University of Southern Queensland

Teaching Programming Strategies Explicitly to Novice Programmers

A dissertation submitted by

Michael de Raadt

for the award of

Doctor of Philosophy

2008

Abstract

The traditional approach to training novice programmers has been to provide explicit programming *knowledge* instruction but to rely on implicit instruction of programming *strategies*. Studies, reported in literature, have discovered universally poor results on standardised tests for novices studying under this traditional approach.

This dissertation describes the explicit integration of programming *strategies* into instruction and assessment of novice programmers, and the impact of this change on their learning outcomes.

An initial experiment was used to measure the performance of students studying under a traditional curriculum with implicitly taught programming *strategies*. This experiment uncovered common flaws in the *strategy* skills of novices and revealed weaknesses in the curriculum. Incorporation of explicit *strategy* instruction was proposed.

To validate a model of *strategies* as being authentic and appropriate for novice instruction, an experiment with experts was conducted. Experts were asked to solve three problems that a novice would typically be expected to solve at the end of an introductory programming course. Experts' solutions were analysed using Goal/Plan Analysis and it was discovered that experts consistently applied *plans*, the sub-algorithmic *strategies* suggested by Soloway (1986). It was proposed that *plans* could be adapted for explicit inclusion in an introductory programming curriculum.

Initially a curriculum incorporating explicit *strategy* instruction was tested in an artificial setting with a small number of volunteers, divided into control and experimental groups. The control group was taught using a simplified traditional curriculum and the experimental group were exposed to a curriculum which explicitly included programming *strategies*. Testing revealed that experimental group participants applied *plans* more than control group participants, who had been expected to learn these *strategies* implicitly. In interviews, experimental participants used strategy-related terminology and were more confident in the solutions they had created. These results justified a trial of the curriculum in an actual introductory programming course.

When explicit instruction of programming *strategies* was incorporated into an actual introductory programming curriculum, novices achieved superior results when compared to results from the initial experiment. Novices used *strategies* significantly more when these *strategies* were incorporated explicitly into instructional materials and assessment items.

This series of experiments focussed on explicitly teaching specific programming *strategies* rather than teaching problem-solving more generally. These experimental results demonstrate that explicit incorporation of programming *strategies* may improve outcomes for novices and potentially improve the potential of expert programmers in future.

Certification of Dissertation

I certify that the ideas, experimental work, results, analyses and conclusions reported in this dissertation are entirely my own effort, except where otherwise acknowledged. I also certify that the work is original and has not been previously submitted for any other award, except where otherwise acknowledged.

__________________________________ _______________

Signature of Candidate *Date*

ENDORSEMENT

__________________________________ _______________

Signature of Supervisor *Date*

__________________________________ _______________

Signature of Supervisor *Date*

Relevant Publications

DE RAADT, M., WATSON, R. & TOLEMAN, M. (2002) Language Trends in Introductory Programming Courses. In Proceedings of Informing Science and IT Education Conference. p. 329 - 337.

DE RAADT, M., WATSON, R. & TOLEMAN, M. (2003) Language Tug-Of-War: Industry Demand and Academic Choice. *Australian Computer Science Communications,* 25**,** 137 - 142.

DE RAADT, M., WATSON, R. & TOLEMAN, M. (2003) Introductory programming languages at Australian universities at the beginning of the twenty first century. *Journal of Research and Practice in Information Technology,* 35**,** 163-167.

DE RAADT, M., WATSON, R. & TOLEMAN, M. (2004) Introductory Programming: What's happening today and will there be any students to teach tomorrow? *Australian Computer Science Communications,* 26**,** 277 - 284.

DE RAADT, M., TOLEMAN, M. & WATSON, R. (2004) Training strategic problem solvers. *ACM SIGCSE Bulletin,* 36**,** 48 - 51.

DE RAADT, M., TOLEMAN, M. & WATSON, R. (2006) Chick Sexing and Novice Programmers: Explicit Instruction of Problem Solving Strategies. *Australian Computer Science Communications,* 28**,** 55 - 62.

DE RAADT, M. (2007) A Review of Australasian Investigations into Problem-Solving and the Novice Programmer. *Computer Science Education,* 17**,** 201 - 213.

DE RAADT, M., TOLEMAN, M. & WATSON, R. (2007) Incorporating Programming Strategies Explicitly into Curricula. In Proceedings of the Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007). p. 53 - 64.

Acknowledgments

I would like to thank my supervisors, Mark Toleman and Richard Watson, for the experience they shared, their enthusiasm in my work and, perhaps above all, their patience.

I thank my family, my wife Erica and my two children, Hannah and Isaac, for allowing me to spend time away to complete my research work and for encouraging me to keep going and get the job done.

I thank the CSEd community for taking my ideas seriously and showing me the value of research in this area.

Thanks to Simon for his proofing services. Simon is a published author and leading figure of the CSEd community.

Thanks also to Peter Dunn for his assistance on statistics matters.

I thank my Lord Jesus, to whom I pray, for giving me the strength to achieve my absolute potential and benefit those in the world around me.

Table of Contents

Glossary

The following terms are used throughout this dissertation. They are presented here to clarify their meaning. The terms are described and explored more deeply in Chapter 2 with references to sources. The terms are presented in alphabetic order.

- **Comprehension**
  In the context of programming, *comprehension* is the ability to read and understand code or program related information. This may involve simulating execution of a program to manually compute a result.
- **Course**
  A single period of instruction, usually contained within a semester. Equivalent to the terms unit, paper or subject used in some institutions.
- **Curriculum**
  Teaching materials, assessment items and method of delivery, related to a single *course* (see above).
- **Expert**
  A programmer who writes programs on a regular or daily basis. For the purposes of this dissertation it is not important when a *novice* programmer (see below) becomes an expert is not critical, as comparisons are made only between programmers at the extremes of this scale.
- **Explicit Instruction**
  Instruction where the the instructor openly describes, usually in some documented form, what the student is to learn and how to go about that learning.
- **Generation**
  In the context or programming, *generation* involves the creation of code or program related information, potentially implementing a solution to a problem.
- **Goal**
  An identified part of a solution that needs to be achieved for a programming problem.
- **Implicit Instruction**
  Instruction within a scenario where a student is expected to undertake learning without being given a full context for what it is they are to learn or how.
- **Knowledge**
  See Programming Knowledge below.
- **Novice**
  A programmer learning how to program for the first time. In this dissertation *novices* are students undertaking a first *course* (see above) in programming.
- **Plan**
  A fundamental form of *programming strategy* (see below). The means of

achieving a *goal* (see above) within a solution to a programming problem. Plans are normally contained in the tacit knowledge of experts.

- **Problem Solving**
  A mechanism for achieving a solution to a programming problem. Within the scope of this document, *problem solving* is not intended to be interpreted more generally.
- **Programming Knowledge**
  A programmer's understanding and potential to apply the syntax and semantics of a programming language and any related language features.
- **Programming Strategy**
  A general term for a range of programming problem solving approaches including *plans* (see above), patterns, algorithms and other methodologies. A *programming strategy* is an abstracted method for applying programming knowledge to solve a problem.
- **Strategy**
  See Programming Strategy above.

1. Introduction

As an information society we rely on computers and software used on computers. The potential to create new software governs our progress as an information society. The responsibility for creating software falls on information technology professionals and key among these are programmers.

Like computers, the art of programming is relatively new, yet it is fundamentally entrenched in our information-dependent world. Expert programmers have the potential to solve many information-related problems. Programmers have the potential to overcome new problems and advance the world of computing a little further with each new solution.

Guiding novices to gain expertise builds a stronger field of expert programmers and strengthens our potential as an information society into the future. Constructing more complex and higher quality software relies on instructors training expert programmers who are more capable and more confident.

1.1 Motivation

Programming is a cognitively demanding task and training novices can be a challenging undertaking. Novices must learn the programming *knowledge* (syntax and language features) and programming *strategies* (ways to apply this *knowledge* in order to solve programming problems).

For many years introductory programming instructors have anecdotally reported failure rates that are higher than most institutions generally tolerate (Lister, 2000, Carbone et al., 2000, Guzdial and Soloway, 2002). Much research in computing education has focused on ways to overcome high failure rates in introductory programming.

Seeking concrete evidence, a number of studies have attempted to quantify the level of skill of novice programmers at the end of an introductory programming course. A multinational study of novice programmers (McCracken et al., 2001) showed universally poor results on a standardised test conducted at institutions across the world. This study did not identify sources of the inadequacies demonstrated, but it did offer an opportunity to accept the failings of the past and a chance to begin to develop new curricula which could better encourage novices to reach expected standards.

Observed novices have produced poor results in standardised program *generation* tests, with many novices demonstrating a fragile *knowledge* (Lister et al., 2004) and most novices failing to demonstrate programming *strategies* (Lister et al., 2006). There are at least two possible causes for this behaviour.

One possibility is that there are some bright students, but most students simply do not possess the mental capacity to meet the standard set for them.

Another possibility is the curricula used in traditional introductory programming courses, and the methods used to deliver them, fail to adequately teach most students programming *knowledge* and *strategies*.

It is likely that both of these causes are contributing in some way; however blaming novices for their own failure will not improve outcomes so we must consider ways of improving curricula to address these failings.

Kuittinen and Sajaniemi (2003) contend that many attempts to ease novice diffficulties have simply sought to teach a traditional curriculum in more exciting ways. Increased student enthusiasm resulting from such innovations does not seem to have improved the outcomes of struggling students around the world. Rather than making ad-hoc changes to parts of an aging curriculum in the hope of finding some magic formula, a better objective is to systematically discover the conceptual barriers that cause novices to struggle and to consider new curricula elements that encourage novices to overcome these barriers.

Ultimately, the underlying motivation for attempting to improve introductory programming curricula is to support the development of more competent novices who will hopefully go on to become better expert programmers, creating superior software and benefiting the greater community.

1.2 Aims

Traditional curricula include explicit instruction of programming *knowledge*. Novices are taught the constructs and associated facilities of a language in a well expressed manner. By contrast, programming *strategies*, the means of applying programming *knowledge* to solve problems, are taught implicitly (Soloway, 1986). It is expected that novices will construct their own programming *strategies* by obtaining programming *knowledge*, solving problems, then reflecting on this process. It has been shown that some novices can successfully develop programming *strategies* over a period of implicit learning (Rist, 1991). However, outcomes for the majority of novices remain poor (McCracken et al., 2001, Lister et al., 2004, Lister et al., 2006).

Explicit instruction has been shown to be far more powerful than implicit instruction. When compared with implicit instruction, explicit instruction has been proven to produce faster learning, higher accuracy and an understanding of the underlying principles of the concepts being learnt in specific disciplines (Baddeley, 1997). Expressing and instructing programming *strategies* in a more explicit fashion may improve outcomes for novices. Before this can be achieved, an set of programming *strategies*, used by experts but relevant to novices, needs to be expressed.

It has been suggested that experts hold programming *strategies* in a tacit form in their minds (Soloway, 1986). Experts build this tacit collection of *strategies* over time and apply and adapt it for new problems as they arise. If a concrete set of *strategies* is proposed for teaching, it needs to be validated against the *strategies* applied by experts.

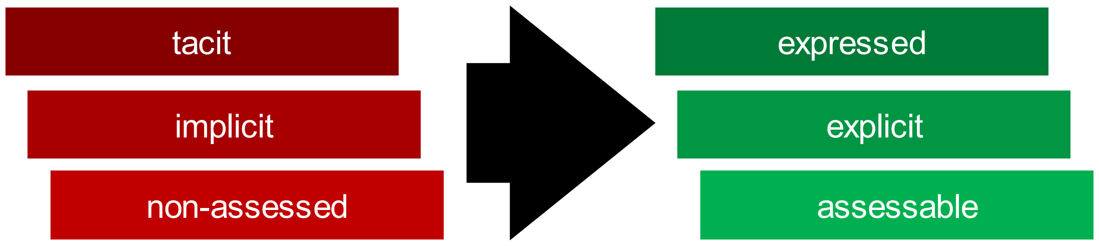


Figure 1.1. Including programming *strategies* in curricula

The aim of work described in this dissertation is to improve the curricula and pedagogy used for training novice programmers by adding curricular elements designed to overcome difficulties faced by novices. The transition from an implicit traditional approach to an explicit approach, pictured in Figure 1.1, aims to:

1. use the tacit programming *strategies* of experts to validate an expressed set of *strategies*, suitable for novice programmers;
2. move from the implicit delivery of *strategies* to curricula and methods of delivery that explicitly teach authentic programming *strategies* to novices, and test the impact of this move; and
3. target the application of specific *strategies* in assessment of novices' skills, thus promoting the value of *strategies*.

To accomplish these aims, the following tasks were proposed and undertaken.

- Measure the programming *strategy* skill level of novices trained using a traditional curriculum. This can then be used as a benchmark for later comparison with new curricula.
- Identify authentic expert programming *strategies* that are relevant to novice programmers.
- Create a curriculum that explicitly integrates programming *strategy* instruction with programming *knowledge* instruction.
- Determine the impact of explicitly teaching programming *strategies*, comparing this to the skill level of novices trained under a traditional curriculum.
- Demonstrate how programming *strategies* can be assessed as part of regular assignments and examinations.

1.3 Dissertation Structure

This dissertation is divided into three parts. The introductory chapters set the stage for the experimental work described in subsequent core chapters, and findings are brought together in the concluding chapter.

Introduction, Overview and Methodology

The body of this dissertation begins in chapter 2 by exploring research related to introductory programming instruction. A history of introductory programming teaching is given. A number of terms, focusing on aspects relevant to teaching in this area, are defined. Areas where there is a need for further exploration are identified and the potential for the contribution made by this study is explained.

Chapter 3 then outlines the scope of experiments undertaken in this study. Research questions are discussed and Goal/Plan Analysis, the main methodological tool for measuring novice programming *strategy* skill, is described in detail.

Description of Experiments

The core of the work described in this dissertation was achieved through four experiments. The experiments were conducted in sequence, with the findings of each study leading to a need for the following experiment.

Chapter 4 describes an experiment where Goal/Plan Analysis was applied to the code of novices who had studied programming with implicit-only programming *strategy* instruction. Results of this study justify the aim to explicitly express programming *strategies* in the curriculum used. These results also served as a basis for the comparison of results of later experiments.

Chapter 5 presents an experiment which attempted to identify *strategies* used by expert programmers. The experiment targeted *strategies* that are relevant to novice programmers and could be explicitly incorporated into introductory programming curricula.

Chapter 6 describes an experiment that compared two introductory programming curricula: one that explicitly included programming *strategy* instruction and another that followed a traditional implicit *strategy* instruction approach. The experiment explored the feasibility of using a curriculum including programming *strategy* instruction, and attempted to measure and compare the impact of the two curricula on novices.

Chapter 7 shows how programming *strategies* were included in an actual university level introductory programming course. This chapter describes how *strategies* were incorporated in instructional material and how they were assessed for grading purposes. Results of this approach are compared to the benchmark set in the initial experiment showing an increase in the use of *strategies*.

Conclusions and Statement of Contribution

Conclusions are made in chapter 8. Implications of these conclusions for the field of computing education are discussed and the original contribution of this dissertation is identified. Finally, future work is suggested.

2. Problem Solving and Novice Programmers

In the previous chapter the rationale and aims of the study described in this dissertation were presented. This chapter establishes a context for the experiments described in later chapters. The first sections review areas of computing education relevant to this investigation. Then a number of important terms are defined; these terms will be used throughout the dissertation. Programming *strategies* can be expressed in a number of forms; two of these are compared, and the form used in experimentation is identified. Having established a context, the need for investigation in this area is expressed and justified.

2.1 A Brief History of Programming Instruction

> *"If you ask me what accomplishment I'm most proud of, the answer would be all the young people I've trained over the years; that's more important than writing the first compiler."*
> *Grace Hopper*

Computer programming became a practised discipline as computing technologies began to develop. Initially computer programmers were self-taught as there were no educational programs covering programming in the early days of computing. *Computing science*, which includes computer programming, surfaced as an offspring of other academic disciplines such as mathematics. "Computing education emerged from a few optional units in mathematics or engineering to establish its own discipline as Computer Science (CS) in the 60s" (Pham, 1996). Early computing curricula focused primarily on creating programs for data processing as this was the origin of computing, but over time the discipline grew and divided into substantial sub-disciplines (Pham, 1996). As computing technologies have become relevant to more than a handful of specialists, the discipline, and education within the discipline, has been forced to change and adapt.

In 1971 Niklaus Wirth introduced the language Pascal, primarily for teaching novice programmers (Wirth, 1971, Wirth, 1974). Pascal was simple and well structured, and allowed instructors to focus on fundamental concepts involved in the task of programming. Although other academic languages had been presented and used, none have had the impact of Pascal, which during a period of 23 years was taught at some stage in almost all Australian universities (de Raadt et al., 2002).

During the 1980s and into the 1990s the advent of personal computers brought computing within reach of non-computing professionals and this new group of computer users required training. Initially, the best way to provide end-user computing skills was thought to be training in programming. A notion developed that all computer users could benefit from some amount of programming literacy. Because of this, programming was referred to as "the New Latin" (Soloway, 1993). Through the act of learning programming it was believed that students would develop general problem-solving and design skills that could be applied to the remaining facets of their lives. Novices being trained as the expert programmers of the future were joined by students who would be exposed to not more than a semester or two of programming instruction. A new approach was considered necessary to teach both groups. As many tertiary institutions of the era did not have the resources to teach both groups independently, compromises were made which generally involved teaching as many of the features of the Pascal programming language as was practical within an introductory course and hoping students would learn problem solving along the way.

Programming is no longer ‘the New Latin’. While computing skills are even more valuable today general computer users are more commonly directed to learn application skills rather than programming. Current courses for non-programmers attempt to instil capabilities in applications such as word processors, spreadsheets and presentation graphics packages. The teaching of programming is now largely directed once more to students who will later take on computing study and strive to become expert programmers or at least require some programming skills as part of their professional training.

Since 2003, Pascal is no longer taught in any Australian university – choosing a language that will attract students is considered, by most instructors, as a higher priority than the pedagogical benefits a language can give (de Raadt et al., 2002). Languages used in introductory programming courses are most commonly industry-relevant languages. A large number of papers have suggested that one language is superior to another because it possesses desirable features (eg. Bergin, 2000, Biddle and Tempero, 1998, Chandra and Chandra, 2005, Hadjerrouit, 1998, Stroustrup, 1999) or because changing to the new language seemed to encourage better results from students (eg. Hitz and Hudec, 1995, Andreae et al., 1998). What is shown in literature is likely to be a reflection of the debates that have undoubtedly taken place within the meeting rooms and corridors of teaching institutions.

In the late 1980s and early 1990s programmers began to explore Object-Oriented Programming (OOP), a new programming paradigm that combines data with behaviour related to that data. Many instructors felt that this paradigm was more analogous to natural human understanding and therefore could benefit novices. Many papers during the 1990s debated just that (eg. Reid, 1993, Decker and Hirshfield, 1994, Kölling et al., 1995). A new language, Java, introduced in 1994, embodied the object-oriented paradigm. Java was immediately successful because it also provided strong integration with the World Wide Web, which was rapidly growing at the same time. Java became popular and students wanted to learn it. Universities within Australia were facing competition and strict government funding regulations (Pham, 1996). Attracting students by teaching them what they wanted to learn contributed largely to the curricula of the late 1990s.

2.1.1 Failure Rates

Many anecdotal reports state failure rates in introductory programming courses are higher than acceptable, gaining attention within tertiary institutions. Seeking empirical evidence the ‘McCracken group’, involving multinational participants from the UK, USA, Israel, Poland and Australia, came together as an ITiCSE 2001 working group (McCracken et al., 2001). The group studied competency of novices after a one- or two-semester introductory course in programming. The group established a set of standard test questions with set evaluation criteria which were applied at participating institutions. The average score was 21% leading to a conclusion that “many students do not know how to program at the conclusion of their introductory courses” (p. 125). The study did not identify sources of the inadequacies demonstrated by novices or potential fixes for these problems.

However, the McCracken group did offer an understanding that these problems are universal and there is now an opportunity to develop new curricula which could produce better outcomes in novices.

2.2 Recent teaching in Australian and New Zealand

As preliminary work for this study an investigation was made into several aspects of introductory programming courses within universities in Australia and New Zealand. The work was referred to as the 'Census' as it attempted to capture information about all such courses in the region[1]. The Census was first conducted in 2001 and covered 57 introductory programming courses at 37 Australian universities (de Raadt et al., 2002, de Raadt et al., 2003b, de Raadt et al., 2003a). The second Census was conducted in 2003 and covered 85 courses from 39 Australian and eight New Zealand universities (de Raadt et al., 2004). Both instances of the Census covered languages and paradigms taught, tools used and numbers of students. A third census is being conducted in 2008.

2.2.1 Declining Student Numbers

A trend showing a reduction in the number of students undertaking introductory programming courses was observed between the 2001 Census and 2003 Census. The average decline in students between the two instances of the Census was 28% (de Raadt et al., 2004).

While having fewer students does not diminish the importance of improving instruction of programming, it does place pressure on instructors, many of whom are at risk of losing their jobs.

2.2.2 Industry-Relevant Languages

In both undertakings of the Census the most popular languages used in introductory programming courses were Java, C/C++, Visual Basic and Haskel. The first three of these languages could be classified as industry-relevant languages as they were widely used in industry at the time (de Raadt et al., 2003b) and not primarily designed with teaching in mind. The three non-commercial languages found by the 2001 Census (Haskel, Eiffel and Ada) were taught predominantly in *sandstone* universities (Australian universities established before 1950 (Ashenden and Milligan, 1999)).

Instructors participating in the 2001 Census were asked why they chose their current language. The responses (summarised in Table 2.1) showed that instructors chose a language either because they perceived it to have industry relevance, or because they believed it would be attractive to students who perceived that it was industry-relevant. This reason appears to have been more important than choosing a language for its pedagogical benefits. At sandstone universities, which can attract students more easily through traditional reputations, pedagogical benefits of a language where mentioned more often as a reason than industry-relevance.

[1] Australasia is an excellent location for such research as social and educational standards match those in other developed countries, yet the scale and number of institutions in this region makes it possible to contact almost all introductory programming instructors.

Table 2.1.Count of reasons given for language choice in all universities (reproduced from de Raadt et al., 2002)

Reason	Count
Industry-relevance/Marketable/Student demand	33
Pedagogical benefits of language	19
Structure of degree/Department politics	16
OOP language wanted	15
GUI interface	6
Availability/Cost to students	5
Easy to find appropriate texts	2

2.2.3 Paradigm

The list of languages taught in introductory programming courses, as discovered by the Census, was dominated by object-oriented languages, with over 80% of instructors choosing an OO language. However, many instructors teaching OO languages did not use an objects-first approach in their teaching, instead remaining with a procedural approach.

Table 2.2. Paradigm used in teaching (reproduced from de Raadt et al., 2004)

Paradigm	Australia		New Zealand	
	By Lang.	Taught	By Lang.	Taught
Procedural	11.7%	53.0%	8.3%	34.0%
Object-oriented	82.2%	36.6%	91.7%	66.0%
Functional	6.1%	10.3%	0%	0%

2.2.4 Approaches to Problem Solving

In the 2003 Census, participants were asked to estimate what percentage of time (in lectures and tutorials) was spent on the teaching of “problem-solving strategies”. Estimates of the proportion of lecture time devoted to the instruction varied greatly between participating instructors. Some participants responded that problem-solving strategies were not part of their course, with several indicating that problems used in their teaching were not of a large enough scale to warrant teaching problem-solving strategies. Other participants reported that their entire lecture time covered problem-solving strategies. Participants were asked what strategies they teach to their students. From the 108 responses given, there were 74 identifiable strategies differing in scale and complexity. As summarised in Table 2.3, the majority of participating instructors described a waterfall problem solving strategy, but even in this there was no consensus. The use of patterns in one course comes closest to the approach described in this dissertation.

Table 2.3. "Problem solving strategies" identified in 2003 Census.

Strategy Type	Different strategies identified	Instructors using
Waterfall problem solving strategies (analyse, design, implement)	39	66
Non-waterfall problem solving strategies (eg test-driven design)	6	6
Learning Strategies (eg working backwards, involving patterns)	6	6
Teaching strategies (eg, showing examples)	12	5
Unclassified	18	18
Total	75	108

This variation in time spent and forms of problem-solving instruction may be due to instructors not having a common definition of what is involved in the teaching of problem-solving strategies at the novice level.

2.2.5 Problem Solving in Textbooks

During the 2003 Census instructors reported using zero (some instructors prescribe no text), one, or more of 49 textbooks discovered by the Census. A copy of each text was requested from publishers and the content of the 40 texts that were delivered was analysed (de Raadt et al., 2005).

The most widely used text is *Simple Program Design* (Robertson, 2004). This text is not targeted at a specific language, but rather covers problem-solving aspects related to programming. This text is used together with a language-based text in most cases.

Problem-solving instruction was presented in varying degrees between the analysed texts (see Table 2.4). Some texts avoid problem solving as a specific topic altogether, focusing only on language syntax instruction. Some authors rely on large numbers of examples and provide little explicit instruction of problem solving. One author bluntly stated "Students learn to program by example" (Sparke, 2003, p. xi). Some texts offer a brief mention of algorithmic problem solving in an early chapter, but this teaching is not obviously integrated in the remainder of the text. Other texts offer instruction in high-level systems analysis or software engineering but little algorithmic problem solving; object-oriented software engineering is a common topic. There are a small number of texts that describe problem solving and attempt to integrate this teaching throughout the text using case studies and examinations of problems.

Table 2.4. Problem solving content in textbooks discovered by 2003 Census

Integration of Problem Solving	Number of texts	Proportion of texts
Problem solving integrated throughout	6	15%
Cursory or no content on problem solving	34	85%
Total texts examined	40	

2.3 Aspects of Novice Instruction

A review of literature related to introductory programming was conducted by Robins, Rountree, & Rountree (2003). This review investigated attempts to apply cognitive psychology research to programming instruction. Robins et al. propose the following aspects of a programmer's ability and use them to compare research in the area.

- expert-novice
- knowledge-strategy
- comprehension-generation

These three aspects will be used and expanded in the following subsections. They are clearly related, although definitive links between these aspects have not been proven. Within the knowledge-strategy aspect, levels of problems are described by the author of this dissertation to allow the scope of study to be clearly defined.

As well as these aspects, another important distinction is made between implicit and explicit instruction. Section 2.3.6 describes this distinction and discusses the value of explicit instruction. These aspects will be referred to in the remainder of the dissertation.

2.3.1 Experts and Novices

An *expert* programmer was defined by Winslow (1996) as a programmer with roughly 10 years experience. Winslow argued that "turning a novice into an expert is impossible in a four year program" and suggests the best product of a three or four year degree is "competence". Rist (1995) argued that programmers demonstrate themselves to be experts when they can produce the best designed solutions to particular problems. For the purposes of this dissertation the assumption is made that students in an introductory course are *novices*, most of whom will be learning programming for the first time. An expert is assumed to be someone who has experience in programming and practices programming on a regular or daily basis. In experiments described in this dissertation, the point at which a novice becomes an expert is not critical as comparisons are made between complete novices and experts.

In this dissertation the means of capturing expertise is through discovering experts' tacit knowledge and representing this in a form that can be understood by novices. Another method for passing expert knowledge to novices through a *cognitive apprenticeship* (Collins et al., 1987). In a cognitive apprenticeship, "learners can see the processes of work" (Collins et al., 1991, p. 1) as achieved by an expert. The idea of using the cognitive apprenticeship model in programming instruction has been proposed in a number of theoretical papers (Caspersen and Bennedsen, 2007, Shabo et al., 1996).

2.3.2 Knowledge and Strategies

Unlike the expert-novice aspect, which can be represented on a continuum, *knowledge* and *strategy* are disparate, but related entities. They are dependent on each other; however they need to be distinguished.

Knowledge involves the "declarative nature" (the syntax and semantics) of a programming language while *strategies* describe how programming *knowledge* is applied (Davies, 1993).

Programming *knowledge* relates to specific constructs and facilities of a given language. A novice usually acquires programming *knowledge* in a single language. Such *knowledge* may be transferable to another language if that language is syntactically similar, but new learning is required when a novice encounters a dissimilar language or paradigm.

Programming *strategies* relate to the application of programming *knowledge* to solve a problem. Ideas expressed in such *strategies* are more abstract than programming *knowledge* and are usually applicable to multiple languages within the same programming paradigm. Programming *strategies* can also be applied between similar paradigms; for instance *strategies* (such as looping strategies) learned in an imperative paradigm can often be applied in an object paradigm, but would not be as easy to apply in functional or event-driven paradigms.

The term *strategy* is a generic term exemplified by problem solving ideas such as *plans* (1986), *patterns* (Wallingford, 1996), algorithms and other methodologies, together with means of integrating these ideas to form a single solution.

Robins et al. (2003) define a distinction between novices who are *effective* or *ineffective*. Effective novices learn to program with little assistance, while ineffective novices fail to learn how to program, or do so only with a great deal of assistance. Robins et al. suggest that the key to novices becoming effective lies in them learning programming *strategies* rather than acquiring programming *knowledge*. Along a similar line, Soloway (1986) states:

> *...language constructs do not pose major stumbling blocks for novices... rather, the real problems novices have lie in "putting the pieces together," composing and coordinating components of a program. (p. 850)*

Soloway then proposes that teaching should reach beyond a focus on syntax (as programming *knowledge*) and focus on programming *strategies*.

Recent studies have attempted to quantify the ability of novices after an introductory programming course. The 'Leeds group' brought together researchers from the UK, USA, Denmark, Finland, Sweden, Australia and New Zealand as an ITiSCE 2004 working group (Lister et al., 2004). The group was attempting to isolate the cause of poor novice results measured by the McCracken group (McCracken et al., 2001) mentioned earlier (§ 2.1.1). The group used a set of multiple-choice questions that focused on program *comprehension* (reading and understanding code). The Leeds authors contended that no problem solving would be required to answer the questions, so if students failed this test, it would indicate a failure in programming *knowledge*. If novices succeeded in the test this would confirm that novices can successfully acquire programming *knowledge* so instructors could put this issue aside and focus their attention on how to improve *strategy* instruction.

Table 2.5. Performance in the Leeds study (reproduced from Lister et al., 2004)

Quartile Score	Range	No. of Students	Percent of Students
1st (top)	10 – 12	152	27%
2nd	8 – 9	135	24%
3rd	5 – 7	142	25%
4th (bottom)	0 – 4	127	23%

Novices who participated in the Leeds study did not perform as poorly as those who participated in the McCracken study, nor did they perform universally well. The distinction between the third and fourth quartiles in the Leeds group study (shown in Table 2.5) is between 4 and 5 correct answers out of the set of 12; a performance little better than guessing.

> *Suppose the students who participated in this study were all studying their first semester of programming at a single institution. Suppose further they were given these 12 MCQs as their exam, and the institution regarded a 25% failure rate as the upper limit of what was acceptable. Then students who scored 5 out of 12 on these MCQs would be progressing to the second semester programming course (Lister et al., 2004, p. 128).*

The Leeds group concluded that many novices possess only fragile programming *knowledge*. The study can be criticised due to a fault in the underlying assumption, made by the authors, that *comprehension* questions do not require problem-solving ability. The comprehension-generation and knowledge-strategy aspects are probably related but, to the authors knowledge, it has not been proven that these aspects are dependent. The Leeds group studied *novice program comprehension*, but made conclusions about *novice programming knowledge*. Regardless, there is undoubtedly some truth in the conjecture that the programming *knowledge* of many novices is flawed.

A following study, the ‘BRACElet project’ (Whalley et al., 2006), extended the Leeds study, using a set of questions created using Bloom's Revised Taxonomy (Anderson et al., 2001) to test programming skill over an identifiable cognitive range. The BRACElet study included questions that were categorised in the Bloom's levels of *apply*, *understand* and *analyse*, with specified sub-categories for each question. More correct answers were given for these better defined questions when compared to results from the Leeds group, but again many novices demonstrated gaps in their programming *knowledge*.

2.3.3 Levels of Problems

Problems that a programmer may face can be differentiated in their level of complexity. The following three classes of problem form a scale according to the complexity of problems. This taxonomy is the invention of the author.

System-Level Problems

Problems at the system level are large, complex and usually unique. Examples of problems at this level might be designing an accounting system for a large corporation, developing a web interface for a government department or developing a widely used end-user application such as an email client. Well established *strategies* have been formulated for designing and implementing solutions to problems of this scale, usually following a Waterfall software development process (analyse, design, implement, test, maintain) (Royce, 1970). New processes such as Extreme Programming (Beck, 2001) might also be useful at this level. Problem solving at this level is too complex for novices in their initial study of programming.

Algorithmic-Level Problems

Problems at the algorithmic level are identifiable parts of a greater problem. (In an academic setting they may be addressed independently.) For such problems, a solution is usually achieved by adopting established algorithms, widely used in the programming community. Solutions to problems of this scale may individually form functions. Generic forms of these functions may be included in standard libraries and perform tasks such as sorting, searching, or maintaining data structures. A novice may be able to start using such *strategies* at the end of an initial course in programming and may use them in greater depth in a second or third course in programming.

Sub-algorithmic-Level Problems

Problems at the sub-algorithmic level are at their most basic. Attempting to decompose and describe a problem below this scale will lead to syntactical definitions of specific language constructs. Solutions to sub-algorithmic problems form parts of more complex solutions. Coded solutions to individual problems at this scale will not usually form entire functions, but several sub-algorithmic solutions may be combined to reach this size. Examples of problems at this scale include guarding a division to avoid division by zero, achieving repetition until a sentinel is found, or swapping the values of a pair of variables. This level of problem is particularly relevant to novices in an initial exposure to programming. While problems are basic at this level, they are still regularly encountered by experts and are therefore relevant to programmers at all levels of expertise.

2.3.4 Comprehension and Generation

In the context of programming, *comprehension* is the ability to read and understand the outcomes of an existing piece of code; *generation* is the ability to create a piece of code that achieves certain outcomes. In studies of how novices learn the *roles of variables*, Kuittinen and Sajaniemi (2003) refer to simulation (tracing through code and predicting its output) as separate to *comprehension* (to "describe what is the purpose of the given program and how it works" (p. 6)). Although there is a subtle difference, for the purposes of this dissertation simulation will be considered as *comprehension*.

Whalley et al. (2006) contend that "a vital step toward being able to write programs is the capacity to read a piece of code and describe it" (p. 249) meaning that a novice must be able to comprehend a solution (and the *knowledge* and *strategies* within it) before they can generate a solution at the same level of difficulty. In other words, novices are more likely to take some time building *comprehension* of a problem solution and later attempting to generate a solution to the same or a similar problem; this may happen concurrently with several programming concepts. This assumption seems natural, perhaps due to the similarity between learning a programming language and learning a natural language where a child will generally learn to read words before they can write them.

Program *comprehension* can be thought of as less cognitively demanding than *generation*. According to the Bloom's Revised Taxonomy (Anderson et al., 2001) *comprehension* tasks can be classified at the lower *understand* and *apply* levels while *generation* involves the cognitively higher *create* level. Oliver, Dobele, Greber, & Roberts (2004) measured the cognitive difficulty of assessments of six courses in an undergraduate computing degree program. The courses included three programming courses studied in first year and early in second year and three networking courses studied in second year. For each course the cognitive difficulty of assessments was measured using Bloom's Taxonomy, with each course being given a single rating between 1.0 and 6.0 according to the average difficulty of its assessment tasks. While this study can be criticised for applying Bloom's Taxonomy as a linear scale, it did conclude that first-year programming courses include assessment tasks more cognitively demanding than those encountered in the later networking courses. This was because assessment tasks in the programming courses frequently reach higher cognitive levels including *apply*, *analyse* and *create*. So while both *comprehension* and *generation* of programs are important skills for a novice, traditional curriculum assessments focus strongly on *generation*, perhaps assuming that *comprehension* will be developed as a prerequisite skill.

2.3.5 Relationships between Aspects

The expert-novice, knowledge-strategy and comprehension-generation aspects are clearly related. According to Brooks (1983), experts and novices can be distinguished by how they undertake *comprehension*. Rist (1995) suggests novices and experts can be differentiated by how they undertake program *generation*. During program *generation* an expert can rely on a tacit body of programming *plans* developed through solving past problems (Soloway, 1986), while a novice has traditionally been expected to conceive and apply *plans*, with varying degrees of success (Rist, 1991). A distinction of expertise by use of *strategy* is proposed by Bailie (1991, p. 277): “one feature clearly distinguishing the novice from the expert programmer is the ability to plan.”

An instructor might present an example problem, say a loop that repeats a fixed number of times. The instructor may then display and describe a coded solution to the problem. A novice might say “I understand how the **`for`** loop works and I can see your program solves the problem, but I don't think I could have dreamed up that solution myself.” This novice has distinguished between their programming *knowledge* and their programming *strategies* (or lack thereof). They have also shown they can comprehend the solution presented by the instructor but do not feel confident in their ability to generate that solution themselves.

In the BRACElet project (mentioned earlier in section 2.3.2), as well as asking novices to predict the outcome of code in a number of questions, participants were also shown a piece of code and asked to “In plain English, explain what the following segment of code does” (Whalley et al., 2006, p. 248). This last question has been referred to as “Question 10” and is sometimes quoted by this name. The responses to Question 10 were categorised according to levels of the SOLO Taxonomy (Biggs and Collis, 1982) which distinguishes levels of understanding. Responses were categorised as shown in Table 2.6.

Table 2.6. SOLO Categorisation of Question 10 responses (reproduced from Whalley et al., 2006)

SOLO category	Description
Relational [R]	Provides a summary of what the code does in terms of the code’s purpose.
Multistructural [M]	A line by line description is provided of all the code. Summarisation of individual statements may be included
Unistructural [U]	Provides a description for one portion of the code (i.e. describes the if statement)
Prestructural [P]	Substantially lacks knowledge of programming constructs or is unrelated to the question
Blank	Question not answered

Performance on Question 10 was consistent with other questions in the study. Approximately 30% of participants in the BRACElet study were able to give a SOLO Relational response for Question 10; 55% gave a Multistructural response; 13% gave a Unistructural response; a small remaining percentage showed only a Prestructural response. These results mean that 70% of novice participants were describing code line-by-line at best. Less than a third of novices were able to identify the overall purpose of the code. The BRACElet project authors propose that novices would need to give a SOLO Relational response to Question 10 before they could generate the same solution themselves (Whalley et al., 2006). Considering the comprehension-generation and knowledge-strategy aspects, this means that before a novice can generate code involving *strategies*, they must first show *comprehension* of the *strategies* in an equivalent piece of code. The results show that only 30% of the participants could comprehend the *strategies* applied in the solution while the remaining participants were relying on programming *knowledge*.

A follow-up paper from the BRACElet project group (Lister et al., 2006) asked instructors (as expert programmers) to explain the code previously given to students. Their responses were then analysed according to the SOLO categorisation given in Table 2.6. Seven of eight participating instructors gave a Relational response, suggesting that the ability to comprehend code at this level is related to programming expertise. This expertise seems to be lacking in 70% of the novices tested in the BRACElet project. This finding is consistent with that of Fix, Wiedenbeck and Scholtz (1993) who identified a contrast between the ability of novices and experts on program *comprehension*. Fix et al. suggest that experts can discover goals, relate goals to previous experience, recall plans, and integrate these to form a program. The BRACElet group went on to suggest that while their study has examined novice and expert potential to think in abstract ways about code, it does not identify how novices could be better trained to perform this task.

Assessment and Aspects of Novice Instruction

When assessing students it is possible to target skills in *comprehension* or *generation*. In a *comprehension* task, novices are given a piece of code and asked questions about it. For example a novice might be shown a piece of code and asked to predict its output. A novice might be asked to identify problems in a flawed piece of code. To test *generation* skills, novices can be asked to create a solution to a given problem.

It is also possible to assess *knowledge* and *strategies* independently. *Knowledge* tasks focus on the syntax and semantics of a language but do not require a novice to solve a problem. *Strategy* questions might ask a novice to identify a *strategy* used to create a problem solution (for example Question 10 from the BRACElet project) or apply *strategies* when solving a problem.

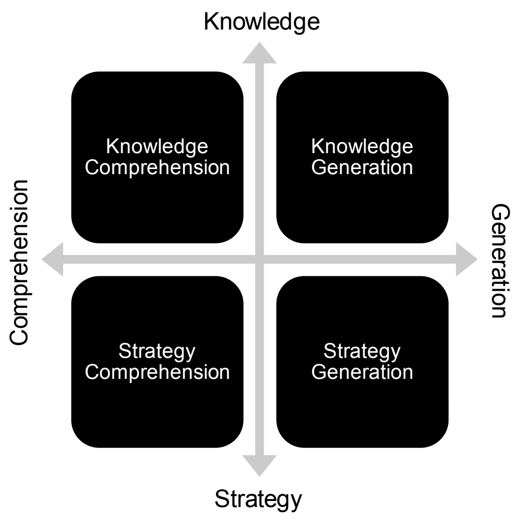


Figure 2.1. Creating programming assessments with consideration of novice instruction aspects

By combining the knowledge-strategy aspect with the comprehension-generation aspect, four types of assessment tasks can be identified, as shown in Figure 2.1. This division is the author's invention.

Exercises can be used to target programming *knowledge* by asking a novice to comprehend a piece of code, where *strategies* have not been applied. Knowledge-comprehension can be tested by asking novices to describe the effect of a particular language construct, such as an **`if`** statement, given in simple context.

Knowledge-generation can be assessed by asking novices to apply a particular language construct in a certain way. For example a novice could be asked to write a **`for`** loop that counts from 0 to 9. Such tasks can be designed without asking novices to solve a problem.

Strategy-comprehension can be assessed by showing novices solutions and asking them to identify *strategies* that were applied in creating the solution.

Strategy-generation can be tested by asking novices to generate a solution that requires the application of certain *strategies*. In programming assignments, strategy-generation is perhaps the most commonly assessed combination.

For examples of questions that target individual areas, see section 7.4.3.

2.3.6 Implicit and Explicit Instruction

The previous sections have addressed aspects on which computing education research can be classified, according to Robins et al. (2003). Another important aspect relevant to this dissertation is how instruction is delivered, which can be described as being implicit, explicit, or a combination of these.

Explicit instruction involves the instructor openly describing, usually in some documented form, what the student is to learn and how to go about that learning. *Implicit* instruction creates a scenario where a student is expected to undertake learning without being given a full context for what it is they are to learn or how.

The following is an example of the distinction between explicit and implicit instruction.

> *My four year old son enjoys playing Uno (a card game) against the computer. Being a father I naturally coach him on how to play. The goal of a player in Uno is to be the first to discard all their cards. In a turn a player must play a card with either the same colour or number as that on the top of the discard pile. If a player cannot match the colour or number of the top card on the discard pile they must draw a card from the pick-up pile or play a Wild card.*
>
> *For many of the turns a player takes the choice of card to play is simple. Occasionally a player must choose from a number of alternatives, and choosing one option over another can give the player an advantage later in the game. Determining the best option is a strategic decision. For example, when a player cannot follow the colour or number of the card at the top of the discard pile they may play a Wild card. When a player uses a Wild card they can choose one of four colours to follow in the subsequent turns. To make an appropriate choice of colour I can coach my son in one of two ways.*
>
> *I can examine the cards my son has and simply tell him which colour to choose. If we were to repeat such coaching several times my son might learn to generalise the advice I am giving him and extrapolate a strategy for choosing a colour after playing a Wild card. This is an implicit approach.*

> *I can inform my son that a good strategy to follow after playing a Wild card is to determine which colour is in the largest majority of his cards and nominate that colour for subsequent turns. Giving my son this strategy in this manner is an example of explicit instruction. After instructing my son in this way, when the opportunity arises, I encourage my son to practice this strategy.*

The *Sapir-Whorf Hypothesis* (Whorf, 1956) proposes that language determines thought: "We dissect nature along lines laid down by our native languages" (p. 212). In order to be able to think about something you need some term to describe it. This proposal might not be entirely true (particularly in light of ambiguity (Pinker, 2007)), but humans are driven to take what is tacit and make it explicit in order to describe it.

During an introductory programming course, novices are expected to develop *knowledge* and *strategies* to allow them to comprehend and generate solutions.

Novices will not become experts during an introductory course, but can be expected to reach a level of competency. A traditional form of programming problem-solving instruction begins with a worked example: novices are shown a simple problem solution from an instructor (or a textbook). Following exposure to a range of problem solutions, a novice is given a problem definition requiring a solution similar to those presented in examples. The novice is expected to devise a solution; they are expected to build *strategies* by undertaking problem solving, applying reasoning about the examples presented. Typically no framework is given to the novice to assist them in building the *strategies* required for a solution. This is an implicit approach to learning *strategies*. By contrast, explicit *strategy* instruction presents concrete techniques that a novice can use to take a problem definition and create a solution. An explicit approach guides the novice to learn and apply *strategies*.

Rist (1991) observed novices in an implicit-only setting as they attempted to construct *strategies* to solve problems and noted that many of the novices succeeded. He also noted that some novices were able to re-apply *strategies* they had developed earlier to new problems in order to solve them more quickly (Rist, 1995). However, achieving this may be possible for only a small group of novices. McCracken et al.(2001) and later studies (Whalley et al., 2006, Lister et al., 2004) have shown that many novices have a fragile programming *knowledge* and lack programming *strategies* at the end of an introductory course.

Beyond computing education a number of studies have compared the outcomes of students under implicit and explicit education. A comparison of explicit and implicit instruction, undertaken by Biederman and Shiffrar (1987), showed a stark quantifiable difference between these two approaches. Chick-sexing is a profession that involves determining the gender of day-old chicks at commercial egg hatcheries. The distinction between male and female chicks remains hard to determine by visual examination of genitalia until one month of age. However, being able to determine the gender of chicks early avoids feed wastage on unwanted males. Professional sexers can classify over 1000 chicks per day and can identify gender in less than a second, with a required level of accuracy. Traditional training of sexers involves six to twelve weeks of implicit instruction, standing along-side an expert instructor, making observations, then attempting the task through trial and error. It can take years for novice chick sexers to achieve the experience and accuracy of an expert professional. In this context Biederman and Shiffrar established an experiment to measure the effectiveness of implicit instruction and compare it to explicit instruction. Initially a benchmark of accuracy was attained through a group of volunteers with no sexing experience who were asked to identify the gender of 18 chicks from genitalia photographs. Performance of these subjects was 60.5%, slightly greater than chance. The same photographs were shown to five sexers, who had been trained in the traditional implicit fashion, who achieved an average performance of 72%. A sexer with vast experience (quoted as 50 years and 55 million chicks) was recruited to identify the gender of chicks from the series of photos. Biederman and Shiffrar then interviewed this man and asked him to explain the visual aspects that prompted his decisions. From this interview a single instruction sheet was created which explicitly described key visual aspects. The volunteers were split into control and experimental groups. The instruction sheet was given to the experimental group to study for one minute. After instruction both groups of volunteers were retested using a second set of photographs. Control subjects showed no improvement in their accuracy over the original measurement. Volunteers in the experimental group, who had read the explicit instruction sheet, averaged accuracy of 84%, which was above that of trained sexers in the initial test. An important aspect of this experiment is that the volunteers are not learning a skill which needed to be generalised before it could be used. The participants were taught strategies which could be applied directly to the task they were being tested on. According to Baddeley (1997) the findings of this chick-sexing experiment demonstrate that a brief period of explicit instruction can be more effective than months of implicit learning.

