inspected. It was therefore important to chose material in a form which the students had experience in. This also avoids teaching a new notation or language which students may spend much of their time trying to understand and become familiar with, instead of finding defects. Since the subjects were competent in C++, material in that language was chosen for the experiment. The decision was further ratified by the availability of high quality material from a local replication of Kamsties and Lott's defect detection experiment [16].

For the training materials, a selection of programs originally used in Kamsties and Lott's experiment were used, since each program had an appropriate specification, a list of library functions used by the program and a comprehensive fault list. These programs were originally written in non-ANSI C and were translated into standard C++ for our experiment. The programs were also edited to remove some faults and add others. The programs used were: `count.cc` (58 lines, 8 faults), `tokens.cc` (128 lines, 11 faults) and `series.cc` (146 lines, 15 faults). A further example, `simple_sort.cc` (41 lines, 4 faults) was created for use in the tool tutorial.

Since the Kamsties and Lott material had already been used in the same class last year, the two programs to be used for the experiment (and, hence, the assessment), were specifically written afresh. One program (`analyse.cc`, 147 lines, 12 faults) was based on the idea of a simple statistical analysis program given in [6]. The second program (`graph.cc`, 143 lines, 12 faults) was written from a specification for a Fortran graph plotting program, originally found in [3]. For each program, a specification written in a similar style to that of the Kamsties and Lott material was also prepared, along with appropriate lists of library functions.

There is no clear consensus on the optimal inspection rate. For example, both Barnard and Price [2] and Russell [26] quote a recommendation of 150 lines of code per hour. On the other hand, Gilb and Graham [12], recommend inspecting between 0.5 and 1.5 pages per hour, translating to between 30 and 90 lines of code. All conclude that lower rates improve defect detection. Each practical session lasted 2 hours, giving an inspection rate of around 70 lines per hour. This figure represents a compromise since subjects were not professional inspectors and could not be expected to perform at the highest recommended rates. At the same time, there was enough time pressure to make the task realistic.

The actual inspection task was to use the program specification and list of library functions to inspect the source code for functionality defects, making use of a checklist. Use of a checklist is standard inspection procedure, and students were supplied with the checklist described below. Students were specifically discouraged from finding defects relating to other qualities, such as efficiency. Each program was seeded with errors of functionality and with checklist violations. For the two experimental programs, defects in one program were matched, in terms of type and perceived difficulty, with defects in the other program, in an effort to match the overall difficulty of the programs. All programs used compiled with no errors or warnings using `CC` under SunOS 4.1.3.

The checklist used was derived from a C code checklist by Marick[21], a C++ checklist by Baldwin[1] (derived from the aforementioned C checklist), the C++ checklist from [15] and a generic code checklist from [9]. From the C and C++ checklists we removed items which we considered to be irrelevant (for example, none of our programs made extensive use of macros), along with esoteric items, such as those dealing with threaded programming and signals. From the generic checklist we removed items not relevant to C++. Duplicates were then removed and the remaining items grouped into a number of categories. An additional category was added concerning differences between the

specification and behaviour of the program. Finally, we performed another edit on the checklist to reduce the number of categories to ten, allowing the checklist to fit on two sides of paper. We felt that a short checklist covering the major points to consider would be more effective than a much longer, more detailed checklist which the subjects would struggle to cover in the time allowed. This follows practice recommended by Gilb and Graham [12]. The checklist is presented in Appendix B.

2.4 Instrumentation

For paper-based inspection, each student was given an individual defect report form like that in Appendix C. For group meetings, the scribe was given a group defect report form like that in Appendix D. For tool-based inspection, ASSIST was used to keep individual defect lists during individual inspection, and a master list at the meeting.

Each practical session was limited to a maximum of 2 hours. Actual finishing times were not collected because the inspection was held under exam conditions. It was felt that many students would stay beyond their actual finishing time to read over their work in case they had 'inspiration', and this proved to be the case. In fact almost all participants made use of the full two hours.

For each subject, data collected were the total number of correct defects found, along with the number of false positives submitted (i.e. defects which subjects incorrectly identify), and similarly for each group. Also calculated were meeting loss (number of defects found by at least one individual in the group, but not reported by the group as a whole), and meeting gain (number of defects reported by group, but not reported by any individual) for each group. Finally, for each defect in each program, the frequency of occurrence was obtained, both in tool-based and paper-based inspection.

2.5 Experiment execution

The practical element of the course ran over a period of ten weeks. The first six weeks were devoted to providing the students with training in software inspection and using ASSIST, as well as refreshing their C++ knowledge. These practical sessions were interspersed with lectures introducing each new topic where appropriate. After inspection of each program was complete, the students were presented with a list of faults in that program. The remaining four weeks were used to run the actual experiment. Appendix A details the exact timetable used. Each practical session was run twice, once for each section of the class, thus ensuring their separation when using different methods on the same program. Both practicals occurred consecutively on the same afternoon of each week.

2.6 Threats to Validity

Threats to internal validity

Any empirical study may be distorted by influences which affect dependent variables without the researcher's knowledge. This possibility should be minimised. The following such threats were considered:

- Selection effects may occur due to variations in the natural performance of individual subjects. This was minimised by creating equal ability groups.
- Maturation (learning) effects concern improvement in the performance of subjects during the experiment. The data was analysed for this and no effect was found. Section 3 describes this analysis in more detail.
- Instrumentation effects may occur due to differences in the experimental materials used. Having both groups of subjects inspect both programs helps counteract the main source of this effect.

Program	analyse.cc		graph.cc	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Subjects	22	21	22	21
Mean	7.68	7.76	6.64	6.00
St. Dev.	1.55	1.92	1.43	2.05
St. Error	0.33	0.42	0.30	0.45
F Ratio	0.02		1.40	
F Prob.	0.88		0.24	

Table 1: Analysis of variance of individual defects scores.

- Presentation effects may occur since both sets of subjects inspect the programs in the same order. It is believed that such an effect is symmetric between both sets of subjects, and that the effect presents less risk than the plagiarism effect possible when the order of presentation is reversed for one set of subjects.
- Plagiarism was a concern since the group phase of each experimental run took place one week after the individual session, hence providing an opportunity for undesired collaboration among students. This was mitigated by retaining all paper materials between phases. With the same purpose in mind, data from the tool regarding the individual phases was extracted immediately after each session. Furthermore, access to the tool and any on-line material was also denied. Finally, any plagiarism effect would be noticeable by any group presenting an above average meeting gain. No such groups were detected.

Threats to external validity

Threats to external validity can limit the ability to generalise the results of the experiment to a wider population, in this case actual software engineering practice. The following were considered:

- The student subjects involved in the experiment may not be representative of software engineering professionals. This was unavoidable since our choice of subjects was limited by available resources.
- The programs used may not be representative of the length and complexity of those found in an industrial setting. The programs used were chosen for their length, allowing them to be inspected within the time available. However, the amount of time given to inspect each program was representative of industrial practice quoted in popular inspection literature.
- The inspection process used may not correspond to that used in industry, in terms of process steps and number of participants. For example, the process used did not involve the author presenting an overview of the product, and a rework phase was not used. However, the detection/collection approach to inspection is a standard process [12].

These threats are typical of many empirical studies, e.g. [25, 16]. These threats can be reduced by performing repeated experimentation with other subjects, programs and processes.

Program	`analyse.cc`		`graph.cc`	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Groups	7	7	7	7
Mean	10.86	10.71	9.57	8.86
St. Dev.	0.69	0.95	1.27	1.07
St. Error	0.26	0.36	0.48	0.40
F Ratio	0.10		1.29	
F Prob.	0.75		0.28	

Table 2: Analysis of variance of group defect scores.

Effect	F Ratio	F Prob.
Order	0.34	0.56
Program	33.78	$\ll$0.01
Order$\times$Program	2.20	0.15

Table 3: Analysis of variance of method order and program.

3 Results and Analysis

3.1 Defect Detection

The raw data from the experiment can be found in Appendix K. Table 1 presents a summary of the data and the analysis of variance of the individual phases of both inspections. For `analyse.cc`, it is obvious that there is very little difference in performance, and this is confirmed by the analysis of variance. For `graph.cc`, the section using paper-based inspection appear to outperform that using the tool, although this difference is not significant. In both cases the null hypothesis concerning individuals must be accepted.

Table 2 presents a summary of the data and the analysis of variance of the group phases of both inspections. These results follow the same pattern as for individual: `analyse.cc` provides very similar results between methods, while `graph.cc` provides a larger difference, but which is not statistically significant. Again, the null hypothesis as applied to groups must be accepted.

Under further investigation, the data from the individual phase of the `graph.cc` inspection failed the the Levene test for homogeneity of variances. However, the robustness of the F test is well documented. For example, Boneau [4] has studied the effects of violating the assumptions which underlie the t test, and generalised these results to the F test. Provided samples sizes are sufficient (around 15) and virtually equal, and a two-tailed test is used, non-homogeneity of variances should not cause difficulties. A similar conclusion is presented Edwards [10]. As a safeguard, the Kruskall-Wallis non-parametric test was applied to all four sets of data, and gave results similar to those for the parametric tests, with no significance.

The data was analysed for any effect stemming from the order in which the methods were used and for any difference caused by the two programs. The results are shown in Table 3. There proved to be no significant difference between subjects who used the tool first and those who used paper first. However, the results indicated a significant difference in the difficulty of `analyse.cc` compared with `graph.cc`. This was also supported by one of the post-experiment questionnaires (described in more detail in Section 3.4). Finally, the data was analysed for any effect from the order in which the methods were used combined with the two different programs. No significant result was found.

Phase	Individual		Group	
Defect No.	Tool	Manual	Tool	Manual
1	13	17	7	7
2	22	21	7	7
3	2	3	1	3
4	16	12	7	6
5	7	7	6	6
6	21	20	7	7
7	21	17	7	7
8	4	8	6	6
9	17	18	7	7
10	13	11	7	6
11	18	18	7	7
12	14	11	7	6

Table 4: Summary defect detection data for `analyse.cc`.