Studies have shown that implicit-only learning can improve a student's performance but it does not create an understanding of underlying systems. In a study closer to programming, Reber (1993) examined implicit learning in the context of language acquisition. According to Reber, children learn the greater part of their native language through implicit means. Second-language instruction is usually achieved through explicit study of the grammar of a new language. Reber used a small, finite-state artificial grammar to test the effectiveness of implicit learning of a second language. An experiment was established involving volunteers divided into control and experimental groups. The experimental group was shown sequences generated from the grammar without being shown the rules of the grammar used to construct the sequences. The control group was shown sequences that were randomly generated and not part of the grammar. After training, both groups were shown 44 sequences, half of which were grammatically correct according to the grammar. The participants were asked to determine which were well formed according to the grammar. Experimental subjects achieved 79% accuracy while members of the control group showed no capacity to accurately distinguish sequences. The results showed that the experimental group had learned the grammar and were able to recognise sequences from it. However, when the experimental group participants were interviewed and asked to describe the grammar they had been exposed to, they were unable to express any understanding of the rules used to generate sequences.

Berry and Dienes conducted a similar experiment (Berry and Dienes, 1993) which asked participants to learn the workings of a simulated transport system through implicit instruction only. Participants showed learning and an ability to operate the system, but when asked to describe the underlying rules of the system, participants were not able to show any understanding.

The previously described experiments indicate the weakness of implicit-only learning and the strength of explicit instruction. It is not the purpose of this dissertation to suggest that explicit instruction be adopted at the expense of implicit learning; programming is still a practical, creative art and much benefit can still be gained through self-discovery. Novice programmers can learn programming *strategies* over time though implicit instruction, but it may be possible to improve the outcomes of novices by adding explicit instruction of programming *strategies*. Husic, Linn and Sloan (1989) discuss how teaching practices influence how students solve problems. If syntax is the focus, students will attempt to solve problems by syntactical means only. “Instructors must achieve a delicate balance between providing opportunities for independent problem-solving and modelling explicit problem-solving strategies” (p. 581). According to Soloway (1986, p. 851), “strategies that experts use need to be made explicit and taught explicitly to students in introductory programming courses.”

In light of previous research, adding explicit instruction to introductory programming curricula may:

- increase student learning speed;
- create a more structured understanding of the problem-solving processes;
- create an enriched vocabulary for describing problems and how to solve them; and
- enable instructors to undertake deeper analysis and assessment of novice programming *strategy* skill.

Another important requirement, suggested by previous research such as the chick sexing experiment, is need to be specific about which *strategies* will be included in the curriculum rather than teaching problem solving as a general abstract task. So not only is there a need to teach programming *strategies* in an explicit manner, the *strategies* to be taught needs to be specifically defined also.

2.4 Explicit Programming Strategies

If it is desirable to include explicit instruction of programming *strategies* in introductory curricula, an instructor must first capture and document these in a form that can be delivered explicitly to novices. Robins et al. (2006) portray *strategies* as being important but ill-defined in literature. A number of attempts have been made to represent *strategies*; these include *plans*, *schema* and *patterns*. This section describes attempts to create explicit representations of programming *strategies* which can be, or are being, delivered to students.

According to Soloway (1986), programming *strategies* are made up of *plans* and the associated means of incorporating these into a single solution. *Goal/Plan Analysis* is the process of describing an ideal solution, which contains appropriate *plans*, and comparing this with the solution of a novice. This analysis allows an instructor to see if a novice has succeeded in learning and applying specific *plans*. Much of the research by Soloway and his colleagues used the idea of *plans* to explore misconceptions that novices exhibit (Spohrer and Soloway, 1986, Spohrer et al., 1985a). PROUST (Johnson and Soloway, 1984) was one of a series of intelligent tutoring systems including the GPCEditor (Guzdial et al., 1998) and SODA (Hohmann et al., 1992). PROUST could perform Goal/Plan Analysis on a Pascal program, comparing its *plan* structure to a structure established by an instructor. Johnson (1986) gave a description of the inner workings of PROUST and also released a catalogue of goals and related *plans*. *Plans*, as a form of programming *strategy*, are a candidate for explicit instruction to novices.

The idea of the schema/plan was not widely used by instructors for many years until the rise of the object paradigm, which brought with it a new sense of reuse and a new term to computing: *patterns* (Wallingford, 1996). According to Clancy and Linn (1999), "learning programming means learning patterns and strategies that enable rapid learning of new programming languages" (p. 37), but novices do not infer *patterns* naturally, and so instructors should "create appropriate exercises and supports so students extract patterns, reuse patterns, develop a disposition to use patterns, and create patterns of their own" (p. 41). Porter and Calder (2003) have proposed *A Pattern-Based Problem-Solving Process for Novice Programmers*. Their approach shows students how to apply *patterns*. Porter and Calder also use a *pattern language* for applying *patterns* to problems and refining solutions. They believe *patterns* have enhanced their curriculum and pedagogical approach. "Patterns lend themselves to the learning of a skill like programming, because they provide the static knowledge plus the means to apply it" (p. 236). Porter and Calder tested their approach on a small number of volunteers divided into control and experimental groups (Porter and Calder, 2004). Participants were asked to undertake an exercise under test conditions. This study showed slightly better outcomes in participants who had been exposed to *patterns* and the *pattern language*, however none of the participants in either group demonstrated any obvious use of the *patterns* or the *pattern language* during testing. A later study by Muller, Haberman and Ginat (2007) showed novices to be more competent in problem decomposition and solution construction after studying under a pattern-oriented instruction approach. In this study novices were shown how *patterns* can be used and were instructed in algorithmic *patterns*. There is a growing community of instructors interested in the *pattern* approach (Wallingford, 2007).

Based on the *plan* ideas of Soloway, Sajaniemi has been refining an explicit description of the *roles of variables* which is being incorporated in introductory programming curricula (Sajaniemi, 2002). Sajaniemi's categorisation of variables by their role (for instance *constant*, *stepper*, *most-recent holder* and so on) is claimed to cover 99% of variables encountered in examples in an introductory programming course. When code is shown to students the role of each variable is identified. A standard visualisation of variable roles has also been created. Kuittinen & Sajaniemi (2003) describe an experiment involving novices divided into three groups, a control group (receiving traditional instruction) and two experimental groups (who were explicitly instructed in roles, with one experimental group also being exposed to animation of roles in examples). After an exam involving *comprehension* and *generation* exercises, an analysis of results found no significant difference between groups on questions. However, when asked to give explanations of their answers, novices in the control group tended to give "operation level descriptions" while novices in the experimental groups gave "data level" descriptions, which reflect a deeper knowledge of a program and represent better comprehension (Pennington, 1987). Sajaniemi & Kuittinen (2005) conclude that novices are able to learn the *roles of variables* and apply them to new situations. They believe this allows novices to generate solutions which contain fewer errors and demonstrate superior programming skills.

Related to programming, Klahr and Carver (1988) found that students explicitly instructed and assess in debugging strategies showed improved debugging ability in later programming courses.

An experimental curricula, described later in this dissertation, uses Soloway's *plans*. *Plans* were chosen over *patterns*, even though *patterns* have become more widespread in recent years. *Patterns* are commonly used in the object paradigm and require a *pattern language* for application. *Plans* can be used in multiple paradigms, including the object paradigm. *Plans* can be expressed simply, particularly at a sub-algorithmic level. In saying this, the focus of this research is not on the types of *strategy* that are taught but on *how* they are taught, and the consequent outcomes for students. It is likely that *patterns*, or another *strategy* representation, could be used to achieve the same programming *strategy* understanding for students as *plans*.

From this point on the term *plan* is used to represent a specific form of *strategy* and the term *strategy* is used in its more generic sense.

2.5 Need for Further Research

Programming instruction is a relatively new practice. Programming curricula have evolved as student cohorts and technologies have changed, and have followed the shifting standards of the computing industry (§2.1). Currently student numbers in computing courses are dwindling, which places pressure on instructors to perform (§2.2).

Instructors do not have a common definition of what constitutes problem solving instruction in an introductory programming course and differ greatly on the extent to which problem solving should be incorporated into courses at this level (§2.2.4). Most existing textbooks contain little content addressing problem solving and most do not integrate this throughout (§0).

Introductory programming instruction is cognitively demanding, with many novices failing to reach expected standards at the end of an initial period of instruction. Studies have shown that novices perform poorly on standardised program *generation* tests (§2.1.1). In program *comprehension* tests, novice performance is better, but still poorer than expected by instructors. This may indicate that the programming *knowledge* of novices is fragile (§2.3.2). When asked to explain the purpose of a given piece of code only 30% of novices were able to give a SOLO Relational response, indicating a possible lack of programming *strategy* skill (§2.3.5). These strategy-related deficiencies could be compounding the effect of poor programming *knowledge* in *generation* exercises.

The traditional approach to teaching programming to novices in an introductory course has been to gradually reveal the constructs and features of a programming language. Most attempts to enhance this approach, in order to improve outcomes for novices, have simply been novel ways of teaching the same curriculum.

> *New efforts to ease and enhance learning have varied in their general approach to improve learning: most studies report effects of new teaching methods and new ways of presenting teaching materials, while reorganization of topics and introduction of new concepts have been far more rare. (Kuittinen and Sajaniemi, 2003, p. 347)*

Considering new concepts and ways of integrating these concepts may improve the potential of novices.

2.5.1 Strategies Appropriate for a Curriculum

Goal/Plan Analysis has been used as a tool for determining weaknesses in a student's code and identifying gaps in their application of *plans* (§2.4). Expressing *strategies* as *plans* provides a representation of *strategies* that can be explicitly incorporated into a curriculum. However, while *plans* are claimed to be based on the tacit *strategies* of experts, this has not been authenticated. Any proposed set of *strategies* needs to be validated as authentic by comparing them to the *strategies* used by expert programmers.

2.5.2 Integrating Strategies into a Curriculum

Studies have investigated the incorporation of programming *strategies* explicitly into introductory programming curricula as *patterns* (Wallingford, 1996, Porter and Calder, 2003) and the *roles of variables (Sajaniemi, 2002, Ben-Ari and Sajaniemi,* *2004)*. Integration of *plans* as *strategies* was never attempted by Soloway, the initiator of *plans,* or his colleagues, but it was something they intended to do (Soloway, 2003). Once a set of *strategies* has been validated as authentic, and expressed in a form that us suitable for dissemination to novices, it can be explicitly integrated into an introductory programming curriculum. The usefulness of the curriculum and its impact on novices needs to be measured and contrasted to those of a traditional curriculum.

2.5.3 Assessing Strategy Ability in Novices

Goal/Plan Analysis is a tool for measuring the *strategy* skill of a novice programmer. However, it is not an appropriate tool for regular assessment in an introductory programming course. Testing novices' programming *strategy* skills, as a means of assessment, can be achieved by isolating programming *knowledge* and programming *strategies* in assessment items and measuring these separately. The consistency of tasks used for such assessment needs to be measured.

The next chapter describes the overall methodology followed in the experiments that form the core of investigation described in this dissertation. The scope of experimentation is defined using the terminology given earlier in this chapter. Research questions are discussed and the method used to answer these questions is given. This leads into later chapters which describe each of the experiments in detail.

3. Experimental Methodology

Chapter 1 of this dissertation identified a rationale and aims for study. Relevant research was described in chapter 2, which also identified important aspects of introductory programming instruction and a need for further research.

This chapter discusses the methodology of the four experiments that were conducted for the purpose of this dissertation. The scope of these experiments is defined in section 3.1. The method of analysis is described in section 3.2. The experiments were undertaken to address a series of research questions which are discussed in section 3.3. Finally a preview of the experiments described in the following four chapters is given in section 3.4.

3.1 Scope of Experimentation

This dissertation studies instruction of problem solving to novice programmers. A categorisation of existing research into teaching of programming within computing education, encountered by the author, is given in Figure 3.1.

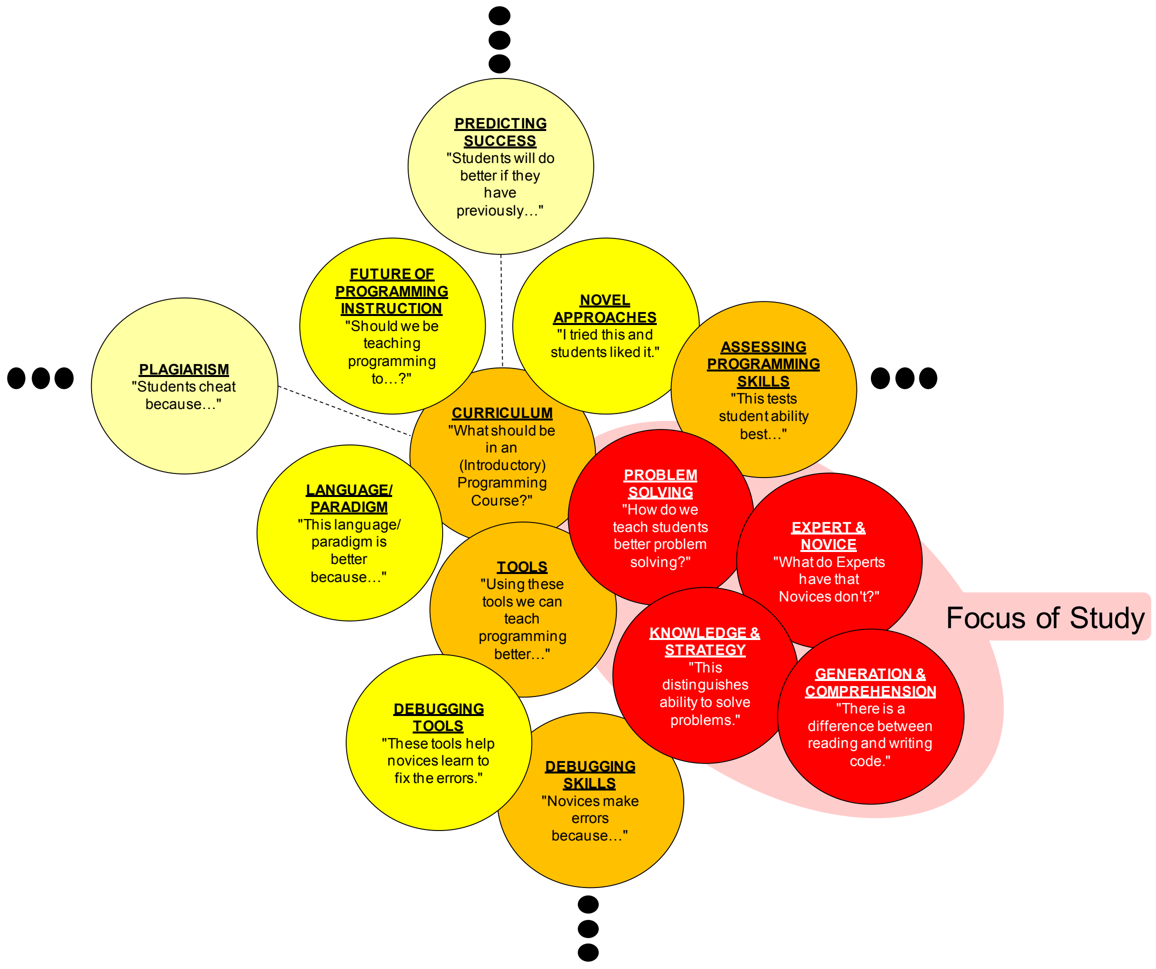

PREDICTING SUCCESS "Students will do better if they have previously…"
FUTURE OF PROGRAMMING INSTRUCTION "Should we be teaching programming to…?"
NOVEL APPROACHES "I tried this and students liked it."
PLAGIARISM "Students cheat because…"
ASSESSING PROGRAMMING SKILLS "This tests student ability best…"
CURRICULUM "What should be in an (Introductory) Programming Course?"
PROBLEM SOLVING "How do we teach students better problem solving?"
LANGUAGE/ PARADIGM "This language/ paradigm is better because…"
EXPERT & NOVICE "What do Experts have that Novices don't?"
TOOLS "Using these tools we can teach programming better…"
KNOWLEDGE & STRATEGY "This distinguishes ability to solve problems."
Focus of Study
GENERATION & COMPREHENSION "There is a difference between reading and writing code."
DEBUGGING TOOLS "These tools help novices learn to fix the errors."
DEBUGGING SKILLS "Novices make errors because…"

Figure 3.1. Some computing education research areas showing focus of this dissertation

This dissertation focuses on the area, highlighted in red in Figure 3.1, relating to the teaching of problem solving to novice programmers. Klahr and Carver (1988) found some success in transferring debugging skills learned in a programming context to a non-programming context. However, as described by Perkins (1992) transfer of learning is limited to contexts closely associated to the learning context. Achieving "near transfer" is far more likely than achieving "far transfer", if transfer can be achieved at all. In this dissertation problem solving is considered only in a programming context rather than in its general sense. Strategies suggested are intended only for use in programming and are not expected to benefit students' problem solving skills in other disciplines.

Specifically within this problem-solving area, experimentation is targeted at sectors of the following aspects (defined in section 2.3).

- **Expert-Novice**
  Experimentation will focus on exploring this area for the benefit of *novices* learning programming in their *initial exposure* (that is in an introductory programming course, sometimes referred to as CS1). Experimentation involving *experts* will be conducted for the purpose of advancing the quality of novice instruction.
- **Knowledge-Strategy**
  This dissertation primarily explores the instruction of *programming strategy* although programming *knowledge* instruction is considered as it is related to (if not required for) the application of programming *strategies*. The level of problems solved by novice programmers in their initial exposure is *sub-algorithmic*, sometimes reaching simpler problems at the *algorithmic* level.
- **Comprehension-Generation**
  Both strategy-comprehension and strategy-generation are considered and explored in this dissertation. The focus of the first three experiments is strategy-generation, driven by the method of analysis (Goal/Plan Analysis see section 3.2.1). In the final experiment, approaches for teaching and assessing both strategy-comprehension and strategy-generation are tested.

In relation to teaching approach along the **implicit-explicit** aspect (defined in section 2.4), this dissertation:

- measures the effect of implicit-only teaching on novices' programming *strategies* (chapter 4);
- describes authentic *strategies* that can be used in explicit instruction (chapter 5); and,
- observes the impact of explicitly teaching programming *strategies* in artificial and actual instruction settings (chapters 6 and 7).

The representation of programming *strategies* chosen for the experiments described here include *plans* used by Johnson and Soloway (1984).

3.2 Experimental Approach

A binding feature of the four experiments described in this dissertation was a common method for experimentation. The common instrument for determining impact of a curriculum on novices was Goal/Plan Analysis, which is described in section 3.2.1 below. This method of analysis was used to study novices' solutions and also to study the solutions of experts in validating the authenticity of the *plans* as an explicit form of programming *strategies*. In the final experiment, alternatives to

Goal/Plan Analysis are suggested as tools for assessing programming *strategy* in novices.

3.2.1 Applying Goal/Plan Analysis

Goals and Plans, and the ability to compose *plans* into a solution, form an enriched vocabulary of programming *strategies* (Soloway, 1986). "Goals and plans – stereotypical, canned solutions – are two key components in representing problems and program solutions" (p. 851).

Goal/Plan Analysis is a method for analysing code created by a novice to determine if they have understood and applied appropriate *strategies* in the code's construction. Goal/Plan Analysis was proposed by Elliot Soloway and his colleagues in early papers (Soloway and Woolf, 1980, Soloway et al., 1982, Soloway et al., 1983b) but arguably the definitive description of Goal/Plan Analysis is given in Soloway (1986). After justifying the motivation for using goals and *plans*, a description of the application of Goal/Plan Analysis is shown through a number of examples in this seminal paper.

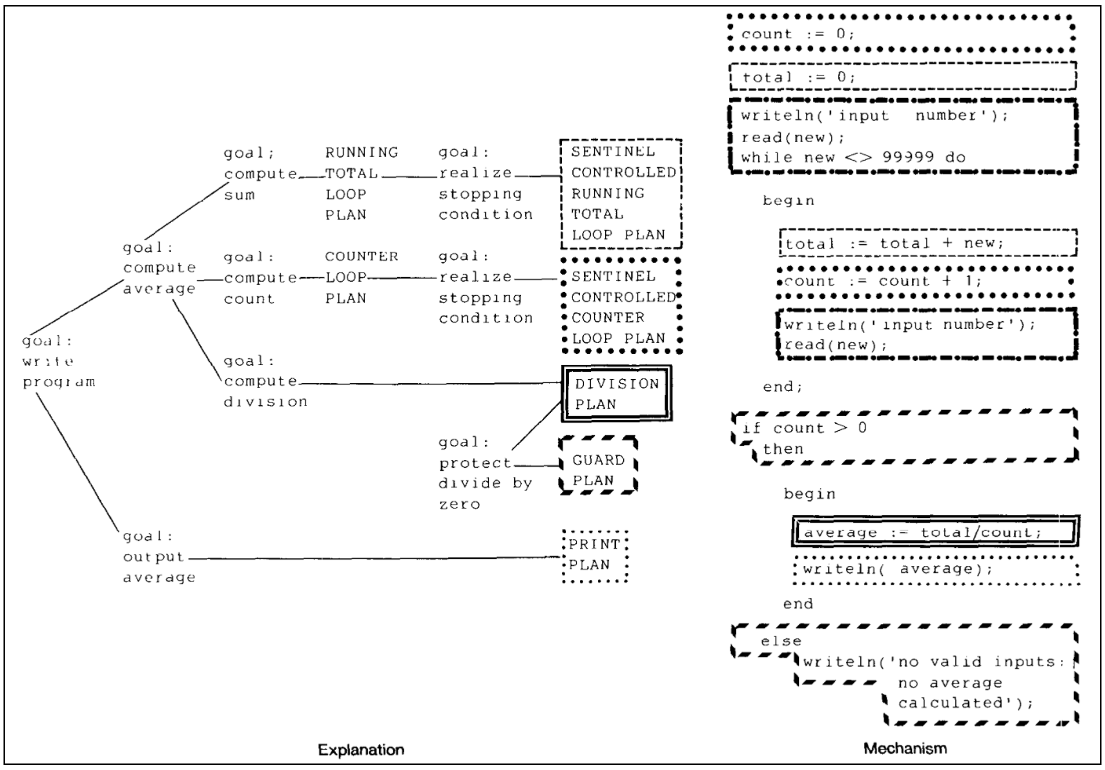
As a knowledge elicitation technique involving experts (Cooke, 1994) Goal/Plan Analysis can be classified as content analysis, a form of protocol analysis. Protocol analysis is an appropriate tool for directly capturing expert's problem solving strategies (Burje, 1998).

With a given problem, the process begins with the instructor determining the goals that need to be achieved to solve the problem. These goals are then mapped to *plans*. In this context, a *plan* is a stereotypical abstract solution to a sub-algorithmic problem. A small set of *plans* are illustrated in Soloway (1986). A more complete set is published in Johnson and Soloway (1984) with the description of the programming tutor PROUST. Some *plans* and other *strategies* were added to create a fuller curriculum for experimentation. A list of *strategies* referred to in this dissertation is given in Appendix A.

Soloway (1986) used the following averaging problem as an example.

> *Write a program that will read in integers and output their average. Stop reading when the value 99999 is input. (p. 851)*

Soloway gives a model solution for this problem, which is reproduced in Figure 3.2.



```
count := 0;
total := 0;
writeln('input  number');
read(new);
while new <> 99999 do
  begin
    total := total + new;
    count := count + 1;
    writeln('input number');
    read(new);
  end;
if count > 0
  then
    begin
      average := total/count;
      writeln( average);
    end
  else
    writeln('no valid inputs:
            no average
            calculated');
```

Figure 3.2. How a solution is derived from goals and *plans*, reproduced from (Soloway, 1986)

The instructor must also define how the *plans* should be integrated to form a solution. Soloway gives the following methods as “gluing together plans [that] have been identified” (p. 856):

- **Abutment**
  Plans, or parts of *plans*, are glued together in sequence, as illustrated on the right-hand-side of the averaging problem shown in Figure 3.2.
- **Nesting**
  One *plan* is completely surrounded by another *plan*. For example, in the averaging program (in Figure 3.2), the OUTPUT PLAN, the *plan* that realises the goal of writing out the average, is nested within the SKIP-GUARD PLAN, which realises the goal of preventing division by zero in the average calculation, which can occur if the count of inputs is zero.
- **Merging**
  At least two *plans* are interleaved. For example, to solve the averaging problem, the input, summing and counting plans are merged.

Soloway also suggests that *plans* need to be tailored to meet the specific goals of a problem. “After all, we do call it ‘software’” (p. 856).

Once a model solution is created from *plans* composed together using the integration methods put forward by Soloway, this model can be compared to a solution given by a novice. The presence or absence of *plans* in the novice's solution, and the correct application of integration methods is noted. Flaws in the novice's solution emerge where *plans* are missing or poorly integrated. The *goal*/*plan* vocabulary can be useful in describing these bugs and correcting a novice's solution.

3.3 Research Questions

Past research presented in chapter 2 showed a need for curricular development which can be linked to the aims of this study, specifically:

- validation of an authentic set of strategies that are suitably expressed for explicit instruction to novices (aim 1);
- development and testing of a curriculum that incorporates programming strategies (aim 2); and
- development and testing of forms of assessments designed to test *knowledge* and *strategy* ability independently (aim 3).

Four experiments were conducted and are described in the following four chapters. A set of research questions was associated with each experiment, and used to guide the design and evaluation of each experiment. The experiments were conducted in a series with the conclusions of each experiment dictating the research questions to be answered in the next.

The research questions relevant to each experiment are listed in an initial section of the chapter relating to that experiment. These questions are also re-stated in section 8.1, where the answers to the questions are presented together. A brief overview of the questions asked for each experiment is now given.

3.3.1 Initial Experiment

Initially a baseline for student learning under a traditional curriculum needed to be measured. The questions listed in section 4.2 directed the experiment described in chapter 4 and are answered in section 4.6. The questions ask about the *strategy* skills of novices trained using implicit-only instruction of programming *strategies* and what this implies about the curriculum used at the time.

3.3.2 Validation of Strategies

In order to establish an appropriate set of authentic programming *strategies* that can be incorporated explicitly into an introductory programming curriculum, the tacit *strategies* of experts needed to be explored. The questions listed in section 5.2 relate to the experiment described in chapter 5 and are answered in sections 5.6 and 5.7. The questions ask if *plans* are consistent with *strategies* applied by experts when solving problems.

3.3.3 Use in an Artificial Course

An attempt to incorporate programming *strategies* explicitly into an introductory course was attempted, initially in an artificial setting. The questions listed in section 6.2 drove the experiment described in chapter 6 and are answered in section 6.6. The questions ask about the feasibility and impact of explicit instruction of *strategies* and the potential to assess novices using Goal/Plan Analysis.

3.3.4 Use in an Actual Course

After successful testing in an artificial setting, the curriculum was used in an actual course. The questions listed in section 7.1.1 relate to experiment described in chapter 7 and are answered in section 7.7. The questions ask about the feasibility and impact of explicitly teaching *strategies* in an actual programming course, and the use of assessment items that attempt to separate *knowledge* and *strategy* skills.

3.4 Overview of Experimentation

There are four main experiments described in this dissertation. The experiments were conducted in sequence with the results of each experiment informing questions to be answered in the next. Retrospectively the sequence of experiments can be viewed as shown in Figure 3.3.

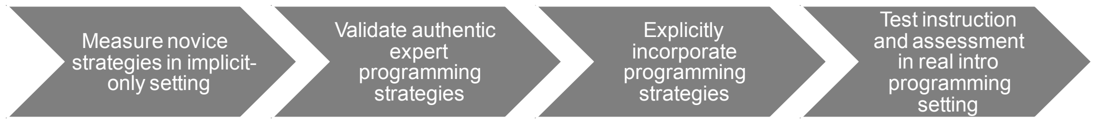

Measure novice strategies in implicit-only setting
Validate authentic expert programming strategies
Explicitly incorporate programming strategies
Test instruction and assessment in real intro programming setting

Figure 3.3. Overview of experiments in a process

Each experiment is described in one of the following four chapters.

In chapter 4 an experiment is described that attempted to measure the effect of implicit-only teaching on novices' programming *strategies*.

Chapter 5 describes an experiment that sought to capture and make explicit *strategies* that can be held as authentic and used by experts.

In chapter 6 an experiment is described that involved the delivery of two curricula, one containing explicit instruction of programming *strategies* and another taking an implicit-only approach. These curricula were delivered in an artificial setting. The impact of the two curricula were measured and compared.

Chapter 7 describes the implementation of a curriculum including explicit instruction of programming *strategies* in an actual setting and the subsequent impact on novices.

Following the description of each experiment the main findings are recapitulated in chapter 8 and are used to offer the contribution made.

The experiments described in chapters 5 and 6 were conducted with individuals, outside normal teaching environments. For these experiments, permission was sought and ethical approval was granted by the USQ Human Research Ethics Committee to proceed under controlled circumstances that protected participants.

4. Goal/Plan Analysis of Programs created by Novices with No Explicit Strategy Instruction

"In the beginner's mind there are many possibilities, but in the expert's mind there are few."
Shunryu Suzuki

Overview

This chapter describes an experiment conducted with 42 introductory programming students. The students had received little explicit instruction in general problem solving (see section 4.1.2) and had not been explicitly exposed to sub-algorithmic programming *strategies* in any form. At the end of a semester of programming instruction, students were asked to write a solution to an averaging problem. Solutions created by students were analysed using Goal/Plan Analysis. Results showed gaps in students' programming *strategies*, implying weaknesses in the curriculum.

4.1 Introduction

Goal/Plan Analysis is a tool for identifying weaknesses in the solutions of novice programmers (see section 3.2.1). Goal/Plan analysis has been used to find common bugs or misconceptions present in the programming *strategies* of a cohort of novices (Spohrer et al., 1985a, Johnson, 1986), but to the author's knowledge, no previous study has applied Goal/Plan Analysis to an entire cohort of students to find the general programming *strategy* skill levels of those students. The aim of the experiment described in this chapter was to achieve this and discern from this weaknesses in the curriculum being delivered.

4.1.1 Participants

Participants were students studying in a first-year introductory programming course. All participating students were attending on-campus classes. Participants included school leavers (recent high school graduates) and mature-aged students. Students were enrolled in a range of discipline areas but were primarily IT and Engineering students.

4.1.2 Setting

This experiment refers to an introductory programming course taught at the University of Southern Queensland. The course was designed for students with no previous programming experience.

At the time of the experiment, the curriculum for the course was focused on the syntactic constructs and facilities of the C programming language, with little coverage of problem solving and no explicit instruction in programming *strategies*. The topics covered in the course were listed as follows.

- Programming Concepts
- Developing Programs with Functions

- Storing Data
- Writing Functions
- Conditions
- Pointers
- Input
- Repetition
- Arrays
- Text Files
- Structures and Abstract Data Types
- Recursion

Each of these topics was covered by a single module in the study materials. One module was taught each week of the course. Study materials consisted of a written 'Study Book' and lecture notes delivered during on-campus lectures and posted on the course website afterward. Each module included paper-based and computer-based tutorial exercises. Some of the exercises required students to solve problems and implicitly learn programming *strategies*. There were three major assignments during the course and each covered a number of modules.

None of the modules was devoted to programming *strategies*; however a description of the problem-solving process was given in the initial module. The context of problem solving was described as *Design*, *Implementation, Compilation* and *Testing*. The lecture notes described *Design* within the problem-solving process as follows.

> *Before the programmer can solve a problem, they must know precisely what the problem is. A good programmer will take time to properly define the problem, including the inputs and outputs the program has. When this is defined, the programmer will design an algorithm on paper or using some computerised tool. An algorithm is a finite sequence of precise instructions that leads to a solution.*

Other than this, a number of programming conventions and tips were discussed in the study materials, though they could not be seen as forming programming *strategies*. These were as follows.

- The "Dangling Else" problem
- Clearing Standard Input
- Meaningful Identifiers
- When to Use What Loop (**`for`**, **`while`**, **`do-while`**)

The final sub-topic above (When to Use What Loop) is closest to a programming *strategy*. This topic referred to each looping construct available in the C programming language and showed examples of the typical use of each construct. The materials did not suggest how a student should apply loops in general, or the set of goals that loops can be applied to achieve.

No sub-algorithmic programming *strategies* (defined in section 2.3.3) were explicitly covered in the course; students were expected to learn these implicitly through exercises. The following is a problem students were asked to solve in a practical session at the end of the Repetition module.

> *Write a program that will allow the user to enter words. Use the* **`%s`** *format sequence in a* **`scanf()`** *call to capture each word one at a time. Find the length of each word using* **`strlen()`**. *To end the user input, the user will enter the string* **`"end"`**. *At the end of the program, output the count of words and the average length of the words.*

In the problem description the student is not asked to reflect on the *strategies* needed to solve the problem, nor how to integrate these *strategies*. The wording of the problem focuses on the syntactical nature of the problem: what functions to use, what format sequence to use. Students are expected to implicitly learn how to create a loop that will stop when the word "end" is encountered. They are expected to discover implicitly how to count the words, sum their lengths and produce an average. Students would not be familiar with any of these *strategies* as they were not covered explicitly in the course. They are expected to merge these ideas into a coherent solution. The *strategies* are not suggested in the problem itself as no vocabulary to express such *strategies* had been established between the instructor and students.

A complete solution would include:

- a sum and count variable both initialised to zero;
- inputs gathered until the sentinel word is encountered (the sentinel should not be included as an input);
- counting of words;
- summing of word lengths;
- merging of input, counting and summing so that the words only need to be entered once; and
- calculation of the average (being sure that the division does not take place if there is a zero count caused by the word "end" being entered as the first input).

From this perspective, the *strategies* required to solve the problem are the same as those needed for the problem used in the experiment described in this chapter. Therefore students had been given the opportunity to learn the *strategies* required to solve the experimental problem through implicit means.

The task described in this experiment required students to complete a solution on paper. Students had experienced writing solutions to programming problems on paper during tutorial classes, so they were familiar with generating code in the experimental context.

4.2 Research Questions

This experiment was motivated by two related questions which are answered in section 4.6.

RQ1. *What is the potential of students who have been exposed to an implicit-only teaching of programming strategies to solve a sub-algorithmic* *problem that requires application of a number of programming strategies for a complete solution?*

RQ2. *What are the deficiencies in the curriculum that are demonstrated by students' solutions to the given problem?*

If it is seen as desirable to incorporate explicit instruction into an introductory programming curriculum, answers to the above questions will provide a benchmark for future comparison of student results.

4.3 The Experimental Problem

The following problem is taken verbatim from a paper by Elliot Soloway (1986).

Write a program that will read in integers and output their average. Stop reading when the value 9999 is input.

In his paper Soloway used this problem to demonstrate how Goal/Plan Analysis can be applied. Examples of student solutions were used to show correct and incorrect application of the specific set of required *plans*. No statistics on the success of any particular cohort was given, so there was no pre-existing benchmark for the problem's difficulty.

This problem was chosen for this experiment because of its previous use, with a well described method of analysis. Using Goal/Plan Analysis it is simple to identify *plans* within a solution to this problem. The problem is simple in its wording, allowing students to complete the problem without needing to refer to further information. The problem itself is language independent. It can be solved using any language under any paradigm. The examples and solutions given below are in C, but this is not the only language that has been used to solve the problem for Goal/Plan Analysis. In chapter 6 of this dissertation a version of the problem is shown in the JavaScript language. The original problem solution (Soloway, 1986) was shown in Pascal (Figure 3.2).

A correct solution to this problem will demonstrate programming *knowledge* and programming *strategies* that a student would be expected to demonstrate at the end of a semester of programming instruction, in order to be awarded a passing grade. The problem includes the ideas of sequence, selection and repetition, input and output, and simple operations, which are likely to appear in any introductory programming curriculum, even in courses that avoid programming *strategy* instruction. To create a complete solution to this problem certain goals would need to be recognised, and these goals would need to be mapped to plans, integrated correctly to form a single solution. The goals and plans and an integration example are described in the following subsections.

Goals

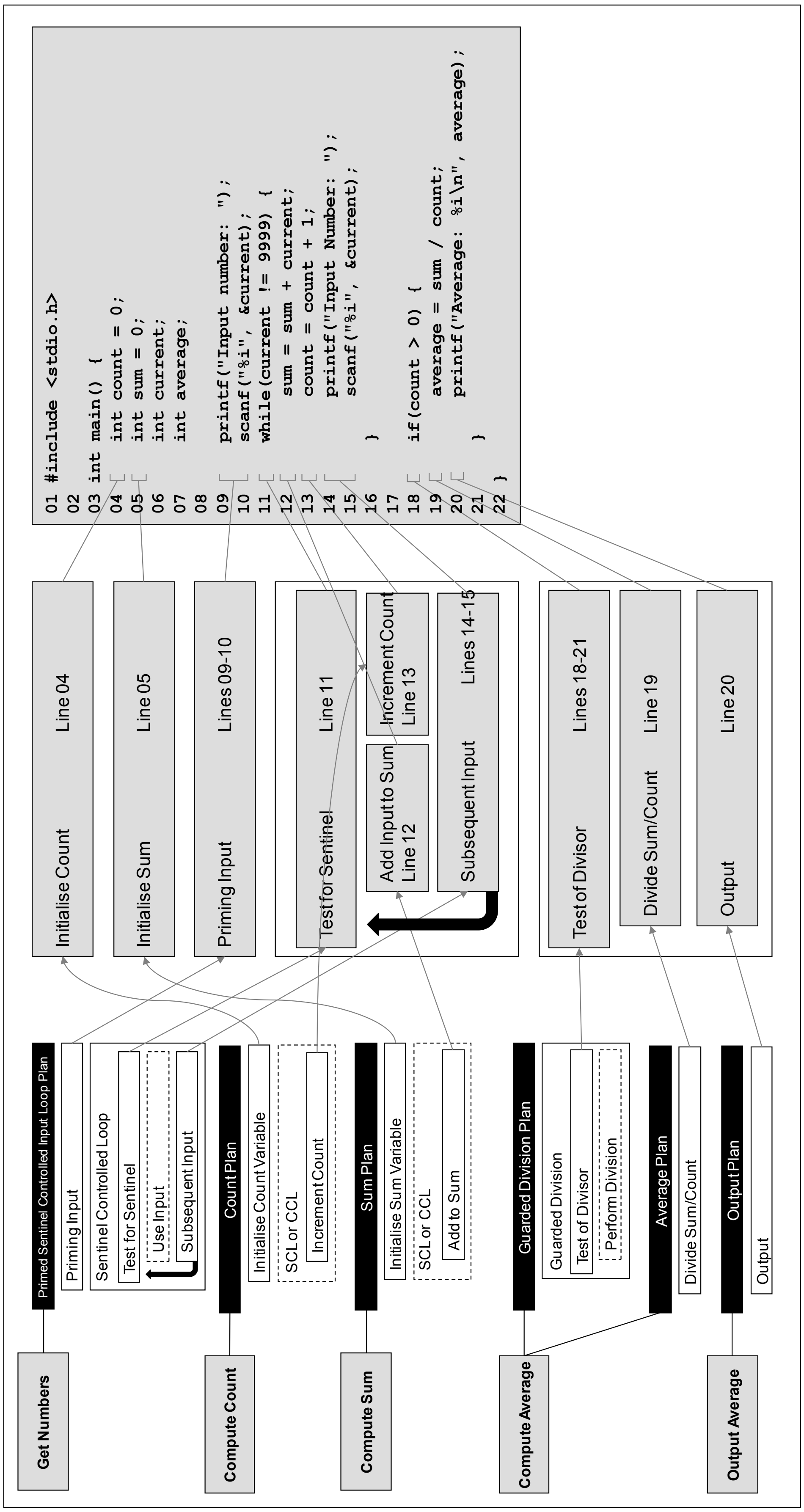
A number of goals are alluded to in the problem description. The goals should be apparent to novices through reading the problem description and without lengthy analysis.

- Input the numbers
- Compute the sum of the numbers
- Compute the count of the numbers
- Calculate the average from the sum and the count (keeping in mind that the count of values could be zero)
- Output the average

Plans

There are a number of *plans* that are needed to completely solve the problem. The absence of any of the following *plans* would reduce the level of completeness. The *plans* needed to completely solve the problem are also listed in Figure 4.1.

- Sentinel-Controlled Input plan
- Sum plan
- Count plan
- Average plan
- Guarded Division plan
- Output plan



```
01 #include <stdio.h>
02
03 int main() {
04 int count = 0;
05 int sum = 0;
06 int current;
07 int average;
08
09 printf("Input number: ");
10 scanf("%i", &current);
11 while(current != 9999) {
12     sum = sum + current;
13     count = count + 1;
14     printf("Input Number: ");
15     scanf("%i", &current);
16 }
17
18 if(count > 0) {
19     average = sum / count;
20     printf("Average: %i\n", average);
21 }
22 }
```

Figure 4.1. Identified goals, associated *plans* and a potential solution

Figure 4.1 shows a flow of design starting with an identification of the goals that need to be achieved. The goals are then mapped to *plans* that can be used to achieve the goals. Each *plan* on its own can achieve little and needs to be correctly integrated into a whole solution; for this problem, it is necessary for *plans* to be abutted to form a correct sequence. Some *plans* are merged; for instance the Input, Sum and Count *plans* need to be applied to the same set of input and thus the Summing and Counting are merged with the Primed Sentinel-Controlled Input Loop plan. Some *plans* are nested, for instance the Average and Output plans which are nested inside the Guarded Division plan.

4.4 Methodology

Students were asked to solve the problem on paper (without computers) as an exam-like activity during tutorial classes, conducted in an ordinary class room. Students were not given any warning and were not expected to have prepared for the activity. Students were not allowed access to resources during the activity.

All students in the course attending classes on-campus were asked to participate. The experiment was conducted during the third last week of the course and two weeks after the module covering *Repetition*, which contained relevant *knowledge* content and exercises (including the exercise shown in 4.1.2 above).

Each student was given a piece of paper with the problem statement printed at the top and a number of lines for the student to write their solution to the problem (see Appendix B). The following statement was then read aloud to the participating students.

> *Please follow the instructions on the paper as closely as possible when writing your program. This task is not graded and will not contribute to your assessment. Completing this task is not required in order to complete the course. By completing the task you will allow us to improve the course for you and other students. You do not need to write your name on the paper. The program should take 10 to 15 minutes to complete.*

There were three tutorial classes during the week. The experiment was consistently conducted and overseen by a single instructor. All students attending these classes accepted the invitation to participate. Although an estimation of time was given in the statement read to students, no time limit was applied; this estimate was made by the instructor before the experiment was started. Specific times were not measured, however, most students required longer than estimated, taking between 15 to 20 minutes to create a solution.

Students were asked not to speak while solving the problem, unless it was to ask a question. A number of students asked questions to clarify the problem statement and these students were answered individually.

4.4.1 Goal/Plan Analysis

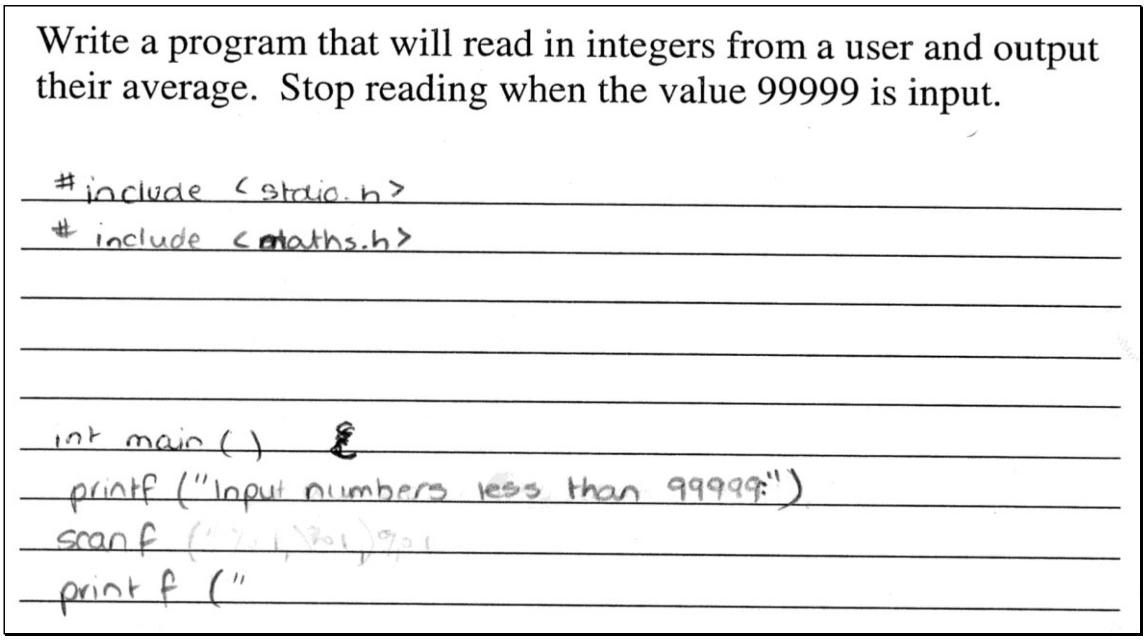
Each solution was analysed using Goal/Plan Analysis (see section 3.2.1). According to the *plans* given for a complete solution in Figure 4.1 above, each of the following plan features (and merging of the sum and count plans*) was judged to be present or absent in each student's solution.

- Initialisation of a sum variable (for Sum plan)
- Initialisation of a count variable (for Count plan)
- A Sum plan in a Primed Sentinel-Controlled Loop
- A Count plan in a Primed Sentinel-Controlled Loop
- A guard against division by zero (for Average plan)
- An Average plan
- An Output plan
- Merging of the Sum and Count plans inside the Primed Sentinel-Controlled Loop*

There are a number of acceptable variations to the example solution shown in Figure 4.1. In some languages variables are automatically initialised; solutions in this experiment were written in C where initialisation is not automatic. The average calculation and output could be combined. A different looping construct, other than **`while`**, could be used in the solution. However even with such variation, it is still possible to recognise the required *plans*.

4.4.2 Examples of Analysis

The following examples show where students have demonstrated, or failed to demonstrate, use of *plans* within their solutions.

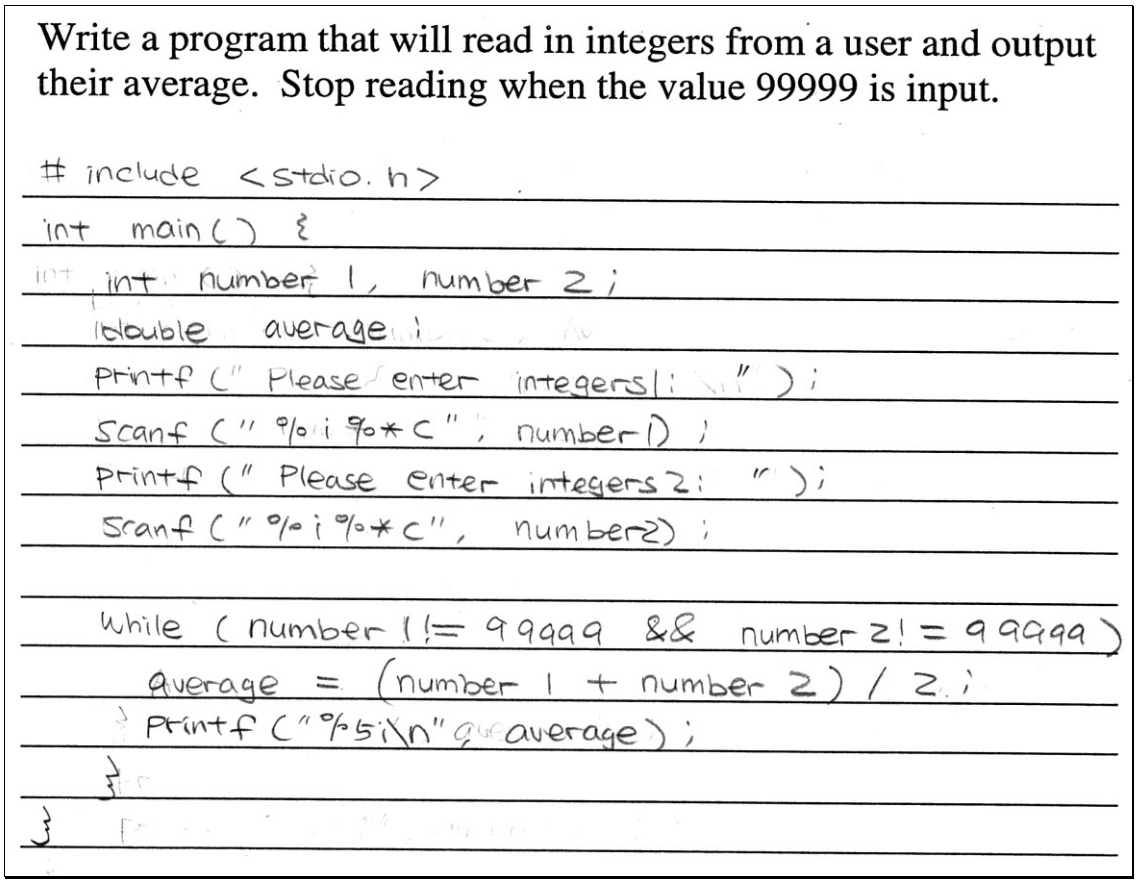


Write a program that will read in integers from a user and output their average. Stop reading when the value 99999 is input.

```
#include <stdio.h>
# include <maths.h>



int main ( )    £
  printf ("Input numbers less than 99999:")
 scanf [illegible]
 print f ("
```

Figure 4.2. A solution showing no apparent *plans*

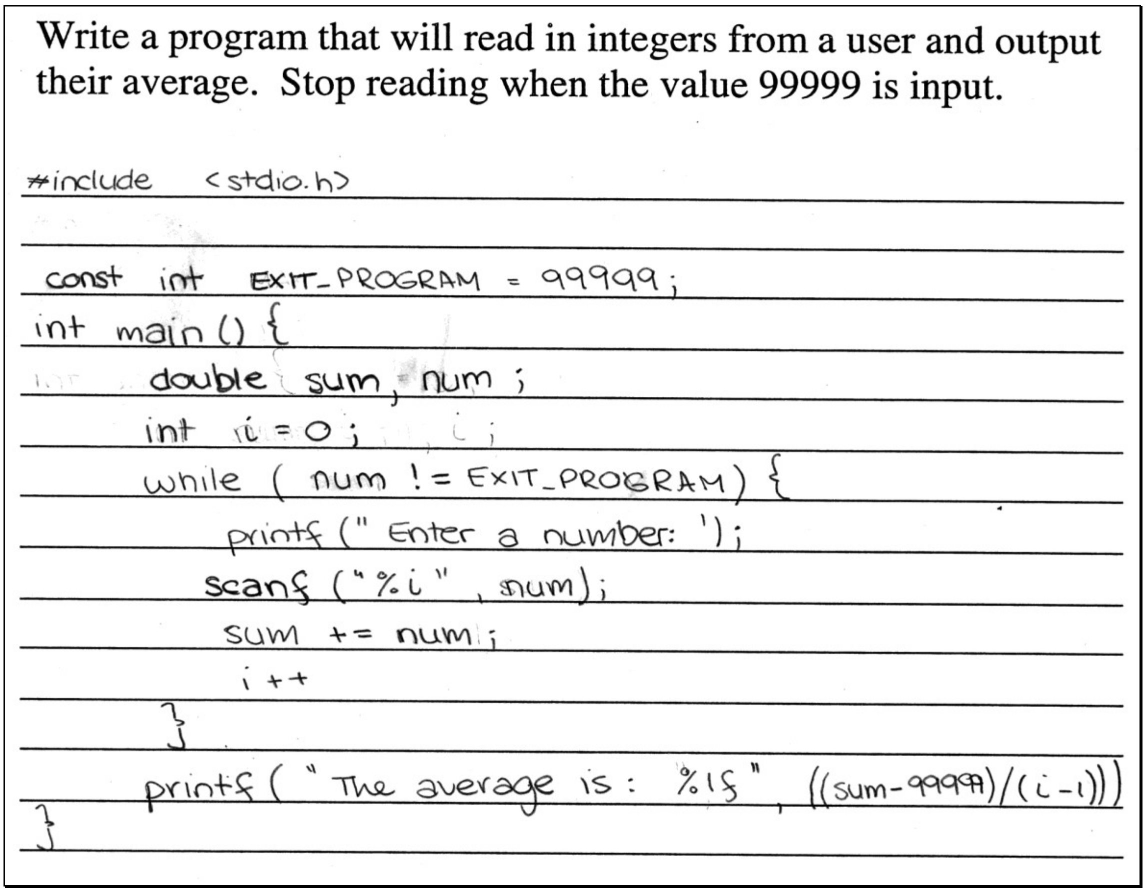
Some students handed in solutions in which no *plans* could be identified. This may have been because the student produced very little code as in Figure 4.2, or code from which *plans* could not be isolated, as in Figure 4.3 where the student appears to have misinterpreted the instructions given.



Write a program that will read in integers from a user and output their average. Stop reading when the value 99999 is input.

```
# include <stdio.h>
int main () {
  int number 1, number 2;
  double average;
  printf (" Please enter integers1: ");
  scanf (" %i %*c", number1);
  printf (" Please enter integers 2: ");
  scanf (" %i %*c", number2);

  while ( number 1!= 99999 && number 2! = 99999)
     average = (number 1 + number 2) / 2;
   printf (" %5i\n" average);
  }
}
```

Figure 4.3. A solution where *plans* are not easily identifiable



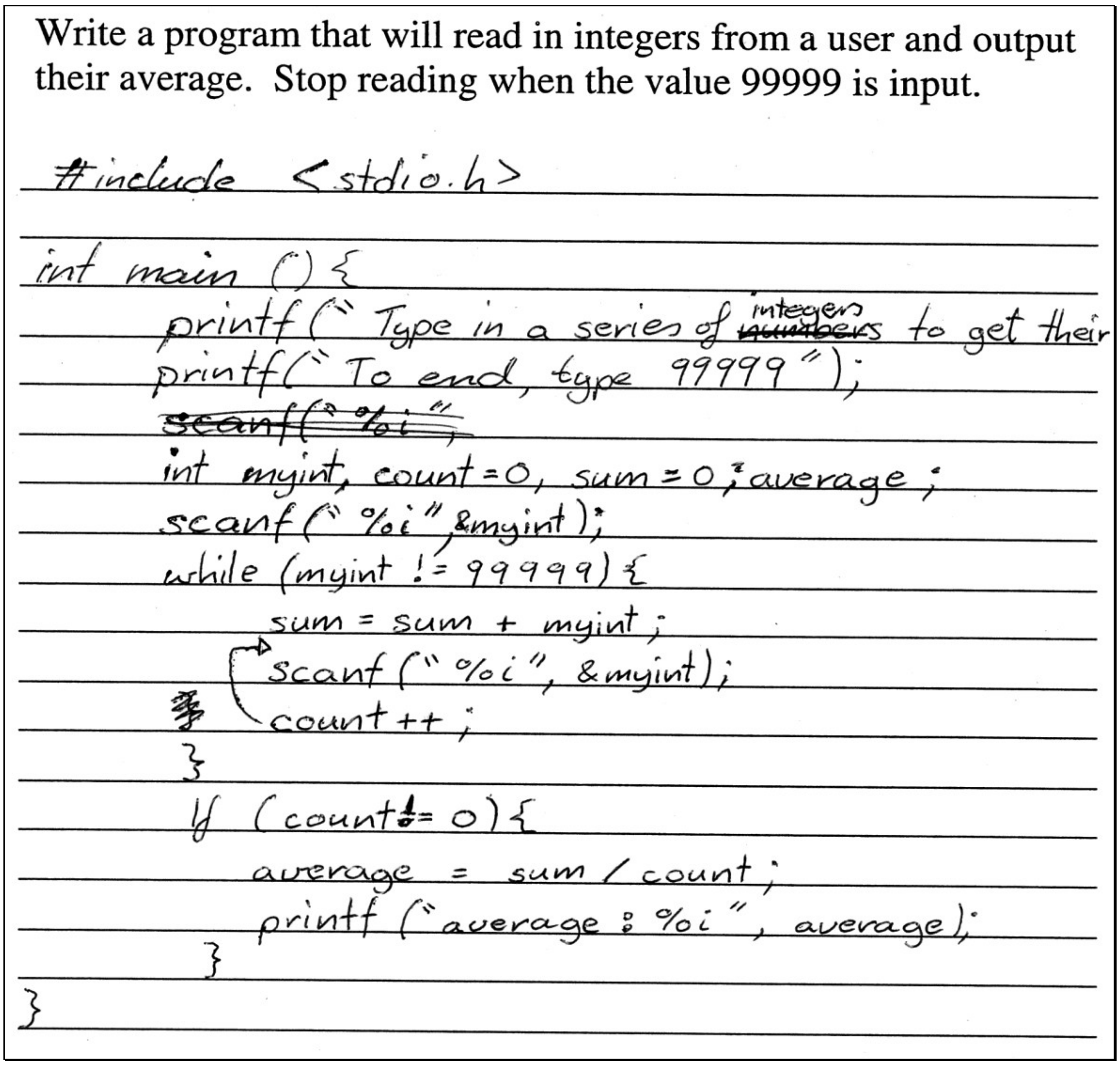
Write a program that will read in integers from a user and output their average. Stop reading when the value 99999 is input.

```
#include   <stdio.h>

const  int   EXIT_PROGRAM  =  99999;
int main () {
      double  sum, num ;
      int  i = 0 ;
      while ( num != EXIT_PROGRAM) {
            printf (" Enter a number: ');
            scanf ("%i" , num);
            sum  +=  num ;
            i ++
      }
      printf ( " The average is :  %lf " , ((sum-9999)/(i-1)))
}
```

Figure 4.4. A solution with a number of flaws

The majority of students created a solution that, on the surface, contained the main ingredients for calculating an average, but also contained a number of flaws that would prevent the program from working in some instances, or from working at all. Figure 4.4 is a typical example. The following flaws can be noted.

- The count variable **`i`** is initialised, but the **`sum`** variable is not; this would result in a solution that produces an incorrect answer in most or all executions.
- The user input is not primed; if the user enters the sentinel value at the first opportunity, the loop would still be entered. The sentinel will also be included in the sum and for this, the solution is forced to compensate when calculating the average.
- There is also a slight possibility that the uninitialised value of the user input could be equal to the sentinel, in which case the user would never be given the opportunity to provide input. While the chance of this is exceptionally small, programs used regularly by a number of users will eventually encounter such circumstances and produce an error that is difficult to identify during testing.
- The division used to calculate the average is not guarded, so if the user has entered the sentinel at the first opportunity, a division by zero would take place and the program would crash, or, worse, produce a false result.


Write a program that will read in integers from a user and output their average. Stop reading when the value 99999 is input.

```
#include <stdio.h>

int main () {
    printf(" Type in a series of ~~numbers~~ integers to get their
    printf(" To end, type 99999 ");
    ~~scanf(" %i",~~
    int myint, count=0, sum=0, average;
    scanf(" %i",&myint);
    while (myint != 99999) {
        sum = sum + myint;
        scanf(" %i", &myint);
        count++;
    }
    if (count != 0) {
        average = sum / count;
        printf ("average : %i ", average);
    }
}
```

Figure 4.5. A solution demonstrating the necessary *plans*

In Figure 4.5 we see a demonstration of the required *plans*.

- The sum and count variables are initialised.
- The loop is primed with an initial user input.
- The sum and count *plans* are within the Primed Sentinel-Controlled Loop and merged so only one set of user input is required.
- The count is tested to guard against division by zero before the average is calculated and output.

4.5 Results

Solutions of 42 participating students were analysed for the presence or absence of the *plans* and of associated *strategies* for incorporating the *plans*.

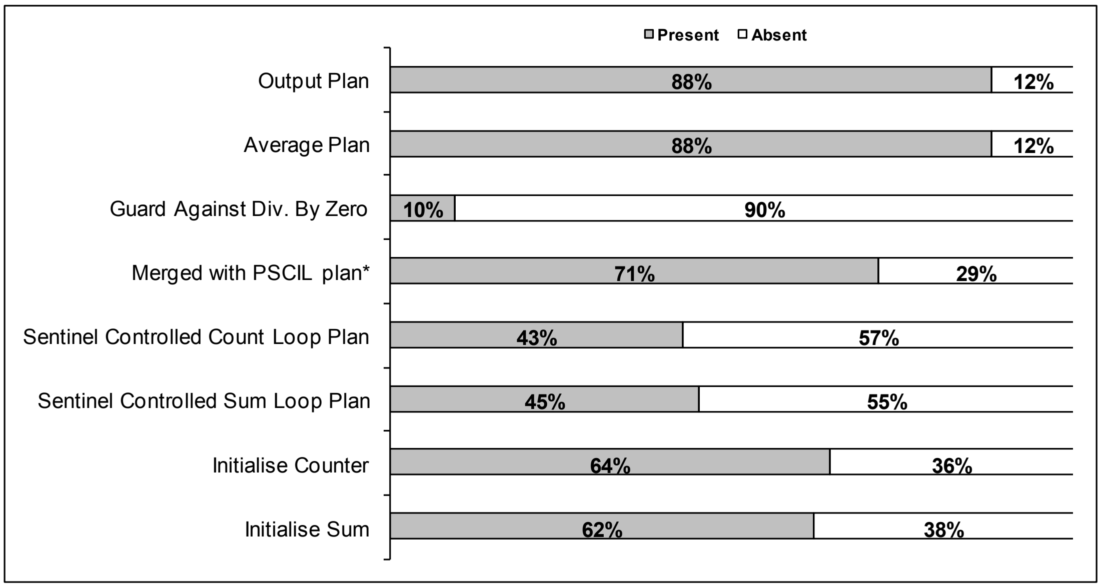


Figure 4.6. Presence or absence of *plans* and use of merging* in student's solutions

Figure 4.6 shows that some *plans* were present in almost all students' solutions while others were seldom applied.

- Most students, but not all, used division to calculate the average and output that amount.
- About a quarter of students failed to merge the summing of inputs with the counting. This is consistent with the findings of Spohrer et al. (1985b) who found students unable to merge such plans created "buggier" solutions.
- About a third of students failed to initialise the sum or count or both.
- Less than half of students produced correct primed sentinel-controlled loops for the summing or counting or both. Many students included the sentinel in their sum and count and would not handle the possibility of the first input being the sentinel.
- Less than 10% of students guarded against a division by zero.

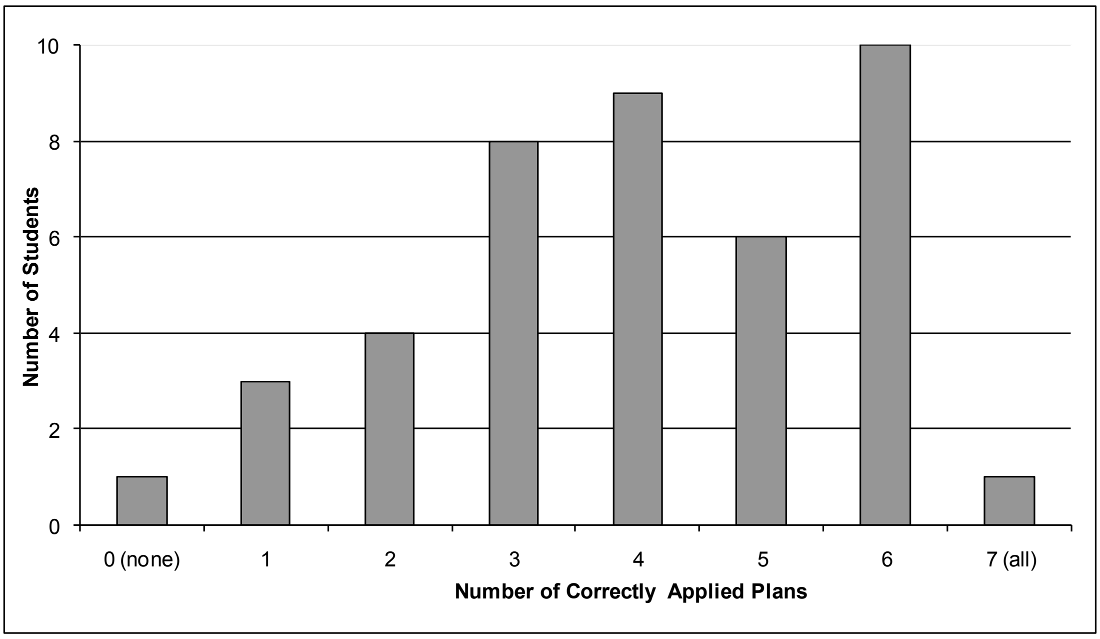


Figure 4.7. Levels of completeness as judged by number of *plans* correctly applied

There were seven *plans* needed for a correct solution (incorporation of the sum and count plans in the sentinel controlled loop is not counted here). Figure 4.7 shows the levels of completeness and the number of students reaching each level. Only one single student of 42 applied all seven expected *plans*. Most students applied between three and six *plans*.

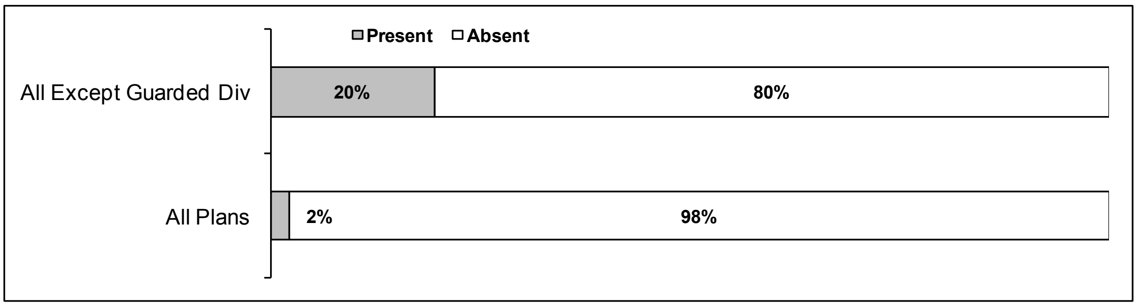


Figure 4.8. Correctness and with exception for Guarded Division plan

A complete solution to Soloway's problem would include all *plans* (see section 4.4.1). Only one student created a fully complete solution. Guarded division is clearly the least applied *plan*. If guarded division is excluded, 23% of students created an acceptable solution (as shown in Figure 4.8). During tool tests, the PROUST tool discovered bugs in 89% of 206 novice solutions (Johnson, 1986) of the "rainfall problem" (Johnson et al., 1983), which includes the averaging problem used here together with maximum and validation components. So high levels of misconceptions is consistent with previous measures of plan use.

Programming *strategies* were not an assessable part of the course. The assignments and the exam focused on *knowledge* components of the course. Applying complete correctness may be appropriate for experts, but it is not commonly applied to novices. If this problem were judged in the same way as a normal exam, with each required *plan* being worth a fraction of a total mark, then the average mark would be 4/7 *plans* or 57%. If the passing mark were 50% then 62% of participants would have received a passing mark. Over 70% of the students actually passed the course.

4.6 Discussion

The two questions that motivated this experiment (see section 4.2) are discussed in the next two sub-sections. This is followed by a number of possible flaws found with the problem during experimentation.

4.6.1 Potential of Students

RQ1. *What is the potential of students who have been exposed to an implicit-only teaching of programming strategies to solve a sub-algorithmic problem that requires application of a number of programming strategies for a complete solution?*

Students failed to demonstrate application of certain important *strategies*. Only one student was able to achieve a fully complete solution to the averaging problem. Students, on average, demonstrated 57% of the expected *plans*.

Participating students were not consistently able to:

- initialise sum and/or count variables,
- use a correct looping *strategy* for the given problem,
- guard against events such as division by zero, or
- merge *plans* that should be achieved together.

4.6.2 Weaknesses of Curriculum

RQ2. *What are the deficiencies in the curriculum that are demonstrated by students' solutions to the given problem?*

This experiment demonstrated weaknesses in the existing teaching approach. Within an implicit-only approach to programming *strategy* instruction, poor looping *strategies* may reflect the unnatural form of looping constructs in modern programming languages. According to Soloway, Bonar, and Ehrlich (1983a), programming looping structures available in programming languages do not reflect the way students envisage repetition. Such misconceptions of looping constructs are described as a poor "cognitive fit" to the looping *plans* required in this experiment. It is not feasible to change the language's looping constructs, so students must be given *strategies* for using existing constructs when solving problems.

It is likely that few students would have experienced the effect of dividing by zero as simple practical exercises can protect students from having to deal with this problem. Rather than being sheltered from encountering errors such as division by zero, novices should be explicitly taught *strategies* that deal with boundary conditions.

This experiment shows that many students had not learned certain programming *strategies* implicitly. The curriculum used, containing only implicit instruction of programming *strategies*, had not allowed those students to learn the required *plans* and demonstrate the application of these *plans*.

4.6.3 Possible Flaws in the Problem

This section raises concerns with the problem statement that arose during the experiment. These possible flaws are not seen as having affected the results observed in this experiment, but could be considered in future replications of the experiment.

- Wording of the problem is very simplistic. This could imply that a less than fully complete solution is acceptable. Checking that there is a sufficient count to calculate an average might be neglected due to the simple wording of the problem.
- Students were allowed to ask questions while completing the task and some showed confusion over whether the sentinel value should be included as an input or not, as this is only implied by the problem description. Stating that the sentinel should not be counted as an input would make the problem statement clearer, but it may also be too suggestive of the *strategies* required.
- The problem statement asks for an average to be calculated which means that numbers need to be input, counted and summed. There is no request within the wording of the problem statement that these three actions need to take place simultaneously, or that it is desirable for a user to be asked to enter the numbers once only, or that a solution that asks the user to enter the numbers twice or more is deficient. It is assumed that the novice will draw these conclusions.
- There is no mention of input validity in the problem statement. Assuming that the user enters some value, should the input be validated or not? To achieve validation requires another, more complex, *strategy*. An input validation *strategy* was not desired, yet the problem does not state that valid input can be assumed. If validation was required it would double the length and complexity of any correct solution and require twice the time to complete. No student attempted validation in solutions analysed for this experiment.

4.7 Implications

This experiment has shown that a cohort of students exposed to an implicit-only teaching of sub-algorithmic programming *strategies* do not produce solutions that consistently demonstrate the required programming *strategies*. Improvements to the curriculum may yield better outcomes for students, overcoming the detected flaws. Past research has shown explicit instruction can be more powerful than implicit instruction (Baddeley, 1997). A more explicit focus on the poorly used *strategies* from this experiment, and programming *strategies* in general, may produce improved outcomes.

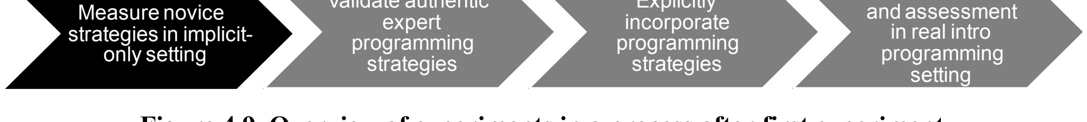

Measure novice strategies in implicit-only setting
Validate authentic expert programming strategies
Explicitly incorporate programming strategies
Test instruction and assessment in real intro programming setting

Figure 4.9. Overview of experiments in a process after first experiment

The next stage of this study attempts to capture a relevant set of programming *strategies* that can then be incorporated explicitly into an introductory programming curriculum. According to Soloway and his colleagues, the source of *goals* and *plans* are experts themselves who have developed a tacit set of canned solutions (Soloway, 1986). In his description of the PROUST system, Johnson (1986) gives a catalogue of *plans*. Chapter 5 describes a comparison of an adapted set of *plans* from this catalogue with the *strategies* demonstrated by experts. This study seeks to confirm that *plans* are an authentic representation of expert *strategies*.

With a set of authentic, concrete, *strategies* it may be possible to explicitly incorporate programming *strategies* into an introductory programming curriculum. The feasibility of such instruction needs to be tested and the impact on students needs to be measured. Chapter 6 describes an experimental curriculum that was tested with students in an artificial setting.

If programming *strategies* can be explicitly incorporated into introductory programming curricula they can possibly be assessed. Methods of assessing programming *strategy* skill may yield a better measurement of the outcomes of students in an introductory programming course than traditional methods. A study of instruction and assessment of programming *strategies* in an actual introductory programming setting is described in chapter 7. Student performance after explicit *strategy* instruction is compared with the results shown in this chapter.

5. Experts and Explicit Strategies

"An expert is a man who has made all the mistakes which can be made, in a narrow field."
Niels Bohr

Overview

A previous experiment (chapter 4) showed a number of common programming *strategy* flaws in novices' solutions to a simple averaging problem. The curriculum used to instruct the novices required the novices to learn programming *strategies* in an implicit way. Including explicit instruction of programming *strategies* in the curriculum might improve outcomes for students. To achieve this, a set of concrete, authentic sub-algorithmic *strategies* was sought. Biederman and Shiffrar (1987) used interviews with an expert chick sexer to gather descriptions that could be presented in an explicit form (see section 2.3.6). Expert programmers are a source of programming *strategies* that could be taught explicitly to novices.

An experiment was conducted with 25 experts, who were asked to solve three well defined problems. *Plans* identified in expert solutions were compared with *plans* used by Soloway and his colleagues (Soloway, 1986, Johnson and Soloway, 1984) through Goal/Plan Analysis. Results showed *plans* appear in solutions created by experts, thus validating Soloway's *plans* as a model of expert programming *strategies*. These *strategies* could be explicitly included in introductory programming curricula to overcome previously identified weaknesses.

5.1 Introduction

An initial experiment (described in chapter 4) showed weaknesses in novices' programming *strategy* skills, exposed by the flaws in solutions to a set programming task. The novices had been instructed using a curriculum that required learning of programming *strategies* implicitly.

The presence of these flaws indicated possible weaknesses in the curriculum used to instruct the novices in programming *strategies*. Expecting students to learn *strategies* implicitly resulted in students acquiring an incomplete set of programming *strategies* and a poor understanding of how to integrate them. This was consistent with results presented by Reber (1993), who examined implicit learning in the context of language acquisition. Reber found that experimental subjects could learn through implicit-only means but when interviewed and asked to describe the underlying system they had been exposed to, they were unable to express any understanding. Rist (1991) observed novices in an implicit-only setting as they attempted to construct their own *strategies* to solve problems and noted that many of the novices succeed. Yet, later research (Whalley et al., 2006) has shown that only 30% of novices at the end of an introductory programming course could comprehend and describe the *strategies* applied in a given piece of code; the remaining students described code line-by-line, relying on programming *knowledge*.

An experiment conducted by Biederman and Shiffrar (1987) showed that taking the tacit understanding of expert chick sexers and providing an explicit representation of this to novice chick sexers greatly improved learning outcomes. This experiment showed that minutes of exposure to explicit instruction can be more effective than weeks of implicit training (Baddeley, 1997).

While chick sexing is far removed from the task of programming, and learning an artificial grammar is trivial by comparison, the implicit-only approach to instruction is similar to the way programming *strategies* are traditionally taught in an introductory programming course. Students are given the basic *knowledge* of a programming language and are then expected to develop programming *strategies* and an understanding of the problem solving processes implicitly through practical exercises.

To overcome programming *strategy* weaknesses discovered in novices, taking programming *strategies* from being instructed in an implicit-only manner and adding explicit instruction may improve outcomes.

Biederman and Shiffrar interviewed a single expert chick sexer and from this were able to create explicit instruction for novice chick sexers. Chick sexing requires only a small number of *strategies* for the identification of chick gender. By comparison, problem solving in programming is a more diverse task. Sub-algorithmic programming *strategies* are at the level most relevant to novice programmers in an introductory course (see section 2.3.3). Even at this simple level there are many *strategies*. It is therefore unlikely that an interview with a single expert programmer would yield all programming *strategies* relevant to novices.

Two well developed forms of programming *strategy* that could be seen as models of expert *strategies* are *plans* and *patterns* (see section 2.4 for more details on representations of *strategies*). *Patterns* can be applied to a range of problems from sub-algorithmic through algorithmic to system-level problems, but tend to be used to define solutions at the algorithmic level and above. *Patterns* are commonly used in the object-oriented paradigm. While the OO paradigm is widely used in industry, and many introductory programming courses adopt an OO language, the majority of Australasian introductory programming courses still introduce programming using the procedural paradigm (see section 2.2.3). *Plans* can be used to describe solutions at sub-algorithmic and algorithmic levels. *Plans* may not be the most suitable form for describing system-level problem solutions. It is the sub-algorithmic level that is most relevant to novice programmers in their first exposure to programming (however, programmers must use sub-algorithmic *strategies* at all stages of expertise). *Plans* can be described without a great deal of extraneous information. No system is needed to describe *plans* or how they are applied. Most *plans* are simply labels for parts of a concise solution along with a description of how these parts are integrated. *Plans* can be applied in almost any paradigm. For this reason, *plans* have been chosen as the form of *strategy* used in the experiments in this and later chapters. However, in stating this, it is likely that these experiments could be replicated using *patterns* or some other representation of *strategies*.

Soloway and his colleagues state that *plans* were identified based on the tacit expertise of programmers (Soloway et al., 1982, p. 52).

> *We, as expert programmers, have constructed a set of such plans, which cover the type of simple looping problems used in introductory programming courses.*

No report of the sourcing of these *strategies* is published so it is assumed that these *strategies* were the product of discussion among academics who were instructors of novice programmers. This experiment sets out to test if *plans* are used by expert programmers. If this model is consistent then *plans* can be viewed as an appropriate form of *strategies* to be instructed explicitly in a curriculum.

In programming related work, Sajaniemi and Prieto (2005) used a card sorting exercise and interviews to validate their model of the roles of variables with experts. In this study professional programmers were asked to organise variables used in set programs into groups. The variables covered six of the previously identified roles (Sajaniemi, 2002) and experts' groupings matched these roles with 85% accuracy.

5.1.1 Participants

As discussed in section 2.3.1 there are several levels of expertise between a novice and an expert programmer. In this study participants were qualified as experts if they were generating publicly used code on a regular or daily basis. The participants were 14 paid professional programmers and 11 academics who were instructors of programming courses.

5.1.2 Setting

The experiment was conducted in the place of work of the participants involved. Where participants were professional programmers the experiment was conducted in a meeting room, or similar, within their normal work building. Where participants were university academics the experiment was conducted in their office away from a computer.

5.2 Research Questions

This experiment was motivated by two related questions which are answered in section 5.6.

RQ3. *Do experts exhibit identifiable plans in their solutions to problems?*

RQ4. *Can an authentic set of strategies, used by experts, be represented in an explicit form, suitable for instruction?*

If the answer to these two questions is positive then it is possible to consider the following issue which is discussed in section 5.7.

RQ5. *Does the potential to represent authentic programming strategies mandate explicit instruction of programming strategies to novices?*

5.3 The Problems

The aim of this experiment was to explore sub-algorithmic *strategies* in solutions created by experts that are relevant to novice programmers. Although the problems used here are small in scale, the *strategies* being elicited could be used by experts on a regular basis within solutions to larger problems. Three problems were chosen that a novice would be expected to solve at the end of an initial semester of programming. For each, the problem statement, identifiable goals and expected *plans* are shown. The problems increase slightly in complexity from Problem 1 to Problem

3. The problems are sufficiently generic to permit solutions in a broad range of languages.

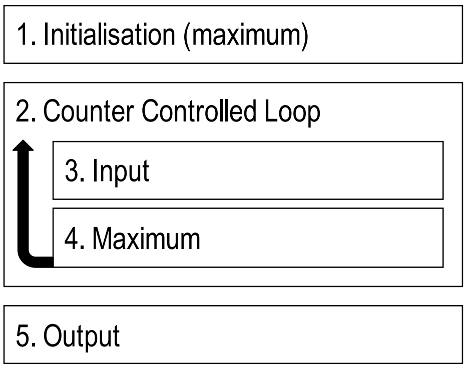
5.3.1 Problem 1

Read in 10 positive integers from a user. Assume the user will enter valid positive integers only. Determine the maximum.

Goals

- Input 10 numbers
- Determine maximum
- Output maximum

Plans



5.3.2 Problem 2

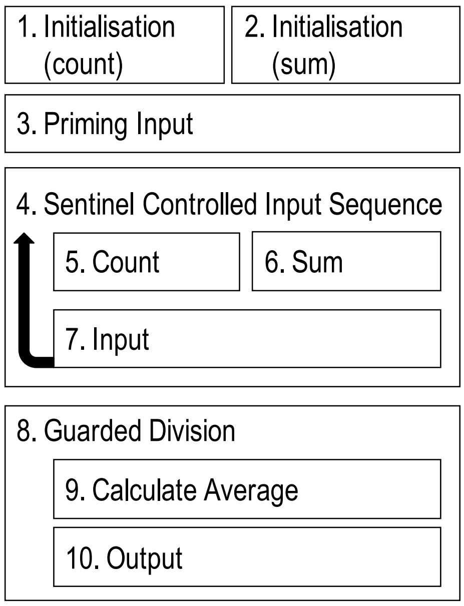
This problem is similar to that presented to novices in the previous experiment (see section 4.3). The problem was refined to eliminate the need for validation, which would otherwise double the complexity of the problem.

Read in any number of integers until the value 99999 is encountered. Assume the user will enter valid integers only. Output the average.

Goals

- Input the numbers (initial and subsequent inputs)
- Compute the sum of the numbers
- Compute the count of the numbers
- Calculated the average from the sum and the count (keeping in mind that the count of values could be zero)
- Output the average

Plans



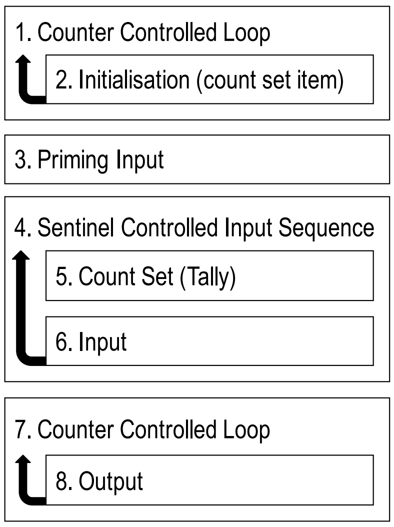
5.3.3 Problem 3

Input any number of integers between 0 and 9. Assume the user will enter valid integers only. Stop when a value outside this range is encountered. After input is concluded, output the occurrence of each of the values 0 to 9.

Goals

- Input numbers (initial and subsequent inputs)
- Count set (tally each number)
- Output set tallies

Plans


1. Counter Controlled Loop
2. Initialisation (count set item)
3. Priming Input
4. Sentinel Controlled Input Sequence
5. Count Set (Tally)
6. Input
7. Counter Controlled Loop
8. Output

5.4 Methodology

As with Biederman and Shiffrar's (1987) chick sexing experiment (see section 2.3.6), participants were asked to solve problems on paper, away from a computer (solution sheets are shown in Appendix C). The focus of analysis of solutions was not on the syntactical correctness of the solutions, but on which *strategies* experts used to solve the problems. Using paper was a means of enforcing this focus.

The three problems were presented on a single sheet of paper each, with the problem statement at the top and lines for answering the problem below this.

Participants were timed to see how long they took to solve each problem. Participants were asked not to rush. Where two or more programmers were participating simultaneously, participants were asked not to collaborate, or race to complete the problem.

A card sorting exercise and interviews have been used as a knowledge elicitation exercise to validate the roles of variables with experts (Sajaniemi and Prieto, 2005). Here, the knowledge elicitation technique used is to apply Goal/Plan Analysis to experts' solutions to set programming problems.

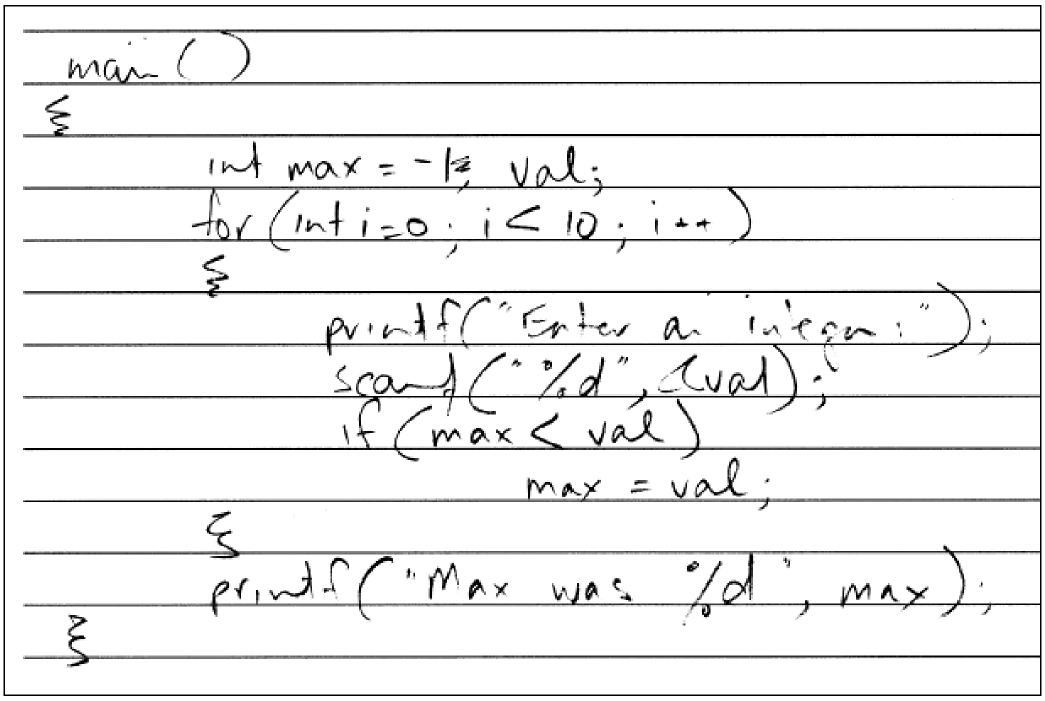
5.4.1 Goal/Plan Analysis

Results were analysed by checking for:

- the presence of each of the relevant *plans* (shown above),
- nesting and merging in appropriate locations, and
- an overall correctness measure of abutment.

Analysis of Solutions to Problem 1

In most cases the presence of a *plan* is easily determined. For instance when searching for a maximum plan, two features are sought: first, an initialisation of a maximum variable; second, a test comparing the current maximum with a new candidate and an assignment if appropriate. With counter-controlled loops, only loops including a test of an incrementing counter variable were accepted.



```
main ( )
{
        int max = -1, val;
        for (int i=0; i< 10; i++)
        {
                printf("Enter an integer:");
                scanf("%d", &val);
                if (max < val)
                        max = val;
        }
        printf("Max was %d", max);
}
```

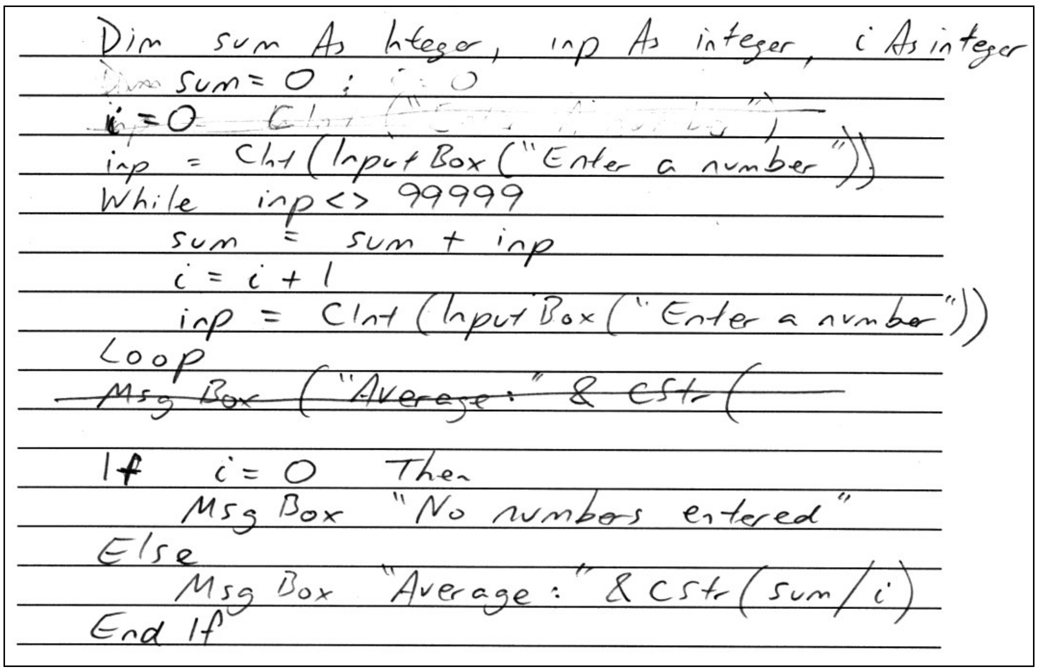
Figure 5.1. A participant's solution to Problem 1

In Figure 5.1 a solution to Problem 1 created by a participant is presented. When performing Goal/Plan analysis on this solution the following features were identified.

- The maximum is initialised; the first input will become the new maximum.
- There is a counter-controlled loop; the `for` loop using the counter `i` will repeat 10 times regardless of user input.
- The user is able to enter input.
- Each input is compared with the current maximum and retained if greater.
- The maximum is output at the end of the program.
- The input and maximum plans are nested inside the counter-controlled loop.

Analysis of Solutions to Problem 2

Sentinel-Controlled Loops were considered to be present only if there was a priming input and the looping construct tested if the first or most recent input could be the sentinel before using it as an input. In this way the sentinel will not be included as an input for summing and counting purposes. A Guarded Division requires a selection construct that will prevent division if the divisor is zero.

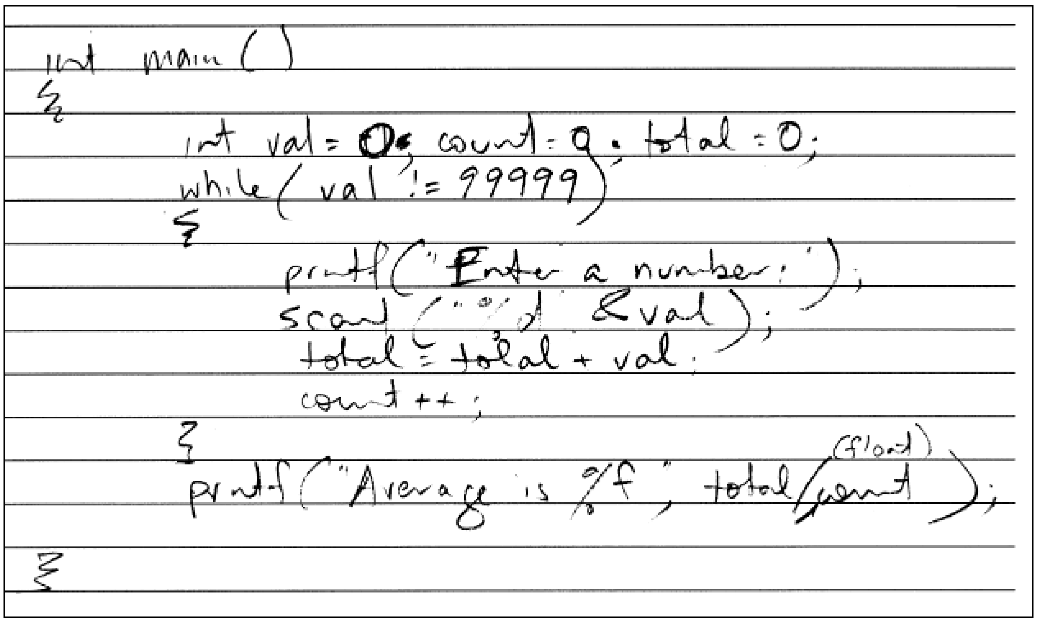


```
Dim sum As Integer, inp As integer, i As integer
Sum = 0 : i = 0
i = 0 ~~CInt(~~
inp = CInt(InputBox("Enter a number"))
While inp <> 99999
        sum = sum + inp
        i = i + 1
        inp = CInt(InputBox("Enter a number"))
Loop
~~MsgBox("Average:" & CStr(~~

If i = 0 Then
        MsgBox "No numbers entered"
Else
        MsgBox "Average:" & CStr(sum/i)
End If
```

Figure 5.2. An acceptable solution to Problem 2

Figure 5.2 shows a participant's solution to Problem 2. From this solution the following features can be identified.

- The sum and count (`i`) are initialised.
- The loop is a Sentinel-Controlled Loop as the test is primed by an initial input and the sentinel will not be included in the sum or count.
- If the count of inputs is zero, the calculation of the average (including a division by the count) will not be performed.



```
int main ()
{
        int val = 0, count = 0, total = 0;
        while ( val != 99999)
        {
                printf("Enter a number: ");
                scanf("%d", &val);
                total = total + val;
                count ++;
        }
        printf("Average is %f", total/(float)count);
}
```

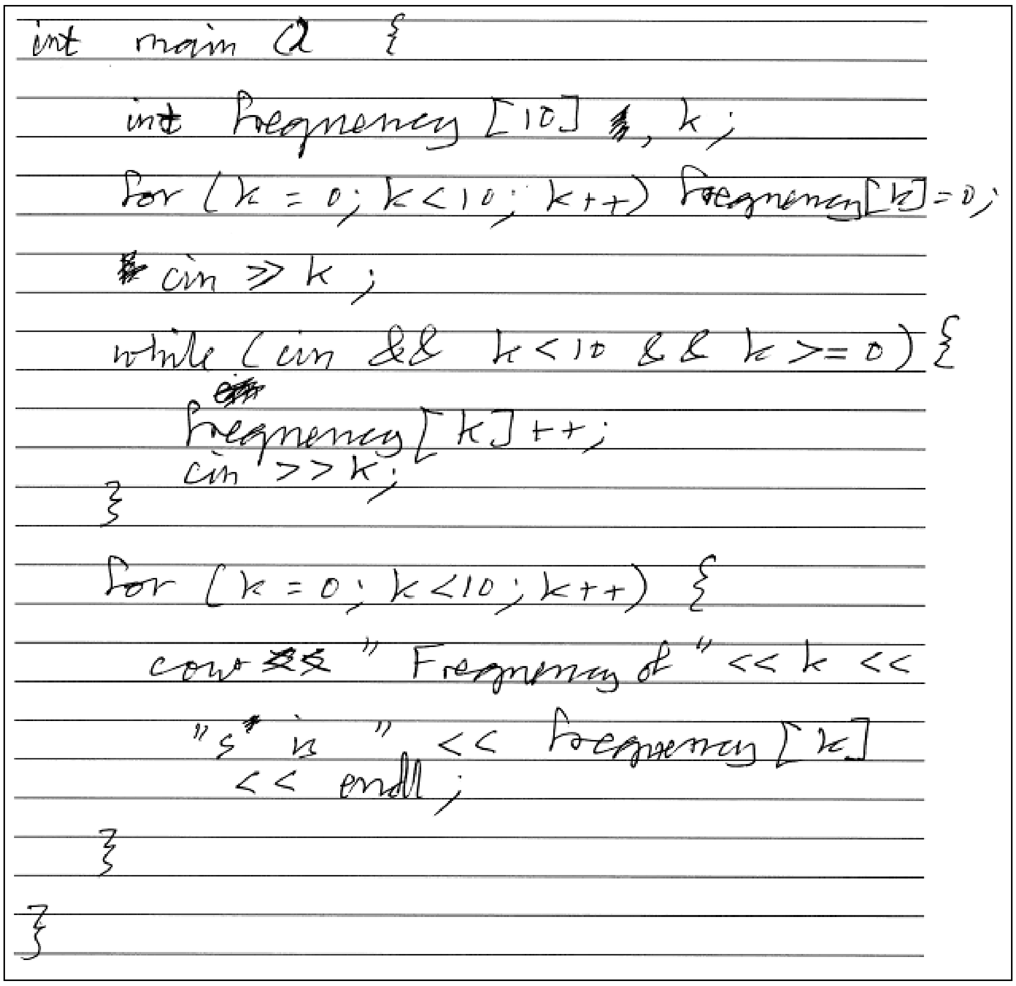
Figure 5.3. A poor solution to Problem 2

Figure 5.3 shows another participant's solution to Problem 2. This solution fails to demonstrate a number of elements that were being identified.