Phase	Individual		Group	
Defect No.	Tool	Manual	Tool	Manual
1	20	20	7	7
2	19	21	7	7
3	13	20	7	7
4	4	4	3	2
5	16	20	7	7
6	7	10	5	6
7	13	15	7	7
8	1	3	0	4
9	4	7	4	6
10	7	7	4	3
11	19	15	7	6
12	3	4	4	5

Table 5: Summary defect detection data for `graph.cc`.

Program	`analyse.cc`		`graph.cc`	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Subjects	22	21	22	21
Mean	3.59	4.19	3.23	3.24
St. Dev.	1.943	1.94	1.9	2.14
St. Error	0.41	0.42	0.4	0.47
F Ratio	1.02		$\approx$0.00	
F Prob.	0.32		0.99	

Table 6: Analysis of variance of individual false positives.

Program	`analyse.cc`		`graph.cc`	
Section	1	2	1	2
Method	Tool	Paper	Paper	Tool
Groups	7	7	7	7
St. Dev.	1.72	1.46	1.34	2.22
St. Error	0.65	0.55	0.51	0.84
F Ratio	0.25		0.08	
F Prob.	0.62		0.78	

Table 7: Analysis of variance of group false positives.

Table 4 summarises the frequency with which each defect was found in `analyse.cc`, both in the individual and in the group phases. Table 5 shows the same data for `graph.cc`. In most cases there is no great difference between the scores achieved with the tool compared to those using paper. The only large difference appears for the third defect in `graph.cc`, which was found by 20 of the paper-based inspectors, yet only 13 of the tool users. This alone represents 35% of the overall difference between tool and paper. This defect concerns a missing function call, which means the program does not print some output specified by the specification. In fact, this is a very easy defect to detect using the search facility of the tool. By entering the function name as the target of the search, the inspector can find calls to that function, almost guaranteeing that it will be found. However, although the mechanics of the find facility were explained, the use of the tool to detect such defects was not explicitly taught to the subjects, since this would bias the experiment in favour of the tool.

3.2 False Positives

In addition to the number of defects found by each subject, the number of false positives reported was also measured. False positives are items reported by subjects as defects, when in fact no defect exists. It is desirable to investigate whether tool-supported inspection alters the number of false positives reported, since an increase would reduce the effectiveness of the inspection. On the other hand, if use of the tool in some way suppressed false positives, the efficiency of the inspection would be increased, with less time wasted on discussing these issues.

Table 6 presents the analysis of variance for the false positive data from the individual phases of each inspection. While the tool appears to provide an improvement over paper for `analyse.cc`, this difference is not significant. On the other hand, the means for `graph.cc` are almost identical, and the ANOVA test confirms this. The analysis of variance or the false positive data of the group phases is shown in Table 7. For both programs, there is no significant difference between the tool-based

Program	analyse.cc		graph.cc	
Group	Gains	Losses	Gains	Losses
1	0	0	0	0
2	1	0	2	1
3	0	0	0	1
4	0	0	1	1
5	0	0	0	1
6	1	1	0	1
7	0	0	0	0
Total	2	1	3	5
8	0	0	1	0
9	0	0	0	2
10	0	0	0	1
11	0	0	0	2
12	0	0	1	0
13	0	0	0	0
14	0	1	1	0
Total	0	1	3	5

Table 8: Meeting gains and losses for both programs. Groups 1–7 perfomed tool-based inspection on `analyse.cc` followed by paper-based inspection on `graph.cc`. Groups 8–14 used the methods in reverse.

and paper-based approaches.

3.3 Meeting Gains and Losses

The final set of data to be analysed concerns meeting gains and losses. Synchronous meetings are frequently cited as necessary because it is believed that factors such as group synergy improve the output of the meeting, manifested in defects being found at the meeting which have not been found during the individual phase. On the other hand, defects may be lost when a participant fails to raise a defect found during the individual phase. We hypothesised there would be no difference between tool and paper-based methods for both gains and losses. Table 8 shows the gains and losses for all group meetings in both inspections. It is clear even without statistical analysis there is no significant differences between methods.

3.4 Debriefing Questionnaires

During the course, subjects were asked to complete four questionnaires. The first two were given after the full practice using ASSIST, one after the individual inspection (Questionnaire 1) and one after the group meeting (Questionnaire 2). These questionnaires focussed on eliciting qualitative feedback on ASSIST. Two further questionnaires were then presented, one after inspection of the first experimental program was complete (Questionnaire 3), the other after inspection of the second experimental program (Questionnaire 4). These questionnaires concentrated on such topics as the overall difficulty of the task and the relative merits of paper-based and tool-based inspection. This section presents some results from these questionnaires.

In Questionnaire 4, the question “Overall did you feel you performed better during individual inspection using manual (paper-based) inspection or ASSIST, or were you equally effective with both methods?” was asked. 39% of respondents claimed to have performed better using paper-based inspection, 39% had no preference, while 22% claimed to have performed better with ASSIST.

When the same question was asked with regard to the group meeting, the number of people preferring paper-based inspection dropped to 19.5%, the number with no preference increased to 61% and the number preferring ASSIST dropped to 19.5%. There is no clear correspondence between preferences for individual and preferences for group: some people who expressed a preference for paper-based individual inspection then went on to select ASSIST for the group meeting, others always preferred paper-based or always preferred ASSIST, yet others moved only one category (i.e. from paper-based to no preference or tool-based to no preference).

The qualitative statements indicating preference make interesting reading. People who indicated a preference for paper-based inspection generally liked the tactile nature of paper, allowing them to scribble notes on the code itself. Others simply preferred reading code on paper instead of on-screen. A number of people found it awkward moving between the code, specification and checklist windows of ASSIST. While the scribe's burden is reduced in ASSIST, one student commented that it was *too* easy to propose defects and put them into the master defect list, and therefore the phrasing of the defect was not usually considered as much as when the scribe had to manually write it down.

People who preferred ASSIST pointed out the following advantages. It was easy for the group as a whole to see exactly where individuals' errors were, and it was also considered easier to compile the master defect list, giving more time for the group to search for further defects. Others found it easier to traverse the code, and a number of people preferred the defect creation/editing facilities. The voting method for resolving defects was also considered to be useful.

People who expressed no preference also made interesting points. For example: "[I]t was easy to look through code when it was on paper, but ASSIST has its advantages such as searching through the document for keywords...[During group meetings] paper-based inspection provoked more discussion, [but] ASSIST made it easier for [the] reader".

Finally, the question "In terms of complexity, how did `graph.cc` compare with `analyse.cc`?" was asked. The five possible responses were: "much more complex", "slightly more complex", "of similar complexity", "slightly less complex" and "much less complex". 49% of students believed `graph.cc` to be "much more complex", while 44% believed it to be "slightly more complex", and only 7% considered it to be "of similar complexity" to `analyse.cc`. These comments support our earlier statistical analysis concerning the difference between programs.

4 Conclusions

Software inspection is an effective defect finding technique and has been widely used. Despite its benefits, it is still deemed to be expensive. Inspection support tools have been proposed as a means to reduce the cost of inspection, yet evaluations of existing tools have been sparse. The question as to whether tool-supported inspection can be as effective as paper-based inspection has yet to be answered with any certainty.

This paper has described a controlled experiment designed to investigate the relative performance of these methods, using a two-stage inspection process. Our null hypotheses stated there would be no significant difference between methods, measured in terms of the number of defects found both by each subject and by each group. The results from our experiment show that a straightforward computer-based approach to inspection does not degrade the effectiveness of the inspection in any way. There is no significant difference in the number of defects found. The number of false positives reported and the amount of meeting gains and losses were also measured, and no significant difference was found. Although the experiment made use of student subjects and inspected short pieces of code, the inspection process used was realistic and the rate of inspection was typical of industry. Statistical power in our experiment was sufficient to allows us to reliably accept the null hypothesis for normalised differences above 13%.

Having compared simple tool-based inspection with paper-based inspection, it is believed that by supplying active defect detection support, computer-based inspection can provide significant gains in effectiveness. Such support may be specific to certain document types, such as static analysers

for code, or applicable to any document, such as new browsing methods. Our current research lies in implementing and evaluating such support.

Acknowledgements

The authors would like to thank Marc Roper and Murray Wood for their assistance in running the experiment and their comments on this paper. We also thank David Lloyd and Ian Gordon for their technical help. Our thanks go to all students who participated in the experiment, who also provided very useful feedback on the usability of ASSIST, as well as finding one or two bugs. Finally, we are grateful to Christopher Lott, Eric Kamsties and Gary Perlman for providing the training material used in the experiment, and for allowing us to reproduce this material.

More information on the ASSIST system can be found at

`http://www.cs.strath.ac.uk/Contrib/efocs/assist.html`

A version is freely available for research purposes. For more details, please email the first author at

`fraser@cs.strath.ac.uk`.

Electronic versions of the material used in the experiment are also available from the first author.

References

[1] J. T. Baldwin. An abbreviated C++ code inspection checklist. Available on the WWW, URL: http://www.ics.hawaii.edu/ johnson/FTR/Bib/Baldwin92.html, 1992.

[2] J. Barnard and A. Price. Managing code inspection information. *IEEE Software*, 11(2):56–69, March 1994.

[3] V. R. Basili and R. W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering*, 13(12):1278–1296, December 1987.

[4] C. A. Boneau. The effects of violations of assumptions underlying the t test. *Psychological Bulletin*, 57(1):49–64, 1960.

[5] L. R. Brothers, V. Sembugamoorthy, and A. E. Irgon. Knowledge-based code inspection with ICICLE. In *Innovative Applications of Artificial Intelligence 4: Proceedings of IAAI-92*, 1992.

[6] H.M. Deitel and P.J. Deitel. *C: How to Program*. Prentice-Hall International, second edition, 1994.

[7] A. Dillon. Reading from paper versus screens: a critical review of the empirical literature. *Ergonomics*, 35(10):1297–1326, October 1992.