- The loop is not a proper Sentinel-Controlled Loop; the input value used in testing is initialised but it is not primed with user input (which could be the sentinel in the first instance). The sum and count will include the sentinel value.
- The division operation used to calculate the average is not guarded.

Analysis of Solutions to Problem 3

For a Set-Counting plan, only methods of classifying and counting inputs, as opposed to capturing and keeping the user's input, were accepted. In some languages initialisation of variables and arrays is done automatically; where this was the case, participants were seen as having fulfilled the initialisation components of the *plans*.



```
int main () {
    int frequency [10] , k;
    for (k = 0; k<10; k++) frequency[k]=0;
    cin >> k;
    while (cin && k<10 && k>=0) {
        frequency [k]++;
        cin >> k;
    }
    for (k = 0; k<10; k++) {
        cout ~~<<~~ " Frequency of " << k <<
          "s is " << frequency [k]
          << endl;
    }
}
```

Figure 5.4. A participant's solution to Problem 3

In Figure 5.4 a participant's solution to Problem 3 is shown. From this solution the following features can be identified.

- A count of numbers 0 to 9 is being kept in an array. The array elements are initialised to zero using a Counter-Controlled Loop at the start of the program.
- The second loop is a Sentinel-Controlled Loop as the test is primed by an initial input and the sentinel (any value outside the range 0 to 9) will not be counted.
- Each input is tallied using the input as an index into the array.
- The counts are output using a Counter-Controlled Loop at the end.

5.5 Results

Two measures were recorded: the times taken by each participant to complete each problem and the presence or absence of the expected strategies in each solution.

In one instance a participant used an event-driven paradigm to solve the problems. Goal/Plan Analysis could not be applied in this case as many of the underlying constructs, such as loops, were not used. This demonstrates that Goal/Plan Analysis is not applicable to all paradigms. The solutions of this participant were not used in the analysis. In three instances participants created a solution to some different problem. In these cases it was clear the experts had misread the instructions rather than being unable to solve the problem. These solutions were disqualified from analysis, but the remaining solutions from these participants were used. This occurred with one professional in Problem 1 and two professionals in Problem 2.

Table 5.1. Average times for problems by expert type in minutes and seconds (and number of each type)

	Prob. 1	Prob. 2	Prob. 3	Overall
Academics	4:50 (11)	4:52 (11)	6:17 (11)	15:58
Professionals	5:33 (12)	5:17 (13)	6:16 (11)	17:06
All	5:13	5:05	6:16	16:34

Table 5.1 shows times taken by participants to solve the problems The number of solutions included for each problem and each expert type is shown in parentheses. The average time to complete all three problems was 16min 34sec. To give a comparison, the initial study described chapter 4 showed novices at the end of a semester's instruction taking 15 to 20 minutes to solve the equivalent of Problem 2 alone. There was a difference in times between participants who were academics and those who were professional programmers, although it was not proven to be significant in a t-test of this sample (t≈0.47, p≈0.32, df=22). The six fastest times were contributed by academics. This may have been due to the simple nature of the problems, which would be familiar to academics but less so to professionals.

Table 5.2. Average plan use problems by expert type

	Prob. 1	Prob. 2	Prob. 3	Overall
Academics	97.7% (11)	82.6% (11)	90.9% (11)	89.7%
Professionals	99.0% (12)	83.9% (13)	96.7% (11)	92.6%
All	98.4%	83.3%	93.8%	91.8%

There was a slight difference in the presence of *plans* between academics and professionals as shown in Table 5.2 (3.1% difference in overall *plan* use) but again, this was not significant in a t-test (t≈0.67, p≈0.35, df=58).

All experts achieved correct abutment (the correct ordering of *plans*) for all problems. For example, no expert placed the output of a maximum before the calculation of the maximum.

Table 5.3. Presence of *plans* for Problem 1

Plan	Presence
Max Initialised	100%
Counter-Controlled Loop	100%
Input Plan	100%
Maximum Plan	100%
Output Plan	87%
Input Nested in Counter-Controlled Loop	100%
Max Plan Nested in Counter-Controlled Loop	100%

Solutions to Problem 1 showed almost universal conformity to the set *plans* (Table 5.3). 23 solutions were analysed. Three participants failed to include output and this may be due to the wording of the problem, which asked for the maximum to be determined but did not specifically ask for an output.

Table 5.4. Presence of *plans* for Problem 2

Plan	Presence
Sum Initialised	92%
Count Initialised	100%
Sentinel-Controlled Input	92%
Sentinel-Controlled Count	92%
Sentinel-Controlled Sum	92%
Guarded Division	33%
Output Plan	92%
Loop Plans Merged	100%
Inputs Nested in Sentinel-Controlled Loop	92%
Output Nested in Guarded Division	33%

Problem 2 showed most participants conforming to the expected *plans* (Table 5.4). 24 solutions were analysed. In some cases individual participants failed to show one *plan*. Where a person failed to show a Sentinel-Controlled Loop, the looping *plans* merged with this loop were considered as not being present, even though they may have attempted to capture a count or sum. One obvious deficiency is shown by the lack of use of Guarded Division. Only one third of participants' solutions contained a Guarded Division plan.

Table 5.5. Presence of *plans* for Problem 3

Plan	Presence
Counter-Controlled Loop (for Initialisation)	91%
Array Initialisation	100%
Sentinel-Controlled Input	86%
Count Set Plan	95%
Counter-Controlled Loop (for Output)	86%
Output Plan	100%
Initialisation nested in Counter-Controlled Loop	91%
Inputs nested in Sentinel-Controlled Loop	91%
Count Set nested in Sentinel-Controlled Loop	86%
Output Nested in Counter-Controlled Loop	95%

Problem 3 showed most participants conforming to the expected *plans* (Table 5.5). 22 solutions were analysed. This problem encouraged the greatest variation in solutions; difference were found in how the data was stored (an array was expected, but some participants used variables), initialisation of the data store (where an array was used, a counter-controlled loop containing element initialisations was expected, but some participants used set notation to initialise the array), and set-counting (the user's input could have been used as index to the array, but some participants used a 'switch'-like construct to increment counts). These variations were allowed where the expected *plans* were still found to be present.

5.6 Discussion

The two research questions posed earlier are answered by the results of this experiment and addressed next. This is followed by discussion of a number of possible flaws found with the problems during experimentation.

5.6.1 Identifiable Strategies

RQ3. *Do experts exhibit identifiable plans in their solutions to problems?*

The results show that, in most instances, experts produce solutions including constructs that could be identified as *plans*. These findings are constrained to the *plans* covered here, but may be consistent with other *plans* relevant to novices in their initial study of programming.

5.6.2 Strategies Made Explicit

RQ4. *Can an authentic set of strategies, used by experts, be represented in an explicit form, suitable for instruction?*

The results indicate that *plans*, when taken as a model of *strategies*, are a valid description of programming *strategies*. Within the scope of this experiment, *strategies* have been described that are relevant to novice programmers while being consistent with those used by expert programmers.

The *strategies* have been expressed in a form that can be explicitly incorporated into an introductory programming curriculum. These *strategies* can be described visually and textually (see figures above and Appendix A). Specific examples of the application of these *strategies* can be given in almost any target programming language.

By identifying *strategies* by name and giving terms to the way they are integrated, a vocabulary can be established that can be used to communicate these *strategies* between instructor and student, among students and among instructors.

Students can be set problems that focus explicitly on these *strategies*, which may help novices to develop programming *strategy* skills at a sub-algorithmic level.

By identifying such *strategies* it is possible to make programming *strategy* skill an assessable component of an introductory programming course. Using Goal/Plan Analysis, a solution can now be regarded as correct because it shows application of expected *strategies*.

5.6.3 Possible Flaws in the Problems

In Problem 1, participants are asked to “determine” but not output the maximum, which was part of an anticipated solution. Some participants did not output the maximum and it is difficult to determine if this was because of the wording of the problem or because they simply neglected to do so.

The poor showing of Guarded Division may have been a product of simplistic problem statements. Participants may also have been affected by being out of their normal programming environment and away from the tools they would use for testing such boundary conditions. After participants had completed the three problems, the expected *plans* were discussed. At this stage the participants' solutions had not been analysed; however, in some cases participants admitted neglecting to include a Guarded Division and saw that it was required. This might be contrasted to a novice who might not apply a Guarded Division plan because they are unable to, or unaware that they need to.

5.7 Implications

In light of these positive findings the following issue is discussed.

RQ5. *Does the potential to represent authentic programming strategies mandate explicit instruction of programming strategies to novices?*

The results of this experiment are a strong indication that programming *strategies* applied by experts can be described and made explicit. With such a model of the *strategies*, it should be possible to create a curriculum that explicitly involves teaching of programming *strategies*.

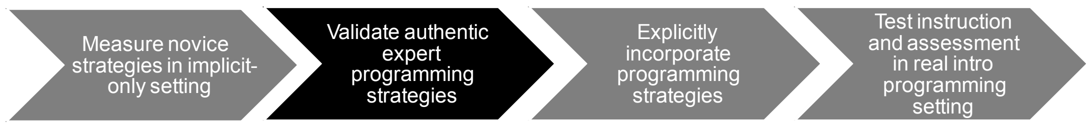

Measure novice strategies in implicit-only setting
Validate authentic expert programming strategies
Explicitly incorporate programming strategies
Test instruction and assessment in real intro programming setting

Figure 5.5. Overview of experiments in a process after second experiment

At this point it would be unjustified to claim that explicit inclusion of programming *strategies* would definitively improve outcomes in student learning, but it would be remiss not to attempt to incorporate such *strategies* into curricula and to examine the effects.

Biederman and Shiffrar's chick sexing experiment (1987) showed that by taking the tacit understanding of experts, making it explicit and using it in instruction, outcomes for students can be improved. Although the setting and target of the chick sexing study differs from programming, the use of the explicit *strategies* can be put to the same use and may lead to similar improvements for novice programmers.

When compared to implicit-only instruction of *strategies*, the potential benefits of explicit *strategy* instruction could be:

- faster learning of *strategies*,
- better performance by novices in solving problems,
- a better understanding of the underlying processes involved in solving a problem at the sub-algorithmic level,
- potential to identify and assess students' programming *strategy* skills, and
- providing instructors and novices with a vocabulary for discussing and learning *strategies*.

The programming *strategies* identified in this experiment should not be seen as independent of programming *knowledge*. It is clear that these *strategies* are built upon knowledge of basic programming constructs, data storage mechanisms and language specific facilities. As such there are dependencies between these *strategies* and the *knowledge* that underlies them. Any curriculum author who incorporates these *strategies* explicitly would need to carefully consider the order in which *knowledge* and *strategy* components are presented.

In the next stage of experimentation described in this dissertation a new curriculum, including explicit instruction of programming *strategies*, is tested in an isolated setting with a small cohort of students. This experiment attempts to measure the impact of explicit instruction of programming *strategies* on novice outcomes. After this, explicit instruction can be incorporated in an actual curriculum and student performance can be compared to the baseline set in the initial experiment.

6. Incorporating Strategies Explicitly into an Artificial Curriculum

"The principle goal of education is to create men who are capable of doing new things, not simply of repeating what other generations have done – men who are creative, inventive and discoverers."
Jean Piaget

Overview

In an initial experiment (chapter 4) a number of programming *strategy* flaws were detected in solutions created by novices. The novices had studied a curriculum that required them to learn programming *strategies* implicitly.

Following the idea that explicit instruction can lead to improved student outcomes (Baddeley, 1997), a model of programming *strategies* that can be explicitly incorporated into an introductory programming curriculum was sought. *Plans* proposed by Soloway (1986) are a model of programming *strategies*. An experiment was conducted to validate that *plans* are an authentic representation of *strategies* used by experts (chapter 5). This experiment showed that *plans* are used by experts. *Plans* are a form of *strategies* that could be explicitly incorporated into a curriculum for novice programmers.

The experiment described in this chapter was conducted to test a curriculum that included programming *strategies* explicitly in lectures, written course materials, exercises and assessment. A control curriculum was also established to allow for comparison and isolation of effects. The two curricula were delivered to two groups of volunteer students who had no previous programming experience. The experimental group showed understanding and application of programming *strategies* in *generation* tests (though not statistically significant), and in interviews used *plan* terminology and showed greater confidence in their solutions to problems.

6.1 Introduction

A previous study (chapter 4) discovered weaknesses in an implicit-only curriculum used in teaching an introductory programming course to novices. Students who participated in the study were asked to create a solution to a simple averaging problem. A number of common flaws were detected when students' solutions were scrutinised under Goal/Plan Analysis.

Participating students were not consistently able to:

- initialise sum and/or count variables,
- use a correct looping *strategy* for the given problem,
- guard against events such as division by zero, or
- merge *plans* that should be achieved together.

Students, on average, were only able to demonstrate application of 57% of the *strategies* required for a complete solution. These flaws implied weaknesses in the curriculum being delivered to the students at the time

Traditional curricula rely on novices acquiring programming *strategies* implicitly. Past studies (Baddeley, 1997, Biederman and Shiffrar, 1987, Reber, 1993) have shown that explicit instruction can be more powerful than implicit-only instruction, so it was proposed that programming *strategies* be taught explicitly. A model for describing programming *strategies* explicitly in an introductory programming curriculum was sought. A second study (chapter 5) uncovered a model of expert programming *strategies* at a sub-algorithmic level. These *strategies* can be explicitly expressed and could be incorporated into introductory programming curricula. This finding prompted the inclusion of explicit programming *strategies* in teaching as it may:

- improve outcomes for students,
- establish a vocabulary for programming *strategy* dissemination, and
- allow students' programming *strategy* skills to be assessed.

The experiment described in this chapter was conducted to test if explicit programming *strategy* instruction can be incorporated into a programming curriculum, and, if this is possible, what effects can be observed. Two curricula were designed to allow comparison and isolation of effects. An experimental curriculum included explicit instruction of programming *strategies* while a control curriculum excluded such instruction. These curricula were delivered over two separate weekend periods, followed by a series of one-on-one interviews with participants.

6.1.1 Participants

Participants were volunteers from the student body at the University of Southern Queensland who were recruited by two means:

- posters hung around the university campus, and
- emails sent to former students of two computing concepts courses for non-computing students.

Participants were asked to undertake an initial survey which gathered demographic data, computing experience, past programming experience and a measure of computing confidence.

This initial data was used to filter students who had previous programming experience. Students with *no* previous programming experience were sought in order to set a common entry point for all participants. Volunteers with previous programming experience were asked to withdraw.

A number of the volunteers withdrew from the weekend courses, mostly due to personal reasons, giving notice before the start of the experiment. Some volunteers unexpectedly failed to attend the course which reduced the group of volunteers to eight, in two groups of four, divided on a self-selecting basis. One of the participants who attended the first weekend had completed a previous course in computer programming and arrived after being asked not to attend. Results were collected from this participant but were not aggregated with results of other participants in this experiment.

6.1.2 Setting

The two weekend courses were conducted in a computing lab. This room included facilities for lecturing, computers for students to undertake practical exercises and desk space between computers for students to complete paper-based exercises.

The two curricula were delivered on consecutive weekends. The curriculum without explicit programming content, the control curriculum, was delivered first and this was followed the next weekend by the curriculum with explicit programming *strategies*. The ordering of the two curricula was arbitrary.

The weekends were divided into sessions with each session covering one to four modules of the course (see the schedule in section 6.4.2). Each session consisted of an initial lecture, with questions from students encouraged. This was followed by paper tasks and practical programming tasks. Later in the course, tasks that involved programming *strategies* were used. Students were given breaks between sessions.

6.2 Research Questions

This experiment was motivated by four related questions, which are answered in section 6.6.

RQ6. *Can programming strategies be explicitly incorporated into an introductory programming curriculum?*

RQ7. *What is the significance of the time consumed by this additional instruction?*

RQ8. *Can programming strategies, explicitly taught in an introductory programming course, be assessed?*

RQ9. *What impact does explicit strategy instruction have on students and their ability to apply strategies when compared to an implicit-only approach?*

RQ10. *Are there any other observable effects or contrasts between students of a traditional curriculum and one with added explicit programming strategy instruction?*

6.3 Description of Curricula

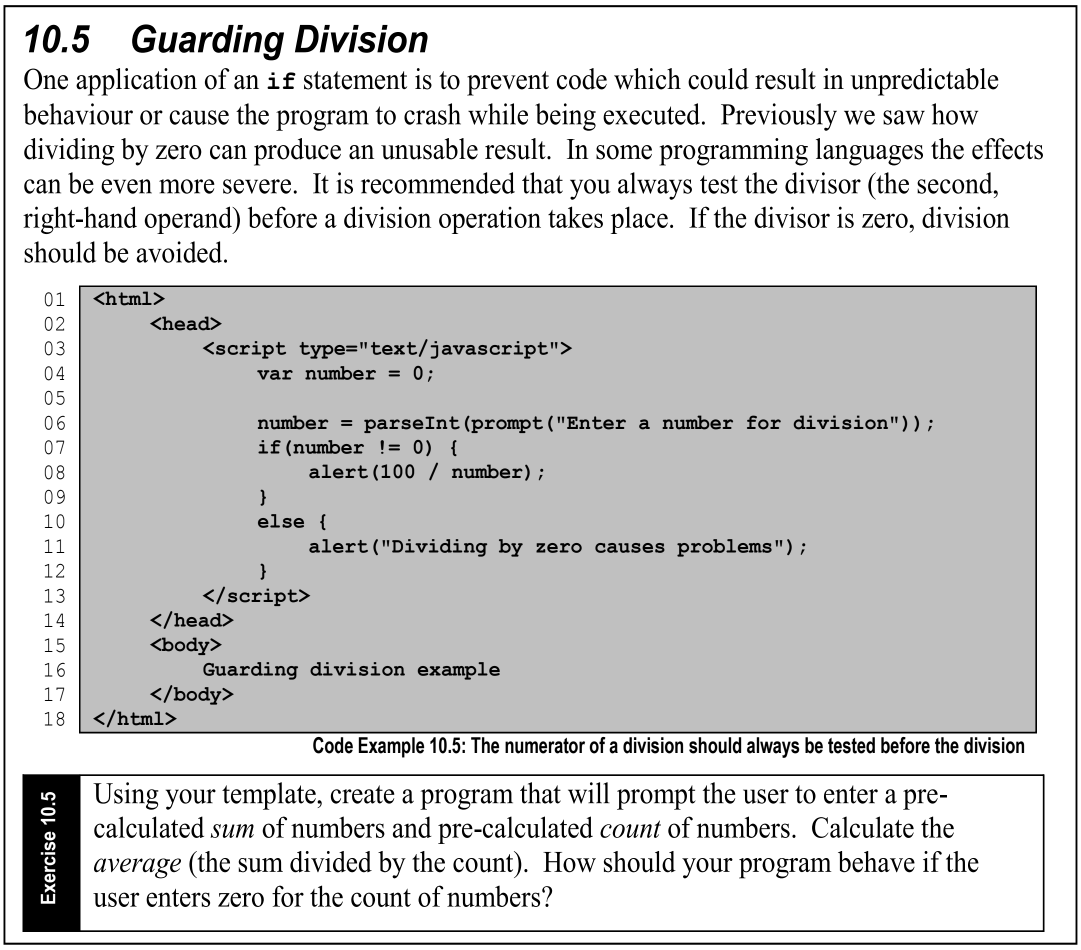
An experimental curriculum was created which contained programming *strategy* instruction explicitly. This curriculum is described further in this section and is included in full in Appendix H. From this, a control curriculum was created by identifying and removing programming *strategy* instruction components.

6.3.1 Explicitly Incorporating Programming Strategies

Programming *strategies* are explicitly incorporated into the curriculum in a number of ways.

Identifying Strategies in the Curriculum

A book of written study materials was created and hardcopies were given to participants. Lecture slides were also created, based on the content of the written study materials. The lecture slides were used during lectures. In written materials and lectures the *strategies* incorporated in the curriculum were named, their benefits were explained and examples of their application were shown. Figure 6.1 shows a section of the written materials provided to students. In this example the Guarded Division plan is identified. An explanation is given for why this *plan* is used, including a reference to an earlier mention of the consequences of dividing by zero. The description tells how the *strategy* is implemented, and an code example, applying this *strategy*, is shown.



10.5 Guarding Division

One application of an `if` statement is to prevent code which could result in unpredictable behaviour or cause the program to crash while being executed. Previously we saw how dividing by zero can produce an unusable result. In some programming languages the effects can be even more severe. It is recommended that you always test the divisor (the second, right-hand operand) before a division operation takes place. If the divisor is zero, division should be avoided.

```
<html>
    <head>
        <script type="text/javascript">
            var number = 0;

            number = parseInt(prompt("Enter a number for division"));
            if(number != 0) {
                alert(100 / number);
            }
            else {
                alert("Dividing by zero causes problems");
            }
        </script>
    </head>
    <body>
        Guarding division example
    </body>
</html>
```

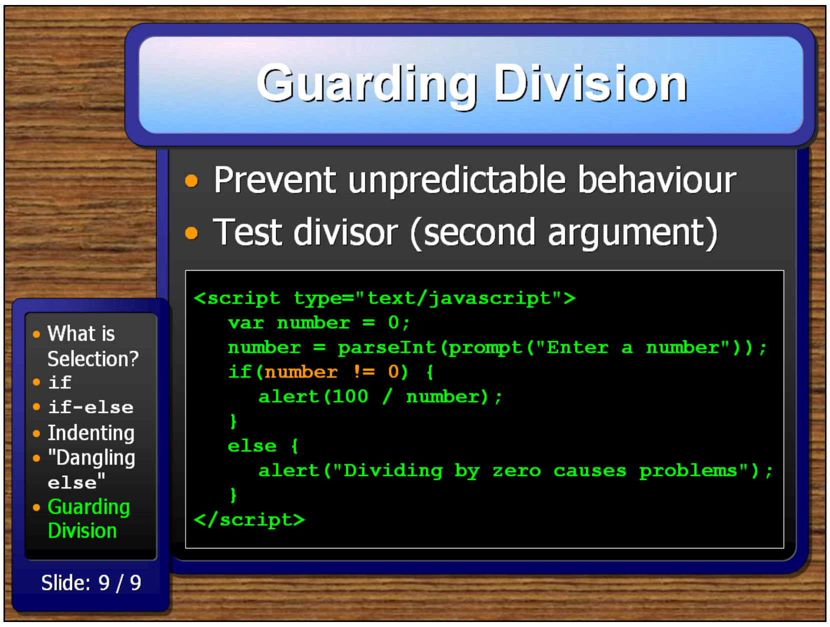
Code Example 10.5: The numerator of a division should always be tested before the division

Exercise 10.5

Using your template, create a program that will prompt the user to enter a pre-calculated *sum* of numbers and pre-calculated *count* of numbers. Calculate the *average* (the sum divided by the count). How should your program behave if the user enters zero for the count of numbers?

Figure 6.1. An extract from the written course materials showing explicit incorporation of programming *strategy* instruction

As well as introducing *strategies*, the means of integrating these *strategies* through abutment, merging and nesting (Soloway, 1986, p. 856) were also covered.



Guarding Division

- Prevent unpredictable behaviour
- Test divisor (second argument)

```
<script type="text/javascript">
   var number = 0;
   number = parseInt(prompt("Enter a number"));
   if(number != 0) {
      alert(100 / number);
   }
   else {
      alert("Dividing by zero causes problems");
   }
</script>
```

What is Selection?
if
if-else
Indenting
"Dangling else"
Guarding Division
Slide: 9 / 9

Figure 6.2. An example of a lecture slide showing incorporation of explicit programming *strategy* instruction

Paper Exercises and Practical Computing Tasks

At the end of each module students were asked to complete paper exercises and computer-based tasks to reinforce the content delivered in lectures and allow students to experience the practical implementation of the *strategies* covered. Instructions for these exercises and tasks were set out in the written materials, for example Exercise 10.5 shown in Figure 6.1. The exercise shown prompts users to explore Guarding Division. In other exercises students are prompted to experiment with the outcome achieved when the *strategy* is not applied or is poorly applied. During the course, as with any normal introductory programming class, the instructor was on hand to answer questions and guide students.

In most cases the exercises and tasks were common to both curricula. In the curriculum without explicit programming *strategies*, students were expected to learn the required programming *strategies* implicitly.

Table 6.1. Comparison of the two curricula tested (items with ~~strike through~~ were absent in the control curriculum)

Module	Section	Experimental Curriculum	Control Curriculum
1		**First JavaScript Program**	**First JavaScript Program**
	1.1.	Hello World!	Hello World!
	1.2.	JavaScript and HTML	JavaScript and HTML
	1.3.	Statements	Statements
2		**Calling Functions**	**Calling Functions**
	2.1.	alert()	alert()
3		**Values**	**Values**
	3.1.	Numbers	Numbers
	3.2.	Strings	Strings
	3.3.	Booleans	Booleans
4		**Variables**	**Variables**
	4.1.	What are Variables	What are Variables
	4.2.	Identifier Rules	Identifier Rules
	4.3.	Declaring Variables with var	Declaring Variables with var
	4.4.	Undefined	Undefined
5		**Assigning Values**	**Assigning Values**
	5.1.	Dynamic Typing	Dynamic Typing
	5.2.	typeof	typeof
	5.3.	Initialising Variables	~~Initialising Variables~~
6		**Operations**	**Operations**
	6.1.	Arithmetic Operators	Arithmetic Operators
	6.2.	Division by Zero – infinity	~~Division by Zero – infinity~~
	6.3.	Postfix Operators	Postfix Operators
	6.4.	Relational Operators (incl. Equality)	Relational Operators (incl. Equality)
	6.5.	Logical Operators	Logical Operators
	6.6.	String Operators	String Operators
7		**Abutment**	~~**Abutment**~~
8		**Debugging**	**Debugging**
		Exercise 8.3	~~Exercise 8.3~~
9		**Functions that Return Values**	**Functions that Return Values**
	9.1.	prompt()	prompt()
	9.2.	parseInt() and parseFloat()	parseInt() and parseFloat()
10		**Selection**	**Selection**
	10.1.	The if Statement	The if Statement
	10.2.	The if-else Statement	The if-else Statement
	10.3.	Indenting and Formatting	Indenting and Formatting
	10.4.	“Dangling else”	“Dangling else”
	10.5.	Guarding Division	~~Guarding Division~~
11		**Repetition (Loops)**	**Repetition (Loops**
	11.1.	while Loop	while Loop
	11.2.	Sentinel-Controlled Loops	~~Sentinel-Controlled Loops~~
	11.3.	for Loop	for Loop
	11.4.	Counter-Controlled Loops	~~Counter-Controlled Loops~~
	11.5.	Finding the Maximum/Minimum	~~Finding the Maximum/Minimum~~
	11.6.	Nesting and Merging	~~Nesting and Merging~~
12		**Arrays**	**Arrays**
	12.1.	Declaring Arrays	Declaring Arrays
	12.2.	Accessing Array Elements	Accessing Array Elements
	12.3.	Initialising Arrays	Initialising Arrays
	12.4.	Arrays for Values	Arrays for Values
	12.5.	Arrays for Categories	Arrays for Categories
	12.6.	Counting Values in a Set	~~Counting Values in a Set~~

Assessment of Programming Strategies

At the end of the course, students were asked to complete the same three programming tasks that were given to experts in the previous study with experts (see section 5.3). These tasks were used as a formal assessment at the end of the course under exam conditions. As well as testing participants' abilities, this was done to explore the potential to assess programming *strategies* as part of a course. *Strategies* necessary to solve the final assessment problems were shown as examples and in exercises and programming tasks.

6.3.2 Format of the Curriculum

The curriculum follows a traditional format, which reveals parts of a given language in a sequence, with new *knowledge* of language concepts being dependent on previously covered *knowledge*. In this format, explicitly incorporating programming *strategies* depend upon underlying *knowledge* being taught beforehand. For instance, for the Guarded Division plan to be introduced, *knowledge* of variables, operators and selection must be covered first. Table 6.1 shows the two curricula with elements excluded from the control curriculum struck out.

Basing the experimental curriculum on a traditional curriculum allowed the creation of a second curriculum without explicit programming *strategies*. In a non-experimental setting, the format of the curriculum could change. For instance, the *strategies* themselves, rather than the underlying language, could be used to govern the structure of the course; in this case *strategies* could be introduced, then underlying language *knowledge* could be taught. If an objects-first approach is taken, *strategies* could be introduced at other stages.

6.3.3 Philosophy behind the Experimental Curriculum

The curriculum was designed to be short and allow students to reach programming *strategies* as soon as possible. The curriculum would not be effective in teaching longer courses, although the explicit incorporation of programming *strategies* could be applied to longer curricula.

The curriculum was focused on programming *strategies* with only a minimal covering of the dependent *knowledge* components. *Knowledge* content was included if it was not fundamentally important for learning the later programming *strategies*. The later exercises focused on the application of programming *strategies*. For those who had not been explicitly instructed in programming *strategies*, exercise times were their opportunity to implicitly learn the needed *strategies*. The assessment at the end of the course focused on the analysis of programming *strategy* skill. In a non-experimental course the focus of exercises and the weighting of examination questions would likely be more balanced between programming *knowledge* components and programming *strategies*.

6.3.4 Language Used with Experimental Curriculum

JavaScript was used as the language that supported the instruction of the curriculum. In their essential form, programming *strategies* are language independent and examples could be given in almost any language. Soloway and his colleagues used Pascal and Lisp to illustrate programming *strategies*. The author has used C/C++ to exemplify programming *strategies* in other work.

Reasons for choosing JavaScript for this experiment were as follows.

- **Potential to reach important concepts rapidly**
  JavaScript has simple facilities for user input and output and a simple model for data storage. This permits rapid progress through foundational *knowledge* concepts that might take longer if a general purpose programming language were used.

- **Simpler to practice than a compiled language**
  JavaScript is interpreted by a web browser as part of a web page with a simple model of execution. Explaining a compiled model of execution was not required. JavaScript programs can be as simple as a single statement and do not need to be contained within a full program context.

- **Attractive to volunteers**
  JavaScript is used to achieve dynamic client-side web pages. Even students who are not studying computing are likely to be familiar with the name 'JavaScript' through use of the World-Wide Web. For this reason it was an incentive to attract experimental volunteers.

- **Expression of programming strategies in another language**
  Plans had not been expressed using JavaScript before. Using JavaScript showed that *plans* could be demonstrated in another language, attesting to the versatility of *plans*.

6.4 Methodology

The method of experimentation began with preliminary demographic, experience and confidence measurements. An examination of programming *strategies* was conducted at the end of each weekend. In the weeks that followed the two weekend sessions, participants were invited to an interview in which they were asked questions about their solutions to gauge their understanding of the *strategies* that were being tested.

6.4.1 Demographic, Experience and Confidence Measures

A number of demographic, experience and confidence measures were conducted via a web survey presented to students when they volunteered. Participants were asked questions to determining the following.

- Gender
- Age
- Computing experience
- Previous programming experience
- Computing confidence

Details of specific questions are given in Appendix D. Computing confidence was captured using a test created by Cretchley (2006) which has been proven as a reliable predictor of computing confidence.

6.4.2 Schedule of Course Delivery

The schedule for both weekends was identical except where programming *strategy* content was covered. In Table 6.2, content covering programming *strategies* is highlighted and was covered only in the course with explicit instruction of programming *strategies*. Participants undertaking the course without explicit programming *strategy* content were intended to be attempting practical exercises during these times. One of the aims of the experiment was to determine if this additional content would impact on the balance of lecture and exercise time. For this reason the schedule was followed as closely as possible on both weekends.

Table 6.2. Schedule of courses (items greyed were not conducted with control curriculum)

Session	Saturday Content	Sunday Content
10:00 – 11:15	Introductions 1 First JS Program 1.1 Hello World 1.2 JavaScript and HTML 2 Calling Functions 2.1 alert()	11 Loops 11.1 while Loop 11.2 Sentinel-Controlled Loops 11.3 for Loop 11.4 Counter-Controlled Loops 11.5 Finding the Maximum 11.6 Nesting and Merging
11:30 – 13:00	3 Values 3.1 Numbers 3.2 Strings 3.3 Booleans 3.4 Undefined 4 Variables 4.1 What are Variables 4.2 Identifier Rules 4.3 Creating variables with var 5 Assigning Values 5.1 Dynamic typing 5.2 typeof 5.3 Initialising Variables	12 Arrays 12.1 Arrays for Values 12.2 Arrays for Categories 12.3 Counting Values in a Set
13:30 – 14:45	6 Operations 6.1 Arithmetic Operators 6.2 Division by Zero - Infinity 6.3 Postfix Operators 6.4 Relational Operators (incl. Equality) 6.5 Logical Operators 6.6 String Operators 7 Abutment 8 Debugging 9 Functions that Return Values 9.1 prompt() 9.2 parseInt()	Testing
15:00 – 16:00	10 Selection 10.1 The if Statement 10.2 The if-else Statement 10.3 Indenting and Formatting 10.4 “Dangling else” 10.5 Guarding Division	

6.4.3 Administering the Final Assessment

After lunch on the Sunday of each weekend course, participants were asked to complete the three programming tasks previously given to experts (see chapter 5). Each problem was presented on a single sheet of paper, with lines below in which students were to complete the solutions to the problems (solution sheets are shown in Appendix C). Participants were able to use as much time as was needed to complete problems.

Problem 1

Read in 10 positive integers from a user. Assume the user will enter valid positive integers only. Determine the maximum.

Problem 2

Read in any number of integers until the value 99999 is encountered. Assume the user will enter valid integers only. Output the average.

Problem 3

> *Input any number of integers between 0 and 9. Assume the user will enter valid integers only. Stop when a value outside this range is encountered. After input is concluded, output the occurrence of each of the values 0 to 9.*

The solutions produced were examined using Goal/Plan Analysis to test for the presence or absence of expected *plans*. This was conducted in the same manner as the earlier experiment with experts, as described in section 5.4.1.

6.4.4 Post-Experiment Interviews with Participants

In the 23-day period after teaching, six participants gave a verbal, one-on-one interview, at their earliest convenience. Students' solution sheets were used as a basis for discussion. Interviews were structured, with set questions as listed in Appendix E. The questions were used as a script, but were intended to encourage discussion which was allowed to continue as long as necessary. The questions used were designed not to be leading. Questions were aimed at discovering participants' interpretations of the problem statements, the *strategies* understood by participants, the articulation of their solution and their confidence in their solution. During interviews participants' responses were recorded as audio files which were transcribed. The interviewer was the teacher and also the author of this dissertation.

The interview transcripts were analysed by looking for references to strategies used (correct or incorrect), use of terminology relating to plans and statements of programming confidence (positive or negative).

6.5 Results

A number of results were gained from this experiment. First, data gathered during registration are shown. During the experiment both curricula were delivered to students. The potential to succeed in this delivery was judged by the time used to deliver the more extensive curriculum, which explicitly incorporated programming *strategies* within the schedule. At the end of each of these sessions participants were asked to complete a set of problems that were examined under Goal/Plan Analysis. Finally an inspection of post-course interviews provides deeper insights into the programming *strategy* potential of the participants after the course.

6.5.1 Data Collected at Registration

The data gathered when participants volunteered for the course are shown in Table 6.3. These data show that the two groups were roughly balanced in gender, age and computing confidence. The two groups differed in responses to computing and web experience self-assessment questions. Experimental group participants showed greater variance in their responses to these experience questions. It is likely that the experimental group was affected more by individual differences. One of the participants indicated they had no previous use of a web browser, even though they used a computer daily. This may have been an error. Participant 29 left early during the data collection period.

One of the intentions for gathering this data was to exclude volunteers who had completed previous formal study in programming. A number of people signed up for the experiment and were rejected because they had studied programming previously. One participant, identified as Participant 14, who was asked not to attend, came anyway. The results of this participant are not presented here. One other participant (21) indicated they had some self-taught programming experience. After discussion with the participant this experience was shown to be a limited amount of HTML writing, which was not seen as significant in this experiment.

Table 6.3. Demographic, experience and confidence data gathered on registration

Group	Participant	Gender	Age Group	Computing Experience	Web Experience	Previous Programming	Computing Confidence 1=low to 5=high
Experimental Group	12	male	Less than 25	Daily use	No use	Never	3.0
	21	male	26 – 35	Daily use	Daily use	Some self-taught	4.6
	29	male	26 – 35	Weekly use	Every few days	Never	3.2
	30	female	Less than 25	Daily use	Daily use	Never	4.4
Average							3.8
Control Group	1	male	Less than 25	Daily use	Daily use	Never	3.6
	6	female	Less than 25	Daily use	Daily use	Never	3.5
	13	male	26 – 35	Daily use	Daily use	Never	3.8
Average							3.6

6.5.2 Time Load of Explicit Programming Strategy Instruction

During teaching of the experimental curriculum that incorporated explicit programming *strategies*, added content required additional time to teach (actual time measures were not recorded), increasing the length of lecture sessions and reducing the time allowed for students to undertake practical work. However, participants undertaking the curriculum with explicit programming *strategies* were still able to complete the set exercises during the time allocated in the schedule. It was possible for the schedule to be followed in both instances of the curriculum.

6.5.3 Time to Complete Tests

While participants were completing solutions to the given problems, each was timed and the times were recorded.

Table 6.4: Times for problems by group in minutes and seconds

	Prob. 1	Prob. 2	Prob. 3	Overall
Participant 12	12:10	07:20	11:30	31:00
Participant 21	15:00	04:05	07:35	26:40
Participant 29		Left early		
Participant 30	10:30	13:00	16:40	40:10
Experimental Group Average	12:33	8:08	11:55	**32:37**
Participant 1	10:40	09:40	09:30	29:50
Participant 6	14:20	19:30	05:40	39:30
Participant 13	15:50		Not recorded	
Control Group Average	13:37	14:35	7:35	**34:40**
All	13:05	10:43	10:11	33:38

Table 6.4 shows the times taken by each participant to complete each problem, collected together in groups. In the experimental group, Participant 29 left after completing part of one solution, explaining that they wished to leave for personal reasons. In the control group Participant 13 completed all problems but the times taken to solve the second two problems were not recorded.

Participants in the experimental group were quicker on average, but both the longest and shortest times were exhibited in the experimental group. There appeared to be no relationship between times and the correctness of solutions presented. With the small population, no generalisations can be made, but the results are reported here to inform later results.

Experts who completed the same three problems took 16min 34sec on average (see section 5.5) and constructed complete solutions. Most participants in this study were not able to create complete solutions.

6.5.4 Goal/Plan Analysis of Participant Solutions

The following tables show results of the Goal/Plan Analysis for each problem. Several of the solutions presented by novice participants in this experiment contained English language text that described the code the participant would like to have written in their solution when they were not sure how to implement these ideas in code. Where this was the case, if the text sufficiently described a *plan*, it was accepted as being present even if it was not described in code. The participants who used text in their code did not create complete or near complete solutions.

Table 6.5. Presence of *plans* and integration for Problem 1

Plan	Presence of Plans/Integration									
	Exp. Participant				Exp. Group Average	Control Participant			Control Group Average	All
	12	21	29	30		1	6	13		
Max Initialised					0%				0%	0%
Counter-Controlled Loop	Y	Y	Y		75%	Y	Y		67%	71%
Input Plan	Y	Y			50%	Y	Y	Y	100%	71%
Maximum Plan	Y				25%				0%	14%
Output Plan	Y	Y			50%			Y	33%	43%
Input Nested in Counter-Controlled Loop	Y	Y			50%	Y			33%	43%
Max Plan Nested in Counter-Controlled Loop	Y				25%				0%	14%
Abutment Correct	Y	Y	Y		75%	Y		Y	67%	71%
Overall	88%	63%	25%	0%	44%	50%	25%	38%	28%	41%

Table 6.5 shows the *plans* present in each participant's solution to Problem 1. The correctness of the integration of the *strategies*, including abutment, is also recorded. Unlike experts (see section 5.5), participants in this experiment did not always integrate these *plans* correctly.

The best Problem 1 solution was created by Participant 12 from the experimental group who, despite never previously undertaking programming study, was able to produce a well coded solution that was nearly complete. This solution, together with those presented by Participant 21, pushed the overall average correctness level for the experimental group above that of the control group despite the abandoned attempt and non-attempt of their group-mates.

One noticeable characteristic was the absence of the initialisation of the maximum variable, which was crucial to the Maximum plan and is required when using JavaScript. Initialisation was explicitly covered in the experimental curriculum. Students undertaking the control curriculum were given the opportunity to learn this *plan* implicitly. Initialisation was important to the later problems and was applied by a number of participants for those problems. It is not clear why it is absent here.

Table 6.6. Presence of *plans* and integration for Problem 2

Plan	Presence of Plans/Integration									
	Participant				Exp. Group Average	Participant			Control Group Average	All
	12	21	29	30		1	6	13		
Sum Initialised	Y				33%	Y	Y		67%	50%
Count Initialised	Y				33%	Y			33%	33%
Sentinel-Controlled Input	Y	Y			67%				0%	33%
Sentinel-Controlled Count	Y				33%		Y		33%	33%
Sentinel-Controlled Sum	Y				33%		Y		33%	33%
Guarded Division			Left Early		0%				33%	0%
Output Plan	Y	Y			67%	Y	Y	Y	0%	83%
Loop Plans Merged	Y				33%	Y			100%	33%
Inputs Nested in Sentinel-Controlled Loop	Y	Y			67%				33%	33%
Output Nested in Guarded Division					0%				0%	0%
Abutment Correct	Y	Y			67%	Y		Y	67%	67%
Overall	82%	36%		0%	39%	45%	36%	18%	33%	36%

Table 6.6 shows the *strategy* correctness of participants' solutions to Problem 2. Participant 29 left after abandoning an attempt at Problem 1, so this participant's solutions were not included in results for this and the next problem.

Again in this problem, an outstanding solution was presented by Participant 12 who correctly solved the problem, with the exception of the Guarded Division plan. No participant in either group applied a Guarded Division plan. This shows that even when it is explicitly incorporated into an introductory programming curriculum, and the consequences of failing to apply the *plan* are discussed, it is still possible for novice programmers to neglect this particular *plan*. This problem was a modified version of the problem given to students in the first experiment (see chapter 4). Students in the earlier study had completed a semester of instruction under a traditional implicit-only model and achieved an average overall completeness of 57% compared to the participants of this experiment who achieved 36%. In the problem statements for Problem 1 and both other problems, students were told they could assume inputs would be valid.

Table 6.7. Presence of *plans* and integration for Problem 3

Plan	Presence of Plans/Integration									
	Participant				Exp. Group Average	Participant			Control Group Average	All
	12	21	29	30		1	6	13		
Counter-Controlled Loop (for Initialisation)	Y			Y	67%				0%	33%
Array Initialisation	Y	Y		Y	100%				0%	50%
Sentinel-Controlled Input	Y				33%				0%	17%
Count Set Plan	Y	Y			67%				0%	33%
Counter-Controlled Loop (for Output)				Y	33%	Y	Y		67%	50%
Output Plan	Y	Y	Left Early		67%				0%	33%
Initialisation nested in Counter-Controlled Loop	Y			Y	67%				0%	33%
Inputs nested in Sentinel-Controlled Loop	Y			Y	67%				0%	33%
Count Set nested in Sentinel-Controlled Loop	Y				33%				0%	17%
Output Nested in Counter-Controlled Loop					0%				0%	0%
Abutment Correct	Y	Y		Y	100%	Y	Y	Y	100%	100%
Overall	82%	36%		55%	58%	18%	18%	9%	15%	36%

Table 6.7 shows the *plan* application for the final problem, Problem 3. Again an outstanding solution was presented by Participant 12 who correctly initialised and filled an array to tally user inputs, but failed to output the content of the array using a loop. Participant 30, who did not attempt Problem 1 and presented a confused solution to Problem 2, managed to apply a number of *plans* for this problem. Participants from the control group showed little ability to demonstrate any of the *plans* that were needed to solve this problem. This problem is arguably the most complex, and it would appear from these results that it is difficult to learn the necessary *plans* implicitly.

One *plan* absent in all solutions was the Counter-Controlled Loop plan to output the occurrences of numbers. This is not truly surprising as most of the solutions for this problem were incomplete and the only near-complete solution did not apply this particular *strategy*. Each of the participants from the experimental group applied a counter-controlled loop to initialise the array used for tallying.

Table 6.8. Overall *plan* use by each group

	Overall Plan Use
Experimental Group	47%
Control Group	28%
All	38%

Table 6.8 shows a comparison of the overall correctness for all problems achieved by each group. There is a distinction in overall results for the two groups, with the experimental group, who were exposed to a curriculum that incorporated programming *strategies* explicitly, achieving a greater result.

Participant 12 produced outstanding solutions to each of the problems. It may be that the incorporation of explicit programming *strategies* suited this participant who, might have performed better than he would have otherwise. One must wonder if they would have done as well in the control group and perhaps reversed the results of the experiment.

With the small number of participants in this experiment no statistically significant evidence can be inferred for the superiority of one curriculum over another. These results are useful as basis for the interviews that followed which allow a deeper and more personal exploration of the participating students' *strategy* understandings.

6.5.5 Interviews

Following the course, participants were asked to attend an interview. Five of the seven participants and Participant 14 (who had previous programming instruction) attended interviews. The list of participants and the length of interviews is shown in Table 6.9.

Table 6.9. Interview participants and interview times

Participant	Group	Time
12	Experimental	30:35
21	Experimental	22:44
30	Experimental	21:56
1	Control	17:40
6	Control	23:14
Average Time		23:14

The interviews probed the understandings of participants that they were perhaps unable to express in code. The interviews followed a fixed script but allowed participants to discuss matters freely. The list of questions is shown in Appendix E. The questions were designed not to be leading. The questions posed to each participant aimed to achieve the following.

- To explore the participant's interpretation of the problem statement
- To examine whether the participant understood the required *strategies*
- To allow the student to articulate their solution
- To elicit a level of confidence in the participant’s solution

Each interview was recorded and transcribed. From an analysis of the transcripts the following observations were made.

Participants Misinterpreted the Validation Simplification Made to Each Problem

Each problem statement contained the text “Assume the user will enter valid integers only.” This additional text was introduced to clarify that no attempt at validation would be necessary. This change was made when these problems were used with expert programmers, but for this experiment it may have confused participants. In interviews, participants were asked what this sentence meant. Three of the five participants misinterpreted this simplification; some felt validation was necessary because of this statement. No participant attempted to validate inputs.

Other parts of the problem statements seemed to be comprehensible to each participant, even if they did not know how to achieve a solution.

Participants Exhibited Understanding of Plans

As well as demonstrating a higher use of *plans* in their solutions to problems, participants from the experimental group verbally described *plans*. For instance, Participant 30 described her application of a Set-Counting plan as follows: “After you've put a number that isn’t in that range it concludes the program and tells the person what numbers you've put into your little boxes. It goes through zero to nine and it tells you how many are in each box.”

Rist (1995) showed that novices can expound and apply *plans* without explicit instruction of programming *strategies*. Some control group participants did learn *plans* implicitly. In an observable instance Participant 6 stated the following which could be seen as a description of a Set-Counting plan using an array: “I've created an array, because I think that for the program to calculate, between 0 and 9, how many times it occurs, it has to have an array for, say if it’s zero, then zero; for one it's one, two three, four... So the array for zero is, like, zero, because arrays start from zero, right? Then, so in the box for zero, say the user enters three times it will refer back to this array zero, it will keep repeating itself in the loop, from then on how many times it gets zero in that box it will get the output.”

Participants Failed to Learn Some Plans

It was clear that participants did not learn all the *plans* they were expected to learn. This was true for participants from the control group who were expected to learn *strategies* implicitly; for example, Participant 6 felt there must be some maximum formula that would take care of the task of calculating maximums: “And probably some formula to determine the highest number (which I don't know how)”.

Experimental group participants also failed to demonstrate application of some *plans*, even though they had been explicitly exposed to them. For example Participant 30, when asked how a maximum could be determined, responded, “Can you make the program look at the digits I guess, so you could determine the maximum. I don't know.” Participant 21, when asked, “What does it mean by determine the maximum?”, responded with, “Perhaps the maximum sum. I'm not really sure.”

Experimental Group Participants Used Plan Terminology and Ideas

On a number of occasions participants from the experimental group (who were exposed to *plans* and related terminology) referred to parts of their code using *plan* terminology, or attempted to describe *plan* terminology without using specific names.

Participant 12, while discussing the integration of counting with input in Problem 2 said “they have to merge with the loop”.

During the interview with Participant 21 discussing loops in Problem 2, the participant cannot remember the terminology for a Sentinel-Controlled Loop but describes it well: “…and then create a loop… get user input outside and inside so that it's, I can't remember the name.” Later Participant 21, while interpreting part of the problem statement, recalls the correct terms and says “Which I did recognise as a sentinel loop.”

The use of Goal/Plan terminology was not universal by any means. Participants from the experimental group still resorted to syntactical description when describing their code and needed to be prompted further to elicit possible *strategy* understandings. Participant 12, who delivered perhaps the best result stated the following syntactical reading of code: “It's a loop, for loop. For counter equals zero. Start from zero again. And counter smaller than numberNum. Counter++. And the message is numArray[counter] equals zero.”

Experimental Group Participants showed Confidence in Solutions

Experimental group participants were more confident in their solutions, or their ability to correct their solutions if given the chance. This is despite the fact that no participant had created a fully complete solution to any of the problems. Participant 21 was confident about all his solutions, even though they were flawed. Participant 30 showed confidence in most of their attempted solutions even though they were flawed; when asked “Does your solution solve the problem?”, she replied, “…Well my solution in my head did, not like the first one, so yes. I did understand this question so I could go through the steps of doing it.”

Participant 12, who was the closest of all participants to solving the problems correctly, was realistic about the correctness of their solutions. During discussion Participant 12 saw the flaws in two of three of his solutions. Interestingly this participant explains his confidence in one of their problems as being the result of understanding the required *strategy*: “I'm very confident in doing this question because I know the right way to structure [it].”

Control Group Participants showed a Lack of Confidence

When asked if they believed if their solutions correctly solved each problem, members of the control group almost universally showed a lack of confidence in the solutions they had created.

Participant 1 lacks confidence in all his solutions except for Problem 2, where they claim more time was needed, even though time was not restricted during the test. When this participant was asked, “Does your solution solve the problem?”, he answered, “Probably, if I got time to add up more things.” This same participant later describes a lack of confidence in their general programming ability: “I'll probably mess it up anyways, because I'm still not sure how to...”, and later expresses a typical gap between design and implementation where *plans* can be applied: “I understand the question. I was thinking through. I got everything right in my head. I just can't put it onto codes.”

The other control group participant interviewed, Participant 6, showed some confidence in one solution, believing, correctly, that the remaining solutions were flawed.

6.6 Discussion

The research questions posed earlier are answered by the results of this experiment and observations made during the experiment.

6.6.1 Incorporating Explicit Programming Strategies

RQ6. *Can programming strategies be explicitly incorporated into an introductory programming curriculum?*

Programming *strategies* can be explicitly incorporated into an introductory programming curriculum. The curriculum used in this experiment, and its successful delivery, is evidence that this can be done.

6.6.2 Balance of Lectures and Practice

RQ7. *What is the significance of the time consumed by this additional instruction?*

As stated in section 6.5.2, the additional instruction in the experimental curriculum did require more time in lecture sessions, but students were still able to complete set exercises by the end of each session. It can therefore be asserted that this additional instruction is balanced by an eased burden on students in completing practical exercises.

This result is useful for our comparison of the curricula, however in regular teaching, lectures and practicals are usually conducted in disjoint time slots, so extending the length of a lecture would not normally impact on practice time.

Having more material in one curriculum over another would increase the burden on student learning, with more content to process. This addition needs to be compared with the effort a student would have to make to develop the needed programming *strategies* in an implicit-only model.

6.6.3 Assessment of Programming Strategies

RQ8. *Can programming strategies, explicitly taught in an introductory programming course, be assessed?*

Goal/Plan Analysis of students' solutions is far from new, but as a means of assessment in a programming course it is novel. This experiment showed that programming *strategies* applied to create solutions can be assessed using Goal/Plan Analysis. A limitation of using Goal/Plan Analysis is that it requires students to generate code before it can be assessed. In early stages, assessing generated code might not be the best method of assessing programming *strategies*.

6.6.4 Impact on Programming Strategy Ability

RQ9. *What impact does explicit strategy instruction have on students and their ability to apply strategies when compared to an implicit-only approach?*

From Goal/Plan Analysis of participants' solutions and through interviews it appeared that students exposed to the experimental curriculum may be more likely to understand and apply *strategies* than participants who were expected to learn *strategies* implicitly.

It was by no means guaranteed that participants explicitly shown programming *strategies* would understand and apply all of these *strategies*. It was also demonstrated that, although less common, participants exposed to an implicit-only curriculum can learn programming *strategies*.

6.6.5 Other Observed Effects

RQ10. *Are there any other observable effects or contrasts between students of a traditional curriculum and one with added explicit programming strategy instruction?*

Two other observations can be made from results shown.

A Vocabulary for Strategies

Some participants in the experimental group, who were exposed to *plan* terminology during their instruction, went on to use this terminology during interviews. If this were applied during an ordinary teaching period with multiple weeks of instruction and assessment, being able to have students use a vocabulary of programming *strategy* terms would be beneficial. Instructors would be able to describe the *strategies* they expect students to apply in tasks. It would be possible to allocate marks for the application of specific *strategies*. Instructors would have the potential to describe and analyse code using such terminology.

Confidence in Solutions

A contrast was found in the confidence of participants. Participants from the experimental group, who had been exposed to programming *strategies* explicitly, were confident about the solutions they presented and the understanding of the *strategies* needed to complete the solutions. Participants from the control group were not so confident. It is not necessarily clear why this is the case. Perhaps because experimental group participants had been exposed to a higher level of programming thought, they might feel that the underlying syntactical implementation is less difficult to achieve. Reber (1993) showed that students exposed to implicit-only instruction can gain aptitude but fail to gain understanding of underlying systems. This seems to be consistent with the experience of participants exposed to implicit-only instruction of programming *strategies* in this experiment who were, in some instances, able to produce partial solutions, but appeared to have a general lack of understanding of programming *strategies* and the processes needed to solve the problems presented.

6.6.6 Flaws in the Experimental Approach

A number of flaws in the experimental approach were realised during and after the experiment.

Size of Groups

The size of the experimental and control groups was sufficient to test the potential to incorporate explicit programming *strategy* content into an introductory programming curriculum and the timing of that incorporation. It was sufficient to allow a small number of participants to experience these curricula and be interviewed on their understandings that developed through this participation.

Although the Goal/Plan Analysis of participants' solutions showed differences between the groups, the size of the population of participants was insufficient to statistically infer the superiority of the experimental curriculum. It is not clear that increasing the size of the participant population would produce consistent reproducible results, which appears to be the bane of many explorations in educational settings (see Hirsch (2002)).

Absorbing Concepts Rapidly

Participants in the study were diligent students. All students were able to follow the course materials and achieve results in paper exercises and practical computer tasks. However, complete solutions in the final assessment, involving *generation* of code to novel problems, appear to have been more than could be expected from students at the end of two days of instruction. Although exercises were given to reinforce concepts covered, these may not have been as effective as if they were completed days or weeks later.

The result of this experiment shows that the *strategy* ability of participants exposed to the experimental curriculum produced an average overall completeness of 39% for Problem 2 compared to students who had been exposed to a semester-long, traditional introductory course in programming, who achieved an average overall completeness of 57% on effectively the same problem.

Generation of Code can be a Poor Measure

The final assessment asked students to generate code to novel problems, applying *strategies* they had learned. Most of the participants were unable to create complete solutions to these problems. This may be attributable to a lag between:

1. exposure to a programming *strategy*,
2. the ability to comprehend that *strategy*, and eventually
3. the ability to generate an implementation that applies that *strategy*.

After only two days, asking participants to generate code might have been a less accurate test than gauging their programming *strategy* skill by other means, such as *comprehension* tests or cases involving errors.

6.7 Implications

This experiment showed that it is possible to create a curriculum that explicitly incorporates sub-algorithmic programming *strategies*. The incorporation of such additional instruction does not create an infeasible burden of time.

There were also noticeable effects on the students participating in the experiment and exposed to this additional instruction. Participants who covered the experimental curriculum appeared more likely to understand and apply the programming *strategies* they had been exposed to. These students used terms from a programming *strategy* vocabulary presented in the curriculum, which could be useful in teaching and assessment if applied to a full-scale course. Participants who covered the experimental curriculum claimed confidence in the solutions they had created and their understanding of the *strategies* used to create them, while students not exposed to explicit programming *strategies* doubted their abilities.

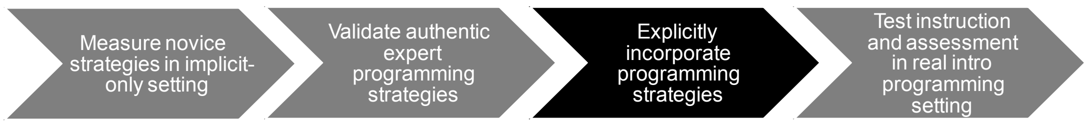

Measure novice strategies in implicit-only setting
Validate authentic expert programming strategies
Explicitly incorporate programming strategies
Test instruction and assessment in real intro programming setting

Figure 6.3. Overview of experiments in a process after third experiment

Having described positive benefits from explicitly incorporating programming *strategy* instruction in an artificial setting, the next stage of experimentation involves applying this approach to an actual course and evaluating student outcomes.

Goal/Plan Analysis is a basic tool for analysing student code and detecting deficiencies in student understanding. It has been used here to measure student solutions and as a basis for a deeper exploration of novice understanding. However, the use of Goal/Plan Analysis is limited and would not be appropriate to assess students at all stages of a full introductory programming course. In an actual course setting, multiple forms of assessment are needed to accurately and consistently measure a student's *strategy* skill. Novices can be encouraged to apply specific *strategies* in assignment instructions and exam questions. The marking criteria used to judge assignments and exams can openly test for use of particular *strategies* and reward their use with credit. Assigning marks for application of *strategies* in assessments and exams will hopefully encourage students to value this component of the curriculum, devoting study time to programming *strategies*.

7. Teaching and Assessing Programming Strategies Explicitly in an Actual Setting

"Good fortune is what happens when opportunity meets with planning."
Thomas Alva Edison

Overview

Previous experiments, described in earlier chapters, led to the following conclusions.

- Under a curriculum relying on implicit instruction of programming *strategies*, novices created solutions that demonstrated strategy-related flaws (chapter 4).
- A representation of sub-algorithmic programming *strategies* used by experts can be expressed in a form that is appropriate for incorporation in an introductory programming curriculum (chapter 5).
- It is possible to incorporate programming *strategies* explicitly into an introductory programming curriculum with observable effects on novices including use of *strategies* in solutions, confidence in solutions and use of a vocabulary of *strategies* (chapter 6).

This chapter describes how programming *strategies* were explicitly incorporated and assessed in an actual introductory programming course. The inclusion of explicit programming *strategy* content began in the second half of 2005 and was refined over a two-and-a-half-year period.

As well as describing how this integration was achieved, comparisons are made between the outcomes of novices under the new curriculum, which included explicit programming *strategies*, and results of novices learning under an implicit-only *strategy* curriculum, as discovered in the initial experiment (chapter 4). Also measured is the relationship between the programming *knowledge* and programming *strategy* components of the course. This is achieved by comparing student results in assessment items that targeted each area independently.

Strategies were successfully integrated into an actual course curriculum and assessed in assignments and examinations. Measurement of novices' *strategy* skill under the new curriculum showed improvement over the benchmark set under the previous traditional curriculum. It was also found that student performance was consistent between *knowledge* and *strategy* examination questions, validating the *strategy* questions used.

7.1 Introduction

An initial experiment showed that novices learning programming *strategies* implicitly created solutions that contained strategy-related flaws (chapter 4). Only a single student out of 42 was able to achieve a complete solution containing all expected *plans* for a classic averaging problem. Overall, students applied only 57% of the expected *plans*.

The presence of these flaws indicated possible weaknesses in the curriculum used to instruct the novices in programming *strategies*. To leverage the potential of explicit instruction (see section 2.3.6) an authentic representation of programming *strategies*, capable of being expressed explicitly to novices, was sought. *Plans* used by Soloway (1986) are a model of programming *strategies* that are supposedly based on the tacit understanding of experts. An experiment was conducted to validate that *plans* are a representation of the sub-algorithmic *strategies* used by experts (chapter 5). This experiment showed that *plans* are used by expert programmers.

Taking *plans* as an authentic representation of programming *strategies*, an experiment was conducted (chapter 6) that compared two curricula: one including programming *strategies* explicitly and a traditional curriculum that required students to learn *strategies* implicitly. The curricula were delivered in an artificial setting. The experiment showed that it is possible to incorporate *strategies* explicitly into a curriculum. Results demonstrated that experimental participants, who had been exposed to explicit *strategy* instruction, used *strategies* more than control group participants (though not significantly so). Novices exposed to explicit *strategy* instruction used a vocabulary of *strategies* to describe their solutions and showed greater confidence than those exposed to a traditional curriculum.

The previous experiment was conducted in an artificial setting with a minimal curriculum. Only a limited set of *strategies* were incorporated, and a greater set is needed for a full introductory programming course held over a semester. A larger set of programming *strategies* needs to be expressed and a method of explicitly incorporating these *strategies* into a full curriculum needs to be developed and tested.

The main testing approach used to gauge *strategy* application in previous experiments was Goal/Plan Analysis. With novices, this approach is limited to analysing solutions generated at or near the end of an introductory programming course. After the previous experiment (chapter 6) it was proposed that analysis of *strategy* skill should be conducted in more flexible ways throughout the course by taking the ideas inherent in Goal/Plan Analysis and using them to assess student work in assignments and examinations. The following are ways *strategies* could be incorporated in assignments and examinations.

- Encouraging students to use particular *strategies* when generating solutions for assignments
- Awarding credit for application of *strategies* in assignment marking criteria
- Using problems that focus on programming *strategies* as part of the final examination
- Analysing examination solutions in a Goal/Plan-Analysis-like manner

As well as encouraging the learning benefits suggested in literature and discovered in the previous experiment, awarding credit for applying *strategies* in assessments may encourage students to value this important component of programming and devote more effort to learning it.

The Leeds group (Lister et al., 2004) attempted to isolate the cause of poor novice results measured by the McCracken group (McCracken et al., 2001). The Leeds group reported that many instructors attribute poor results to poor problem-solving ability in novices. The group attempted to create programming questions that required no problem-solving ability to answer. They felt that if novices succeeded in the test it would confirm that novices can successfully acquire programming *knowledge* so instructors could put this issue aside and focus their attention on improving *strategy* instruction. If novices failed this test, it would indicate a failure in programming *knowledge*. The results of the Leeds group study, and the BRACElet project (Whalley et al., 2006) that followed, showed that many novices exhibit a fragile programming *knowledge* and very few can demonstrate programming *strategy* understanding in a *comprehension* exercise.

After the McCracken, Leeds and BRACElet series of experiments, some concerns still remain. It is not clear if programming *knowledge* instruction must precede *strategy* instruction. It is not clear whether a student with a flawed programming *knowledge* can learn and understand programming *strategies*. If it is possible to isolate programming *knowledge* ability from *strategy* ability it may be possible to define a clearer relationship between these aspects.

7.1.1 Participants

Participants in the current experiment were students studying in a first-year introductory programming course at the University of Southern Queensland. Results shown in this chapter were taken from two cohorts, those studying in second semester of 2005 and those studying in the second semester of 2007.

Participants included a mix of students attending on-campus classes and those studying externally. Results were drawn from examinations undertaken by all students. The size of the 2005 and 2007 cohorts was lower than the 2003 cohort who participated in the initial experiment described in chapter 4, however only on-campus students were used in the initial experiment, so participant numbers are similar.

- 2003 cohort included 42 participants
- 2005 cohort included 36 participants
- 2007 cohort included 45 participants

Participants included school leavers and mature-aged students. Students come from a range of discipline areas but were primarily IT and Engineering students. The mix of students and entry standard had not changed since the initial experiment.

7.1.2 Setting

The setting of the experiment described in this chapter is essentially the same as that of the initial experiment described in chapter 4, except for the inclusion of explicit *strategy* instruction in course materials and assessment items. These changes are described in detail in section 7.3 and 7.4 respectively. The instructor was the researcher and author of this dissertation.

7.2 Research Questions

The current experiment was motivated by the following questions which are answered in section 7.7.

This section is divided into three parts related to the three perspectives taken when conducting this experiment. This three-perspective structure is mirrored in the Methodology, Results and Discussion sections of this chapter.

7.2.1 Integration

The first two questions of this chapter consider the possibility of instructing and assessing programming *strategies* explicitly. Although this has been established on a smaller scale in an artificial setting, it needs to be tested with a complete curriculum in an actual introductory programming course.

RQ11. *Can instruction of programming strategies be explicitly incorporated into instruction in an actual introductory programming course?*

RQ12. *Can programming strategy skill be measured as part of the assessment in an actual introductory programming course?*

7.2.2 Impact

The third question relates to the effect of introducing explicit programming *strategies* to novice programmers. This question will be answered by analysing novice performance on assessments in the course and comparing this to the baseline performance described by the initial experiment (from chapter 4).

RQ13. *What is the impact on novice programmers of incorporating programming strategy explicitly into instruction and assessment?*

7.2.3 Consistency of Knowledge and Strategy Skill

The final question asks if it is possible and appropriate to separate assessment items that relate to *knowledge* from those that relate to *strategies*. This is done by comparing results of assessment items independently covering *knowledge* and *strategy*, and checking they are consistent for novices. This comparison may also shed light on the relationship between *knowledge* and *strategy* skill.

RQ14. *Are novices' results in assessment of programming strategies consistent with their results in assessment of programming knowledge?*

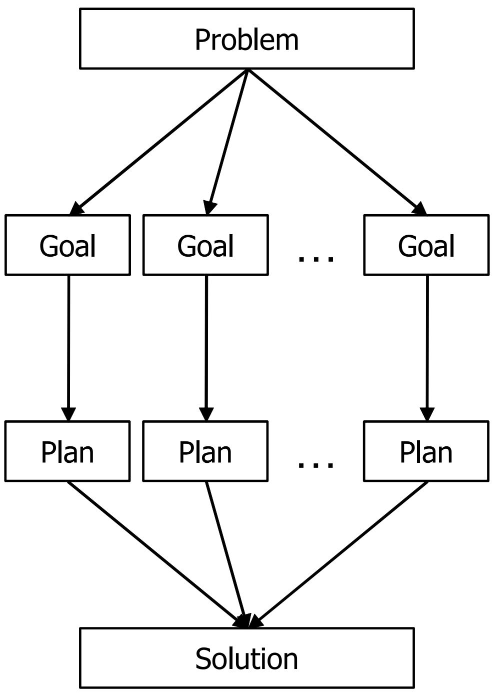
7.3 Integrating Strategy Instruction into Written Materials, Lectures, Tutorials and Practical Classes

Over the two-and-a-half-year period between the second half of 2005 and the end of 2007, programming *strategies* have been incorporated into the curriculum of an introductory programming course at the University of Southern Queensland.

The course is delivered to students on campus (approximately 40% of the student cohort) and students studying externally (potentially anywhere in the world). On-campus students are expected to attend two one-hour lectures followed later in the week by an hour-long tutorial (in a normal classroom) and a two-hour practical class with computers. External students study independently by reading the same written materials, accessing lecture slides with audio online, and undertaking tutorial and practical exercises at home.

1.6.1 Design

An expert programmer will take time to properly design a solution. It is tempting to jump to implementation, but often, without a reasonable design, a programmer can waste time correcting a poor implementation and take far longer than if they had spent a small amount of time on design first.



From a *problem statement* a programmer will identify the *goals* that need to be achieved. These goals can usually be found through a careful reading of the problem statement.

STRATEGY

When the goals of the problem have been identified, a programmer can choose appropriate *plans* that satisfy goals. A plan is a small, independent strategy that the programmer has applied in a past solution. During this course we will be covering programming knowledge and also the strategies that you can use to apply this knowledge. Look for the STRATEGY sidebar to differentiate parts of this book that cover strategies.

Once plans have been identified they need to be combined together to form a solution. Plans can be combined together in three possible ways.

- **Abutment**
  Placing the plans one after another in the correct sequence that will solve the problem.
- **Merging**
  Integrating plans so that common parts are performed together
- **Nesting**
  Placing one plan inside another plan

Figure 7.1. Introduction to *strategies* from the Study Book

Programming *knowledge* is presented in a similar manner to the traditional curriculum presented with the first experiment (described in chapter 4). *Strategies* are interwoven through the course in an explicit manner. In the beginning of the course the distinction between *knowledge* and *strategies* is presented. Figure 7.1 shows an initial description of *plans* as *strategies* within a description of the programming process. *Strategies* are a part of the curriculum and testing students' *strategy* skills forms part of the assessment. Students are informed of this.

Written materials provided to students include notes for each module of the course and exercises for each week. Students are encouraged to read the written materials before attending or listening to lectures provided online (with audio for external students). The lectures complement the written materials and allow opportunities for questions and further explanations. Each week students are expected to undertake written and computer-based exercises, in tutorials and practicals, to reinforce the material for the week.

The following sections describe how programming *strategies* were explicitly incorporated into written materials, lectures and weekly exercises. Assessment of students' *strategy* skills in assignments and in the course examination is described in section 7.4.

7.3.1 Strategy Guide

The major component of written material provided to novices in the course is referred to as a 'Study Book'. The bulk of the Study Book is divided into modules, with one module being covered each week of the course. More detail about the Study Book modules is given in section 7.3.2 below. At the end of the study book two appendices are given: one is a syntax guide and the other collects together all the *strategies* that are covered in the course. This 'Strategy Guide' is included in this dissertation as Appendix A.

The Strategy Guide begins by defining how *strategies* can be integrated. *Abutment*, *nesting* and *merging* are discussed in this introduction. Each *strategy* is then described as a *plan* (some later *strategies* are basic algorithms). The programming *knowledge* required to apply each *plan* is stated at the beginning of each *plan* description. Examples and diagrams are provided for most *strategies*. This Strategy Guide forms a resource for novices studying in the course, and possibly after they have completed the course. All *strategies* assessed in assignments and the examination can be found in this guide; students are told this at the beginning of the course and before the examination. *Strategies* are addressed individually in the modules of the Study Book and lectures, often with a different context or example.

The Strategy Guide contains 18 *plans* ranging in scale from very simple *plans* such as finding an average, through several sub-algorithmic *plans* such as a triangular swap (see Figure 7.2 below for this example), and on to some algorithmic *plans* such as sorting. The Strategy Guide has developed and been refined during its use and in future it should grow and be modified as the need arises. The 18 *strategies* currently in the Strategy Guide are listed below.

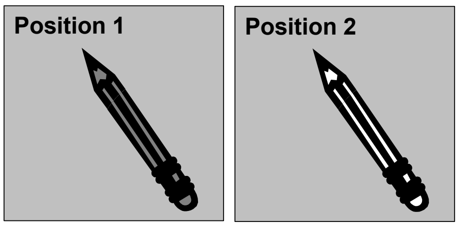
1. Average plan
2. Divisibility plan
3. Cycle Position plan
4. Number Decomposition plan
5. Initialisation plan
6. Triangular Swap plan
7. Guarded Exception plans (including Guarded Division plan)
8. Counter-Controlled Loop plan
9. Primed Sentinel-Controlled Loop plan
10. Sum and Count plans
11. Validation plan
12. Min/Max plans
13. Tallying plan
14. Search algorithm

15. Bubble Sort algorithm
16. Command Line Arguments plan
17. File Use plan
18. Recursion plans (single- and multi-branching)

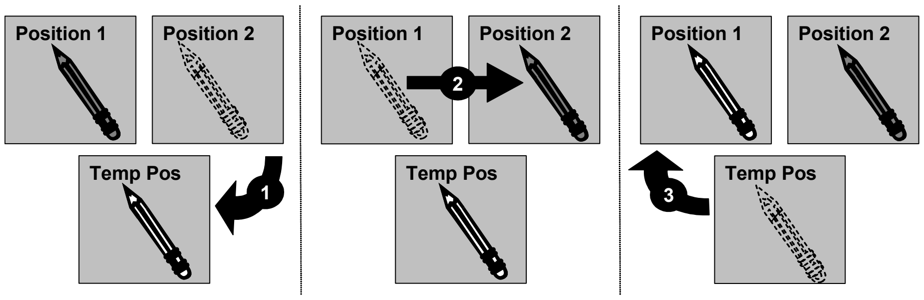
Plan 6. Triangular Swap Plan

This plan requires an understanding of variables and the assignment operator.

Consider how you swap two items. Imagine two pencils in front of you. To swap their positions you would pick up one with one hand, the second with your other hand and then place each in their new positions.



A computer can only perform one action at a time. Now, imagine that you only have one hand; how would you swap the positions of the two pencils now? Keep in mind also that when a variable is assigned a new value, the old value is replaced and cannot be accessed later. Attempting to swap using the above method will result in two copies of the same value.



To achieve a swap a temporary position is needed. One of the pencils could be moved to the temporary position; the second pencil could be moved to its new location; finally the first pencil could be moved from the temporary position to its new position.

Here is an example in the context of a full program.

```
#include <stdio.h>

int main() {
      int firstPosition  = 5; // First position containing value to swap
      int secondPosition = 6; // Second position containing value to swap
      int tempPosition;       // Temporary position for swap

      // Output the numbers after the swap
      printf("Before Swap...\n");
      printf("First: %i, Second: %i\n", firstPosition, secondPosition);

      // Swap the two numbers in a triangular swap
      // 1. Copy the value from the second position to temp
      tempPosition = secondPosition;

      // 2. Copy the value from the first position to the second
      secondPosition = firstPosition;

      // 3. Copy the value from the temp position to the first
      firstPosition = tempPosition;

      // Output the numbers after the swap
      printf("After Swap...\n");
      printf("First: %i, Second: %i\n", firstPosition, secondPosition);
}
```

Here is the output of the above program.

```
Before Swap...
First: 5, Second: 6
After Swap...
First: 6, Second: 5
```

The above results show the values are swapped and not duplicated.

Figure 7.2. An example of a *plan* from the Strategy Guide

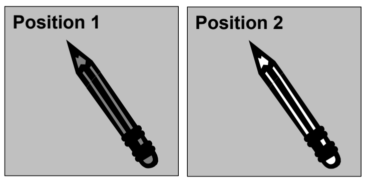
7.3.2 Explicit Incorporation in Written Notes

Within the 12 modules of the Study Book, programming *strategies* are introduced after presenting the programming *knowledge* applied in each *strategy*. In this context the *strategies* show immediately how the *knowledge* can be applied, which, in its purest sense, is the nature of a *strategy*.

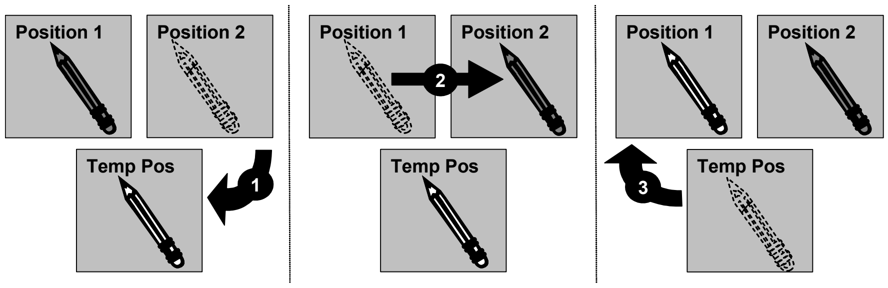
STRATEGY

3.3.4 Triangular Swap Plan

Consider how you swap two items. Imagine two pencils in front of you. To swap their positions you would pick up one with one hand, the second with your other hand and then place each in their new positions.



A computer can only perform one action at a time. Now imagine that you only have one hand; how would you swap the positions of the two pencils now? Keep in mind also that when a variable is assigned a new value, the old value is replaced and cannot be accessed later. Attempting to swap using the above method will result in two copies of the same value.



To achieve a swap a temporary position is needed. One of the pencils could be moved to the temporary position; the second pencil could be moved to its new location; finally the first pencil could be moved from the temporary position to its new position.

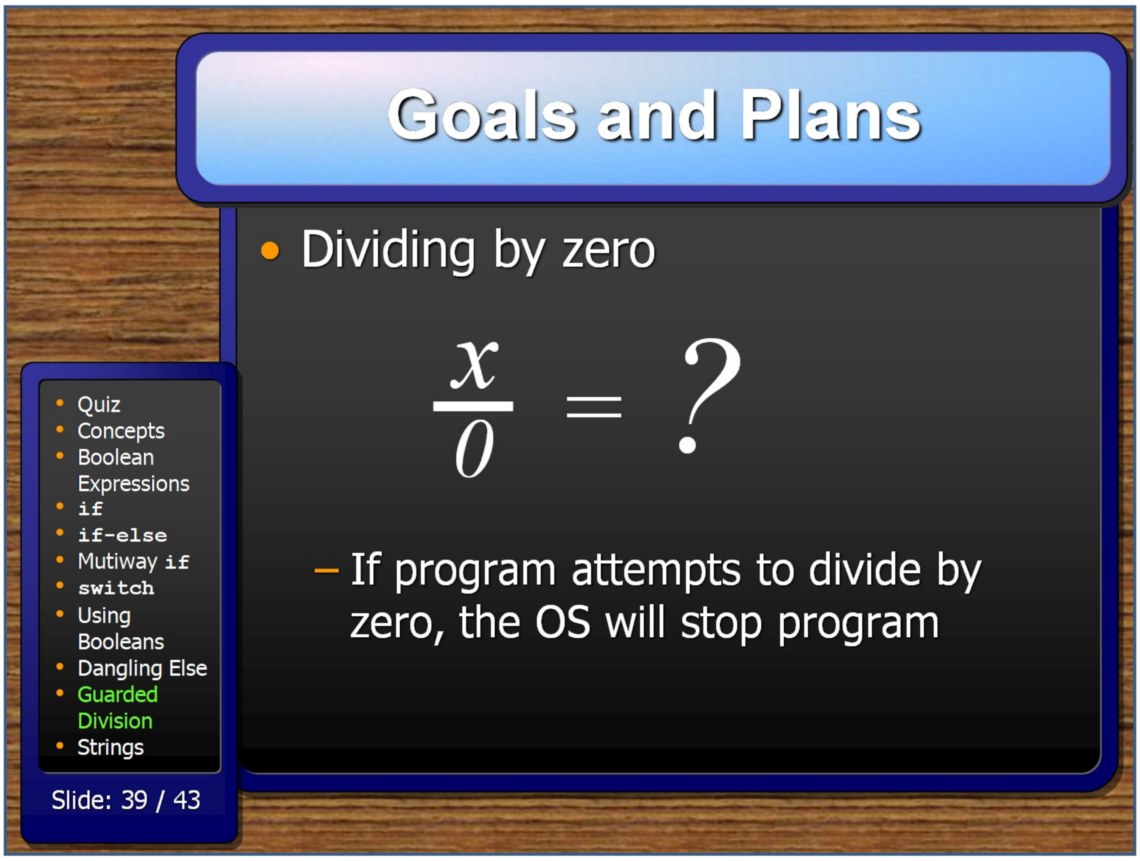
Figure 7.3. An example of part of a *strategy* from the Study Book within a teaching module

Figure 7.3 shows an extract from the Study Book including the same Triangular Swap plan shown previously. This is followed by a code example showing the *plan* applied. Note the bar down the left that distinguishes parts of the Study Book as covering a programming *strategy*; other parts of the Study Book, covering programming *knowledge* and other content, do not show this bar.

The Triangular Swap plan is shown after students cover variables and assignment as programming *knowledge* components. This takes place in the third module, which is covered during the third week of the course. This *plan* is discussed in lectures, reinforced in tutorial and practical exercises and assessed in assignments and the examination. The Triangular Swap plan appears again when the Bubble Sort Algorithm is presented in a later module of the course. This demonstrates how identifying *strategies* and creating a vocabulary for *strategies* can allow instructors to use this vocabulary, and in doing so, reinforce *strategies* when they appear later in the course.

7.3.3 Explicit Incorporation in Lectures

During lectures, *strategies* are presented and discussed after relevant programming *knowledge* content had been covered. Lectures are presented in person to a class of on-campus students. The lecture is also recorded and the slides and audio are presented together as a 'Breeze' flash presentation and placed on the course website.



Goals and Plans

- Dividing by zero

$$\frac{x}{0} = ?$$

– If program attempts to divide by zero, the OS will stop program

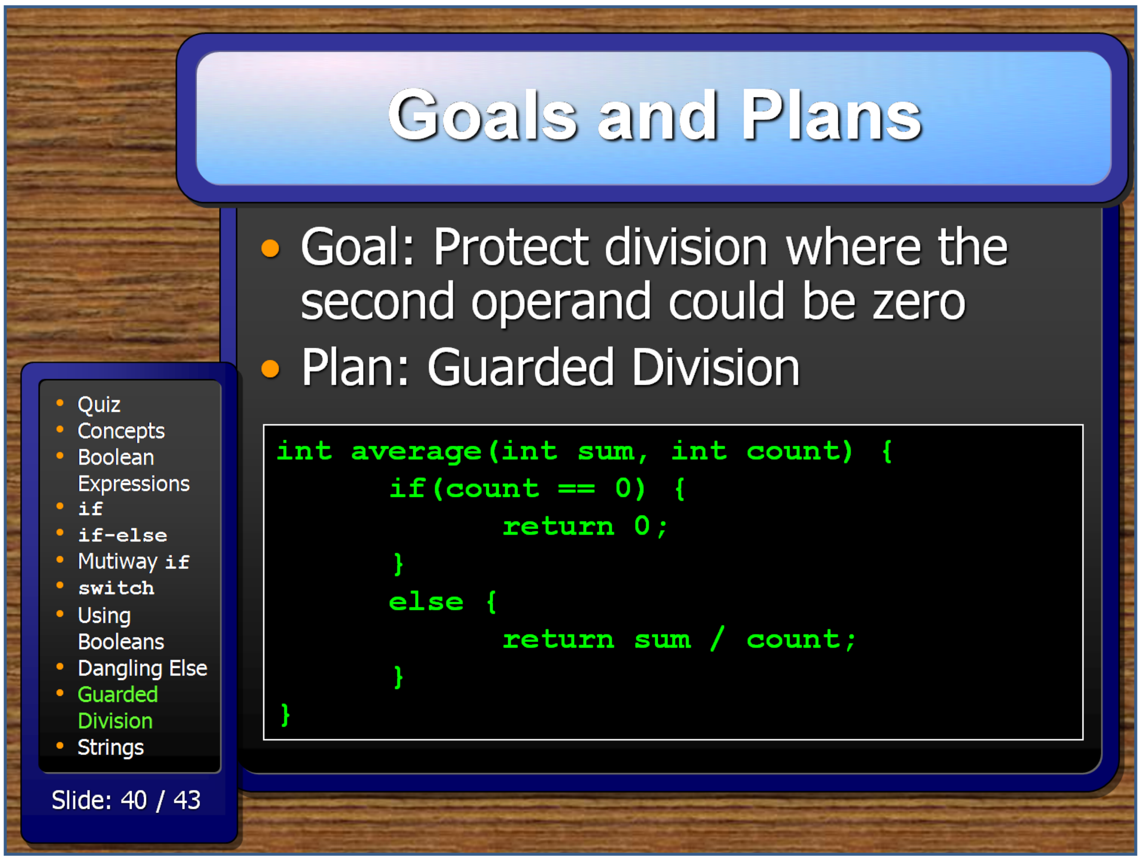
- Quiz
- Concepts
- Boolean Expressions
- `if`
- `if-else`
- Mutiway `if`
- `switch`
- Using Booleans
- Dangling Else
- Guarded Division
- Strings

Slide: 39 / 43

Figure 7.4. Example of a lecture slide showing the Guarded Division plan (slide 1 of 2)

The example shown in Figure 7.4 is one of two related slides. On the left of the slide the outline of the lecture is shown and the current topic, 'Guarded Division', is highlighted. Observe that much of the previous content of the lecture has covered programming *knowledge*. Before a guarded division can be applied novices must be aware of the **`if`** statement and the division operator (covered in a previous module). In the following slide (shown in Figure 7.5 below) students are shown how to apply this plan[2]. This *strategy* is reinforced in the tutorial class held later that same week and is assessed in assignments and sometimes in the examination.

[2] The Guarded Division plan is covered early in the course (at week 3) so the implementation shown in Figure 7.5 when the divisor is zero is a naive one. The function returns the average if a non-zero count is given, but when the count is zero the function returns zero. Ideally an exception would be generated in response to this event.



Goals and Plans

- Goal: Protect division where the second operand could be zero
- Plan: Guarded Division

```
int average(int sum, int count) {
      if(count == 0) {
            return 0;
      }
      else {
            return sum / count;
      }
}
```

Figure 7.5. Example of a lecture slide showing the Guarded Division plan (slide 2 of 2)

7.3.4 Strategies in Tutorial and Practical Exercises

Strategies are practiced in Tutorial and Practical classes. Exercises for these classes are listed in the Study Book.

13. Fill in the blanks in the following code which swaps the values of two character variables and then outputs the variables new values.

```
#include <stdio.h>

int main() {
     char letter1 = 'a'; // First letter
     char letter2 = 'b'; // Second letter
     char temp = '-';    // Temporary position

     // Swap the two letters in a triangular swap
     ______________________________
     ______________________________
     ______________________________

     // Output the letters
     ______________________________
}
```

Figure 7.6. Example exercise from Module 3 requiring Triangular Swap plan

The example shown in Figure 7.6 above requires students to apply a Triangular Swap plan to swap two character values. The *plan* name is mentioned explicitly in the code (in a comment) and three blanks imply the use of the triangular swap. Later in the course this *strategy* is used again in an exercise where students write a function that takes two pointers and orders the values to which they point.

Computer Exercises

6. Copy the *Guarding Division* function example from page 15 that will calculate an average. Add a **`main()`** function that will call the **`average()`** function. It should still work when the value passed to **`count`** is zero.
   6.1 Remove the guarding **`if-else`** stateme nt so all that remains in the function is the **`return`** statement. Now test the function sending zero as the value of **`count`**. When the program is compiled and run, the operating system should shut the program down and display an error.
   6.2 Restore the guard to the function and test that it works correctly again.

Figure 7.7. Example exercise from Module 5 testing the Division by Zero plan

Figure 7.7 contains an example of an exercise that asks the students to experiment with the Guarded Division plan. This exercise encourages novices to experience the consequences (a program crash) resulting from dividing by zero. Through this, novices will hopefully come to appreciate the necessity of protecting the division with a guard.

In a description from an earlier instance of the same course, prior to adding *strategy* content explicitly, the following exercise was given as an example.

> *Write a program that will allow the user to enter words. Use the* **`%s`** *format sequence in a* **`scanf()`** *call to capture each word one at a time. Find the length of each word using* **`strlen()`**. *To end the user input, the user will enter the string* **`"end"`**. *At the end of the program, output the count of words and the average length of the words.*

This example was used in section 4.1.2 to demonstrate how novices were expected to learn programming *strategies* implicitly in order to solve problems. The problem statement describes what needs to be achieved, but does not suggest how a solution should be constructed. As a contrast, a new version is shown in Figure 7.8 below. In the new version students are given the same initial requirement with a few programming *knowledge* embellishments (such as the size of an array). Following this, in the third and fourth paragraphs of the problem statement, *strategy* instructions are given. Students are expected to use a Primed Sentinel-Controlled Loop to achieve repetition; this *plan* is named and its use is directed. The students are also reminded to guard the division when calculating the average. Note that students are expected to know what a sentinel-controlled loop and guarded division are at this stage. This problem relies on students possessing a vocabulary that includes the term ‘sentinel’, which is used to define the value that, when encountered, will stop the repetition.

Students are deliberately led to practise application of particular *strategies* for these problems in the same way that an instructor might encourage students to use a particular language construct, such as a **`for`** loop. In the examination, students are expected to apply required *strategies* without being led in this manner.

Computer Exercises

8 Write a program that will allow the user to enter words. Use the **`%s`** format sequence in a **`scanf()`** call to capture each word one at a time (this will skip whitespace between words). You don't have to keep the user inputs in memory; you only need to deal with each word one at a time. Create an array with 256 characters for the input word. Set the maximum word size as a constant.

Find the length of each word using **`strlen()`**. To end the user input, the user will enter the string "end" (you will have to use **`strcmp()`** to test for this). You will need to include **`string.h`** to use these functions. Set the sentinel word as a constant.

At the end of the program, output the count of words, the total number of letters and the average length of the words. Be sure to use a sentinel controlled loop and guard the calculation of the average word length. Keep all numeric values as integers.

Your program should work if several words are entered before the sentinel, or if the sentinel is entered as the first input. Test your program by entering "end" as the first word. Try entering more than one word per line of input.

Figure 7.8. Example exercise from Module 8 requiring the Sentinel-Controlled Loop and Guarded Division plans. Highlighting (added for this figure only) shows *strategy* content

7.4 Assessing Strategy Skill in Assignments and Examinations

As well as being introduced explicitly into instructional materials, programming *strategies* also became explicitly assessed in the course. This section describes how programming *strategies* have been included in assignment instructions and marking criteria as well as how examinations have been designed and marked to include testing of strategy-related abilities.

When teaching *strategies* explicitly, the challenge for instructors is to create problems that focus on particular programming *strategies*. Achieving this allows novices to demonstrate specific *strategies* in assignments and the examination.

7.4.1 Assignment Instructions

In assignment instructions students are given tasks that require them to apply specific programming *strategies*. Figure 7.9 below is an extract from an assignment's instructions where students are asked to use a Primed Sentinel-Controlled Loop to input characters entered by a user until the end-of-line is encountered.

- In your program, create the following functions.

…

```
void decryptEncryptLine(int shift);
```

- This function will shift alphabetic characters by the amount of the shift. The function performs in the same manner for encryption and decryption. If the shift is a positive amount, this will shift characters forward (encrypt characters) and if negative it will shift them back (decrypt characters).
- The function will input and process each character one at a time until a newline character is detected. Use a **primed sentinel controlled loop**. Do not try to store or process entire lines.

Figure 7.9. An extract from the instructions for a programming assignment highlighting the requirement for a specific programming *strategy*

7.4.2 Assignment Marking Criteria

As well as requiring specific *strategies* to be applied in the creation of solutions, the marking schema used to evaluate solutions also explicitly includes references to specific *strategies*.

In the course described here students participate in electronic peer-review as part of each assignment. Each marking scheme is constructed well in advance and released as part of the assignment instructions. Students are therefore aware of how their submission will be judged before they submit. They can see that they will receive marks for applying specific programming *strategies*. Being involved in peer-review, students are also expected to be able to judge if a peer-student has correctly applied a specific *strategy* where required by criteria.

Criteria relating to programming *strategies* are mixed with other criteria in each marking scheme. Figure 7.10 below is an extract from the marking scheme for the same assignment that was used in the previous section.

…

Check that no variables are declared outside functions. This does not include global constants.

☐ **A Primed Sentinel Controlled Loop is used to process menu options in the `main()` function**
The function should contain a priming input before the loop and a subsequent input at the end of the loop. If the user enters the quit option in the first instance, the loop body should not be entered.

☐ **A Primed Sentinel Controlled Loop is used to gather characters for input until the end of a line in the `decryptEncryptLine()` function**
The function should contain a priming input before the loop and a subsequent input at the end of the loop. If the user enters a blank line, the loop body should not be entered.

☐ **Code is indented consistently and no line is longer than 80 characters**

…

Figure 7.10. An extract from the marking scheme stating that a particular programming *strategy* is required in the solution for a programming assignment

7.4.3 Examination Questions

The philosophy used to create questions for the examination is based on the aspects listed in section 2.3. Questions attempt to distinguish abilities in *knowledge* and *strategies*, and separately test *comprehension* and *generation*. As noted in section 2.3.5, by combining these aspects, four types of question can be defined as shown in Figure 7.11 (a reproduction of Figure 2.1).

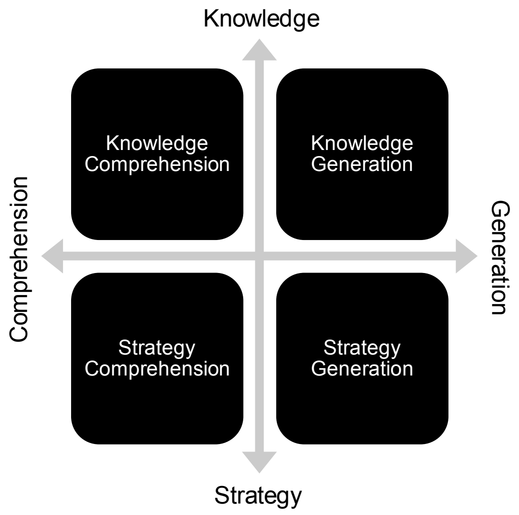


Figure 7.11. Four types of examination questions based on novice instruction aspects

Targeting questions to one of these four areas is not always simple. Some questions may stray over the boundaries between areas. The focus of the question can be reinforced by criteria defining how the answer is awarded marks (see section 7.4.4).

Knowledge-Comprehension Questions

In order to test *knowledge* and *comprehension* an examination question must focus on language syntax but not ask the novice to generate any code. The question should test that the student understands an example shown to them, possibly by simulating how the code would be executed. An example of a knowledge-comprehension examination question is shown in Figure 7.12 below.

QUESTION 1 **(10 marks, 12min)**

What will the following output?

```
#include <stdio.h>

int testFunc(int *ptr, int num);

int main() {
      int x=7, y=3, z=5;
      printf("%i %i\n", x, y);
      z = testFunc(&y, x);
      printf("%i %i %i\n", x, y, z);
}

int testFunc(int *ptr, int num) {
      int temp;
      printf("%i %i\n", *ptr, num);
      temp = num;
      num = *ptr;
      *ptr = temp;
      printf("%i %i\n", *ptr, num);
      return num + (*ptr);
}
```

Figure 7.12. Example of a Knowledge-Comprehension examination question

Knowledge-Generation Questions

Knowledge-generation questions should require novices to generate code but not solve a problem requiring any programming *strategies*. The question should prompt the novice to create code that demonstrates their understanding of specific language constructs. An example of such a question is given as Figure 7.13 below.