[8] E. P. Doolan. Experience with Fagan's inspection method. *Software–Practice and Experience*, 22(2):173–182, February 1992.

[9] R. G. Ebenau and S. H. Strauss. *Software Inspection Process*. McGraw-Hill, 1994.

[10] A. L. Edwards. *Statistical Methods*. Holt, Rinehart and Winston, Inc, second edition, 1967.

[11] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[12] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[13] J. W. Gintell, J. Arnold, M. Houde, J. Kruszelnicki, R. McKenney, and G. Memmi. Scrutiny: A collaborative inspection and review system. In *Proceedings of the Fourth European Software Engineering Conference*, September 1993.

[14] J. W. Gintell, M. B. Houde, and R. F. McKenney. Lessons learned by building and using Scrutiny, a collaborative software inspection system. In *Proceedings of the Seventh International Workshop on Computer Aided Software Engineering*, July 1995.

[15] W. S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, 1995.

[16] E. Kamsties and C. M. Lott. An empirical evaluation of three defect-detection techniques. Technical Report ISERN-95-02, International Software Engineering Research Network, May 1995.

[17] J. C. Knight and E. A. Meyers. An improved inspection technique. *Communications of the ACM*, 36(11):51–61, November 1993.

[18] F. Macdonald. ASSIST V1.1 User Manual. Technical Report EFoCS-22-96, Department of Computer Science, University of Strathclyde, February 1997.

[19] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. A review of tool support for software inspection. In *Proceedings of the Seventh International Workshop on Computer Aided Software Engineering*, pages 340–349, July 1995.

[20] F. Macdonald, J. Miller, A. Brooks, M. Roper, and M. Wood. Automating the software inspection process. *Automated Software Engineering: An International Journal*, 3(3/4):193–218, August 1996.

[21] B. Marick. A question catalog for code inspections. Available via anonymous FTP from cs.uiuc.edu as /pub/testing/inspect.ps, 1992.

[22] V. Mashayekhi. *Distribution and Asynchrony in Software Engineering*. PhD thesis, University of Minnesota, March 1995.

[23] J. Miller, J. Daly, M. Wood, M. Roper, and A. Brooks. Statistical power and its subcomponents — missing and misunderstood concepts in empirical software engineering research. *Information and Software Technology*, 1997. To appear.

[24] J. Miller, M. Wood, M. Roper, and A. Brooks. Further experiences with scenarios and checklists. Technical Report EFoCS-20-96, Department of Computer Science, University of Strathclyde, 1996.

[25] A. A. Porter, L. G. Votta, and V. R. Basili. Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Transactions on Software Engineering*, 21(6):563–575, June 1995.

[26] G. W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Software*, 8(1):25–31, January 1991.

[27] D. Tjahjono. Comparing the cost effectiveness of group synchronous review method and individual asynchronous review method using CSRS: Results of pilot study. Technical Report ICS-TR-95-07, University of Hawaii, January 1995.

A Timetable

The experiment was run over a period of ten weeks. Six weeks were used to train students in software inspection and the use of ASSIST, and to refresh their C++ knowledge. Four weeks were used to run the actual experiment. The following timetable was used:

- **Week 1** Individual inspection of `count.cc`.
- **Week 2** Individual inspection of `tokens.cc`, introducing checklist.
- **Week 3** Group meeting to collect the results of Week 2 individual inspection.
- **Week 4** Tutorial introduction to both individual inspection and group meeting using ASSIST with `simple_sort.cc`.
- **Week 5** Individual inspection of `series.cc` using ASSIST.
- **Week 6** Group meeting using ASSIST to collect the results of Week 5 individual inspection.
- **Week 7** Individual inspection of `analyse.cc`. Section 1 made use of ASSIST, while Section 2 performed the inspection on paper.
- **Week 8** Group meeting to collect the results of Week 7 individual inspection, using ASSIST or paper as appropriate.
- **Week 9** Individual inspection of `graph.cc`. Section 1 performed the inspection on paper, while Section 2 made use of ASSIST.
- **Week 10** Group meeting to collect the results of Week 9 individual inspection, using ASSIST or paper as appropriate.

B C++ Checklist

1. **Specification**
   - Is the functionality described in the specification fully implemented by the code?
   - Is there any excess functionality implemented by the code but not described in the specification?
   - Is the program interface implemented as described in the specification?
2. **Initialisation and Declarations**
   - Are all local and global variables initialised **before use**?
   - Are variables and class members i) required, ii) of the appropriate type and iii) correctly scoped?
3. **Function Calls**
   - Are parameters presented in the correct order?
   - Are pointers and & used correctly?
   - Is the correct function being called, or should it be a different function with a similar name?
4. **Arrays**
   - Are there any off-by-one errors in array indexing?
   - Can array indexes ever go out-of-bounds?
5. **Pointers and Strings**
   - Check that pointers are initialised to NULL
   - Check that pointers are never unexpectedly NULL
   - Check that all strings are identified by pointers and are NULL-terminated at all points in the program
6. **Dynamic Storage Allocation**
   - Is too much/too little space allocated?
7. **Output Format**
   - Are there any spelling or grammatical errors in displayed output?
   - Is the output formatted correctly in terms of line stepping and spacing?
8. **Computation, Comparisons and Assignments**
   - Check order of computation/evaluation, operator precedence and parenthesising
   - Can the denominator of a division ever be zero?
   - Is integer arithmetic, especially division, ever used inappropriately, causing unexpected truncation/rounding?
   - Are the comparison and boolean operators correct?
   - If the test is an error-check, can the error condition actually be legitimate in some cases?
   - Does the code rely on any implicit type conversions?
9. **Flow of Control**

- In a *switch* statement, is any case not terminated by *break* or *return*?
- Do all *switch* statements have a *default* branch?
- Are all loops correctly formed, with the appropriate initialisation, increment and termination expressions?

10. **Files**

    - Are all files properly declared and opened?
    - Is a file not closed in the case of an error?
    - Are EOF conditions detected and handled correctly?

C Individual Defect Report Form

Complete this form in BLOCK CAPITALS in blue or black ink. Each defect description must be complete and accurate. Any description not satisfying the above criteria will be IGNORED.

Name: **Group:**
Start time: **End time:**

Defect No. 1 Time: Location:

Defect No. 2 Time: Location:

Defect No. 3 Time: Location:

Defect No. 4 Time: Location:

Defect No. 5 Time: Location:

Defect No. 6 Time: Location:

Defect No. 7 Time: Location:

Defect No. 8 Time: Location:

Defect No. 9 Time: Location:

Defect No. 10 Time: Location:

Defect No. 11 Time: Location:

Defect No. 12 Time: Location:

Defect No. 13 Time: Location:

Defect No. 14 Time: Location:

Defect No. 15 Time: Location:

Defect No. 16 Time: Location:

Defect No. 17 Time: Location:

Defect No. 18 Time: Location:

Defect No. 19 Time: Location:

Defect No. 20 Time: Location:

Defect No. 21 Time: Location:

Defect No. 22 Time: Location:

Defect No. 23 Time: Location:

Defect No. 24 Time: Location:

Defect No. 25 Time: Location:

Defect No. 26 Time: Location:

D Master Defect Report Form

Complete this form in BLOCK CAPITALS in blue or black ink. Each defect description must be complete and accurate. Any description not satisfying the above criteria will be IGNORED.

Group:
Moderator: **Scribe:**
Reader: **Inspector (if present):**
Start time: **End time:**

Defect No. 1 Time: Location:

____
____
____
____

Defect No. 2 Time: Location:

____
____
____
____

Defect No. 3 Time: Location:

____
____
____
____

Defect No. 4 Time: Location:

____
____
____
____

Defect No. 5 Time: Location:

____
____
____
____

Defect No. 6 Time: Location:

____
____
____
____

Defect No. 7 Time: Location:

Defect No. 8 Time: Location:

Defect No. 9 Time: Location:

Defect No. 10 Time: Location:

Defect No. 11 Time: Location:

Defect No. 12 Time: Location:

Defect No. 13 Time: Location:

Defect No. 14 Time: Location:

Defect No. 15 Time: Location:

Defect No. 16 Time: Location:

Defect No. 17 Time: Location:

Defect No. 18 Time: Location:

Defect No. 19 Time: Location:

Defect No. 20 Time: Location:

Defect No. 21 Time: Location:

Defect No. 22 Time: Location:

Defect No. 23 Time: Location:

Defect No. 24 Time: Location:

Defect No. 25 Time: Location:

E Training Program count.cc

E.1 Specification for program count

Name

count – count lines, words, and characters

Usage

count filename [filename ...]

Description

count counts the number of lines, words, and characters in the named files. Words are sequences of characters that are separated by one or more spaces, tabs, or line breaks (carriage return).

If a file supplied as an argument does not exist, a corresponding error message is printed and processing of any other files continues. If no file is supplied as an argument, **count** reads from the standard input.

The computed values are given for each file (including the name of the file) as well as the sum of all values. If only a single file or if the standard input is processed, then no sum is printed. The output is printed in the order lines, words, characters, and either the file name or the word "total" for the sum. If the standard input is read, the fourth value (name) is not printed.

Options

None.

Examples

On a single file:

```
% count data
     84      462     3621 data
```

On multiple files:

```
% ./count file1 file2 file3
      3       24      120 file1
      5       49      196 file2
     17      175      787 file3
     25      248     1103 total
```

Author

Original by Erik Kamsties and Christopher Lott. C++ conversion and update by Fraser Macdonald.

E.2 Library functions used in `count.cc`

- `open()`

  Opens the corresponding I/O stream.

- `char get(void)`

  `get` inputs one character from the designated stream and returns this character as the result of the function call. If end-of-file on the stream is encountered, `get` returns EOF.

- `good()`

  Returns true if the corresponding I/O stream is available for use.

- `width()`

  Sets the field width and returns the previous width for this stream. The width setting applies only to the next stream insertion or extraction.

E.3 count.cc

```
 1  #include <iostream.h>
 2  #include <fstream.h>
 3  #include <stdlib.h>
 4
 5  void main (int argc, char* argv[])
 6  {
 7    int   c, i, inword;
 8    ifstream InputFile;
 9    long  linect, wordct, charct;
10    long  tlinect = 1, twordct = 1, tcharct = 1;
11
12    i = 1;
13    do {
14      if (argc > 1) {
15        InputFile.open(argv[i]);
16        if (!InputFile.good()) {
17          cout <<  "can't open " << argv[i] << endl;
18          exit(1);
19        }
20      }
21      linect = wordct = charct = 0;
22      inword = 1;
23      while ((c = InputFile.get()) != EOF) {
24        ++charct;
25        if (c == '\n')
26          ++linect;
27        if (c == ' ' || c == '\t' || c == '\n')
28          inword = 0;
29        else if (inword == 0) {
30          inword = 1;
31          ++wordct;
32        }
33      }
34      cout.width(7);
35      cout << linect;
36      cout.width(7);
37      cout << wordct;
38      cout.width(7);
39      cout << charct;
40      if (argc > 1)
41        cout << " " << *argv << endl;
42      else
43        cout << endl;
44      InputFile.close();
45      tlinect += linect;
46      twordct += wordct;
47      tcharct += charct;
48    } while (++i < argc);
49    if (argc > 1) {
50      cout.width(7);
```

```
51      cout << linect;
52      cout.width(7);
53      cout << twordct;
54      cout.width(7);
55      cout << tcharct << " total" << endl;
56    }
57    exit(0);
58  }
```

E.4 Faults in `count.cc`

1. Fault in line 10: The variables are initialised with 1, should be with 0.

   Causes failure: The sums are incorrect (off by one).

2. Fault in line 14: The variable “InputFile” is not initialised in the case that the input should be taken from “stdin”.

   Causes failure: The program cannot read from stdin.

3. Fault in line 17: The error message is sent to “stdout” instead of “stderr”.

   Causes failure: Error messages appear on the standard output (stdout) instead of the standard-error output (stderr).

4. Fault in line 18: Component is terminated with “exit (1)”, where “continue” should have been used.

   Causes failure: If a file is not found, the program stops there instead of continuing on to other files; also, no sum is printed.

5. Fault in line 22: The variable “inword” is initialised with 1 instead of 0.

   Causes failure: Depending on whether the first symbol in a file is whitespace, the program reports that files with n words have either n or n - 1 words.

6. Fault in line 41: *argv is used instead of argv[i].

   Causes failure: The program prints its own name instead of the file name when reporting the counts.

7. Fault in line 49: Argc is compared with 1, but should be compared with 2.

   Causes failure: The program prints out sums even when only a single file was processed.

8. Fault in line 51: Instead of “tlinect” the variable “linect’ is used.

   Causes failure: The sums are not computed correctly. For example:

```
% ./count file2 file2 file2
      1       2      14 ./count
      1       2      14 ./count
      1       2      14 ./count
      1       7      43 total
```

F Training Program `tokens.cc`

F.1 Specification for program `tokens`

Name

tokens – sort and count alphanumeric tokens

Usage

tokens [–ai] [–c chars] [–m count]

Description

tokens reads its input from the standard input and counts all alphanumeric tokens. The tokens are then printed in alphabetic order, along with their counts. Options may be specified to tailor the output. If incorrect options are given, **tokens** will print a usage message.

Options

- "–a": Allow only alphabetic characters in tokens (no digits 0–9).
- "–c chars": Allow chars to be part of tokens.
- "–i": The **–i** flag causes the program to ignore the difference between upper and lower case by mapping all input to lower case.
- "–m count": The **–m** flag indicates the minimum count needed for the entry to be printed.

Examples

In its simplest form with no options:

```
% tokens
this is a_test
      1 a
      1 is
      1 test
      1 this
```

Allowing alphabetic characters only:

```
% tokens -a
test number 2
      1 number
      1 test
```

Using the "–c" option to allow "+" in tokens:

```
% tokens -c+
one on+ + +
      2 +
      1 on+
      1 one
```

Using "–i" to ignore the difference between lower and upper case characters, and using "–m" to set a minimum count of two:

```
% tokens -i -m2
Orange apple orange orange apple banana
      2 apple
      3 orange
```

Author

Original by Gary Perlman. C++ conversion and update by Fraser Macdonald.

F.2 Library functions used in `tokens.cc`

- `void assert(int expression)`

  Tests the value of the supplied expression. If the value is 0, assert prints an error message and terminates program execution.

- `int atoi(const char* s)`

  Converts the string given as argument to its `int` representation.

- `char get(void)`

  `get` inputs one character from the designated stream and returns this character as the result of the function call. If end-of-file on the stream is encountered, `get` returns EOF.

- `getopt(int argc, char **argv, char *optstring)`
  `extern char *optarg`
  `extern int optind`

  `getopt()` returns the next option letter in argv that matches a letter in `optstring`. `optstring` must contain the option letters the command using `getopt()` will recognise; if a letter is followed by a colon, the option is expected to have an argument, or group of arguments, which must be separated from it by white space.

  `optarg` is set to point to the start of the option argument on return from `getopt()`. When all options have been processed, `getopt()` returns -1.

  For example,

  ```
  c = getopt(argc, argv, "b:fp:")
  ```

  indicates that three options may be given to the program. The first is "–b", which should have an argument, followed by "–f", then "–p", which should also have an argument (e.g. "–f20"). Each successive call to `getopt()` will return one of these letters, if they are present, and will set `optarg` to point to any argument that may be present for this option. When all options present have been parsed, `getopt()` will return -1.

  See the entry for `getopt` in Section 3 of the man pages for more details.

- `int strcmp(const char *s1, const char *s2)`

  Compares the two strings passed as arguments, returning an integer greater than, less than, or equal to zero when s1 is respectively greater than, less than or equal to s2.

- `char *strdup(char *s)`

  Returns a pointer to a new string which is a duplicate of the string pointed to by s.

- `int width(const int newwidth)`

  Sets the field width and returns the previous width for this stream. The width setting applies only to the next stream insertion or extraction.

F.3 tokens.cc

```
 1  #include <iostream.h>
 2  #include <string.h>
 3  #include <assert.h>
 4
 5  int     Ignore = 0;
 6  int     Mincount = 0;
 7  int     Alphabetic = 0;
 8  char    MapAllowed[256];
 9
10  typedef struct tnode
11  {
12    char   *contents;
13    int    count;
14    struct tnode *left;
15    struct tnode *right;
16  } TNODE;
17
18  void treeprint(TNODE *);
19  TNODE *install(char *, TNODE *);
20  char *getword(void);
21  void tokens(void);
22
23  int main(int argc, char **argv)
24  {
25    int c, errcnt = 0;
26    extern char *optarg;
27
28    while ((c = getopt(argc, argv, "ac:im:")) != EOF)
29      switch(c)
30        {
31        case 'a': Alphabetic = 0; break;
32        case 'c':
33          while (*optarg)
34            {
35              MapAllowed[*optarg] = *optarg;
36              optarg++;
37            }
38          break;
39        case 'i': Ignore = 1; break;
40        case 'm': Mincount = atoi(optarg); break;
41        }
42    if (errcnt)
43      {
44        cerr << "Usage: " << *argv << " [-i] [-c chars] [-m count]" << endl;
45        return(1);
46      }
47    for (c = 'a'; c <= 'z'; c++)
48      MapAllowed[c] = c;
49    for (c = 'A'; c <= 'Z'; c++)
50      MapAllowed[c] = Ignore ? c : c - 'A' + 'a';
```

```
 51    if (!Alphabetic)
 52      for (c = '0'; c <= '9'; c++)
 53        MapAllowed[c] = c;
 54    tokens();
 55    return(0);
 56  }
 57
 58  void treeprint(TNODE *tree)
 59  {
 60    if (tree != NULL)
 61      {
 62        treeprint(tree->left);
 63        if (tree->count > Mincount) {
 64          cout.width(7);
 65          cout << tree->count;
 66          cout << "\n" << tree->contents << endl;
 67        }
 68        treeprint(tree->right);
 69      }
 70  }
 71
 72  TNODE *install(char *string, TNODE *tree)
 73  {
 74    int cond;
 75    assert(string != NULL);
 76    if (tree == NULL)
 77      {
 78        if (tree = new TNODE)
 79          {
 80            tree->contents = strdup(string);
 81            tree->count = 1;
 82          }
 83      }
 84    else
 85      {
 86        cond = strcmp(string, tree->contents);
 87        if (cond < 0)
 88          tree->left = install(string, tree->left);
 89        else if (cond == 0)
 90          tree->count++;
 91        else
 92          tree->right = install(string, tree->left);
 93      }
 94    return(tree);
 95  }
 96
 97  char *getword(void)
 98  {
 99    static char string[1024];
100    char *ptr = string;
101    int c, count = 0;
102    for (;;)
```

```
103      {
104        c = cin.get();
105        if (c == EOF)
106          if (ptr == string)
107            return(NULL);
108          else
109            break;
110        if (!MapAllowed[c])
111          if (ptr == string)
112            continue;
113          else
114            break;  // end of word
115        *ptr++ = MapAllowed[c];
116      }
117    *ptr = NULL;
118    return(string);
119  }
120
121  void tokens(void)
122  {
123    TNODE *root = NULL;
124    char *s;
125    while (s = getword())
126      root = install(s, root);
127    treeprint(root);
128  }
```

F.4 Faults in `tokens.cc`

1. Fault in function “main”, line 27 (circa): The array “MapAllowed” is never initialised.

   Failure: Non-alphanumeric symbols could be mistakenly accepted, depending on the contents of MapAllowed.

2. Fault in function “main”, line 31: The variable “Alphabetic” is given the value 0 instead of 1.

   Failure: The argument “-a” has no effect.