QUESTION 4 **(10 marks, 17min)**
Write a `main()` function that input an integer from a user and then use a `switch` statement to respond to the user's input with one of the following outputs:
 Where `0` is entered, output `hello`
 Where `1` is entered, output `bye`
 Where any other value is entered, output `invalid`

Figure 7.13. Example of a Knowledge-Generation examination question

Strategy-Comprehension Questions

Strategy-comprehension questions are perhaps the most difficult to define. These questions must test the *strategy* potential of a novice without asking them to generate any code. Possible ways to achieve this include the following.

- Asking novices to identify or describe *strategies* used in a given solution
- Asking novices to relate common *strategies* applied across multiple solutions
- Asking novices to identify how a *strategy* has been incorrectly applied in, or is absent from, a solution

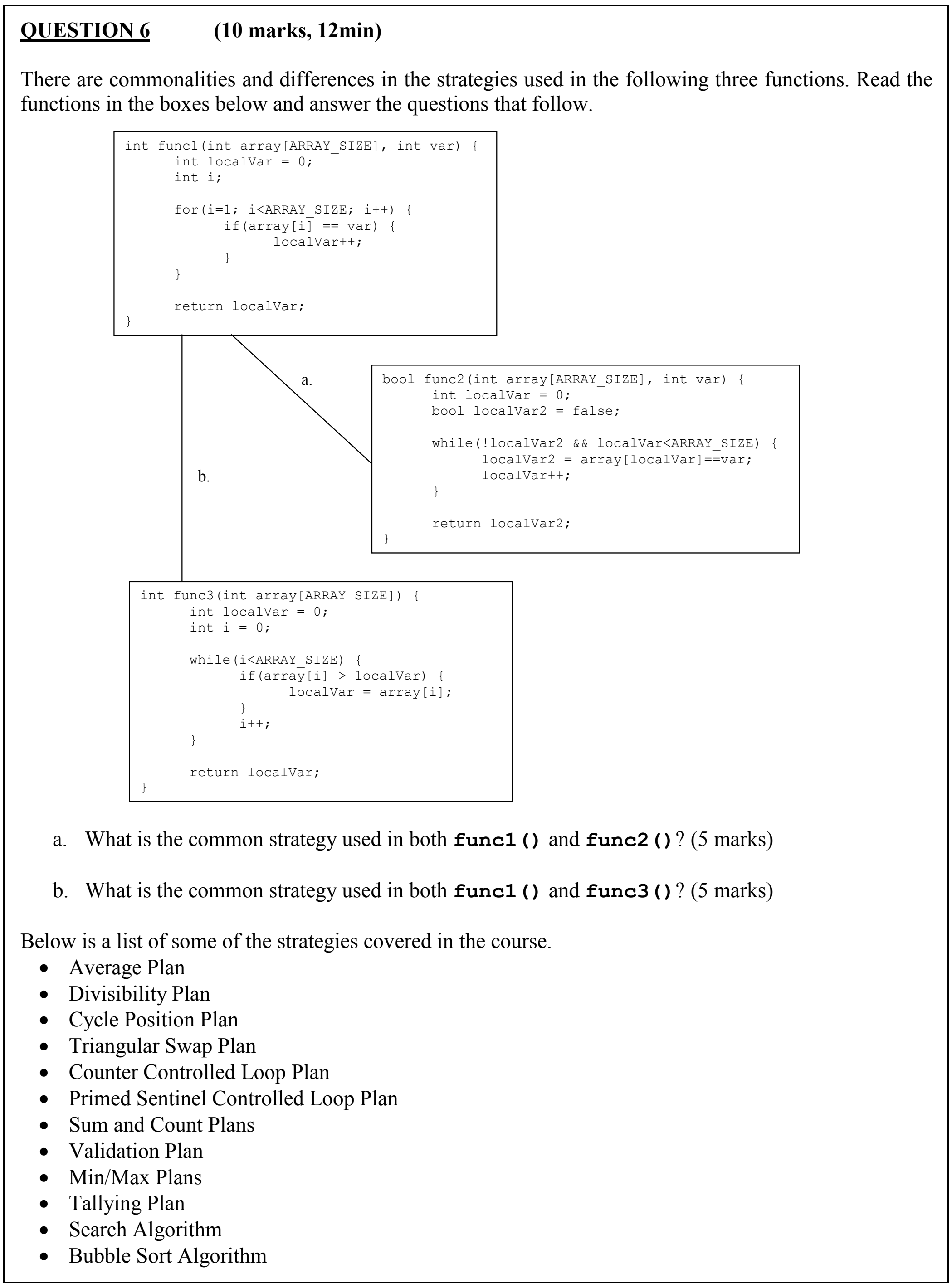
QUESTION 5 **(5 marks, 18min)**

The following function contains a logic error. In a few words, describe what the error is **and** how you would remedy the error. Do not re-write the whole function.

```
int getAverage(int sum, int count) {
      return sum/count;
}
```

Figure 7.14. Example of a Strategy-Comprehension examination question

In Figure 7.14 we see an example of a strategy-comprehension question that asks the novice to identify the strategy-related error in the code and state how the error could be corrected. The error can occur when the argument count has a value of zero, which would cause a division by zero. There is no guard to protect against this. To remedy this problem the student should apply a guard against division by zero. The exact ‘Guarded Division’ terminology is not critical if the novice can express this solution using other words.



QUESTION 6 **(10 marks, 12min)**

There are commonalities and differences in the strategies used in the following three functions. Read the functions in the boxes below and answer the questions that follow.

```
int func1(int array[ARRAY_SIZE], int var) {
      int localVar = 0;
      int i;

      for(i=1; i<ARRAY_SIZE; i++) {
            if(array[i] == var) {
                  localVar++;
            }
      }

      return localVar;
}
```

a.

```
bool func2(int array[ARRAY_SIZE], int var) {
      int localVar = 0;
      bool localVar2 = false;

      while(!localVar2 && localVar<ARRAY_SIZE) {
            localVar2 = array[localVar]==var;
            localVar++;
      }

      return localVar2;
}
```

b.

```
int func3(int array[ARRAY_SIZE]) {
      int localVar = 0;
      int i = 0;

      while(i<ARRAY_SIZE) {
            if(array[i] > localVar) {
                  localVar = array[i];
            }
            i++;
      }

      return localVar;
}
```

a. What is the common strategy used in both **`func1()`** and **`func2()`**? (5 marks)

b. What is the common strategy used in both **`func1()`** and **`func3()`**? (5 marks)

Below is a list of some of the strategies covered in the course.
- Average Plan
- Divisibility Plan
- Cycle Position Plan
- Triangular Swap Plan
- Counter Controlled Loop Plan
- Primed Sentinel Controlled Loop Plan
- Sum and Count Plans
- Validation Plan
- Min/Max Plans
- Tallying Plan
- Search Algorithm
- Bubble Sort Algorithm

Figure 7.15. An example of a Strategy-Comprehension examination question

Figure 7.15 shows a second example of a strategy-comprehension question. Here novices are asked to identify the *strategies* used in the three given functions and pick the common *strategies* between the identified pairs. This is a challenging problem as it requires novices to identify *strategies* even when they may be applied using different syntax within the code.

Strategy-Generation Questions

Strategy-generation questions are probably what most instructors think of when they write a *generation* question in an examination. It is important, though, that problems allow novices to apply specific *strategies* they have learned in the course.

QUESTION 7 **(20 marks, 24min)**

Write a function, using the following prototype, which will prompt the user and read in a valid positive integer. If the user enters invalid input, or a negative integer, the function will tell them their input was invalid and prompt them to enter another value. The function will repeat this until the user enters a valid input.

```
int getValidPositiveInteger();
```

For your reference, the following lines of code will clear the standard input stream.

```
scanf("%*[^\n]");
scanf("%*c");
```

QUESTION 8 **(20 marks, 24min)**

Write a `main()` function that will read in integers and output their average. Input will be gathered using the `getValidPositiveInteger()` function as described above (do not re-write that function). Stop reading when the value 99999 is entered (this is not to be used as an input).

Figure 7.16. Another example of Strategy-Generation examination questions

Figure 7.16 gives an example of two questions that formed a series from an examination. The first question asks the novice to demonstrate a Validation plan. The Validation plan involves a Sentinel-Controlled Loop plan where a valid input is the sentinel.

The second question is essentially the same classic averaging problem defined by Soloway (1986) and used in the initial experiment shown in chapter 4. This question avoids the pitfalls found when this question was used in earlier experiments (chapters 4, 5 and 6): inputs are validated by the function they have attempted to answer earlier (in question 7) and it is clear that the sentinel should not be used as an input. This question requires novices to apply the following *plans*, each of which is covered explicitly in the course.

- Primed Sentinel-Controlled Loop plan
- Sum plan
- Count plan
- Guarded Division plan
- Average plan
- Output plan

7.4.4 Marking the use of Strategies in the Examination

When assessing the use of *strategies* in an examination it is critical that the marking scheme does not fall back on syntactical measures. The marking criteria for *strategy* related questions should seek the application of specific *strategies* or *comprehension* of those *strategies*. Strategy-generation questions should target specific *strategies* and the marking scheme for these question should award marks where the required *strategies* have been applied, rather than for syntactical correctness.

Distinguishing how knowledge-related and strategy-related questions are marked forces a greater focus on particular areas from Figure 7.11 at the beginning of this section.

7.5 Methodology

The experimentation described in this chapter can be considered from three perspectives, which can be related back to the research questions stated earlier:

- to test the possibility of explicitly incorporating and assessing programming *strategies* in an actual introductory programming course (RQ11 and RQ12);
- to measure the impact of explicit programming *strategy* instruction and assessment on novices by comparing results produced under the new curriculum with benchmark measurements from the initial experiment (RQ13); and
- to measure the validity of strategy-related questions used to assess *strategy* skill in students undertaking the new curriculum (RQ14).

The method for achieving these three aims is described in the following sub-sections.

7.5.1 Integration

The first and second research questions (RQ11 and RQ12) raised in section 7.1.1 consider the possibility of integrating *strategy* content into an actual introductory programming course. The success of this integration, drawing on examples presented earlier, is discussed in section 7.6.1. Observations are made on student response to the newly incorporated materials and assessment.

7.5.2 Impact

The third research question (RQ13) seeks to measure impact of the new curriculum relative to curriculum measured in the initial experiment (chapter 4). Students who participated in the initial experiment had studied using a curriculum that required them to learn *strategies* implicitly. In the initial experiment students were asked to create a solution to a classic averaging problem. Several *strategy* gaps were detected in student solutions indicating flawed understandings of the required *strategies*. Of particular interest was the lack of application of a Guarded Division plan.

Comparison of performance under the new curriculum with the benchmark performance was achieved through two examination questions. One question was included in the examination that followed the first integration of explicit programming *strategy* instruction in the second half of 2005 and another from the most recent examination at the end of 2007.

Guarded Division Problem (2005 Examination)

One of the major flaws in novice *strategy* skill, detected in the initial experiment (described in chapter 4), was poor use of guarded division. A 2005 examination question shown as Figure 7.14 (section 7.4.3, page 97) is a strategy-comprehension question that targets the Guarded Division plan. This question yields either a correct or incorrect response. Student responses to this question were analysed and compared to application of Guarded Division in the initial experiment.

Averaging Problem (2007 Examination)

A 2007 examination question shown as Question 8 in Figure 7.16 (section 7.4.3, page 99) was a strategy-generation question that repeated the averaging problem given to novices in the initial experiment (described in chapter 4). Solutions to this question were analysed using the same approach as used in the initial experiment. Eight features were analysed in student solutions: seven *plans*, and the correct merging of *plans*. The presence or absence of each of these features was checked in all attempts. The features measured were as follows.

- Initialisation of a sum variable
- Initialisation of a count variable
- A Sum plan in a Primed Sentinel-Controlled Loop
- A Count plan in a Primed Sentinel-Controlled Loop
- A guard against division by zero
- An Average plan
- An Output plan
- Merging of the Sum and Count plans inside the Primed Sentinel-Controlled Loop

For more detail on how these features can be identified in a solution, see section 4.4. Results from this analysis are compared to results from the initial experiment to gauge the impact of introducing explicit *strategy* instruction and assessment.

The circumstances surrounding the initial testing were slightly different to a final examination. The initial experiment was conducted under examination-like conditions (students were not permitted to talk to each other or draw on resource materials), but in tutorial classes during the course. Final examinations are held at the end of the course, giving students more time between exposure and testing of the necessary *plans*. These differences need to be kept in mind when comparing performance between these tests.

Results of these two examination question comparisons are shown in section 7.6.2.

Avoiding Bias

Neither of these two specific questions had been used in the course prior to the examinations. The closest problem resembling the averaging problem was the average word length exercise given in tutorials and shown in Figure 7.8 (section 7.3.4, page 93). The course materials covered each of the required *strategies*. Students had opportunities to practice each of the required *strategies*. These *strategies* were not emphasised more than any other *strategies* taught in the course.

In the two examination questions, students are not led to use any specific *strategies*; they are expected to have learned which *strategies* to apply at this stage (during the exam).

7.5.3 Consistency of Knowledge and Strategy Skill

The final research question (RQ14) sought to validate testing of *strategy* skill by checking the consistency of students' performances in knowledge-related and strategy-related questions. Using a controlled examination structure that focuses on *knowledge* and *strategies* independently (see section 7.4.3) allows for a comparison of skills between these two areas.

Examination questions from the 2005 and 2007 examinations were used; both examinations were held following explicit instruction of strategies. Marks awarded for questions targeting *knowledge* and *strategies* were compared proportionally and correlations were noted. Using this information it was possible to compare the relative performance of students between knowledge-related and strategy-related questions. If the questions used to measure *strategy* skill are valid then student performance should generally be consistent between these two question types. Results of this comparison are shown in section 7.6.3.

The distinction between performance in *knowledge* and *strategy* potential in novices is significant as previous research has dwelt upon this relationship (Lister et al., 2004). A clearer picture of the relationship between these two aspects may help instructors and computing education researchers in future.

7.6 Results

Results are presented below, again divided by the three perspectives used earlier. First the success of integrating programming *strategies* in an actual introductory programming course is discussed. Specific strategy-related responses elicited under the traditional and new curriculum are then compared. Finally, the consistency of students' *knowledge* and *strategy* performance is analysed.

7.6.1 Integration

Integrating explicit *strategy* instruction and assessment into an actual introductory programming course was achieved. The examples of curricular materials shown in section 7.3 and the assessment items described in section 7.4 demonstrate how this was achieved. The Strategy Guide used is given as Appendix A. The assessment items shown in section 7.4 were added to assignments and examinations during the two-and-a-half-year period after the previous experiment. A full examination and marking scheme are provided as appendices F and G respectively.

Perhaps the most arduous part of integrating *strategies* explicitly was in conceiving well focused assessment items. It is challenging to create problems that required students to apply specific *plans*, while maintaining interesting problems. Even so, a set of problems was developed to assess *strategy* skill in assignments and examinations. The validity of examination questions used to assess programming *strategy* skill is discussed in section 7.7.3.

Students accepted the new instruction as part of the course; no student protested against the inclusion of *strategies* as legitimate content. As each new cohort undertook the new curriculum, they were not aware that it was different to the traditional curriculum that preceded it. Students did not protest against having their *strategy* skills assessed. As mentioned earlier (see section 7.4.2), assignments involved peer review, so students were being asked to evaluate the work of their peers. Students were asked to complete reviews that required them to judge the presence or absence of *strategies* in the work of their peers.

Occasionally the author, as instructor, would quiz the class or individual students on strategy-related comprehension during the course. For instance, when shown a piece of code, such as a Bubble Sort, which contains *plans* covered earlier in the course, the instructor would ask, “What type of loop is this?”, or “What is this plan for swapping a pair of variables called?” Students were able to use a vocabulary of *strategies* to respond.

Tutorial and practical classes allowed for some observation of students working on *strategies* in a face-to-face setting. Tutorials allowed students to occasionally answer and discuss strategy-related questions and students showed no sign that *strategy* instruction was to be appreciated differently to *knowledge* instruction in the course. In practical classes it is possible to detect that a student has a flawed *strategy* understanding and to challenge that flaw based on the *strategy* content expressed in the course. For example, when a student fails to guard a division used to calculate an average of user inputs, asking the student to test their program by providing no inputs (providing a sentinel as the first input), then asking them why their program does not work, inevitably causes them to recall the necessary *plan*.

7.6.2 Impact

Two specific questions were used to compare *strategy* skill under the previous and new curricula. The questions were drawn from two examinations, one which took place at the end of 2005 after the first instance of the course to include explicit *strategy* instruction, and one in the most recent instance at the end of 2007.

Guarded Division Problem (2005 Examination)

During the initial experiment (from chapter 4) a particularly poorly applied *plan* was the Guarded Division plan, with only four students out of 42 applying this *plan*. In the second semester 2005 examination, under the new curriculum, the strategy-comprehension question given as Figure 7.14 (section 7.4.3, page 97) was used to specifically target comprehension of the Guarded Division plan after explicit instruction. This question showed a function used to calculate an average; however, there was no guard around the division so it was susceptible to failure if the count of values was zero. Students were asked to identify the flaw and suggest a remedy.

Table 7.1. Change in Guarded Division ability under new curriculum

	Correct	Proportion
Application in generation experiment (chapter 4) before explicit *strategy* instruction	4 of 42	10%
Comprehension in 2005 exam under new curriculum	25 of 36	69%

Results from Table 7.1 show the poor application of the Guarded Division plan under implicit-only *strategy* instruction and the current potential of students to comprehend this *plan* after explicit instruction. After explicit *strategy* instruction, correct answers to the Guarded Division were provided by 25 of 36 students. This indicates that most students had learned and could comprehend the Guarded Division plan, knowing where it should be applied.

Testing *comprehension* of a *strategy* (as in this problem) is not directly comparable to *generation* of that *strategy* (as with the initial experiment). However, knowing that 69% of students comprehend the Guarded Division plan should be kept in mind when considering the results of a direct comparison using a generation task in the next subsection. This direct comparison is achieved with a question that required students to generate a solution that applies the Guarded Division plan within an averaging problem.

Averaging Problem (2007 Examination)

During the examination from second semester 2007 the questions shown in Figure 7.16 (section 7.4.3, page 99) were used. From this figure Question 8 repeats the averaging problem used in the initial experiment (chapter 4).

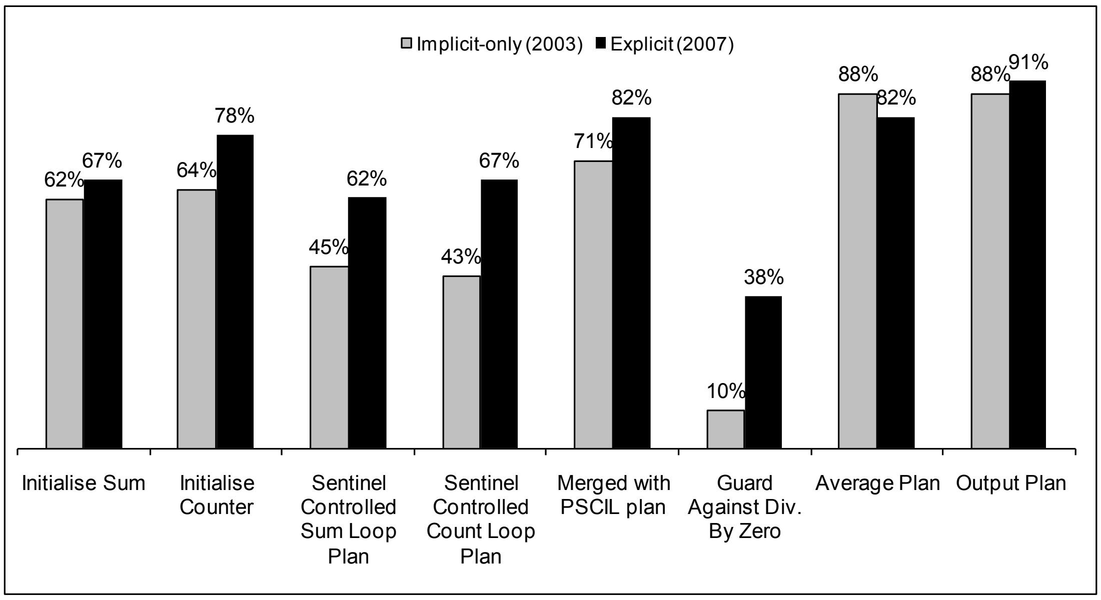


Figure 7.17. Comparison of *plan* use in averaging problem under curricula including implicit-only and explicit *strategy* instruction

Solutions to this problem were analysed under Goal/Plan Analysis, with the same list of *plans* sought (and merging of the sum and count plans in the SCL). Figure 7.17 distinguishes results between the initial test, where novices learned programming *strategies* in an implicit-only manner, and the examination under the new curriculum, which included programming *strategies* explicitly. Student results show consistent improvement in all *plans* except one. The Guarded Division plan is still the most poorly applied *plan*, with only 38% of participants using this *plan* even after explicit instruction in this *plan*. However, according to a chi-squared test, this is a significant increase ($\chi^2 \approx 9.47$, $p \approx 0.002$, $k=1$), almost fourfold from the initial experiment, and this level is higher than the level demonstrated by experts (as seen in Table 5.4 from chapter 5, page 56). There was also a significant chi-squared increase in use of the Sentinel-Controlled Count Loop plan ($\chi^2 \approx 4.98$, $p \approx 0.03$, $k=1$).

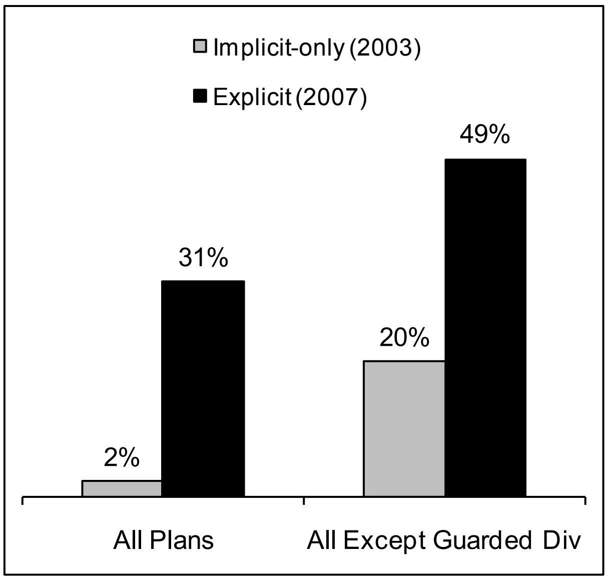


Figure 7.18. Comparison of complete and near-complete correctness in averaging problem with and after without *strategy* instruction

Figure 7.18 compares the completeness (use of all expected *plans*) from the initial experiment and results from the averaging question in an examination under a curriculum with explicit programming *strategies*. Under the new curriculum, the proportion of correct solutions increased from 2% (1 of 42) to 31% (14 of 45) which is a significant chi-squared increase ($\chi^2 \approx 12.56$, $p \approx 0.0004$, k=1). If the most poorly applied *plan*, Guarded Division, is ignored the proportion of complete and near-complete answers has increased from 20% (10/42) to 49% (22/45) which is also a significant chi-squared increase ($\chi^2 \approx 5.88$, $p \approx 0.02$, k=1).

Table 7.2. Testing for improvement between cohorts

Exam	Average Plan Application
Implicit-only (2003)	4.0 of 7 plans (57%)
Explicit (2007)	4.8 of 7 plans (69%)

There is an improvement in the average proportion of application of the seven expected *plans* between the student cohorts. As shown in Table 7.2, prior to explicit instruction of programming *strategies*, students applied 57% of the expected *plans* on average. With explicit instruction of programming *strategies*, this increased to 69% of the expected *plans* on average. Using a two-sample t-test (one-tailed) there is evidence of a statistically significant improvement between the two cohorts (df=85, $t \approx 1.66$, $p \approx 0.02$).

7.6.3 Consistency of Knowledge and Strategy Skill

By definition, a programming *strategy* is a way of applying programming *knowledge* (Davies, 1993). This infers that *strategy* skill is dependent on *knowledge* skill. If this inference holds, students should perform equally or better in knowledge-related questions compared to strategy-related questions

Table 7.3. Testing dependency of *strategies* on *knowledge*

Exam	*n*	Average Knowledge Performance	Average Strategy Performance	Pearson Correlation	P Value
Explicit 2005	36	72%	68%	0.74	0.000001
Explicit 2007	45	64%	57%	0.81	0.000004

The results in Table 7.3 were calculated by grouping examination questions as either *knowledge* questions or *strategy* questions according to the definitions given in section 7.4.3 and finding each student's proportional marks for the two groups of questions. The results are shown for two examinations. The first from2005 was the first time *strategies* were explicitly incorporated into the curriculum (following the experiment described in chapter 6). The second from 2007 is the most recent and the semester from which student results were taken for section 7.6.2.

The results show that on average, students perform slightly better in *knowledge* questions than in *strategy* questions. Results also show that there is a significant correlation between novices' results on knowledge-related and strategy-related questions in the both examinations. By combining these results we can say that a novice will consistently perform slightly better in knowledge-related questions than strategy-related questions, but in general, a novice who performs poorly will generally perform poorly in both, and a student who performs well will perform well in both. This observation is certainly not without exception. Some students did perform better in strategy questions thank knowledge questions, but rarely so.

7.7 Discussion

In this section we use the experimental results from section 7.6 to answer the research questions posed in section 7.2.

7.7.1 Integration

RQ11. *Can instruction of programming strategies be explicitly incorporated into instruction in an actual introductory programming course?*

While it did take some time and effort to transform the traditional curriculum, adding explicit *strategy* content, this was demonstrated to be possible. The amount of *strategy* content is not necessarily fixed; it needs to be further refined. Sharing these *strategies* with other instructors will allow this development. It is useful to reiterate that *strategies* can be used with most imperative and object-oriented languages so they would suit the majority of introductory programming courses, requiring little change for different languages.

The author has been asked to reflect, from experience, how programming strategies can be integrated *well* rather than poorly. The author has no evidence to support one method over another, so the following are only suggestions.

- Students should be informed of the approach to learning they should take when studying in a course. Students need to be told that they are expected to learn the *strategies* covered in the course. If this is to be assessed, students need to know this also.

- If our objective is to teach programming *strategies*, then our assessments should be constructively aligned (Biggs, 1999) with this expected learning outcome. Reward students for applying strategies. Assessment is an instructor's currency. Marks force surface learners (Biggs, 1987) to learn what an instructor sees as important. Marks show deep learners what an instructor sees as important. Pre-defined marking schemes, published with assignment instructions, are an excellent way of showing students how they will be assessed. Advanced students can be rewarded with bonus marks for extension activities.
- Refer to programming strategies rather than underlying syntax where possible. For instance, one could say "use a for loop to achieve that" when a more strategic instruction would be "use a counter-controlled loop to achieve that".
- Like programming *knowledge*, *strategies* need to be practiced. Naming them is not enough. Students need to see examples and undertake practical exercises that focus on strategies.

RQ12. *Can programming strategy skill be measured as part of the assessment in an actual introductory programming course?*

It is possible to measure programming *strategy* ability in novices with tests that address both *comprehension* and *generation*. A number of different forms of assessment have been demonstrated for programming assignments and examinations. Most assessment methods used in the new curriculum resemble traditional curriculum assessment items, but with careful problem design and objective criteria for evaluation, assessment items can be used to focus testing of *knowledge* and *strategies* independently.

7.7.2 Impact

RQ13. *What is the impact on novice programmers of incorporating programming strategy explicitly into instruction and assessment?*

The results show a significant improvement in students' use of *strategies* under a curriculum where *strategies* are covered explicitly. There is a strong improvement in overall completeness of solutions to the averaging problem tested between the initial experiment (chapter 4) and an examination under the new curriculum. There is a specific improvement in the use of the most poorly applied *strategy*, the Guarded Division plan, although its application is still relatively low.

7.7.3 Consistency of Knowledge and Strategy Skill

RQ14. *Are novices' results in assessment of programming strategies consistent with their results in assessment of programming knowledge?*

The tests show that results gained in strategy-related questions are consistent with results gained in questions covering programming *knowledge*. This may be seen as a measure of validity in the method of testing *strategy* skill. In overall performance, there was consistency found between knowledge-related and strategy-related responses.

A novice who performs poorly in *knowledge* questions will generally perform poorly in *strategy* questions and a student who performs well in *knowledge* questions will perform well (but slightly worse) in *strategy* questions. This finding supports the assumption that programming *knowledge* is a prerequisite for programming *strategies*.

7.8 Implications

This experiment has shown that it is possible to instruct and assess programming *strategies* in an actual introductory programming course. Teaching programming *strategies* in this way creates a vocabulary that can be used in teaching and assessment. This vocabulary allows *strategies* to be reused and reinforced after they are presented. Students learn and apply programming *strategies* more consistently when they are presented in an explicit manner than when they are learned implicitly.

This study has also shown that *strategies* can be a valid part of assessment and can therefore be a valued part of an introductory programming curriculum that aims to train novice programmers to apply programming *strategies*. The methods of *strategy* skill assessment used can be applied to both *comprehension* and *generation* exercises and conducted throughout a course. Strategy-related questions in examinations can elicit results consistent with questions that assess programming *knowledge* skill. *Strategy* skill testing can also be achieved in regular assignments.

With a more precise vocabulary for defining a complete solution to a problem, instructors can now avoid vague terms such as ‘elegance’ and ’connoisseurship’ when evaluating the work of a novice; instead, instructors can point out what *strategies* are absent or misapplied in novices’ solutions.

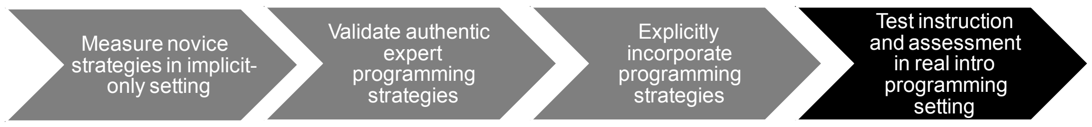

Measure novice strategies in implicit-only setting
Validate authentic expert programming strategies
Explicitly incorporate programming strategies
Test instruction and assessment in real intro programming setting

Figure 7.19. Overview of experiments in a process after the fourth experiment

As can be seen in Figure 7.19, this experiment brings to an end the series of four experiments that form the core investigation of this dissertation.

8. Findings and Contribution of this Study

This study set out to create new curricular elements that could help overcome barriers documented in literature and thus help to alleviate the poor outcomes for students that are caused by these barriers. Encouraging novices to become better programmers could ultimately benefit the programming profession and in turn benefit the community that depends on programmers.

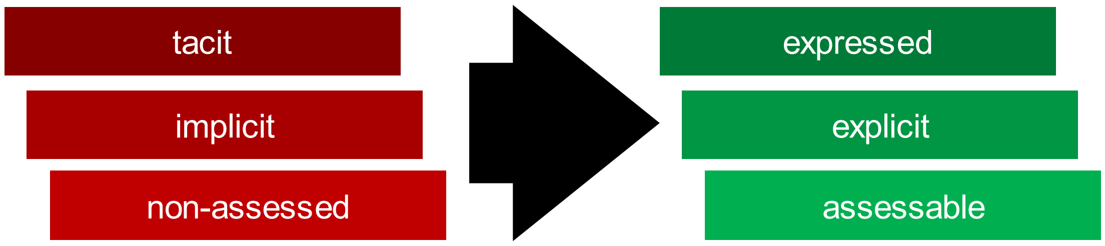


Figure 8.1. Including programming *strategies* in curricula

Figure 8.1 (reproduced from Figure 1.1) depicts the aims of this study: to improve introductory programming curricula by plainly expressing programming *strategies* and instructing and assessing these *strategies* explicitly.

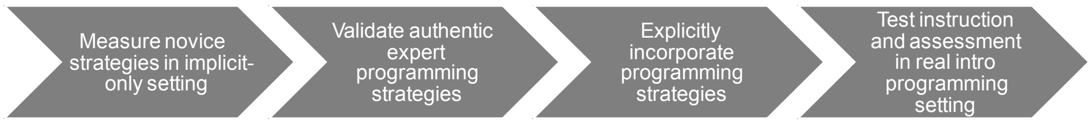

Measure novice strategies in implicit-only setting
Validate authentic expert programming strategies
Explicitly incorporate programming strategies
Test instruction and assessment in real intro programming setting

Figure 8.2. Overview of experiments as a process

The aims of this study were explored in a series of four experiments summarised in Figure 8.2 (reproduced from Figure 3.3), which contributed the core findings of the study. Section 8.1 consolidates the experimental work described in chapters 4 to 7. For each experiment research questions are revisited and results are described. In section 8.2, the contribution of the work in this study is identified in the context of computing education research. Finally, in section 8.3, suggestions are made for future extensions to the work done in this study.

8.1 Findings of this Study

The results of experimentation, described as answers to research questions, form the overall findings of this study. These can now be elaborated.

8.1.1 Initial Experiment

The initial experiment, described in chapter 4, set out to find a benchmark of programming *strategy* skill for students learning under a traditional curriculum with *strategies* taught implicitly. In this experiment the following research questions were asked.

RQ1. *What is the potential of students who have been exposed to an implicit-only teaching of programming strategies to solve a sub-algorithmic problem that requires application of a number of programming strategies for a complete solution?*

When asked to create a solution for a classic averaging problem, many students failed to demonstrate application of important *strategies*. Only a single student was able to achieve a fully complete solution to the problem. On average, students applied 57% of the expected *plans*.

In particular, participating students were not consistently able to:

- initialise sum and/or count variables,
- use a correct looping *strategy* for the given problem,
- guard against events such as division by zero, or
- merge *plans* that should be applied together.

RQ2. *What are the deficiencies in the curriculum that are demonstrated by students' solutions to the given problem?*

The initial experiment showed that many novices had not learned the specific programming *strategies* covered in the experiment. The curriculum relied on implicit instruction of programming *strategies* and had not allowed most students to learn *plans* to a level where they could demonstrate the application of these *plans*.

The initial experiment found common programming *strategy* flaws in solutions created by novices across an entire student cohort. The novices had studied a curriculum that required them to learn programming *strategies* implicitly. Biederman and Shiffrar (1987) found that explicit instruction can be far more effective than implicit-only instruction. When considered in the context of an introductory programming course, this suggested that if programming *strategies* could be expressed and incorporated explicitly into an introductory curriculum, this might improve the programming *strategy* potential of novices.

8.1.2 Expert Strategies Experiment

The findings of the initial experiment were used to justify instruction of *strategies* in an explicit manner. But before this instruction could be undertaken, an authentic representation of *strategies,* consistent with those found in solutions created by experts, was needed.

The *strategies* used by experts were explored in the second experiment, described in chapter 5. *Plans* proposed by Soloway (1986) were proposed as a model of the sub-algorithmic programming *strategies* used by experts. To test this proposal, experts were asked to solve three sub-algorithmic-level problems which were analysed to answer the following questions.

RQ3. *Do experts exhibit identifiable plans in their solutions to problems?*

Experts' solutions to the three problems included code that could be identified as applications of the expected *plans*. This finding indicated that *plans* are used by expert programmers.

RQ4. *Can an authentic set of strategies, used by experts, be represented in an explicit form, suitable for instruction?*

By finding that *plans* are consistent with the solutions of experts, it is justifiable that *plans* be used as an expression of expert *strategies*. *Plans* are a simple form of *strategies* that can be incorporated explicitly into an introductory programming curriculum.

RQ5. *Does the potential to represent authentic programming strategies mandate explicit instruction of programming strategies to novices?*

Based on the advantages of using explicit instruction (Biederman and Shiffrar, 1987, Baddeley, 1997, Berry and Dienes, 1993) and indications that novices can become more effective by focusing on programming *strategies* (Robins et al., 2003, Soloway, 1986), it is justifiable to incorporate programming *strategies* explicitly into the introductory programming curriculum. *Plans* are an authentic representation of experts' programming *strategies* at a sub-algorithmic level, and can be used to explicitly represent *strategies*.

8.1.3 Artificial Curriculum Experiment

With justification and a validated set of *strategies*, an experiment was conducted to measure the potential to incorporate programming *strategies* in an introductory programming curriculum, initially in a limited manner. A third experiment, described in chapter 6, was conducted in an artificial setting to test a curriculum that included programming *strategies* explicitly in lectures, written course materials, exercises and assessment. The following questions were used to guide this investigation.

RQ6. *Can programming strategies be explicitly incorporated into an introductory programming curriculum?*

In the experiment, two groups (an experimental and control group) were trained over separate weekend periods. Both groups were exposed to a common base curriculum, which included programming *knowledge* content and exercises. The experimental group were also exposed to additional content which explicitly covered a limited set of programming *strategies*. By describing and using a curriculum that included *strategies* explicitly, it was shown that such integration can be achieved.

RQ7. *What is the significance of the time consumed by this additional instruction?*

Introducing additional material did increase the time consumed by lecture sessions. However, the same schedule was followed for the experimental and control weekend sessions, even with the additional explicit instruction in the experimental curriculum. All participating students were able to complete exercises before the end of each session, indicating a reduction in the time taken by experimental participants to complete exercises. This indicates that the additional time needed for explicit strategy instruction was not significant and did not cause undue burden on students or instructors.

RQ8. *Can programming strategies, explicitly taught in an introductory programming course, be assessed?*

Goal/Plan Analysis was used to measure all participants' application of programming *strategies* in *generation* exercises at the end of the course. Goal/Plan Analysis was found to be limited as an assessment method as it requires students to generate code before *strategy* skill can be assessed, and is only useful towards the end of a course.

RQ9. *What impact does explicit strategy instruction have on students and their ability to apply strategies when compared to an implicit-only approach?*

This experiment found that when programming *strategies* are taught explicitly, students may be more likely to understand and apply these *strategies* than when students are expected to learn *strategies* implicitly. Novices taught programming *strategies* explicitly used strategies more often in their solutions, although not significantly so.

RQ10. *Are there any other observable effects or contrasts between students of a traditional curriculum and one with added explicit programming strategy instruction?*

Students shown programming *strategies* explicitly used *strategy* terms from the strategy-related vocabulary presented in the curriculum during interviews. In an actual introductory programming curriculum, such a vocabulary could be used between instructors and students to aid teaching and assessment.

From interviews it was found that students who covered the curriculum containing explicit *strategies* showed confidence in the solutions they had created and their understanding of the *strategies* used to create them, while students not exposed to this curriculum doubted their abilities.

These findings were justification for incorporating programming *strategies* in a full introductory programming curriculum.

8.1.4 Explicit Programming Strategy Instruction in an Actual Course

Following the successful integration of explicit programming *strategy* content in an artificial curriculum, and noting the effects of this integration, a full-scale integration was undertaken with an actual introductory programming course. This integration is described in chapter 7. The following questions were considered when measuring the potential for, and effectiveness of, integrating programming *strategies* explicitly.

RQ11. *Can instruction of programming strategies be explicitly incorporated into instruction in an actual introductory programming course?*

Programming *strategies* were successfully integrated as explicit content in an actual introductory programming course. This integration was achieved by inserting programming *strategies* at points following prerequisite programming *knowledge*. Programming *strategies* were described in written materials, discussed in lectures, and practised during tutorials and practicals. A Strategy Guide, collating all of the *strategies* covered in the course, was also produced and given to students.

RQ12. *Can programming strategy skill be measured as part of the assessment in an actual introductory programming course?*

Programming *strategy* skill can be measured at different times during a course through assignments and examinations. With careful problem design and objective criteria for evaluation, assessment items can be used to focus on testing *knowledge* and *strategies* independently.

RQ13. *What is the impact on novice programmers of incorporating programming strategy explicitly into instruction and assessment?*

Student performance under the new curriculum was compared to the benchmark measured in the initial experiment from chapter 4. Results showed significant improvements in strategy-related performance under the new curriculum. There was an improvement in overall completeness of *plans* applied in novices' solutions to the averaging problem.

RQ14. *Are novices' results in assessment of programming strategies consistent with their results in assessment of programming knowledge?*

Results from student responses to strategy-related questions are consistent with questions covering programming *knowledge*. A novice who performs poorly in *knowledge* questions will generally perform poorly in *strategy* questions. A student who performs well in *knowledge* questions will generally perform well (but slightly worse) in *strategy* questions. As well as adding credibility to testing of programming *strategies*, this finding also adds evidence to the assumption that programming *knowledge* is a prerequisite for programming *strategies*.

The inclusion of explicitly described programming *strategies* positively impacted the programming *strategy* potential of novices who undertook this new curriculum. By separating testing of programming *knowledge* and *strategies*, the consistency of the method for assessing programming *strategies* was verified. Distinguishing programming *knowledge* skill from programming *strategy* skill added evidence to the fundamental tenet that programming *knowledge* is prerequisite to programming *strategies*.

8.2 Contribution

Studies have shown universally poor results by novices on standardised tests conducted at institutions across the world (McCracken et al., 2001). Novices produce poor results in standardised program *generation* tests, with many novices having a fragile *knowledge* (Lister et al., 2004) and most novices failing to demonstrate programming *strategies* (Lister et al., 2006).

The work undertaken for this dissertation contributes to the field of computing education research by:

- improving understanding of the distinction between programming *knowledge* and programming *strategies* (§2.3.2, §7.5.3);

- identifying a level of problems relevant to novices (§2.3.3);
- discovering that novices develop a flawed set of programming *strategies* when learning *strategies* implicitly (RQ1, §4.6.1);
- describing deficiencies in a traditional implicit-only curriculum with respect to instruction of programming *strategies* (RQ2, §4.6.2);
- demonstrating that *plans* are programming *strategies* that appear in solutions created by experts (RQ3, §5.6.1);
- representing *strategies* in a form that can be explicitly incorporated into a curriculum (RQ4, §5.6.2);
- demonstrating that programming *strategies* can be explicitly incorporated into an introductory programming curriculum without undue time pressure through a controlled experiment that compared a traditional implicit-only *strategy* curriculum to a curriculum incorporating programming *strategies* implicitly (RQ6, §6.6.1, RQ7, §6.6.2);
- experimentally testing the effects of instructing programming *strategies* explicitly to novices, which including difference in performance (RQ9, §6.6.4), increased confidence and use of a vocabulary of *strategies* (RQ10, §6.6.5);
- describing how programming *strategies* can be explicitly incorporated into teaching materials (RQ11, §7.3) and assessment items used for an actual introductory programming course (RQ12, §7.4);
- measuring improved outcomes produced by explicitly instructing programming *strategies* by comparing students' results with a baseline standard set under an implicit-only curriculum (RQ13, §7.7.2); and
- adding evidence to the logical inference that programming *knowledge* must precede programming *strategies* (RQ14, §7.7.3).

These contributions are intended to improve outcomes for novices by improving the curricula delivered to novice programmers. With a well defined and justified method for instructing novices in programming *strategies*, poor standards of performance, measured around the world, might be improved.

Ultimately, assisting novice programmers to construct a more consistent and coherent body of programming *strategies* may aid them in later programming study, and guide their development as experts who can advance the art of programming.

8.3 Future Work

The experiment described in chapter 6, which was conducted in an artificial setting, provided justification to incorporate explicit programming *strategy* instruction in an actual introductory programming course. A number of flaws were identified in the experimental methodology. Conducting the experiment again would allow for reproduction of results, adding confidence to the findings of that experiment. If this were to be undertaken the following changes could be made.

- Seek larger groups of participants
- Conduct the experiment over a longer period to allow absorption of concepts
- Use methods of assessment not limited to Goal/Plan Analysis throughout the course to measure programming *strategy* skill

The set of *plans* used in the curriculum, described in chapter 7 and given in full in Appendix A, might be useful to instructors. This set of *plans* could be further developed and improved by:

- formalising the representation of *plans*,
- extending the set of *plans*, and
- developing a repository of assessment item examples that test programming *strategy* skill.

To make programming *strategies* accessible, these ideas may be published in a programming textbook, to be used by instructors and novices.

By explicitly teaching programming *strategies* and separating these *strategies* from programming *knowledge*, it may be possible to investigate the impact of such teaching on the learning styles and meta-cognition of novice programmers. The following research questions could be explored.

- Can programming *strategy* performance be linked to a 'deep approach' to learning (Biggs, 1987)?
- Can students' potential to solve problems be explored without asking them to generate solutions?
- Are students aware of which problems they can solve?
- Can gaps in programming *strategy* ability be identified and repaired before summative assessment?
- Is there a lag between instruction, *comprehension* and *generation* of programming *knowledge* and *strategies*, and if so, can this be measured?

Sub-algorithmic programming *strategies* could conceivably become as much a part of future novice instruction as programming syntax. By turning away from failing traditional curricula and creating new curricula with well justified content and pedagogically sound delivery, programming instructors might begin to deliver more acceptable student outcomes.

References

ANDERSON, L. W., KRATHWOHL, D. R., AIRASIAN, P. W., CRUIKSHANK, K. A., MAYER, R. E., PINTRICH, P. R., RATHS, R. & WITTROCK, M. C. (Eds.) (2001) *A Taxonomy for Learning, Teaching and Assessing. A Revision of Bloom's Taxonomy of Educational Objectives,* New York, USA, Addison Wesley Longman, Inc.

ANDREAE, P., BIDDLE, R., DOBBIE, G., GALE, A., MILLER, L. & TEMPERO, E. (1998) Surprises in Teaching CS1 with Java (School of Mathematical and Computing Sciences, Technical Report CS-TR-98/9). Wellington, Victoria University of Wellington.

ASHENDEN, D. & MILLIGAN, S. (1999) *The good universities guide: Universities, TAFE and private colleges in 2000.,* Australia, Hobsons.

BADDELEY, A. (1997) *Human Memory: Theory and Practice (Revised Edition),* East Sussex, UK, Psychology Press.

BAILIE, F. K. (1991) Improving the modularization ability of novice programmers. In Papers of the twenty-second SIGCSE technical symposium on Computer science education. p. 277 - 282.

BECK, K. (2001) Extreme Programming. Accessed October 6, 2008, http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,66192,00.html

BEN-ARI, M. & SAJANIEMI, J. (2004) Roles of Variables as Seen by CS Educators. In Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2004). p. 52 - 56.

BERGIN, J. (2000) Java-- GOOD, BAD, and NOT C++. Accessed 30th August, 2006, http://csis.pace.edu/~bergin/Java/SomegoodthingsaboutJava.html

BERRY, D. C. & DIENES, Z. (1993) *Implicit Learning: Theoretical and Empirical Issues,* East Sussex, UK, Lawrence Erlbaum Associates Ltd.

BIDDLE, R. & TEMPERO, E. (1998) Java pitfalls for beginners. *ACM SIGCSE Bulletin,* 30**,** 48 - 52.

BIEDERMAN, I. & SHIFFRAR, M. M. (1987) Sexing Day-Old Chicks: A Case Study and Expert Systems Analysis of a Difficult Perceptual-Learning Task. *Journal of Experimental Psychology: Learning, Memory and Cognition,* 13**,** 640 - 645.

BIGGS, J. B. (1987) *Student Approaches to Learning and Studying,* Melbourne, Australian Council for Educational Research.

BIGGS, J. B. (1999) *Teaching for Quality Learning at University,* Buckingham, Open University Press.

BIGGS, J. B. & COLLIS, K. F. (1982) *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome),* New York, Academic Press.