3. Fault in function “main”, line 41 (circa): No default branch for the case statement.

   Failure: Arguments other than those defined in the specification do not cause a usage message to be printed.

4. Fault in function “main”, line 44: The argument “-a” is not documented in the usage message.

   Failure: The usage message says nothing about the “-a” argument.

5. Fault in function “main”, line 50: “-i” option not implemented correctly, the branches of the “?” operator are transposed.

   Failure: Upper case and lower case characters are always distinguished, irrespective of the use of the “-i” option.

6. Fault in function “treeprint”, line 63: The symbol “>” should be “>=”.

   Failure: If a boundary value n is given with the “-m” argument, the value $n+1$ is used instead of n.

7. Fault in function “treeprint”, line 66: The escape sequence “\n” is used instead of “\t”.

   Failure: The output is not formatted correctly. Each token should appear on a line with its count. As written, the program outputs the token, then the count on a new line.

8. Fault in function “install”, line 82 (circa): The left and right branches of the tree should be initialised to NULL.

   Failure: No failure apparent, but this may be implementation dependent, i.e. depending on the definition of NULL. Also checklist violation.

9. Fault in function “install”, line 92: The function install is called with incorrect parameters. tree− >left should be tree− >right.

   Failure: New tokens are inserted into the tree incorrectly. The output is therefore generally unreliable.

10. Fault in function “getword”, line 101: The variable count is declared but never used.

    Failure: None apparent, but checklist violation.

11. Fault in function “getword”, line 103/117: The length of the input is not checked.

    Failure: The program dumps core if a very long word is read.

G Training Program `simple_sort.cc`

G.1 Specification for program "simple_sort"

Name

simple_sort – sort a list of numbers entered by the user

Usage

simple_sort

Description

simple_sort starts by prompting for the number of items to be sorted. The program then reads reads in the list of numbers from the user, sorts them into ascending numerical order, then prints the sorted list.

Options

None.

Example

Sorting a list of ten numbers:

```
% simple_sort
Enter the number of data values: 10
Data item 0: 5
Data item 1: 6
Data item 2: 7
Data item 3: 8
Data item 4: 2
Data item 5: 3
Data item 6: 9
Data item 7: 1
Data item 8: 4
Data item 9: 10

Sorted list:
Data item 0: 1
Data item 1: 2
Data item 2: 3
Data item 3: 4
Data item 4: 5
Data item 5: 6
Data item 6: 7
Data item 7: 8
Data item 8: 9
Data item 9: 10
```

Restrictions

The number of elements which can be sorted is currently limited to 100.

Author

Fraser Macdonald.

G.2 simple_sort.cc

```
 1 #include <iostream.h>
 2
 3 const int TABLESIZE = 100;
 4
 5 void swap(int& x, int& y)
 6 {
 7  int temp = x;
 8  x = y;
 9  y = temp;
10 }
11
12 int max(int x, int y)
13 {if (x > y) return x; else return y;}
14
15 int main()
16 {
17   int size;
18   int table[TABLESIZE];
19
20   cout << "Enter the number of data values: ";
21   cin >> size;
22
23   if(size >= TABLESIZE)
24     cout << "Too many elements, maximum is " << TABLESIZE << endl;
25   else {
26     for(int i = 0; i < size; i++){
27       cout << "Data item " << i << ": ";
28       cin >> table[i];
29     }
30     for(i = size - 1; i > 0; i--)
31       for(int j = 0; j <=i - 1; j++)
32         if(table[j] > table[j+1])
33           swap(table[j], table[j+1]);
34
35     cout << endl << "Sorted lits:" << endl;
36     for(i = 0; i < size; i++)
37       cout << "Data item " << i << ": " << table[i] << endl;
38   }
39   return(0);
40 }
41
```

G.3 Faults in `simple_sort.cc`

1. Fault in function “swap”, line 5 : The parameters are passed by value, not by reference.

   Failure: “swap” doesn’t correctly swap the numbers, therefore the sort is not carried out correctly.

2. Fault in function “max”, line 12 : The function “max” is defined but never used.

   Failure: None apparent, but checklist violation.

3. Fault in function “main”, line 23 : >= should be >.

   Failure: The program only accepts one less than the true maximum number of elements.

4. Fault in function “main”, line 35 : “list” is spelled incorrectly in the message.

   Failure: The program displays incorrect output.

H Training Program `series.cc`

H.1 Specification for program "series.cc"

Name

series – generate a series of numbers

Usage

series start end [stepsize]

Description

series prints the real numbers from start to end, one per line. **series** begins with start to which `stepsize` is repeatedly added or subtracted, as appropriate, to approach, possibly meet, but not pass `end`.

If all arguments are integers, only integers are produced in the output. The stepsize must be nonzero; if it is not specified, it is assumed to be 1. Negative step sizes are made positive. In all other cases, series prints an appropriate error message. If the wrong number of arguments are given, series prints a usage message.

series accepts numbers in several formats: integer, real (where either the whole or fractional part may be omitted) and exponential (an integer or real, suffixed with 'e' or 'E' followed by a (signed) integer exponent). Any number with a fractional part consisting only of zeroes is converted to an integer (e.g. 1.0000 is treated as 1). All numbers may optionally be prefixed by a plus or minus. Examples of acceptable numbers include:

```
+23
-2.4
26.0
92.
.348
1.0E3
34e-2
```

Example

To count from 1 to 100:

```
% series 1 100
1
2
3
....
98
99
100
```

To do the same, but backwards:

```
% series 100 1
100
99
98
...
```

```
3
2
1
```

To count from 1.5 to 4.5 in steps of 0.5

```
% series 1.5 4.5 0.5
1.5
2
2.5
3
3.5
4
4.5
```

Limitations

The reported number of significant digits is limited. If the ratio of the series range to the stepsize is too large, several numbers in a row will be equal.

The maximum length of a series is limited to the size of the maximum integer that can be represented on the machine in use. Exceeding this value has undefined results.

Author

Original by Gary Perlman. C++ conversion and update by Fraser Macdonald.

H.2 Library functions used in `series.cc`

- `double atof(char *nptr)`

  Converts the initial portion of the string pointed to by `nptr` to double representation. The function returns the converted value.

- `double fabs(double x)`

  Computes the absolute value of number x

- `int isdigit(char c)`

  Returns non-zero integer if the character `c` is a decimal digit.

- `int isspace(char c)`

  Returns non-zero integer if the character `c` is a standard whitespace character. The standard whitespace characters are: space(' '), form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t') and vertical tab ('\v').

H.3 series.cc

```
1 #include <stdlib.h>
2 #include <ctype.h>
3 #include <iostream.h>
4
5 #define IS_NOT  0
6 #define IS_INT  1
7 #define IS_REAL 2
8 #define IS_EXP  3
9 #define IS_RAT  4
10 #define FZERO   10e-10
11
12 #define startstr  argv[1]
13 #define endstr    argv[2]
14 #define stepstr   argv[3]
15
16 #define fzero(x) (fabs (x) < FZERO)
17
18 int isinteger (char *string);
19 int number(char *string);
20
21 void main (int argc, char **argv)
22 {
23   long NumItems = 0;
24   double Value = 0.0;
25   double Start = 0.0;
26   double End = 0.0;
27   double Step = 1.0;
28   int NumArgs = argc;
29   int OnlyInt = 0;
30
31   switch (NumArgs)
32     {
33     case 3:
34       if (! number(startstr)) {
35   cout << "Argument #1 not a number: " << startstr << endl;
36   exit(1);
37       }
38       if (! number(startstr)) {
39   cout << "Argument #2 not a number: " << endstr << endl;
40   exit(1);
41       }
42       if (! number(stepstr)) {
43   cout << "Argument #3 not a number: " << stepstr << endl;
44   exit(1);
45       }
46       break;
47     case 2:
48       if (! number(startstr)) {
49   cout << "Argument #1 not a number: " << endstr << endl;
50   exit(1);
51       }
```

```
52        if (! number(endstr)) {
53    cout << "Argument #2 not a number: " << endstr << endl;
54    exit(1);
55        }
56        break;
57      default:
58        cout << "Usage: " << argv << " start end stepsize" << endl;
59        exit(1);
60      }
61
62    Start = atof(startstr);
63    End = atof(endstr);
64    OnlyInt = isinteger(startstr) && isinteger(endstr);
65    if (NumArgs == 3) {
66        Step = fabs(atof(stepstr));
67        if (! fzero(End - Start) && fzero(Step))
68    cout << "stepsize must be non-zero" << endl;
69    exit(0);
70        OnlyInt = OnlyInt && isinteger(stepstr);
71    }
72
73    if (fzero(End - Start))
74      NumItems = 2;
75    else
76      NumItems = (long) (End - Start / Step + 1.0 + FZERO);
77
78    for (int Item = 0; Item < NumItems; Item++) {
79        Value = Start + Step * (double) Item;
80        if (fzero(Value))
81  cout << 0.0 << endl;
82        else
83  cout << Value << endl;
84    }
85
86    exit(0);
87  }
88
89  int isinteger (char *string)
90  {
91   return (number(string) == IS_INT);
92  }
93
94  int number (char *string)
95  {
96      int     answer = IS_REAL,
97              before = 0,
98              after = 0;
99      char    *ptr = NULL;
100
101     while (isspace(*string))
102         string++;
103     if (*string == NULL)
```

```
104         return (IS_NOT);
105     if (*string == '+' || *string == '-') {
106         string++;
107         if (isdigit(*string) && *string != '.')
108              return (IS_NOT);
109     }
110     if (isdigit(*string)) {
111         before = 1;
112         while (isdigit(*string))
113             string++;
114     }
115     if (*string == '.') {
116         string++;
117         ptr = string;
118         while (*ptr == '0')
119             ptr++;
120         while (isspace(*ptr))
121             ptr++;
122         if (*ptr == NULL)
123             return (IS_INT);
124         answer = IS_REAL;
125         if (isdigit(*string)) {
126             after = 1;
127             while (isdigit (*string))
128                  string++;
129         }
130     }
131     if (!before && !after)
132         return (IS_NOT);
133     if (*string == 'E' || *string == 'e') {
134         answer = IS_EXP;
135         string++;
136         if (*string == '+' || *string == '-')
137             string++;
138         if (!isdigit(*string))
139             return (IS_NOT);
140         while (isdigit(*string))
141             string++;
142     }
143     while (isspace(*string))
144        string++;
145     return(answer);
146 }
```

H.4 Faults in program "series"

1. Fault in line 9: `IS_RAT` is defined and not used.

   Failure: None, but checklist violation.

2. Fault in line 28: `NumArgs` is initialised to `argc` instead of `argc - 1`

   Failure: With one or two arguments, a segmentation fault occurs. Any other number of arguments always produces a usage message.

3. Fault in function `main()`, line 29 (and others): The variable `Onlyint`, plus all associated code has absolutely no effect on the output of the program.

   Failure: None, but checklist violation.

4. Fault in function `main()`, line 35, 39, 43, 49, 53: Error messages are sent to `stdout` instead of `stderr`

   Failure: None visible, but depends on environment in which program is run.

5. Fault in function `main()`, line 38: The variable `startstr` is used in the if-condition instead of `endstr`.

   Failure: The program does not recognise a non-numeric second argument as an error.

6. Fault in function `main()`, line 49: The variable `endstr` is used in the output instead of `startstr`.

   Failure: Although a non-numeric first argument is recognised as an error, the corresponding error message shows the second argument.

7. Fault in function `main()`, line 58: `argv` should be `*argv`

   Failure: Instead of the program name, the program prints the address of its name.

8. Fault in function `main()`, line 67: Mismatch with specification.

   Failure: The specification states that `stepsize` should always be non-zero. In fact, the programs accepts zero for `stepsize` if the difference between the start and end values is also zero.

9. Fault in function `main()`, line 67–69: Missing {} for the if clause.

   Failure: If three arguments are given, the program always terminates without any input, since `exit(0)` is always executed.

10. Fault in function `main()`, line 74: The variable `NumItems` is set to 2 instead of 1.

    Failure: If the distance between the first and second parameter is evaluated to zero, then two lines are produced as output although only one was expected.

11. Fault in function `main()`, line 76: The calculation `End - Start` should be enclosed in parentheses.

    Failure: The value assigned to `NumItems` is incorrect in all cases except where the step size is 1, since the calculation is performed in the wrong order (division is performed first).

12. Fault in function `main()`, line 78–84: Treatment of the case "end < start" was forgotten.

    Failure: No output is produced in the case that the starting value is greater then the ending value.

13. Fault in function `number()`, line 96: `answer` is initialised to `IS_REAL`, should be `IS_INT`.

    Failure: `number()` will only return `IS_INT` if the number given is of the form 1.0, 51.000, etc. Normal integers will be mistakenly classified as reals.

14. Fault in function `number()`, line 107: `isdigit(*string)` should be `!isdigit(*string)`

    Failure: `number()` will not recognise numbers starting with signs, such as +1.4, -23, etc. The exceptions are anything of the form +.2, -.982, etc., i.e. any number with a point straight after the sign.

15. Fault in function `number()`, line 145: The string should be checked for any remaining characters after a number has been parsed.

    Failure: If a number is terminated by non-numeric characters, `number()` does not return `IS_NOT`.

I Experiment Program `analyse.cc`

I.1 Specification for program `analyse`

Name

analyse – perform simple analysis on survey data

Usage

analyse file

Description

analyse performs simple statistical analysis on a file containing survey responses. Each response is an integer from 0 to 9 (inclusive). The program calculates four statistics: mean, median, mode and standard deviation.

For the mean, the calculation is presented, along with the answer. For the median, both the unsorted and sorted arrays of responses are printed (formatted in rows of up to twenty numbers), followed by the value of the median. The median is the central value of the ordered data set. If the number of elements in the data set is even, the median is the average of the two most central values. For the mode, a histogram of frequencies of each response is drawn, followed by the value of the mode and its frequency. The mode is the most frequently occurring response. If more than one response shares the highest frequency, the lowest valued response is chosen. For the standard deviation, the answer is simply presented. It is calculated according to the formula $\sqrt{\frac{S}{N}}$, where S is the sum of the squares of the differences between each data element and the mean, and N is the number of elements in the data set.

The input file consists of a list of integer responses in the range 0 to 9, one to each line. The first line of the file contains an integer indicating the number of responses in the file. This value must be greater than zero, otherwise an error message is printed. If no filename is given, the program prints a usage error. If the file cannot be accessed, an appropriate error message is printed.

Example

A typical data file might be:

```
8
6
5
8
9
5
8
5
1
```

The first line indicates that eight responses are included in the file. This is then followed by each of the eight responses. The output for this file is:

```
% analyse data
Input file data being processed.

End of input file data
```

```
Mean = 47/8 = 5.875

The unsorted array is

 6 5 8 9 5 8 5 1

The sorted array is

 1 5 5 5 6 8 8 9

Median is 5.5

Response  Frequency          Histogram

                                      1    1    2    2
                                 5    0    5    0    5

       0          0
       1          1          *
       2          0
       3          0
       4          0
       5          3          ***
       6          1          *
       7          0
       8          2          **
       9          1          *

Mode is 5, occurring 3 times.

Standard deviation is 2.36841
```

Restrictions

The histogram output will only properly display a maximum frequency of 25 responses.

Author

Written by Fraser Macdonald, based on an example from *C: How to Program* by Deitel and Deitel.

I.2 Library functions used in `analyse.cc`

- `int eof(void)`

  Returns 1 if end-of-file has been encountered in the corresponding stream, otherwise returns 0.

- `double fabs(double x)`

  Computes the absolute value of floating point number `x`.

- `int good(void)`

  Returns 1 if the corresponding I/O stream is available for use, otherwise returns 0.

- `int open(char* s)`

  Opens the corresponding I/O stream.

- `int setw(int x)`

  Sets the field width and returns the previous width for this stream.

- `double sqrt(double x)`

  Computes the non-negative square root of `x`. An error occurs if the argument is negative.

I.3 analyse.cc

```
 1  #include <iostream.h>
 2  #include <iomanip.h>
 3  #include <fstream.h>
 4  #include <math.h>
 5
 6  void Mean (int Array[], int Size);
 7  void Median (int Array[], int Size);
 8  void Mode(int Array[], int Size);
 9  void StandardDeviation(int Array[], int Size);
10  void BubbleSort(int Array[], int Size);
11  void PrintArray(int Array[], int Size);
12
13  int main(int argc, char **argv)
14  {
15      int Item = 0,
16          Size = 0,
17          Count = 0;
18      int *Responses = NULL;
19      ifstream InputFile;
20      char *Filename = NULL;
21
22      if (argc != 2) {
23        cerr << "Usage: " << argv[1] << " file" << endl;
24        return(-1);
25      }
26      Filename = argv[1];
27      InputFile.open(Filename);
28      if (!InputFile.good()) {
29        cerr << "File error." << endl;
30        InputFile.close();
31        return(-1);
32      }
33      cout << "Input file " << Filename << " being processed." << endl << endl;
34      InputFile >> Size;
35      if (Size > 0) {
36        cerr << "Number of responses must be greater than 0." << endl;
37        InputFile.close();
38        return(-1);
39      }
40      else
41        if (!(Responses = new int [Size])) {
42          cerr << "Memory allocation failure." << endl;
43          return(-1);
44        }
45      while (!InputFile.eof()) {
46        InputFile >> Item;
47        Responses[Count++] = Item;
48      }
49      InputFile.close();
50      Mean(Responses, Size);
51      Median(Responses, Size);
```

```
 52     Mode(Responses, Size);
 53     StandardDeviation(Responses, Size);
 54     cout << "End of input file " << Filename << endl << endl;
 55     return(0);
 56  }
 57
 58  void Mean(int Array[], int Size)
 59  {
 60     int Total = 0;
 61
 62     for (int j =  0; j < Size; j++)
 63        Total += Array[j];
 64     cout << "Mean = " << Total << "/" << Size << " = "
 65          << Total / Size << endl << endl;
 66  }
 67
 68  void Median(int Array[], int Size)
 69  {
 70     int Median;
 71
 72     BubbleSort(Array, Size);
 73     cout << "The sorted array is" << endl;
 74     PrintArray(Array, Size);
 75     Median = Array[Size / 2];
 76     cout << "Median is " << Median << endl << endl;
 77  }
 78
 79  void Mode(int Array[], int Size)
 80  {
 81     int Rating, j, h,
 82         Largest = 0,
 83         ModeValue = 0;
 84     int Frequency[10];
 85
 86     for (j = 0; j < Size; j++)
 87        Frequency[Array[j]]++;
 88
 89     cout << "Responce";
 90     cout << setw(11) << "Frequency";
 91     cout << setw(19) << "Histogram" << endl << endl;
 92     cout << setw(54) << "1    1    2    2" << endl;
 93     cout << setw(54) << "5    0    5    0    5" << endl << endl;
 94
 95     for (Rating = 0; Rating <= 9; Rating++) {
 96        cout << setw(8) << Rating;
 97        cout << setw(11) << Frequency[Rating] << "          ";
 98        if (Frequency[Rating] > Largest)
 99           Largest = Frequency[Rating];
100           ModeValue = Rating;
101        for (h = 1; h < Frequency[Rating]; h++)
102            cout << "*";
103        cout << endl;
```

```
104     }
105
106     cout << endl << "Mode is " << ModeValue << ", occurring " << Largest
107          << " times." << endl << endl;
108  }
109
110  void StandardDeviation(int Array[], int Size)
111  {
112      double Total = 0.0,
113             Mean = 0.0,
114             StdDev = 0.0;
115
116      for (int j = 0; j < Size; j++)
117         Total += (double) Array[j];
118      Mean = Total / (double) Size;
119
120      for (j = 0; j < Size; j++)
121         Total = Total + fabs((double) Array[j] - Mean)
122                       * fabs((double) Array[j] - Mean);
123      StdDev = sqrt(Total / (double) Size);
124      cout << "Standard deviation is "  << StdDev << endl << endl;
125  }
126
127  void BubbleSort(int Array[], int Size)
128  {
129     int Pass, Hold, j;
130
131     for (Pass = 1; Pass < Size; Pass++)
132        for (j = 0; j < Size - 1; j++)
133           if (Array[j] > Array[j+1]) {
134              Hold = Array[j];
135              Array[j] = Array[j+1];
136              Array[j+1] = Hold;
137           }
138  }
139
140  void PrintArray(int Array[], int Size)
141  {
142     for (int j = 0; j < Size; j++) {
143         if (j % 20 == 0) cout << endl;
144         cout << setw(2) << Array[j];
145     }
146     cout << endl << endl;
147  }
```

J Experiment Program graph.cc

J.1 Specification for program graph

Name

graph – draw a graph

Usage

graph file

Description

Given an input file of ordered pairs (x, y) of either positive or negative integers as input, the program displays the list of points read in and plots them on a grid with a horizontal x-axis and a vertical y-axis which are appropriately labeled, and have 'tick' marks every five units. A plotted point on the grid appears as an asterisk (*), and the grid is scaled to fit into an area with a maximum width of 40 characters and a maximum height of 20 characters.