BROOKS, R. E. (1983) Towards a theory of the comprehension of computer programs. *International Journal of Man–Machine Studies,* 18**,** 543 – 554.

BURJE, J. E. (1998) Knowledge Elicitation Tool Classification. Accessed 25 September, 2008, http://web.cs.wpi.edu/~jburge/thesis/kematrix.html

CARBONE, A., HURST, J., MITCHELL, I. & GUNSTONE, D. (2000) Principles for designing programming exercises to minimise poor learning behaviours in students. In Proceedings of the on Australasian computing education conference. p. 26 - 33.

CASPERSEN, M. E. & BENNEDSEN, J. (2007) Instructional design of a programming course: a learning theoretic approach. In Proceedings of the third international workshop on Computing education research (ICER2007). p. 111 - 122.

CHANDRA, S. S. & CHANDRA, K. (2005) A comparison of Java and C#. *Journal of Computing Sciences in Colleges,* 20**,** 238 - 254.

CLANCY, M. J. & LINN, M. C. (1999) Patterns and Pedagogy. In The proceedings of the thirtieth SIGCSE technical symposium on Computer science education. p. 37 - 42.

COLLINS, A., BROWN, J. S. & HOLUM, A. (1991) Cognitive apprenticeship: Making thinking visible. Accessed 1 October, 2008, http://pride.wilsonsd.org/pol/ri/collins.pdf

COLLINS, A., BROWN, J. S. & NEWMAN, S. E. (1987) *Cognitive apprenticeship: teaching the craft of reading, writing, and mathematics,* Cambridge, Massachusetts, USA, University of Illinois at Urbana-Champaign.

COOKE, N. J. (1994) Varieties of knowledge elicitation techniques. *International Journal of Human-Computer Studies,* 41**,** 801 - 849.

CRETCHLEY, P. (2006) Does computer confidence relate to levels of achievement in ICT-enriched learning models? *Education and Information Technologies.* New York, USA, Springer.

DAVIES, S. P. (1993) Models and theories of programming strategy. *International Journal of Man-Machine Studies,* 39**,** 237 - 267.

DE RAADT, M., TOLEMAN, M. & WATSON, R. (2005) Textbooks Under Inspection (Working Paper). Accessed November 4, 2007, http://www.sci.usq.edu.au/research/workingpapers/sc-mc-0715.ps

DE RAADT, M., WATSON, R. & TOLEMAN, M. (2002) Language Trends in Introductory Programming Courses. In Proceedings of Informing Science and IT Education Conference. p. 329 - 337.

DE RAADT, M., WATSON, R. & TOLEMAN, M. (2003a) Introductory programming languages at Australian universities at the beginning of the twenty first century. *Journal of Research and Practice in Information Technology,* 35**,** 163-167.

DE RAADT, M., WATSON, R. & TOLEMAN, M. (2003b) Language Tug-Of-War: Industry Demand and Academic Choice. *Australian Computer Science Communications,* 25**,** 137 - 142.

DE RAADT, M., WATSON, R. & TOLEMAN, M. (2004) Introductory Programming: What's happening today and will there be any students to teach tomorrow? *Australian Computer Science Communications,* 26**,** 277 - 284.

DECKER, R. & HIRSHFIELD, S. (1994) The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS1. In Selected papers of the twenty-fifth annual SIGCSE symposium on Computer science education. p. 51 - 55.

FIX, V., WIEDENBECK, S. & SCHOLTZ, J. (1993) Mental representations of programs by novices and experts. In Proceedings of the Conference on Human Factors in Computing Systems. p. 74 - 79.

GUZDIAL, M., HOHMANN, L., KONNEMAN, M., WALTON, C. & SOLOWAY, E. (1998) Supporting Programming and Learning-to-Program with an IntegratedCAD and Scaffolding Workbench. *Interactive Learning Environments,* 6**,** 143 - 180.

GUZDIAL, M. & SOLOWAY, E. (2002) Teaching the Nintendo generation to Program. *Communications of the ACM,* 45**,** 17 - 21.

HADJERROUIT, S. (1998) Java as first programming language: a critical evaluation. *ACM SIGCSE Bulletin,* 30**,** 43 - 47.

HIRSCH, E. D., JR. (2002) Classroom Research and Cargo Cults. *Policy Review,* 115**,** 51 - 69.

HITZ, M. & HUDEC, M. (1995) Modula-2 versus C++ as a first programming language--some empirical results. In Papers of the 26th SISCSE technical symposium on Computer science education. p. 317 - 321.

HOHMANN, L., GUZDIAL, M. & SOLOWAY, E. (1992) SODA: A computer-aided design environment for the doing and learning of software design *Lecture Notes in Computer Science.* Berlin / Heidelberg, Springer.

HUSIC, F. T., LINN, M. C. & SLOANE, K. D. (1989) Adapting Instruction to the Cognitive Demands of Learning to Program. *Journal of Educational Psychology,* 81**,** 570 - 583.

JOHNSON, W. L. (1986) *Intention Based Diagnosis of Novice Programming Errors,* Los Altos, California, USA, Morgan Kauffman Publishers, Inc.

JOHNSON, W. L. & SOLOWAY, E. (1984) PROUST: Knowledge-based program understanding. In Proceedings of the 7th international conference on Software engineering. p. 369 - 380.

JOHNSON, W. L., SOLOWAY, E., CUTLER, B. & DRAPER, S. (1983) Bug Catalogue: I (Technical Report). Yale University, Computer Science Department.

KLAHR, D. & CARVER, S. M. (1988) Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology,* 20**,** 362 - 404.

KÖLLING, M., KOCH, B. & ROSENBERG, J. (1995) Requirements for a first year object-oriented teaching language. In Papers of the 26th SISCSE technical symposium on Computer science education. p. 173 - 177.

KUITTINEN, M. & SAJANIEMI, J. (2003) First Results of An Experiment on Using Roles of Variables in Teaching. In Papers from the Joint Conference at Keele University (EASE & PPIG 2003). p. 347 - 357.

LISTER, R. (2000) On Blooming First Year Programming and its Blooming Assessment. In Proceedings of the on Australasian computing education conference. p. 158 - 162.

LISTER, R., ADAMS, E. S., FITZGERALD, S., FONE, W., HAMER, J., LINDHOLM, M., MCCARTNEY, R., MOSTRÖM, J. E., SANDERS, K., SEPPÄLÄ, O., SIMON, B. & THOMAS, L. (2004) A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin,* 36**,** 119 - 150.

LISTER, R., SIMON, B., THOMPSON, E., WHALLEY, J. L. & PRASAD, C. (2006) Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *ACM SIGCSE Bulletin,* 38**,** 118 - 122.

MCCRACKEN, M., WILUSZ, T., ALMSTRUM, V., DIAZ, D., GUZDIAL, M., HAGAN, D., KOLIKANT, Y. B.-D., LAXER, C., THOMAS, L. & UTTING, I. (2001) A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin,* 33**,** 125 - 180.

MULLER, O., HABERMAN, B. & GINAT, D. (2007) Pattern-Oriented Instruction and its Influence on Problem Decomposition and Solution Construction. In Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2007). p.

OLIVER, D., DOBELE, T., GREBER, M. & ROBERTS, T. (2004) This Course Has A Bloom Rating Of 3.9. In Proceedings of the Sixth Australasian Computing Education Conference (ACE2004). p. 227 - 231.

PENNINGTON, N. (1987) Comprehension Strategies in Programming. IN OLSON, G., SHEPPARD, S. & SOLOWAY, E. (Eds.) *Emperical Studies of Programmers: 2nd Workshop.* Norwood, New Jersey, USA., Ablex Publishing Co.

PERKINS, D. N. & SALOMON, G. (1992) Transfer of Learning. *International Encyclopedia of Education, Second Edition.* Oxford, England, Pergamon Press.

PHAM, B. (1996) The changing curriculum of computing and information technology in Australia. In Proceedings of the Second Australasian Conference on Computer Science Education. p. 149 - 154.

PINKER, S. (2007) *The Stuff of Thought: Language As a Window Into Human Nature*, Viking Adult.

PORTER, R. & CALDER, P. (2003) A Pattern-Based Problem-Solving Process for Novice Programmers. In Fifth Australasian Computing Education Conference (ACE2003). p. 231 - 238.

PORTER, R. & CALDER, P. (2004) Patterns in learning to program: an experiment? In Proceedings of the Sixth Conference on Australasian Computing Education. p. 241 - 246.

REBER, A. S. (1993) *Implicit Learning and Tacit Knowledge,* New York, USA, Oxford University Press.

REID, R. J. (1993) The object oriented paradigm in CS1. In Proceedings of the Twenty-fourth SIGCSE Technical Symposium on Computer Science Education. p. 265 - 269.

RIST, R. S. (1991) Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programmers. *Human-Computer Interaction,* 6**,** 1 - 46.

RIST, R. S. (1995) Program Structure and Design. *Cognitive Science,* 19**,** 507 – 562.

ROBERTSON, L. A. (2004) *Simple Program Design,* Australia, Thompson Nelson.

ROBINS, A., HADEN, P. & GARNER, S. (2006) Problem distributions in a CS1 course. In Proceedings of the 8th Austalian conference on Computing education (ACE2006). p. 165 - 173.

ROBINS, A., ROUNTREE, J. & ROUNTREE, N. (2003) Learning and Teaching Programming: A Review and Discussion. *Computer Science Education,* 13**,** 137 - 173.

ROYCE, W. W. (1970) Managing the development of large software systems: concepts and techniques. In Proceedings of IEEE WESCON, 1970. p. 328 - 338.

SAJANIEMI, J. (2002) An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs. In Proceedings of IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02). p. 37 - 39.

SAJANIEMI, J. & KUITTINEN, M. (2005) An Experiment on Using Roles of Variables in Teaching Introductory Programming. *Computer Science Education,* 15**,** 59 – 82.

SAJANIEMI, J. & PRIETO, R. N. (2005) Roles of Variables in Experts' Programming Knowledge. In Proceedings of the 17th Workshop of the Psychology of Programming Interest Group (PPIG2005). p. 145 - 159.

SHABO, A., GUZDIAL, M. & STASKO, J. (1996) Computer science apprenticeship: creating support for intermediate computer science students. In Proceedings of the 1996 international conference on Learning sciences (ICLS'96). p. 308 - 315.

SOLOWAY, E. (1986) Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM,* 29**,** 850 - 858.

SOLOWAY, E. (1993) Should We Teach Students to Program? *Communications of the ACM,* 36**,** 21 - 24.

SOLOWAY, E. (2003) Personal Communication.

SOLOWAY, E., BONAR, J. & EHRLICH, K. (1983a) Cognitive Strategies and Looping Constructs: An Empirical Study. *Communications of the ACM,* 26**,** 853 - 860.

SOLOWAY, E., EHRLICH, K. & BLACK, J. B. (1983b) Beyond Numbers: Don't Ask "How Many"... Ask "Why". In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. p. 240 - 246.

SOLOWAY, E., EHRLICH, K. & BONAR, J. (1982) Tapping into tacit programming knowledge. In Proceedings of the Conference on Human Factors in Computers Systems. p. 52 - 57.

SOLOWAY, E. & WOOLF, B. (1980) Problems, Plans and Programs. In Proceedings of the Eleventh ACM Technical Symposium on Computer Science Education. p. 16 - 24.

SPARKE, G. (2003) *The Java Way : An Introduction to Programming in Java,* Australia, Nelson ITP.

SPOHRER, J. C. & SOLOWAY, E. (1986) Novice mistakes, are the folk wisdoms correct. *Communications of the ACM,* 29**,** 624 - 632.

SPOHRER, J. C., SOLOWAY, E. & POPE, E. (1985a) A goal/plan analysis of buggy Pascal programs. *Human-Computer Interaction,* 1**,** 163 - 207.

SPOHRER, J. C., SOLOWAY, E. & POPE, E. (1985b) Where the bugs are. In Proceedings of the CHI '85 conference on Human factors in computing systems. p. 47 - 53.

STROUSTRUP, B. (1999) Learning Standard C++ as a New Language. *The C/C++ Users Journal,* May**,** 43 - 54.

WALLINGFORD, E. (1996) Toward a first course based on object-oriented patterns. In Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education. p. 27 - 31.

WALLINGFORD, E. (2007) The Elementary Patterns Home Page. Accessed 19th November, 2007, http://cns2.uni.edu/~wallingf/patterns/elementary/

WHALLEY, J. L., LISTER, R., THOMPSON, E., CLEAR, T., ROBINS, P., KUMAR, P. K. A. & PRASAD, C. (2006) An Australasian Study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. In Proceedings of the Eighth Australasian Computing Education Conference (ACE2006). p. 243 - 252.

WHORF, B. L. (1956) *Language, Thought and Reality,* Cambridge, Mass, M.I.T. Press.

WINSLOW, L. E. (1996) Programming Pedagogy -- A Psychological Overview. *SIGCSE Bulletin,* 28**,** 17 - 22.

WIRTH, N. (1971) The programming language Pascal. *Acta Informatica,* 1**,** 35 - 63.

WIRTH, N. (1974) On the Composition of Well-Structured Programs. *ACM Computing Surveys,* 6**,** 247 - 259.

List of Figures

List of Tables

Appendices

Table of Appendices

Appendix A. Programming Problem Solving Strategies Reference

Introduction

This appendix contains a number of useful strategies relevant to an introductory programming course, but also necessary to solve problems of a more complex nature. The list is not complete, but contains strategies that are well defined and malleable enough be manipulated to suit particular problems.

This appendix should be seen as a tool-kit for solving problems at a sub-algorithmic level. The plans at this scale usually do not constitute an entire algorithm (although some approach this level) but usually form part of a greater algorithm.

This reference is not meant to be a complete curriculum; it is merely a short reference guide.

Certain programming language knowledge (constructs and functions) are required before each plan can be applied. These dependencies are listed *in italics* at the beginning of each plan.

Strategies Reference Table of Contents

Plan Integration

Before introducing the plans, it is important to discuss how plans can be integrated into a whole solution. There are three ways of combining plans.

Abutment

Abutment is placing plans or steps within plans one after the other. The sequence of these defines the necessary order that must be followed to be successful. For example, if we wish to perform calculations on user inputs, we must first get the inputs before we can perform the calculation.

Merging

Often two plans need to be achieved together. Step within the two plans may be intertwined in their order so that they can be achieved together. A processor can only achieve one instruction at a time so these steps cannot be achieved simultaneously, but the steps can be placed one after another in arbitrary order. For example, if we were wishing to calculate an average of a set of numbers we need to count the numbers and sum the numbers. Rather than inputting and processing the set of numbers twice, we can merge these two plans and achieve them together.

Nesting

Where one plan is contained within another, the inner plan is said to be nested inside the outer plan. For example, if we were summing numbers we may nest the summing plan within one of the specific looping plans. If we were to calculate an average, we may nest this within a Guarded Division plan to avoid division by zero in the average calculation.

Plan 1. Average Plan

This plan requires an understanding of the division operator.

Finding the average of a series of numbers is a common task in programming. To calculate the average we need the sum of the numbers and the count of the numbers. Assuming we have these two values we calculate the average by dividing the sum by the count.

`average = sum / count`

Here is an example in the context of a full program.

```
#include <stdio.h>

int main() {
      int sum = 15;   // Stores the some of some numbers
      int count = 3;  // Stores the count of those numbers
      int average;    // Will store the calculated average

      // Calculate the average
      average = sum / count;

      // Output the average
      printf("Average: %i\n", average);
}
```

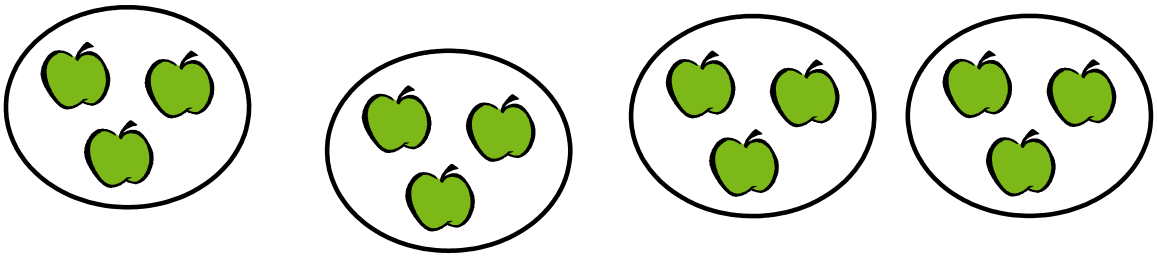
Here is the output of the above program.

```
Average: 5
```

Plan 2. Divisibility Plan

This plan requires an understanding of the mod operator and selection statements.

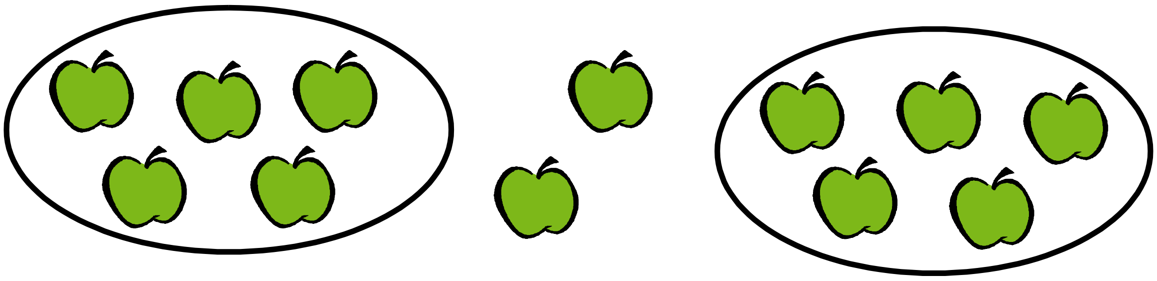
If we wish to see if one number is evenly divisible by another, we can use the mod operator. If this operator produces a result of zero we know that the first operand is divisible by the second. The mod operator gives us the remainder after division. If there is no remainder we know that the first operand is divisible by the second. In a real world application, if we were to group objects, say apples, we may wish to know if we can form complete groups from the number of apples at hand. If we have 12 apples we can divide this into 4 groups of 3 with no remainder.



We can apply the same to numbers in code, for example…

`12 % 3` results in **`0`** so we can say **`12`** is divisible by **`3`**

We can also see when a number is not divisible by another. If we group 12 apples in to groups of 5 we are left with 2 apples remaining.



Again we can apply the same to numbers in code, for example…

`12 % 5` results in **`2`** so we can say **`12`** is not divisible by **`5`**

Here is an example in the context of a full program.

```
#include <stdio.h>

int main() {
      int numberToCheck = 12;  // A number to check for divisibility
      int firstDivisor = 3;    // A sample divisor to use
      int secondDivisor = 5;   // Another sample divisor to use
      int result;              // Will store the result of mod operation

      // Check the divisibility using first divisor
      result = numberToCheck % firstDivisor;
      printf("Result using %i: %i\n", firstDivisor, result);

      // Check the divisibility using second divisor
      result = numberToCheck % secondDivisor;
      printf("Result using %i: %i\n", secondDivisor, result);
}
```

Here is the output of the above program.

```
Result using 3: 0
Result using 5: 2
```

The above results show that 12 is divisible by 3 but 12 is not divisible by 5.

Here is a program that tests if numbers are even. An even number is divisible by two.

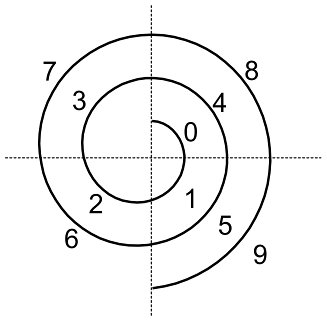
```
#include <stdio.h>

int main() {
      int firstNumberToCheck  = 4;   // Number to check divisibility by 2
      int secondNumberToCheck = 5;   // Another "

      // Check if first number is even
      if(firstNumberToCheck%2 == 0) {
             printf("%i is even\n", firstNumberToCheck);
      }
      else {
             printf("%i is not even\n", firstNumberToCheck);
      }

      // Check if second number is even
      if(secondNumberToCheck%2 == 0) {
             printf("%i is even\n", secondNumberToCheck);
      }
      else {
             printf("%i is not even\n", secondNumberToCheck);
      }
}
```

Here is the output of the above program.

```
4 is even
5 is not even
```

Plan 3. Cycle Position Plan

This plan requires an understanding of the mod operator.



It is possible to form a series of numbers into a cycle. Each number will then have a relative position within the cycle. For example we can to take a series of numbers beginning with zero and group them by fours. Each number would then have a relative position within each cycle from zero to three. In the figure above we see such a cycle. The numbers are in four groups and each group has a relative. Numbers with position 0 are { 0, 4, 8, … }, numbers with position 1 are { 1, 5, 9, … } and so on.

We can determine the position of a number in a cycle using the mod operator. As a general rule numbers can be brought into a cycle of size *n* by applying mod *n*.

x % n gives the position of *x* in a cycle of size *n*

For example if we want to create a size 3 we can apply mod 3 and we can then find positions of numbers in this cycle.

...

`9 % 3` gives **`0`**

`10 % 3` gives **`1`**

`11 % 3` gives **`2`**

`12 % 3` gives **`0`** …and so on.

One useful application of this idea is to bring random numbers into a range. In the C/C++ language random numbers are generated in a range from 0 to the largest possible integer value (with 4 byte integers this is 2147483647). If we want to generate a random number in a specified range, we can take the random number given by the standard library function **`rand()`** and find its position in a specified cycle.

x % n gives the a value in the range 0 to *n-1*

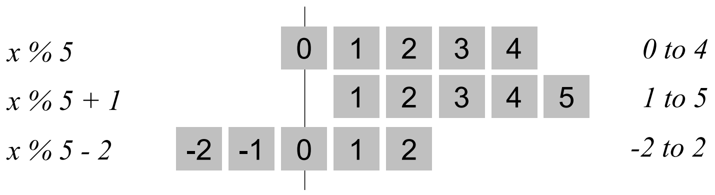
If we wanted to have a random number between 0 and 4 we can apply mod 5.

```
myRand = rand() % 5;
```

If we want a random number between 1 and 5 we can shift the previous range by adding 1 to the result.

```
myRand = rand() % 5 + 1;
```

We can also shift such a range in a negative direction. The diagram below shows a range and how it can be visualised when shifted.



We can create a function that generates a random number between 1 and 10 as follows.

```
int rand1to10() {
      return rand()%10 + 1;
}
```

We can generalise this function to apply settable upper and lower limits.

```
int myRand(int lowerLimit, int upperLimit) {
      return rand()%(upperLimit-lowerLimit+1) + lowerLimit;
}
```

Plan 4. Number Decomposition Plan

This plan requires an understanding of the mod and division operators.

We can use the division and mod operators to tear numbers apart. For example, if we want to find the last two digits of 12345 we can apply mod 100. For decimal digits the following rules apply.

`x % 10`	gives	the last digit
`x % 100`	gives	the last two digits
`x % 1000`	gives	the last three digits
`x % 10000`	gives	the last four digits …and so on.

Applying a similar idea we can discover the first digits of a number using the division operator. Using a 5 digit number, the following rules apply.

`x / 10000`	gives	the first digit
`x / 1000`	gives	the first two digits
`x / 100`	gives	the first three digits
`x / 10`	gives	the first four digits.

To find the third last digit of a decimal number we can apply the following operation.

```
thirdLastDigit = x % 1000 / 100;
```

Plan 5. Initialisation Plan

This plan requires an understanding of variables and the assignment operator.

Initialisation is commonly applied within other plans.

Failing to initialise variables before they are used can lead to errors.

It is recommended that you initialise all variables when you declare them.

In the following example **`sum`** is initialised to 0 as this is an appropriate sum before summing commences.

```
int sum = 0;
```

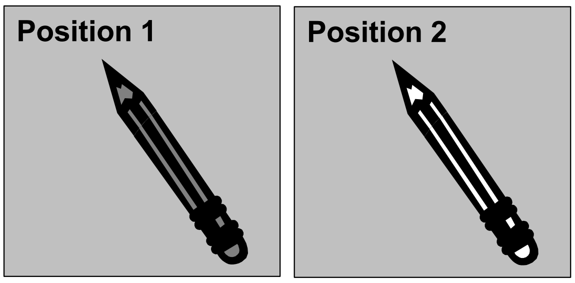
In some plans it may be necessary to initialise an array of items. For instance, here we are initialised an array used to tally letters in a message.

```
#include <stdio.h>

int main() {
      int letterCount[26]; // Array to store count of letters
      int i;               // Iterative counter

      // Initialise array of counts
      for(i=0; i<26; i++) {
             letterCount[i] = 0;
      }

      ...
}
```

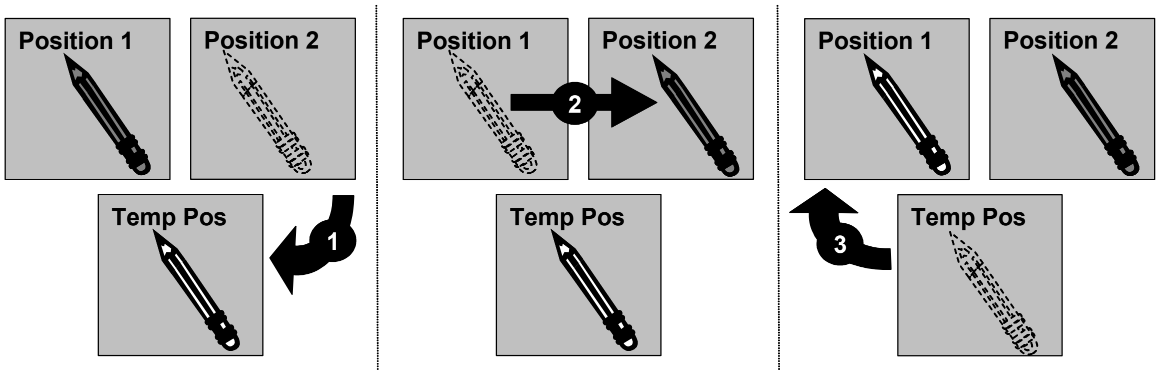
Plan 6. Triangular Swap Plan

This plan requires an understanding of variables and the assignment operator.

Consider how you swap two items. Imagine two pencils in front of you. To swap their positions you would pick up one with one hand, the second with your other hand and then place each in their new positions.



A computer can only perform one action at a time. Now, imagine that you only have one hand; how would you swap the positions of the two pencils now? Keep in mind also that when a variable is assigned a new value, the old value is replaced and cannot be accessed later. Attempting to swap using the above method will result in two copies of the same value.



To achieve a swap a temporary position is needed. One of the pencils could be moved to the temporary position; the second pencil could be moved to its new location; finally the first pencil could be moved from the temporary position to its new position.

Here is an example in the context of a full program.

```
#include <stdio.h>

int main() {
      int firstPosition  = 5; // First position containing value to swap
      int secondPosition = 6; // Second position containing value to swap
      int tempPosition;       // Temporary position for swap

      // Output the numbers after the swap
      printf("Before Swap...\n");
      printf("First: %i, Second: %i\n", firstPosition, secondPosition);

      // Swap the two numbers in a triangular swap
      // 1. Copy the value from the second position to temp
      tempPosition = secondPosition;

      // 2. Copy the value from the first position to the second
      secondPosition = firstPosition;

      // 3. Copy the value from the temp position to the first
      firstPosition = tempPosition;

      // Output the numbers after the swap
      printf("After Swap...\n");
      printf("First: %i, Second: %i\n", firstPosition, secondPosition);
}
```

Here is the output of the above program.

```
Before Swap...
First: 5, Second: 6
After Swap...
First: 6, Second: 5
```

The above results show the values are swapped and not duplicated.

Plan 7. Guarded Exception Plans (including Guarded Division Plan)

This plan requires an understanding of the `if` *statement.*

When a program compiles and runs, there are still opportunities for things to go wrong. Usually such *logic errors* occur around or outside *boundaries* of the data being worked on. Such boundaries include:

- Absence of data where some is expected,
- Negatives or zero where positives are expected,
- Too much data where a finite amount is expected, and
- Values outside an acceptable range.

To create reliable, "bullet proof" programs, these boundary conditions need to be considered.

There are also time where a program may encounter data that, when used in operations, will cause the operating to stop the program.

In mathematics, if a number is divided by zero the result is undefined. If a program attempts to divide by zero, the operating system will close the program down. Whenever we perform a division where the second operand could be zero, we must test the second operand before performing the division and prevent the division from taking place if it is zero.

Here is an example in the context of a full program.

```
int main() {
      int firstOperand;  // First operator for division
      int secondOperand; // Second operator for division

      // Gather inputs for division
      printf("Enter two integers for division: ");
      scanf("%i %i", &firstOperand, &secondOperand);

      // Test second operand
      if(secondOperand != 0) {

            // Perform division
            printf(
                  "%i divided by %i is %i",
                  firstOperand,
                  secondOperand,
                  firstOperand / secondOperand
            );
      }
}
```

Here is the output of the above program when the value 5 is given as the second operand.

```
Enter two integers for division: 10 5
10 divided by 5 is 2
```

When a zero value is given for the second operand, no output is produced and the program ends.

```
Enter two integers for division: 10 0
```

Here is another example that incorporates Guarded Division into a function which calculates an average from a given sum and count.

```
int average(int sum, int count) {

      // Test against dividing by zero
      if(count == 0) {
            return 0;
      }

      // Perform division as normal
      else {
            return sum / count;
      }
}
```

Plan 8. Counter Controlled Loop Plan

This plan requires an understanding of looping constructs.

A Counter Controlled uses a counter variable which is incremented until a set number of repetitions is achieved. The loop will continue regardless of any other event that may occur during repetition.

The following example reads in 10 integers from a user and calculates the sum. The program will continue regardless of what the user inputs. We usually use **`for`** loops to achieve counter controlled loops.

```
#include <stdio.h>

const int NUMBER_OF_INPUTS = 10;

int main() {
      int i = 0;      // Loop iterator
      int sum = 0;    // Sum of numbers input
      int userInput; // Input from user

      // Calculate the sum
      for(i=0; i<NUMBER_OF_INPUTS; i++) {
             printf("Enter a number: ");
             scanf("%i", &userInput);
             sum += userInput;
      }

      // Output the sum
      printf("Sum: %i\n", sum);
}
```

Counter Controlled loops are often used with arrays. When this happens the loop iterator can serve the dual purpose of being an index into the array. For an example of this see the initialisation of an array in Plan 5.

Plan 9. Primed Sentinel Controlled Loop Plan

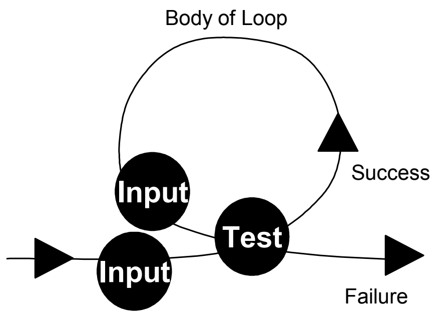
This plan requires an understanding of looping constructs.

A Primed Sentinel Controlled Loop allows repetition until an event takes place or some target value (the sentinel) is discovered.

Here is an example including a primed sentinel-controlled loop. Not that the loop tests **`userInput`** to determine if it should continue looping. The variable is being compared to the sentinel value **`SENTINEL`**. The value of **`userInput`** is primed with an initial user input before the loop begins. Although this adds some redundancy (the input statement appears twice) there can be efficiency savings made when the user enters the sentinel value in the first instance (which is not uncommon).

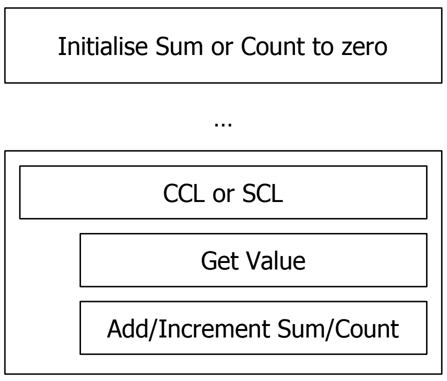
```
#include <stdio.h>

const int SENTINEL = 9999;

int main() {
      int sum = 0;   // Sum of numbers input
      int userInput; // Input from user

      // Get the first user input
      printf("Enter a number (%i to end): ", SENTINEL);
      scanf("%i", &userInput);

      // Calculate the sum
      while(userInput != SENTINEL) {
             sum += userInput;
             printf("Enter a number (%i to end): ", SENTINEL);
             scanf("%i", &userInput);
      }

      // Output the sum
      printf("Sum: %i\n", sum);
}
```

If the user where to enter the sentinel value as their first input, the loop would never be entered. The sum will also be correct as we are checking each user input before it is added to the sum. This avoids accidentally including the sentinel value in the sum.



Plan 10. Sum and Count Plans

This plan requires an understanding of looping constructs and initialization.



Two frequently practiced programming activities are summing or counting values. These simple processes are easily achieved, but also easily messed up. Both plans are achieved by using a variable to accumulate the sum or count as values are encountered. The key to both is assuring that the sum or count variable is initialised to zero. Failing to initialise such a variable will not stop your program from compiling. In many instances an uninitialised variable will have a value of zero so the program will work, but it will not work all the time. Just remember:

INITIALISE SUM AND COUNT VARIABLES

Below is an example which inputs and sums 5 numbers from a user. Note a Counter Controlled loop is used to control repetitions as we know how many are desired before the looping begins.

```
#include <stdio.h>

const int NUMBER_OF_INPUTS = 5;

int main() {
      int userInput = 0; // Input from user
      int sum = 0;       // Sum of inputs INITIALISED
      int i;             // Iterative counter

      // Counter Controlled loop to repeat inputs
      for (i=0; i<NUMBER_OF_INPUTS; i++) {

             // Prompt for input
             printf("Please enter an integer: ");
             scanf("%i", &userInput);

             // Add input to sum
             sum += userInput;
      }

      // Output the sum
      printf("Sum of numbers entered: %i\n", sum);
}
```

The output of the above program will resemble the following.

```
Please enter an integer: 1
Please enter an integer: 2
Please enter an integer: 3
Please enter an integer: 4
Please enter an integer: 5
Sum of numbers entered: 15
```

The following is an example which counts numbers entered by a user unit the value 9999 is encountered as a sentinel.

```
#include <stdio.h>

const int SENTINEL = 9999;

int main() {
      int userInput = 0; // Input from user
      int count = 0;     // Count of inputs INITIALISED

      // Prompt for initial input
      printf("Please enter an integer: ");
      scanf("%i", &userInput);

      // Test for sentinel
      while( userInput != SENTINEL ) {

             // Count input
             count++;

             // Subsequent input
             printf("Please enter an integer: ");
             scanf("%i", &userInput);
      }
      printf("You entered %i inputs\n", count);
}
```

The output of the above program will resemble the following.

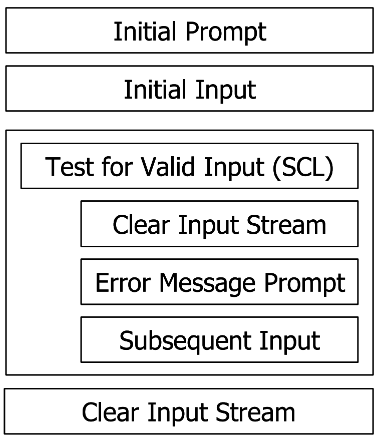
```
Please enter an integer: 1
Please enter an integer: 2
Please enter an integer: 3
Please enter an integer: 9999
You entered 3 inputs
```

Plan 11. Validation Plan

This plan requires an understanding of loops and the ***`scanf()`*** *function (or equivalent).*

When dealing with inputs from users one can never assume they will enter what is expected. It is therefore important, for critical systems, to validate that users have entered what they were expected to enter, and repeat inputs, with appropriate messages, in the case where users enter invalid inputs.

The plan shows here prompts the user and accepts an initial input. The value is then tested as the condition of a Sentinel Controlled loop where the sentinel is a valid input.



Testing for validity can take two forms:

- Testing if a valid input type has been entered, for instance, if an integer is expected, it is important to know that one has been entered.
- Once the first test has been satisfied, and where a value within a specified range is expected, then the value of the input should be tested.

The user will usually enter a valid input in the first instance, but if they do not, in the loop an error message is output and a subsequent input is gathered. This looping can continue indefinitely until the user enters a valid value.

After each input (within the loop and after the loop) the input stream is cleared. If the user has entered additional, unwanted data, either accidentally or maliciously, then it will be removed before the next input is sought.

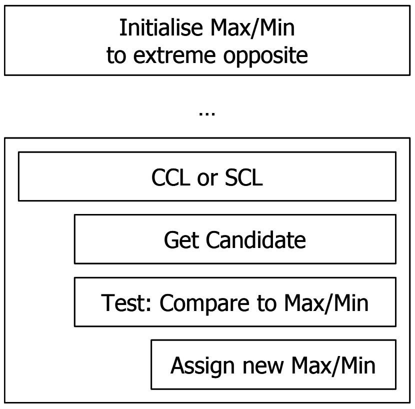
Here is an example function that gathers a valid integer in a specified range.

```
int getValidIntegerInRange(int lowestAllowed, int highestAllowed) {
      int userInput = 0;      // Input from user
      int inputsGathered = 0; // Number of inputs from scanf()

      // Prompt for initial input
      printf(
            "Please enter an integer between %i and %i: ",
            lowestAllowed, highestAllowed
      );
      inputsGathered = scanf("%i", &userInput);

      // Test for valid input
      while(
            inputsGathered !=1 ||
            userInput < lowestAllowed ||
            userInput > highestAllowed
      ) {

            // Clear standard input
            scanf("%*[^\n]");
            scanf("%*c");

            // Error message prompt
            printf(
                  "Invalid input. "
                  "Please enter an integer between %i and %i: ",
                  lowestAllowed, highestAllowed
            );
            inputsGathered = scanf("%i", &userInput);
      }

      return userInput;
}
```

Note that where inputs are gathered from the user, the return value from **`scanf()`** is also captured. The function **`scanf()`** will attempt to input values according to the format string, storing the values at the addresses provided. The return value of **`scanf()`** is not an input value, but the number of values that have been successfully input and stored. Using this we can determine if an appropriate value has been entered by the user. See the description of **`scanf()`** in Appendix 1 for more detail.

Plan 12. Min/Max Plans

This plan requires an understanding of looping constructs and the ***`if`*** *statement.*



To find the minimum or maximum from a number of user inputs, it is not necessary to keep all candidates, just the current min/max at any stage.

This process starts by selecting an initial value for the min/max variable. If searching for a maximum, initialise to the minimum possible value. If searching for the minimum, initialise to the maximum possible value. In that way the first value encountered will become the new min/max. Alternately the first value encountered (if it can be guaranteed there will be a single value) can be used as the initial value for the min/max.

As each candidate is presented within a loop (a counter controlled loop or sentinel controlled loop) it needs to be compared with the current-max/min. If searching for a maximum and the candidate is greater than the current maximum, then the candidate will be assigned as the new current-maximum.

The following example inputs 5 numbers between 0 and the largest integer value allowed. Inputs are gathered from a user using **`getValidIntegerInRange()`** as shown in Plan 11 above. The **`maxNumber`** variable is used to store the current maximum and it is initialised to 0 which is the smallest input allowed.

```
#include <stdio.h>
#include <limits.h>

const int NUMBERS_TO_READ = 5;

int getValidIntegerInRange(int lowestAllowed, int highestAllowed);

int main() {
      int i;             // Iterative counter
      int input;         // Validated Input from user
      int maxNumber = 0; // Current maximum initialised to
                         //   minimum possible value

      // Get inputs from user
      for(i = 0; i < NUMBERS_TO_READ; i++) {
             input = getValidIntegerInRange(0,INT_MAX);

             // Compare with current max and assign if greater
             if(input>maxNumber) {
                    maxNumber = input;
             }
      }

      // Output the max
      printf("The maximum was: %i\n", maxNumber);
}

int getValidIntegerInRange(int lowestAllowed, int highestAllowed) {
      ...
```

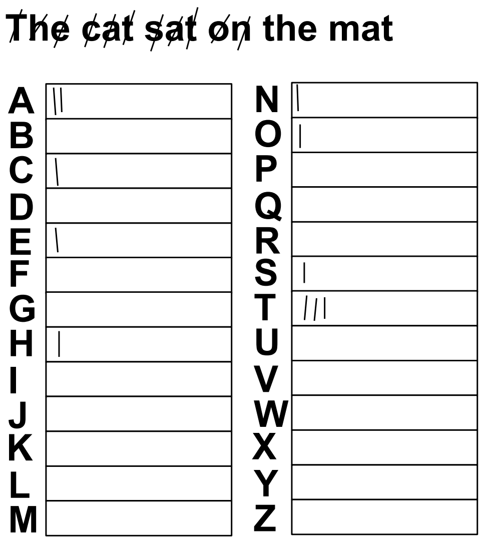
Note that each input is compared with the current maximum. Where a candidate is found to be greater than the current maximum it replaces the current maximum and is used for future comparisons.

Plan 13. Tallying Plan

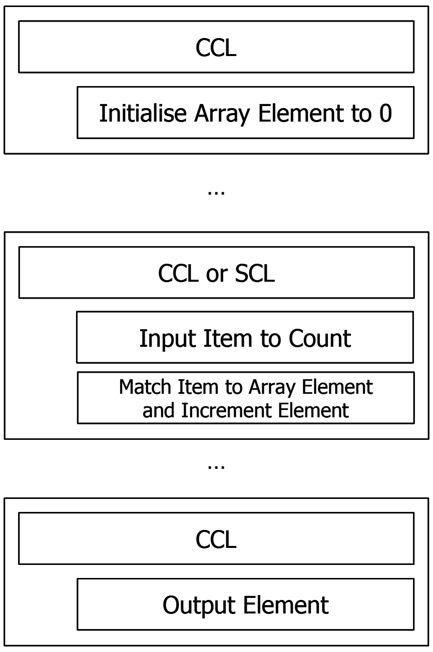
This plan requires an understanding of arrays and looping constructs.

As well as being able to store individual values in an array we can also use arrays to represent counts of occurrences of a set of values.

For instance if I asked you to count each letter in the sentence, "The cat sat on the mat", you could set up a sheet and tally each letter in the sentence. We start off with a blank sheet where the tally each letter is empty (zero). We process each letter in turn, crossing it off in the sentence as it is processed. When we encounter a letter, we place a tally mark in the box on our sheet that relates to that letter. We can continue this until all the letters are processed, at which stage the number of tally marks next to each letter is the number of occurrences of that letter.



We can apply a similar strategy in code using an array.


CCL
Initialise Array Element to 0
...
CCL or SCL
Input Item to Count
Match Item to Array Element and Increment Element
...
CCL
Output Element

We will create an array with enough elements to represent the set of values we are counting. If we are counting the letters of the alphabet we need an array with 26 elements. Before we start counting we must first initialise the array to be sure the count of all values is zero.

We can then process the values, matching them to the relevant element of our array and 'adding another tally mark' (incrementing the count) for that value.

When we have processed all items of interest the values in the array will be the counts of the items encountered. If we wish we can output the counts of the letters encountered.

The following code is an example of such a strategy.

```
#include <stdio.h>
#include <ctype.h>

const int SENTINEL = 9999;

int main() {
      int letters[26];  // Array for tallying letters encountered
      int i;            // Iterative counter
      char inputLetter; // Letter from user

      // Initialise all array elements to 0
      for(i=0; i<26; i++) {
            letters[i] = 0;
      }

      // Process the user input until end of line
      printf("Please input a sentence...\n");
      scanf("%c", &inputLetter);
      while(inputLetter != '\n') {
            if(isalpha(inputLetter)) {
                  letters[tolower(inputLetter)-'a']++;
            }
            scanf("%c", &inputLetter);
      }

      // Output occurrences of letters which have occured once or more
      for(i=0; i<26; i++) {
            if(letters[i] > 0) {
                  printf("%c: %i\n", 'a'+i, letters[i]);
            }
      }
}
```

Notice first that the array is initialised, the values are counted and then the counts are output. See the language reference for descriptions of **`isalpha()`** and **`tolower()`**.

The array used is an array of integers, which is appropriate as we are storing counts of letters and not the letters themselves. The array elements are referenced by index and the indices are integers, so this means we have to translate each character into a number to find the array element that relates to that letter. We can associate each alphabetic letter with a number in order starting from 'a' being 0, 'b' being 1 and so on. To achieve this we can convert each letter to lower case and deduct the value of 'a' as follows.

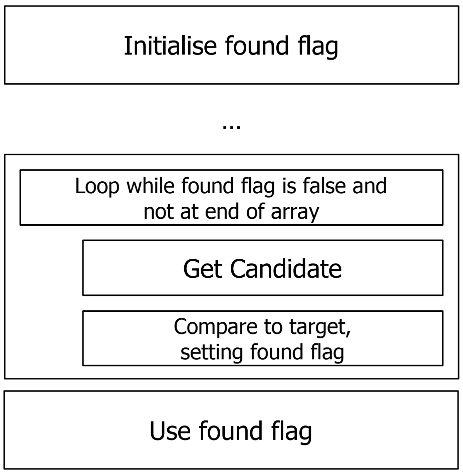
```
'a' - 'a' → 0
'b' - 'a' → 1
'c' - 'a' → 2
     ...
'z' - 'a' → 25
```

Once we have a letter's position in the alphabet we can use this as the index into the array to access the array element that relates to that letter of the alphabet. When we are counting a particular letter, we will translate it into a number, find the array element and increment its value. This is achieved in the statement from the above example shown below.

```
                  letters[tolower(inputLetter)-'a']++;
```

Plan 14. Search Algorithm

This plan requires an understanding of looping constructs and arrays.


Initialise found flag
...
Loop while found flag is false and not at end of array
Get Candidate
Compare to target, setting found flag
Use found flag

This plan and the next are approaching the scale of a full algorithm and could exist independently as useful functions.

The key to efficient searching is to search only the parts of the search space (say the elements of an array) necessary to discover the value sought. Of course, if the location of the target value is unknown then the amount of searching required cannot be predicted, but, if we are seeking the presence of a target value we should be able to stop searching after we discover the value. In the case that the target value is not present, searching will continue until the end of the search space is reached.

One way to achieve this is through a combination of a sentinel controlled loop that searches for the target value as a sentinel and a counter controlled loop that stops when the end of the search space is reached. We can use a Boolean flag to control the test for the target value and the value of this flag after the search will tell us if the target value is present. Here is an example function that searches an array for a target value.

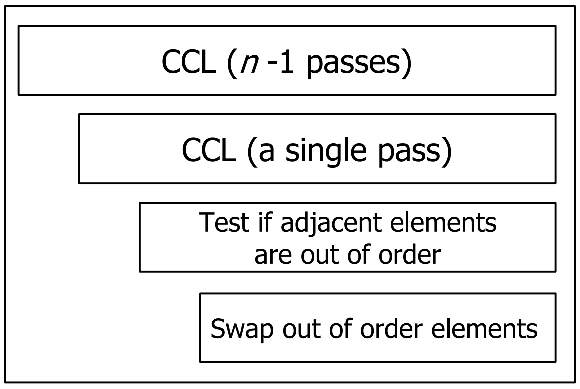
```
bool search(int targetValue, int array[], int arrayLength) {
      bool found = false; // Boolean search flag
      int i = 0;          // Iterative counter

      // Search until found or end of array
      while(!found && i<arrayLength) {

            // Match array element to target value
            found = array[i]==targetValue;
            i++;
      }
      return found;
}
```

Of course, this approach will only work if we are seeking the presence of a target value. If we wish to count the occurrences of a value we will need to search the entire search space, so no saving can be made.

```
int countValues(int targetValue, int array[], int arrayLength) {

      int i;       // Iterative counter
      int count=0; // Times targetValue has been encountered

      // Search entire array for occurrences of target value
      for(i = 0; i < arrayLength; i++) {
            if( array[i] == targetValue ) {
                  count++;
            }
      }

      // Return the count of occurrences
      return count;
}
```

Plan 15. Bubble Sort Algorithm

This plan requires an understanding of looping constructs and arrays.