Vertical scaling is handled as follows. The total height of the graph is calculated as the difference between the largest y value (or zero if the largest is negative) and the smallest y value (or zero if the smallest value is positive). If this height is less than the maximum height of the graph, no scaling is carried out and the graph is plotted with with vertical spacing of one line per integral unit (e.g., the point (3, 6) should be plotted on the sixth line above the origin; two lines above the point (3, 4)). Note that the origin (point (0, 0)) corresponds to the intersection of the axes (the x-axis is referred to as the Oth line). The origin is represented by a '+' and the graph is drawn to ensure that the origin and axes always appear.

If the height is greater than the maximum height of the graph, the scale for vertical spacing is set to the maximum possible height divided by the total height required. This scaling factor is the applied to every point on the graph and the result rounded appropriately to ensure the point lies within the correct interval. For example, if the the graph was required to display the points (1, 1) and (1, 99) the total height is 100 (since the origin must also be displayed). The scaling factor is then $20/100 = 0.2$. (1, 1) is displayed on the 0th line (which covers the interval 0 to 4) and (1, 99) is displayed on the 19th line (which covers 95–99). Negative coordinate values are treated in a similar way. Horizontal scaling is handled similarly.

If two or more of the points to be plotted would show up as the same asterisk in the grid, the number of points occurring on that grid position appears instead of the asterisk. Points whose asterisks will lie on an axis or other marker show up in place of that item.

The input file consists of list of integer coordinates, with each x-coordinate followed by the corresponding y-coordinate. If no filename is given, the program prints a usage error. If the file cannot be accessed, an appropriate error message is printed. If an odd number of coordinates are present in the file, an appropriate error message is printed.

Example

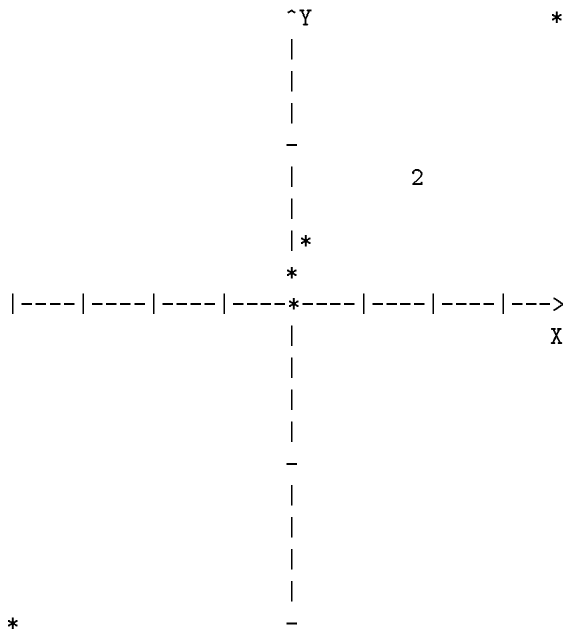
A typical data file might be:

```
-100
-100
0
0
4
19
5
```

```
20
99
99
49
49
48
48
```

The printed output from this file is:

```
% graph data
(48, 48)
(49, 49)
(99, 99)
(5, 20)
(4, 19)
(0, 0)
(-100, -100)
```



Restrictions

The program will only correctly deal with up to nine overlapping points, since numbers beyond this occupy more than a single grid position.

Author

Written by Fraser Macdonald, based on a specification from Basili and Selby's *Comparing the effectiveness of software testing techniques*, IEEE Transactions on Software Engineering, 13(12), December 1987.

J.2 Library functions used in `graph.cc`

- `int abs(int x)`

  Computes the absolute value of integer `x`.

- `int good(void)`

  Returns 1 if the corresponding I/O stream is available for use, otherwise returns 0.

- `int open(char* s)`

  Opens the corresponding I/O stream.

- `int Round(float x)`

  If `x` is less than zero, `Round` returns the smallest integral value not less than `x`. If `x` is greater than or equal to zero, `Round` returns the largest integral value not greater than `x`.

J.3 graph.cc

```
 1  #include <iostream.h>
 2  #include <fstream.h>
 3  #include <stdlib.h>
 4
 5  #define GWIDTH  40
 6  #define GHEIGHT 40
 7
 8  typedef struct pointnode {
 9     int X;
10     int Y;
11     struct pointnode *Next;
12  } PointNode;
13
14  typedef PointNode *PointNodePtr;
15
16  extern int Round(float x);
17
18  void InsertPoint(int XValue, int YValue, PointNodePtr PointList);
19  void PrintPointList(PointNodePtr PointList);
20  void PlotGraph(PointNodePtr PointList, float XShift, float YShift,
21                 float XScale, float YScale, char Output[GWIDTH][GHEIGHT]);
22  void DrawGraph(PointNodePtr PointList);
23
24  int main(int argc, char **argv)
25  {
26      PointNode *PointList = NULL;
27      ifstream InputFile;
28      char *Filename = NULL;
29      int X = 0, Y = 0;
30
31      if (argc != 2) {
32        cerr << "Usage: " << argv[0] << " file" << endl;
33        return(-1);
34      }
35      Filename = argv[1];
36      InputFile.open(Filename);
37      if (InputFile.good()) {
38        cerr << "File error on " << Filename << endl;
39        InputFile.close();
40        return(-1);
41      }
42      while (InputFile >> X)
43        if (InputFile >> Y)
44           InsertPoint(X, Y, PointList);
45        else {
46           cerr << "Error: no Y value for X = " << X << endl;
47           InputFile.close();
48           return(-1);
49        }
50      InputFile.close();
51      DrawGraph(PointList);
```

```
52      return(0);
53  }
54
55  void InsertPoint(int XValue, int YValue, PointNodePtr PointList)
56  {
57   PointNodePtr NewNode = NULL;
58
59   if (NewNode = new PointNode) {
60      NewNode->X = XValue;
61      NewNode->Y = YValue;
62      NewNode->Next = PointList;
63      PointList = NewNode;
64   }
65   else
66      cerr << "Error allocating memory." << endl;
67  }
68
69  void PrintPointList(PointNodePtr PointList)
70  {
71   PointNodePtr Current = PointList;
72
73   while(Current != NULL) {
74      cout << "(" << Current->Y << ", " << Current->X << ")" << endl;
75      Current = Current->Next;
76   }
77   cout << endl;
78  }
79
80  void PlotGraph(PointNodePtr PointList, float XShift, float YShift,
81                 float XScale, float YScale, char Output[GWIDTH][GHEIGHT])
82  {
83   int x = 0, y = 0;
84   PointNodePtr Current = PointList;
85
86   for (y = 0; y < GHEIGHT; y++)
87      Output[Round(XShift)][y] = '|';
88   Output[Round(XShift)][GHEIGHT - 1] = '^';
89   Output[Round(XShift) + 1][GHEIGHT - 1] = 'Y';
90
91   for (x = 0; x < GWIDTH; x++)
92      Output[x][Round(YShift)] = '-';
93   Output[GWIDTH - 1][Round(YShift)] = '>';
94   Output[GWIDTH - 1][Round(YShift) - 1] = 'X';
95   Output[Round(XShift)][Round(YShift)] = '+';
96
97   while (Current != NULL) {
98      x = Round((float) Current->X * XScale + XShift);
99      y = Round((float) Current->Y * YScale + YShift);
100     switch(Output[x][y]) {
101        case '-' : case '|' :
102        case '+' : case ' ' : Output[x][y] = '*'; break;
103        case '*' : Output[x][y] = '2'; break;
```

```
104         default : Output[x][y] = Output[x][y] + 1; break;
105      }
106      Current = Current->Next;
107  }
108 }
109
110 void DrawGraph(PointNodePtr PointList)
111 {
112  int   SmallestX = 0, LargestX = PointList->X,
113        SmallestY = 0, LargestY = PointList->Y,
114        Width = 0, Height = 0,
115        x = 0, y = 0;
116  float XScale = 1.0, YScale = 1.0,
117        XShift = 0.0, YShift = 0.0;
118  PointNodePtr Current = PointList;
119  char Output[GWIDTH][GHEIGHT];
120
121  while (Current != NULL) {
122     if (Current->X < SmallestX) SmallestX = Current->X;
123     if (Current->X > LargestX) LargestX = Current->X;
124     if (Current->Y < SmallestY) SmallestY = Current->Y;
125     if (Current->Y > LargestY) LargestY = Current->Y;
126     Current = Current->Next;
127  }
128
129  Width = LargestX - SmallestX + 1;
130  Height = LargestY - SmallestY + 1;
131  if (Width > GWIDTH) XScale = (float) Width / (float) GWIDTH;
132  if (Height > GHEIGHT) YScale = (float)  Height/ (float) GHEIGHT;
133  if (SmallestX < 0) XShift = (float) abs(SmallestX) * XScale;
134  if (SmallestY < 0) YShift = (float) abs(SmallestY) * YScale;
135
136  PlotGraph(PointList, XShift, YShift, XScale, YScale, Output);
137
138  for (y = 0; y < GHEIGHT; y++) {
139     for (x = 0; x < GWIDTH; x++)
140        cout << Output[x][y];
141     cout << endl;
142  }
143 }
```

Group no.	Subjects
1	1, 2, 3
2	4, 5, 6
3	7, 8, 9
4	10, 11, 12
5	13, 14, 15
6	16, 17, 18
7	19, 20, 21, 22
8	23, 24, 25
9	26, 27, 28
10	29, 30, 31
11	32, 33, 34
12	35, 36, 37
13	38, 39, 40
14	41, 42, 43

Table 9: Allocation of subjects to groups.

K Raw Data

This appendix presents the raw data collected from the experiment. Table 9 shows the allocation of subjects to groups. Table 10 summarises the individual phases of the experiment, showing which program each subject performed each technique on. It also summarises the number of defects found and the number of false positives found by each subject. Table 11 shows the same data for the group phase of each inspection.

Table 12 presents the raw data for the individual phase of the `analyse.cc` inspection. Each row contains the data for one subject, with the defect numbers appearing in the column headers. An "X" indicates that the given defect was found by that subject. Similar tables are given for the group phase of the `analyse.cc` inspection (Table 14), the individual phase of the `graph.cc` inspection (Table 13), and the group phase of the `graph.cc` inspection (Table 15).

Subject No.	E1 Method	E1 Result	E2 Method	E2 Result
1	T	10/11	P	9/10
2	T	7/10	P	5/8
3	T	9/10	P	7/8
4	T	9/12	P	8/12
5	T	9/14	P	7/11
6	T	6/10	P	7/9
7	T	9/14	P	10/13
8	T	9/13	P	6/9
9	T	5/9	P	6/13
10	T	7/10	P	4/9
11	T	7/7	P	6/8
12	T	6/13	P	7/8
13	T	7/13	P	6/7
14	T	9/9	P	6/9
15	T	9/11	P	8/8
16	T	6/10	P	6/11
17	T	6/10	P	4/11
18	T	10/14	P	8/12
19	T	8/11	P	6/10
20	T	8/15	P	7/12
21	T	8/13	P	6/8
22	T	5/9	P	7/11
23	P	10/14	T	8/12
24	P	4/11	T	5/8
25	P	8/13	T	5/12
26	P	7/13	T	7/7
27	P	8/10	T	7/9
28	P	7/9	T	4/10
29	P	10/14	T	10/10
30	P	7/14	T	3/9
31	P	8/14	T	5/9
32	P	8/11	T	5/11
33	P	7/12	T	4/9
34	P	7/12	T	9/10
35	P	11/17	T	7/9
36	P	6/8	T	6/7
37	P	7/11	T	4/9
38	P	7/12	T	7/11
39	P	12/14	T	9/10
40	P	6/11	T	4/8
41	P	9/15	T	6/7
42	P	5/5	T	3/7
43	P	9/11	T	8/10

Table 10: Results for individuals. E1 refers to the first phase of the experiment, inspecting `analyse.cc`. E2 refers to the second phase, where `graph.cc` was inspected. Method is either (P)aper or (T)ool. The results are given as the total number of correct defects out of the total number submitted.

Group No.	E1 Method	E1 Result	E2 Method	E2 Result
1	T	11/12	P	10/10
2	T	12/14	P	11/15
3	T	10/14	P	10/12
4	T	11/15	P	9/10
5	T	10/13	P	11/14
6	T	11/16	P	8/12
7	T	11/17	P	8/10
8	P	11/15	T	10/12
9	P	10/12	T	7/8
10	P	11/15	T	9/11
11	P	11/16	T	8/15
12	P	11/13	T	9/12
13	P	12/16	T	10/12
14	P	9/10	T	9/9

Table 11: Results for groups. E1 refers to the first phase of the experiment, inspecting `analyse.cc`. E2 refers to the second phase, where `graph.cc` was inspected. Method is either (P)aper or (T)ool. The results are given as the total number of correct defects out of the total number submitted.

	Defect Number											
Subject	1	2	3	4	5	6	7	8	9	10	11	12
1	X	X	X	X	X	X	X	X	X		X	X
2		X		X		X	X		X		X	X
3	X	X		X	X	X	X	X	X	X	X	X
4	X	X	X		X	X	X		X		X	X
5		X				X	X		X	X		X
6		X				X	X		X	X		X
7		X		X	X	X	X		X	X	X	X
8	X	X		X		X	X		X	X	X	X
9		X				X	X		X	X		
10		X			X	X	X		X	X	X	
11	X	X		X		X		X			X	X
12		X		X		X	X		X		X	
13		X		X		X	X			X	X	X
14	X	X		X		X	X		X	X	X	X
15	X	X		X			X	X	X	X	X	X
16	X	X				X	X				X	X
17	X	X				X	X		X		X	
18	X	X	X	X	X	X	X		X	X	X	
19	X	X		X	X	X	X		X			X
20	X	X		X		X	X	X	X		X	
21	X	X		X		X	X		X	X	X	
22		X		X		X	X			X		
23	X	X			X	X	X	X	X	X	X	X
24		X				X			X	X		
25	X	X	X			X	X		X		X	X
26	X	X				X	X		X		X	X
27	X	X		X	X	X	X				X	X
28	X	X		X		X		X	X		X	
29	X	X	X	X	X	X		X	X	X	X	
30	X	X				X	X		X	X	X	
31	X	X		X	X	X	X		X		X	
32	X	X		X		X	X	X			X	X
33	X	X		X		X	X		X	X		
34	X	X		X		X	X	X	X			
35	X	X		X	X	X	X	X	X	X	X	X
36	X	X				X	X			X	X	
37		X				X	X		X	X	X	X
38	X	X		X		X	X		X			X
39	X	X	X	X	X	X	X	X	X	X	X	X
40		X		X				X	X		X	X
41	X	X		X		X	X		X	X	X	X
42		X				X	X		X		X	
43	X	X			X	X	X		X	X	X	X

Table 12: Raw data for individual phase of `analyse.cc` inspection.

	Defect Number											
Subject	1	2	3	4	5	6	7	8	9	10	11	12
1	X	X	X		X	X	X			X	X	X
2		X	X		X	X	X					
3	X	X	X			X	X		X		X	
4	X	X		X	X	X	X		X		X	
5	X	X	X		X	X	X				X	
6	X	X	X		X		X			X	X	
7	X	X	X		X		X	X	X	X	X	X
8	X	X	X		X		X				X	
9	X	X	X	X	X						X	
10	X	X	X		X							
11	X	X	X		X	X	X					
12	X	X	X	X	X			X				X
13	X	X	X		X				X	X		
14	X	X	X				X			X	X	
15	X		X	X	X	X	X	X				X
16	X	X	X		X	X					X	
17		X	X		X					X		
18	X	X			X	X	X		X	X	X	
19	X	X	X		X				X		X	
20	X	X	X		X	X	X				X	
21	X	X	X		X		X				X	
22	X	X	X		X		X		X		X	
23	X	X		X	X		X			X	X	X
24	X	X			X				X		X	
25	X	X			X		X				X	
26	X	X	X		X	X	X				X	
27	X	X	X	X	X		X				X	
28	X	X					X				X	
29	X	X	X		X	X	X		X	X	X	X
30	X	X	X									
31	X	X			X		X				X	
32	X	X	X		X						X	
33		X	X			X		X				
34	X	X	X		X	X	X		X	X	X	
35	X	X	X	X	X		X				X	
36	X	X	X		X					X	X	
37	X	X	X								X	
38	X	X	X		X	X	X				X	
39	X	X		X	X	X	X			X	X	X
40	X					X	X				X	
41	X	X	X		X					X	X	
42	X				X						X	
43	X	X	X		X		X		X	X	X	

Table 13: Raw data for individual phase of `graph.cc` inspection.

	Defect Number											
Group	1	2	3	4	5	6	7	8	9	10	11	12
1	X	X		X	X	X	X	X	X	X	X	X
2	X	X	X	X	X	X	X	X	X	X	X	X
3	X	X		X	X	X	X		X	X	X	X
4	X	X		X	X	X	X	X	X	X	X	X
5	X	X		X		X	X	X	X	X	X	X
6	X	X		X	X	X	X	X	X	X	X	X
7	X	X		X	X	X	X	X	X	X	X	X
8	X	X	X		X	X	X	X	X	X	X	X
9	X	X		X	X	X	X	X	X		X	X
10	X	X	X	X	X	X	X	X	X	X	X	
11	X	X		X	X	X	X	X	X	X	X	X
12	X	X		X	X	X	X	X	X	X	X	X
13	X	X	X	X	X	X	X	X	X	X	X	X
14	X	X		X		X	X		X	X	X	X

Table 14: Raw data for group meeting with `analyse.cc`.

	Defect Number											
Group	1	2	3	4	5	6	7	8	9	10	11	12
1	X	X	X		X	X	X		X	X	X	X
2	X	X	X	X	X	X	X	X	X		X	X
3	X	X	X		X		X	X	X	X	X	X
4	X	X	X		X	X	X	X			X	X
5	X	X	X	X	X	X	X	X	X	X		X
6	X	X	X		X	X	X		X		X	
7	X	X	X		X	X	X		X		X	
8	X	X	X	X	X		X		X	X	X	X
9	X	X	X		X		X				X	X
10	X	X	X		X	X	X		X		X	X
11	X	X	X		X	X	X		X		X	
12	X	X	X	X	X	X	X			X	X	
13	X	X	X	X	X	X	X			X	X	X
14	X	X	X		X	X	X		X	X	X	

Table 15: Raw data for group meeting with `graph.cc`.