CCL (*n* -1 passes)
CCL (a single pass)
Test if adjacent elements are out of order
Swap out of order elements

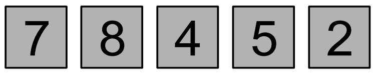
There are a many different algorithms which can be used to put elements in order. The Bubble Sort is presented here as it is easy to comprehend and use.

This algorithm works by looping through the array comparing each element with the following one, and swapping the values where necessary. Each pass through the array brings it closer to being sorted. The looping and swapping process must occur as many times needed to ensure the array is completely sorted. If we loop through the array *n-1* times (where *n* is the length of the array), it is guaranteed to be sorted.

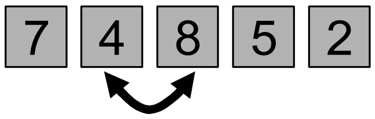
The process can be summarized as follows.

- Start at beginning of the array
- Compare first and second elements
- If out of order swap
- Compare the second and third elements
- If out of order swap
- Continue comparing adjacent pairs in the array, from beginning to end; this constitutes a single pass.
- Perform *n-1* passes to completely sort the array.

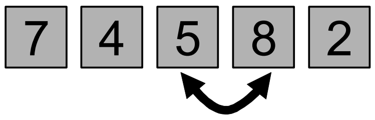
Consider the following array.

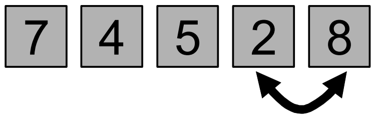


Starting at the beginning we compare the first two values. They are in order so we do not swap them. The second and third values are out of order and must be swapped. The outcome is shown below.



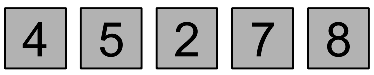
We continue comparing and swapping adjacent values if needed until we get to the end of the array.



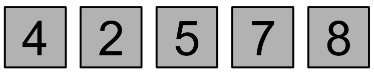


The state of the array after one pass is shown above. We will complete four passes through the array. The state of the array after each pass is shown below.

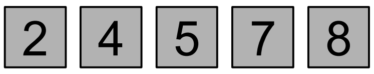
After second pass



After third pass



After fourth (final) pass



The following program will perform a *bubble sort* on an array of integers to put them in ascending order.

```
#include <stdio.h>

const int MAX_LENGTH = 5;

int main() {
      int array[MAX_LENGTH] = {9,8,2,5,4}; // Unsorted array
      int i, j;                            // Loop iterators
      int temp;                            // For swapping

      // Pass through the array MAX_LENGTH-1 times
      for( i = 0; i < MAX_LENGTH-1; i++){

            // For each pair of consecutive numbers
            for( j = 0; j < MAX_LENGTH-1; j++) {

                  // Test if the pair is out of order
                  if ( array[j] > array[j+1] ) {

                        // Swap using triangular swap
                        temp = array[j];
                        array[j] = array[j+1];
                        array[j+1] = temp;
                  }
            }
      }

      // Output the array after sorting
      for(i = 0; i < MAX_LENGTH; i++){
            printf("%i ",array[i]);
      }
      printf("\n");
}
```

Notice the above code contains two for loops, one inside the other. The outer loop ensures that *n-1* passes are performed. Each iteration of the outer loop, the inner nested loop compared each adjacent value in the array and swaps it if necessary.

Bubble sort is not the most efficient sorting algorithm. For large and unordered data faster sorting algorithms are available. The efficiency of the Bubble Sort algorithm can be improved by applying the following two modifications.

- Reduce the number of comparisons by one for each pass. After the first pass the greatest value will be pushed to the rightmost element. After two passes, the final two elements will contain the two greatest values in sorted order and so on. To achieve this, the value of **`i`** can be deducted from the upper limit of the inner loop.
  `j < MAX_LENGTH-1-i;`
- For an array that contains values that are nearly already sorted, it is possible to reach a sorted state before *n-1* passes have been made. The array can be determined to be in a sorted state when a complete pass has been performed in which no swaps are made. A Boolean flag **`swapsMade`** can be used which is set to **`false`** at the beginning of each pass. If it is still false at the end of the pass, no swaps have been made and the array is in sorted order. This flag can be incorporated into the test of the outer loop.

Plan 16. Command Line Arguments Plan

This plan requires an understanding of command line arguments and the ***`if`*** *statement.*

If information provided to a program from the command line is crucial to the successful running of the program, then the number of arguments needs to be checked at the beginning of program execution.

```
#include <stdio.h>

int main(int argc, char *argv[]) {

      // Check for the correct number of arguments
      if ( argc < 2 ) {
            printf("USAGE: %s secondArgument\n", argv[0]);
            exit(1);
      }

      // Rest of program
      ...
}
```

The arguments to the **`main()`** are **`argc`** (the number of command line arguments) and **`argv`** (an array of strings, each containing and argument). The code above shows a test for the minimum number of command line arguments needed. In this case the program expects two arguments and any extras will be ignored. If the user runs the program and does not supply a second argument, then an error message is output and the program exits. Note that the name of the executable file will be stored in **`argv[0]`** and this is used in the error message; the name of the executable could change, but the error message will always be correct.

Once the number of command line arguments has been checked, the validity of the values supplied may then also need to be checked.

Plan 17. File Use Plan

This plan requires an understanding of files and the if statement.

When using input files, where data sourced from those files is critical to the running of a program, the following 5 Step Plan should be taken. This plan takes checks that the file is available for use. It closes the stream when it is no longer needed; this is important to avoid data loss.

1 Create a stream (FILE) pointer

```
FILE *inputStream;
```

2 Open a file and attach the stream

```
inputStream = fopen("myfile.txt","r");
```

3 Test the stream, this testing the file opening

```
if (inputStream == NULL) {
      printf("Error opening file");
      exit(1);
}
```

4 Use the stream for input or output (this will of course vary according to the needs of the input stream)

5 Close the stream

```
fclose(inputStream);
```

Plan 18. Recursion Plans (single- and multi-branching)

*This plan requires an understanding of the **`if`** statement and calling functions.*

A recursive function is one which calls itself, either directly or indirectly. Recursive functions are very simple, but can achieve quite complex solutions by solving a problem a small part at a time. Recursion is a way of achieving repetition in a program.

Recursive functions have two parts: a stopping case and a recursive case. An **`if`** statement is used to determine which case should be used as shown in the skeleton below.

```
int exampleRecursiveFunction( ...ARGUMENTS... ) {

      // Stopping case
      if( TEST TO SEE IF RECURSION SHOULD STOP ) {
             ...;
      }

      // Recursive case
      else {
             ...
             exampleRecursiveFunction( ... );
             ...
      }
}
```

The recursive case contains a recursive function call. Each time the recursive function is called, the arguments passed should be slightly different to those used to call the current function. In that way progress is made towards the end of recursion.

The stopping case is reached when some end has been achieved. It contains no further recursive function calls.

The following function is a recursive function that counts down from any positive number to zero.

```
void countDown(unsigned int number) {

      // Stopping case
      if(number == 0) {
             printf("0\n");
      }

      // Recursive case
      else {
             printf("%i\n", number);
             countDown(number - 1);
      }
}
```

The stopping case for this function occurs when the value of **`number`** is zero. If we called this function once and passed it the value zero, it would use the stopping case immediately and end. If a greater number is passed the recursive case will be used and the recursive function call within that passes a number one less each time. In this way the stopping case will eventually be reached.

We could start the recursive process, starting at the number 3, by calling the **`countDown()`** function from the **`main()`** and passing the value 3.

```
int main() {

     // Start the count down at 3
     countDown(3);
}
```

The output of this program would be as follows.

```
% a.out
3
2
1
0
%
```

Below is an example of another recursive function that can be used to calculate factorials. The factorial of an integer is the integer multiplied by all the positive integers less than it to one. We denote the factorial of a number using an exclamation (!) like as follows.

$$5! = 5 \text{ x } 4 \text{ x } 3 \text{ x } 2 \text{ x } 1$$

The factorial for 4! can be expressed as follows.

$$4! = 4 \text{ x } 3 \text{ x } 2 \text{ x } 1$$

If we wanted to, we could now express 5! as follows.

$$5! = 5 \text{ x } 4!$$

You can see the recursive nature of this equation already. We can make this a general equation as follows. This is our recursive case.

$$n! = n \text{ x } (n-1)!$$

We also need to express a stopping case for this, which is when n is 1.

$$1! = 1$$

This is a mathematical definition of a recursive process. If we were to run it through for say 4! it would look as follows.

$$4! = 4 \text{ x } 3!$$

$$3! = 3 \text{ x } 2!$$

$$2! = 2 \text{ x } 1!$$

We know that 1! is equal to one. We can now start working our way back up.

$$2! = 2 \text{ x } 1! \rightarrow 2 \text{ x } 1 \rightarrow 2$$

$$3! = 3 \text{ x } 2! \rightarrow 3 \text{ x } 2 \rightarrow 6$$

$$4! = 4 \text{ x } 3! \rightarrow 4 \text{ x } 6 \rightarrow 24$$

So 4! is 24. We can write a function that calculates factorials using the process we have described as follows.

```
int factorial(unsigned int number) {

      // Stopping case
      if (number <= 1) {
            return 1;
      }

      // Recursive case
      else {
            return number * factorial(number - 1);
      }
}
```

You will notice that with this function, as well as actions being achieved on the way to the stopping case, calculations are happening through the return values after the stopping case has been reached and while working back to the original function call. In order to complete the expression in the recursive case…

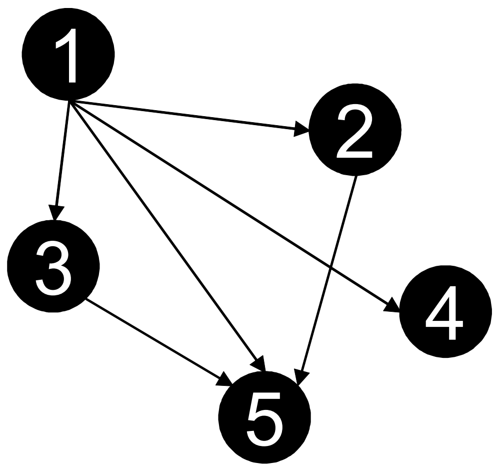
```
            return number * factorial(number - 1);
```

…the factorial function needs to be called. We must wait for this function to end and return a result before we can complete the expression.

This function is an example of single branching recursion. The recursive case contains only a single function call, so the recursive process will continue until a single stopping case is reached, after which the calls will roll back to the original function call.

A multi-branching recursive function contains more than one recursive function call in the recursive case. This is useful for problems where from a particular point there may be several following points that need to be probed and from each of those points further points need to be probed and so on. There may be multiple stopping points that can be reached in such cases also. Consider for example, a directed graph. A directed graph is described by its points and the vertices between points that run in one direction only. The vertices are like one-way streets that join one place to another.

The picture below describes a directed graph. The starting point is 1 and the ending point is 5. We can represent this information textually as shown with each vertex having a starting and ending point and a series of directed vertices that make up the graph.



```
1

5

1 2

1 3

2 5

3 5

1 4

1 5
```

Our task is to find how many paths lead from the starting point to the ending point assuming that there are no cycles in the graph. We can represent a graph as follows.

```
struct directedGraph {               // Describes a directed graph
     vertex vertices[MAX_VERTICES]; // The vertices that make up the graph
     int numVertices;               // The number of vertices
     int startPoint;                // The starting point
     int endPoint;                  // The end/target point
};
```

We can then create a recursive function that, when started at the start point, will discover how many paths lead to the end point.

```
int countPaths(directedGraph graph, int currentPoint) {
     int countPathsFromHere=0; // Paths in the graph starting here

     // Stopping case
     if(currentPoint == graph.endPoint) {

            // A complete path has been found
            return 1;
     }
     else {

            // Probe all paths that start here
            for(int i=0; i<graph.numVertices; i++) {
                   if(graph.vertices[i].from == currentPoint) {
                          countPathsFromHere += countPaths(
                                 graph,
                                 graph.vertices[i].to
                          );
                   }
            }

            // Return the number of completed paths staring here
            return countPathsFromHere;
     }
}
```

Assuming we have read in a graph into a structure variable called `graph` we could start this recursive process as follows, printing out the number of paths returned.

```
printf("%i\n", countPaths(graph,graph.startPoint));
```

Recursion is a less efficient way of achieving repetition than when using loops. However when a problem is being solved that is recursive by nature, writing recursive solutions can be far simpler than writing an iterative solution for the same functionality. Where the depth of recursion is on too deep, recursive solutions can be quite acceptable.

Strategies Index

Appendix B. Solution Sheet for Initial Goal/Plan Analysis Experiment

Write a program that will read in integers from a user and output their average. Stop reading when the value 99999 is input.

Appendix C. Answer Sheets for Problem Solving Experiments

Read in 10 positive integers from a user. Assume the user will enter valid positive integers only. Determine the maximum.

Read in any number of integers until the value 99999 is encountered. Assume the user will enter valid integers only. Output the average.

Input any number of integers between 0 and 9. Assume the user will enter valid integers only. Stop when a value outside this range is encountered. After input is concluded, output the occurrence of each of the values 0 to 9.

Appendix D. Web Survey Demographics and Computing Confidence Questions

To classify age, participants were asked "Which of the following ranges does your age fall into?" The following age ranges were presented.

- Less than 25
- 26 to 35
- 36 to 45
- 46 to 55
- 56 or over

To gauge each participant's general computing experience the following questions were asked.

- What is your level of computer use?
- How often would you use a web browser?

To each of these questions, the possible responses were as follows.

- No use
- Irregular use
- Weekly use
- Use every few days
- Daily use

It was desired for participants to have no previous programming experience. To distinguish this, participants were asked "Have you programmed before (not including HTML)?" and were able to select from the following answers.

- Never
- Some self-taught
- Formal training

Computing confidence was thought to be a possible differentiating factor in this experiment. A series of statements were presented to participants to measure their confidence with computers. The confidence statements were from a computing confidence test created by Cretchley (2006) and were presented in an unmodified manner. Previous evaluation of this test by Cretchley had proven it to be a reliable predictor of computing confidence. The statements were presented as follows (with negatively phrased statements identified.

- I have less trouble learning how to use a computer than I do learning other things.
- When I have difficulties using a computer, I know I can handle them.
- I am not what I would call a computer person. (phrased negatively)
- I enjoy trying things on a computer.
- It takes me longer to understand computers than the average person. (phrased negatively)
- I have never felt myself able to learn how to use computers. (phrased negatively)
- I find having to use computers frightening. (phrased negatively)
- I find many aspects of using a computer interesting and challenging.
- I don't understand how some people seem to enjoy so much time on a computer. (phrased negatively)
- I have never been very excited about computers. (phrased negatively)
- I find using computers confusing. (phrased negatively)

Possible responses to these confidence statements were as follows.

- Strongly Disagree
- Disagree
- Neutral
- Agree
- Strongly Agree

Each response has a value from 1 to 5. For positively phrased questions a response of “Strongly Disagree” is valued as 1, a “Neutral” response is 3 and a “Strongly Agree” response is valued at 5. For negatively phrased questions, this is reversed. The value of each response is summed to give a confidence measure that can be compared among participants.

As well as acting as a filter for volunteers who had previously completed a programming course, one of the intentions of this initial data was to balance the representation of participants in the control and experimental groups. Ultimately, balancing was unnecessary as participants, who grouped themselves according to their preferred dates, showed an even level of confidence between groups (there was no significant difference in average confidence levels between the two groups).

Appendix E. Post Experiment Interview Questions

Questions about the Maximum Problem

1. What is this problem statement asking?
2. What is meant by "positive integers"?
3. What does it mean by the user entering "valid positive integers only"?
4. What does it mean by "Determine the maximum"?

Discussion about Participant's Solution to the Maximum Problem

1. Lead me through your solution ... what does this part do?
2. Does your solution solve the problem?
3. Are there any improvements that could be made?

Questions the Averaging Problem

1. What is this problem statement asking?
2. What is meant by "Read any number of integers"?
3. What does it mean by "until the value 9999 is encountered"?
4. What does it mean by "Output the average"?

Discussion about Participant's Solution to the Averaging Problem

1. Lead me through your solution ... what does this part do?
2. Does your solution solve the problem?
3. Are there any improvements that could be made?

Questions about the Set Counting Problem

1. What is this problem statement asking?
2. What is meant by "Stop when a value outside this range is encountered"?
3. What does it mean by "output the occurrence of each of the values 0 to 9"?

Discussion about Participant's Solution to the Set Counting Problem

1. Lead me through your solution ... what does this part do?
2. Does your solution solve the problem?
3. Are there any improvements that could be made?

Appendix F. Exam Questions for Assessment Experiment

A cover page preceded this in the actual examination paper.

<u>NOTE</u>

There is a list of function specifications and other useful information on a page at the end of this exam paper.

<u>QUESTION 1</u> (10 marks, 12min)

What will the following output?

```
#include <stdio.h>

int testFunc(int *ptr, int num);

int main() {
      int x=7, y=3, z=5;
      printf("%i %i\n", x, y);
      z = testFunc(&y, x);
      printf("%i %i %i\n", x, y, z);
}

int testFunc(int *ptr, int num) {
      int temp;
      printf("%i %i\n", *ptr, num);
      temp = num;
      num = *ptr;
      *ptr = temp;
      printf("%i %i\n", *ptr, num);
      return num + (*ptr);
}
```

<u>QUESTION 2</u> (10 marks, 12min)

There are errors on three lines of the code below. Identify the lines with errors by number and give a corrected version for each of those lines.

```
01 #include <stdio.h>
02
03 const int NUM_ITEMS = 4;
04
05 int main() {
06      int i=0;
07      double items[NUM_ITEMS] = {1.2, 3.4, 5.6, 7.8}
08
09      for(i=0, i<=NUM_ITEMS, i++) {
10           printf("%d\n", items[i]);
11      }
12 }
13
```

Question 3 is on the next page.

QUESTION 3 **(10 marks, 12min)**

The following code is relevant to the instructions that follow.

```
#include <stdio.h>

const int ROWS = 3;
const int COLS = 5;

void print2DArray(const int array[ROWS][COLS]);

int main() {
      int arrayToPrint[ROWS][COLS] = {
            {7,8,2,5,4},
            {5,9,2,5,4},
            {9,3,2,1,7}
      };

      print2DArray(arrayToPrint);
}

// Your function would go here
```

In the context of the code above, create the function **`print2DArray()`** with the following prototype so that it prints the content of the array it is passed with a space between each number and with each row on its own line.

```
      void print2DArray(const int array[ROWS][COLS]);
```

Question 4 is on the next page.

QUESTION 4 **(10 marks, 12min)**

The following code is relevant to the instructions that follow.

```
#include <stdio.h>

const int MAX_NAME_LENGTH = 256;
const int NUM_EMPLOYEES = 4;

enum EmployeeType {manager, underling};

struct Employee {
	EmployeeType eType;
	int employeeID;
	char name[MAX_NAME_LENGTH+1];
	double payRate;
};

void raise(Employee *empPtr, double amount);

int main() {
	int i=0; // Iterative counter

	Employee employees[NUM_EMPLOYEES] = {
		{underling, 324, "Phil In",   23.00},
		{manager,   327, "Boss Hog",  59.00},
		{underling, 329, "Joe Dirt",  22.00},
		{manager,   332, "Phil King", 78.50}
	};

	for(i=0; i<NUM_EMPLOYEES; i++) {
		raise(&employees[i], 1.50);
		printf("%s %.2lf\n", employees[i].name, employees[i].payRate);
	}
}

// Your function would go here
```

In the context of the code above, create the function **`raise()`** with the following prototype so that it increases the **`payRate`** of a single employee by the given **`amount`**, but only if they are a *manager.*

```
void raise(Employee *empPtr, double amount);
```

Question 5 is on the next page.

QUESTION 5 **(10 marks, 12min)**

Read the following code to answer the questions that follow.

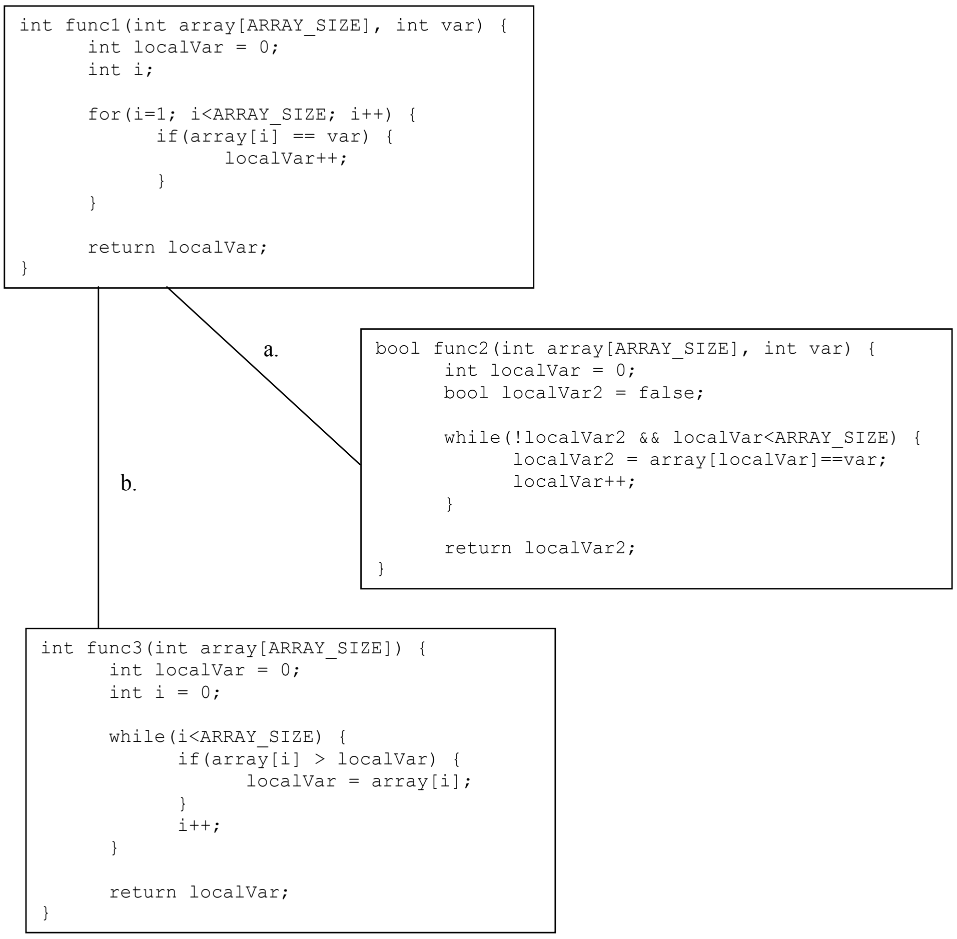
```
#include <stdio.h>

int mysteryFunction(int num1, int num2);

int main() {
      printf("%i\n", mysteryFunction(3,4));
}

int mysteryFunction(int num1, int num2) {

      // Stopping case
      if(num2 <= 0) {
            return 0;
      }

      // Recursive case
      else {
            return num1 + mysteryFunction(num1, num2-1);
      }
}
```

a. What will the code above output? (6 marks)
b. What would be a better identifier for the function `mysteryFunction()`? (4 marks)

Question 6 is on the next page.

QUESTION 6 (10 marks, 12min)

There are commonalities and differences in the strategies used in the following three functions. Read the functions in the boxes below and answer the questions that follow.



```
int func1(int array[ARRAY_SIZE], int var) {
      int localVar = 0;
      int i;

      for(i=1; i<ARRAY_SIZE; i++) {
            if(array[i] == var) {
                  localVar++;
            }
      }

      return localVar;
}
```

```
bool func2(int array[ARRAY_SIZE], int var) {
      int localVar = 0;
      bool localVar2 = false;

      while(!localVar2 && localVar<ARRAY_SIZE) {
            localVar2 = array[localVar]==var;
            localVar++;
      }

      return localVar2;
}
```

```
int func3(int array[ARRAY_SIZE]) {
      int localVar = 0;
      int i = 0;

      while(i<ARRAY_SIZE) {
            if(array[i] > localVar) {
                  localVar = array[i];
            }
            i++;
      }

      return localVar;
}
```

a. What is the common strategy used in both **`func1()`** and **`func2()`**? (5 marks)
b. What is the common strategy used in both **`func1()`** and **`func3()`**? (5 marks)

Below is a list of some of the strategies covered in the course.

- Average Plan
- Divisibility Plan
- Cycle Position Plan
- Triangular Swap Plan
- Counter Controlled Loop Plan
- Primed Sentinel Controlled Loop Plan
- Sum and Count Plans
- Validation Plan
- Min/Max Plans
- Tallying Plan
- Search Algorithm
- Bubble Sort Algorithm

Question 7 is on the next page.

QUESTION 7 (20 marks, 24min)

Write a function, using the following prototype, which will prompt the user and read in a valid positive integer. If the user enters invalid input, or a negative integer, the function will tell them their input was invalid and prompt them to enter another value. The function will repeat this until the user enters a valid input.

```
int getValidPositiveInteger();
```

For your reference, the following lines of code will clear the standard input stream.

```
scanf("%*[^\n]");
scanf("%*c");
```

QUESTION 8 (20 marks, 24min)

Write a `main()` function that will read in integers and output their average. Input will be gathered using the `getValidPositiveInteger()` function as described above (do not re-write that function). Stop reading when the value 99999 is entered (this is not to be used as an input).

There is a list of function specifications and other information is on the next page.

Relevant functions from stdio.h

`int printf(char *format, ...);`
Produces output to standard output according to string **`format`**. Returns the number of characters printed.

`int scanf(char *format, ...);`
Reads from standard input according to the string **`format`**, assigning data to the variables pointed to by the second and subsequent arguments. Returns the number of input items assigned.

Format Sequences

`%c`	`char`
`%i`	`int`
`%s`	string
`%u`	`unsigned int`
`%li`	`long int`
`%lu`	`long unsigned int`
`%hi`	`short int`
`%hu`	`short unsigned int`
`%f`	`float`
`%lf`	`double`
`%%`	percentage symbol

Field Width

For Integers
Placing a number between the **`%`** and format specifier for a format sequence will cause the integer to be output right-justified in a field width with that number of spaces. For example **`%3i`** will output numbers in a field width of 3 spaces. If the integer being output is longer than the field width, the field width will be 'pushed out' to accommodate the integer.

For Floating Point Numbers
A field width can be created in the same way as with integers. A precision (number of decimal places after the decimal point) can be specified by putting a point after the field width and a number of decimal places after that. For example **`%5.2lf`** will output a double with a field width of 5 spaces and a precision of two digits. It is possible to specify precision without a field with, for example **`%.2lf`**.

Format Flags

`-`	left justify
`+`	force printing sign
`0` (zero)	pads field width with spaces

Escape Sequences

`\n`	newline
`\t`	tab
`\"`	double quotes
`\\`	backslash

End of paper

Appendix G. Final Exam Answers for Assessment Experiment

A1 (2 marks each for first 3 lines, 4 marks for final line) Total 10

```
7 3
3 7
7 3
7 7 10
```

A2 (Lines 7, 9 and 10 contain errors, corrected versions below, 2 marks for missing ; at end of 7, 4 marks for using ; instead of , and using < instead <= at 9 (alt change NUM_ITEMS to NUM_ITEMS-1), 2 marks for %lf instead of %d) Total 10

```
01 #include <stdio.h>
02
03 const int NUM_ITEMS = 4;
04
05 int main() {
06      int i=0;
07      //double items[NUM_ITEMS] = {1.2, 3.4, 5.6, 7.8}
        double items[NUM_ITEMS] = {1.2, 3.4, 5.6, 7.8};
08
09      //for(i=0, i<=NUM_ITEMS, i++) {
        for(i=0; i<NUM_ITEMS; i++) {
10           //printf("%d\n", items[i]);
             printf("%lf\n", items[i]);
11      }
12 }
```

A3 (2 marks for counter variables, 4 marks for nested loos, 2 marks for correct printf in inner loop, 2 marks for printf at end of line) Total 10

```
void print2DArray(const int array[ROWS][COLS]) {
      int i=0, j=0;

      for (i=0; i<ROWS; i++) {
            for(j=0; j<COLS; j++) {
                  printf("%i ", array[i][j]);
            }
            printf("\n");
      }
}
```

A4 (2 marks for if, 2 marks for correct use of enumerated type, 3 marks for empPtr->eType, 3 marks for increase in pay rate) Total 10

```
void raise(Employee *empPtr, double amount) {
      if(empPtr->eType == manager) {
            empPtr->payRate += increase;
      }
}
```

A5 (6 for correct answer to a, 4 marks for correct answer to b) Total 10

```
a. 12
b. multiply() (or a name with an equivalent meaning)
```

A6 (5 marks for each correct answer) Total 10

a. SEARCH ALGORITHM
Both functions are searching for a value. func1() returns count of occurrences, func2() returns its presence.

b. PRIMED SENTINEL CONTROLLED LOOP
Both functions use a counter-controlled loop

A7 (5 marks for initial input, 4 marks for checking correct number of inputs, 2 marks for validating input is positive, 4 marks for clearing standard input after invalid input, 5 marks for subsequent input) Total 20

```
int getValidPositiveInteger() {
        int input=0;
        int valuesInput=0;

        printf("Enter an integer: ");
        valuesInput = scanf("%i",&input);
        while(valuesInput==0 || input<0) {
                scanf("%*[^\n]");
                scanf("%*c");
                printf("Invalid input. Enter an integer: ");
                valuesInput = scanf("%i",&input);
        }
        scanf("%*[^\n]");
        scanf("%*c");

        return input;
}
```

A8 (2 marks for initalising sum, 2 marks for initialising count, 2 marks for initial input, 2 marks for checking input is not sentinel, 2 marks for sum plan, 2 marks for count plan, 2 marks for subsequent input, 4 marks for guard on division, 2 marks for average calculation) Total 20

```
int main() {
        int input;
        int sum=0;
        int count=0;

        input = getValidPositiveInteger();
        while(input != SENTINEL) {
                sum += input;
                count++;
                input = getValidPositiveInteger();
        }

        if(count>0) {
                printf("Average: %i\n", sum/count);
        }
        else {
                printf("No numbers input\n");
        }
}
```

Appendix H. Experimental Curriculum Written Materials

The curriculum is presented over the following pages as it was presented to students during the experiment described in chapter 6. In the document, highlighting shows explicit strategy content elements that were removed to create the control curriculum.

JavaScript Reference

Author:	Michael de Raadt
	University of Southern Queensland
Email:	deraadt@usq.edu.au
Last Updated:	Tuesday 31st May, 2005
Curriculum:	A

1. First JavaScript Program

The process of writing a program in JavaScript is as follows.

1. Open a text editor (like Notepad)
2. Enter the JavaScript program within an HTML document
3. Save the document with a ".html" extension
4. Open the HTML document in a web browser.

When the browser opens the document it will run the JavaScript program or report errors in the code.

Writing programs is usually accomplished iteratively in small chunks. Start with a very basic program; save and open in a browser. Make a small change to the program, save then refresh the browser.

1.1. Hello World!

A traditional program to start programmers in a new language is one which outputs the message "Hello World!"

Exercise 1.1

Copy the following code (Code Example 1.1) into your text editor taking care not to introduce changes. The line numbers to the left of each line need not be entered; they are there so we can refer to a specific line in the code. Also, two symbols appear in the example which should be used as follows.

- Where the » symbol appears, press the TAB key; and
- Where the ¶ symbol appears, press the ENTER key.

In future examples, these symbols will not be shown explicitly, but will be used will be used when you write such code.

Save it as an HTML document with a filename like "hello.html"; open the document in your web browser.

```
01  <html>¶
02  »   <head>¶
03  »   »   <script type="text/javascript">¶
04  »   »   »   alert("Hello World!");¶
05  »   »   </script>¶
06  »   </head>¶
07  </html>¶
```

Code Example 1.1: Hello World!

1.2. JavaScript and HTML

HTML stands for HyperText Markup Language. Locating JavaScript code in an HTML document allows us to write and run simple programs easily.

An HTML document starts with an opening **`<html>`** tag and ends with a closing **`</html>`** tag. The document is divided into two parts.

The second part is the body, enclosed in **`<body>`**...**`</body>`** tags. The body contains text that will be shown in the browser window. Learning how to format and organised text in the body is interesting, but will not be covered here. In later examples we will use the body to add a label describing what JavaScript code we are testing.

The first part of an HTML document is the head, enclosed in **`<head>`**...**`</head>`** tags. The head contains extra information not shown in the browser window. It is here that we locate our JavaScript code. Within the head we add **`<script>`**...**`</script>`** tags to identify the start and end of our code. We also identify the scripting language used by adding the **`type="text/javascript"`** attribute to the starting **`<script>`** tag.

```
<html>
     <head>
         <script type="text/javascript">

                         JavaScript Code Here

         </script>
     </head>
     <body>

                         HTML Here

     </body>
</html>
```

Code Example 1.2: JavaScript in an HTML document

Exercise 1.2

As we progress through this study, we will use the above format repeatedly. We will enter JavaScript code in the block identified by the label "JavaScript Code Here" and possibly add a simple label in the block identified by "HTML Here".

Using the file you created in Exercise 1.1 (called "hello.html") remove the line containing **`alert(...);`** and replace it with a blank line. This will be the section label "JavaScript Code Here" in Code Example 1.2.

Below the **`</head>`** tag, add a new line, press TAB and type **`<body>`**. Add a blank line and on another new line press TAB and then add a closing **`</body>`** tag. The blank line will be the section labelled "HTML Here" above. We will use this section to write a simple description of future programs.

Choose "Save As..." from the File menu and name the file "template.html". When creating a new JavaScript program, open the template and save it under a new name, then start adding code.

1.3. Statements

A JavaScript program is made up of *statements*. A statement usually starts at the beginning of a line and ends with a semi-colon (`;`). In Code Example 1.1 there is a single statement at line 04. Most programs have several statements.

2. Calling Functions

Built into JavaScript are a number of functions which achieve common, basic tasks like:

- Gathering input data from a user
- Displaying output to a user
- Discovering information about data
- Converting data from one form to another

To use a function, it is not necessary to know how it is constructed or how it achieves its task, you just need to know how to *call* the function. To call a function you need to know:

1. Its name (which very briefly describes the function's purpose)
2. What arguments (inputs to the function) are needed
3. What you might get back from the function

2.1. `alert()`

There are a number of functions a JavaScript program can use to get information to a user. One such function is **`alert()`**.This function takes a message and outputs it in a window that pops up within the user's browser. An example of such a window is show in Figure 2.1.

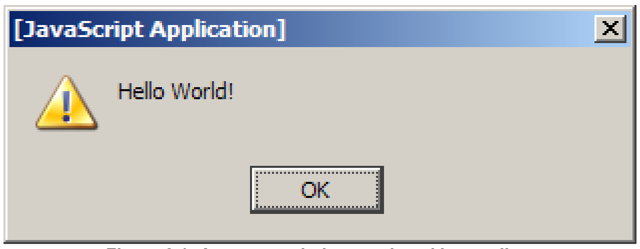


Figure 2.1: An output window produced by a call to `alert()`

In Code Example 1.1 (the first code example in this document) the **`alert()`** function is used on line 04 to produce the window above.

The **`alert()`** function has only one argument, the message to be output. As we will see later, it is possible to combine values together to form a single message output by this function.

Exercise 2.1

Open "template.html" and add a call to **`alert()`** in the JavaScript section. As an argument to the function, in between the parentheses **`()`** add your name surrounded by double quotes (`""`). Be sure the statement (the line) ends with a semi-colon (`;`).

In the HTML section (on the line after the opening **`<body>`** tag) Add text describing what the JavaScript program does. Text in this section is not part of the JavaScript program; only text between the opening **`<script>`** and closing **`</script>`** tags is regarded as JavaScript code. What text you add in the HTML section is up to you.

3. Values

In any programming language it is useful to distinguish different types of values so they can be treated differently in different circumstances.

3.1. Numbers

Numbers include integers (whole numbers like 1) and floating point numbers (numbers with a fraction after a decimal point like 1.23).

3.2. Strings

A string is a series of characters. A string can have many characters, a single character, or no characters at all (an empty string). To create a string, we use quotes to show the start and end of a string. Single or double quotes can be used as long as the same quotation mark is used at the start and end.

3.3. Booleans

There are only two Boolean values: **`true`** and **`false`**. These values do not need to be surrounded by quotes.

In Code Example 3.1 an example of each of the values above is output using **`alert()`**. On lines 04 and 05 strings are output using double and single quotes. On line 06 a number is output. On line 07 a Boolean value is output.

```
01 <html>
02      <head>
03           <script type="text/javascript">
04                alert("string content in double quotes");
05                alert('string content in single quotes');
06                alert(123);
07                alert(true);
08           </script>
09      </head>
10      <body>
11           Values example
12      </body>
13 </html>
```

Code Example 3.1: An example showing different values

Exercise 3.1

Using your template file, replicate Code Example 3.1. Does it output the values you expected it to?

Make the following changes.

1. On line 06 change the number from **`123`** to **`123.456`**. What happens?
2. On line 07 change the value of **`true`** to **`false`**. What happens?
3. Remove the quotes from around the string in the first call to **`alert()`** on line 04. What happens? Put the quotes back. What happens now?
4. Add another call to **`alert()`** with the argument **`abc`** (not in quotes). What happens?

4. Variables

4.1. What are Variables

A variable allows the storage of a value for later in the program. A variable can store any of the values shown in section 3. A variable is a piece of information named by an *identifier*. Where the identifier of a variable is located in a program, the value of the variable will be looked-up and used in its place.

4.2. Identifier Rules

There are some rules which constrain the identifiers you use.

1. Can contain:
    a. alphabetic characters **`A`** to **`Z`** and **`a`** to **`z`**
    b. numerals **`0`** to **`9`**
    c. underscores **`_`**
2. Cannot contain spaces, punctuation, quotes, or any characters not shown in 1 above.
3. Can start with:
    a. an alphabetic character **`A`** to **`Z`** and **`a`** to **`z`**
    b. an underscore **`_`**
4. Cannot start with a numeral or any character not shown in 3 above.

It should also be noted that identifiers are *case sensitive*, so a variable with an identifier **`userName`** will be a completely separate variable to one with an identifier **`UserName`**. Be careful; it is easy to accidentally misspell an identifier.

It is a good programming practice to use meaningful identifiers for variables. While it may be easier to name a variable **`x`** or **`myVar`**, such identifiers carry no description of the value they contain. It is better to identify a variable with a description of its contents. Multiple words can be used with second and subsequent words in the identifier staring with an uppercase letter. For example if one wished to store the name of the user, possible identifiers include **`nameOfUser`** or **`userName`**. Programmers tend to develop their own style for such aspects of programming and use the same style consistently.

Exercise 4.1

Are the following identifiers legal or not? If not, why not?

1. **`example-number`**
2. **`exampleNumber1`**
3. **`1exampleNumber`**
4. **`example_number`**
5. **`example number`**

4.3. Declaring Variables with `var`

If you wish to declare a variable, the best place to do this is at the start of your program. If you do this, the declaration will easy to find later.

The following form can be used to declare variables.

`var variableIdentifer;`

or

`var variableIdentifier = value;`

In the examples above **`variableIdentifier`** would be replaced by the identifier of the variable and **`value`** would be replaced with an initial value. Initialising variables will be discussed further in section 5. Examples of variable declarations are shown in Code Example 4.1.

```
01  <html>
02      <head>
03          <script type="text/javascript">
04              var exampleString = "string content in quotes";
05              var exampleNumber = 123;
06              var exampleBoolean = true;
07              var exampleVariable;
08
09              alert(exampleString);
10              alert(exampleVariable);
11          </script>
12      </head>
13      <body>
14          Variables example
15      </body>
16  </html>
```

Code Example 4.1: Declaring variables

4.4. Undefined

If a variable is declared and not given an initial value, it will not be given a default value. If a variable is given no value and an attempt is made to get the value out of the variable, the value **`undefined`** will be given. In Code Example 4.1 a variable is declared without an initialisation at line 07. In line 10 the value of the variable is accessed to be output. As the variable has not yet been assigned a value, the value **`undefined`** will be output.

Exercise 4.2

Before you start writing any code, look at Code Example 4.1. On a piece of paper, write what you think the program will output.

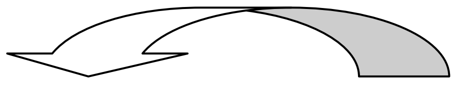
Using your template to replicate Code Example 4.1. Does it output the values as you expected it to?

Make the following changes.

1. Add another call to **`alert()`** to output **`exampleNumber`**.
2. Add another call to **`alert()`** to output **`exampleBoolean`**.
3. Create a new variable which will contain your name. Use an appropriately descriptive identifier which follows the rules shown in section 4.2. Assign the new variable a string (use quotes or double quotes) containing your name. Add another call to **`alert()`** to output the value of the variable.

5. Assigning Values

It is possible to give a variable a value when it is declared, and also to change its value later in the program. The form of an assignment is as follows.



```
variableIdentifier = value;
```

The value on the right is determined first. This could be from a number of sources. This value is then assigned to the variable identified on the left.

5.1. *Dynamic Typing*

Not only can the value of a variable change during the course of a program, but also the type of value may change. So a variable initialised with a string can later be assigned a number or a Boolean value. An example of this is show in Code Example 5.1.

```
<html>
    <head>
        <script type="text/javascript">
            var exampleVariable;

            exampleVariable = "string content in quotes";
            alert(exampleVariable);
            exampleVariable = 123;
            alert(exampleVariable);
            exampleVariable = true;
            alert(exampleVariable);
        </script>
    </head>
    <body>
        Dynamic typing example
    </body>
</html>
```

Code Example 5.1: The value and type of a variable can change

5.2. *`typeof`*

It is possible to determine if a variable currently contains a number, a string, a Boolean value, or no value at all (an **`undefined`** value). To do this, put the word **`typeof`** before the variable name (separated by a space). Code Example 5.2 is the same as the previous example, except instead of outputting the new values, the type of the variable is output at each stage.

```
<html>
    <head>
        <script type="text/javascript">
            var exampleVariable;

            alert(typeof exampleVariable);
            exampleVariable = "string content in quotes";
            alert(typeof exampleVariable);
            exampleVariable = 123;
            alert(typeof exampleVariable);
            exampleVariable = true;
            alert(typeof exampleVariable);
        </script>
    </head>
    <body>
        typeof() example
    </body>
</html>
```

Code Example 5.2: Discovering the type of a variable

Exercise 5.1

On a piece of paper write down the *identifier*, *value* and *type* of the following variables.

1. `var exampleInteger = 5;`
2. `var myName = "Michael";`
3. `var myLetter = 'M';`
4. `var emptyString = ""`
5. `var exampleTruthValue = true;`
6. `var exampleVariable;`

Exercise 5.2

On a piece of paper create variable declarations based on the following descriptions.

1. A number with identifier **`maxFound`** and value 0.
2. A string called **`name`** with your name as the value.
3. A Boolean variable called **`found`** with initial value **`false`**.

Exercise 5.3

Look at Code Example 5.2. On a piece of paper, write what you think the program will output.

Using your template to replicate Code Example 5.2. Does it output the values as you expected it to?

Make the following changes.

1. Change the double quotes on line 07 to single quotes. What happens?
2. Remove the contents of the string leaving only the quotes. What happens?
3. Change the number on line 09 to **`123.456`**. What happens?
4. Change the Boolean value on line 11 from **`true`** to **`false`**. What happens?

Not in implicit curriculum

5.3. Initialising Variables

When a variable is created its value is **`undefined`** until it is assigned a value. Using a variable that contains an undefined value can cause errors. Also, using value of one type (like a string) where another is expected (like a number) can have unexpected effects. It is therefore good practice to always initialise variables when they are created.

6. Operations

Operations are used to perform calculations and to combine values. There are four types of operators: arithmetic, relational, logical and string operators.

6.1. Arithmetic Operators

The form of operations you are probably most familiar with are arithmetic operations; operations on numbers. The following table describes the arithmetic operators available in JavaScript.

Operator	Name	Purpose
+	Plus	To add two numbers
-	Minus	To subtract one number from another
*	Multiply	To multiply two numbers
/	Divide	To divide two numbers
%	Mod	To find the remainder after integer division

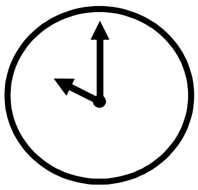
Table 6.1: Arithmetic Operators

Each of the operators above can be used with two values (operands), one on each side. We call these *binary* operators. The Minus operator can also be used to negate the sign of a single variable from positive to negative and vice-versa. In this case we refer to the Minus operator as a *unary* operator.

```
<html>
     <head>
          <script type="text/javascript">
               var number = 5;

               alert(1.2 + 3.4);
               alert(1.2 - 3.4);
               alert(1.2 * 3.4);
               alert(1.2 / 3.4);
               alert(12 % 5);
               alert(-number);
          </script>
     </head>
     <body>
          Operations example
     </body>
</html>
```

Code Example 6.1: Examples of arithmetic operators

The Mod (or Modulo) operator provides the remainder after a division. For example, say we had 12 apples and we wanted to divide this into groups of 5; how many would we have left-over? The 12 apples can give two full groups of 5 with 2 apples left-over.



Using the Mod operator we are able to bring large numbers to a position in a cycle. The Mod operator is sometimes called the clock operator. Consider a clock which shows the time at 10 o'clock. If asked what time will it be in 80 minutes, we do not say 10:80, we say it will be 11:20. We can use Mod to perform such a calculation as follows.

```
endMinute = (startMinute + minutesSpent)%60;
```

The Mod operator only works with whole numbers which we refer to as *integers*.

Exercise 6.1

On paper, write down what the following JavaScript statements will output.

1. `alert(1 + 2.5);`
2. `alert(1 – 2.5);`
3. `alert(2 * 3);`
4. `alert(1 / 2);`
5. `alert(5 % 3);`
6. `alert(9 % 3);`

Not in implicit curriculum

6.2. Division by Zero – `infinity`

When a number is divided by zero, the mathematical result is irrational. In JavaScript when a number is divided by zero, the special value **`infinity`** is given as the result. Care must be taken to avoid using **`infinity`** later in another operation as this may crash your program.

```
<html>
     <head>
          <script type="text/javascript">
               alert(123 / 0);
          </script>
     </head>
     <body>
          Division by zero example
     </body>
</html>
```

Code Example 6.2: Dividing by zero results in infinity

6.3. Postfix Operators

A common operation is increasing a variable's value by one (*increment*) or reducing its value by one (*decrement*). One way to achieve an increment as follows.

```
numberVariable = numberVariable +1;
```

A simpler short form is provided using the *unary* **`++`** operator.

```
numberVariable++;
```

A similar operator (**`--`**) is provided for decrementing. Both operators are demonstrated in Code Example 6.3.

```
<html>
     <head>
          <script type="text/javascript">
               var number = 0;

               alert(number);
               number++;
               alert(number);
               number--;
               alert(number);
          </script>
     </head>
     <body>
          Postfix operations example
     </body>
</html>
```

Code Example 6.3: Increment and decrement operators

Exercise 6.2

On paper, write down what the Code Example 6.3 will output.

6.4. Relational Operators (incl. Equality)

A relational operator takes two values (usually numbers) and compares them. The result of such an operation will be a Boolean value of **`true`** or **`false`**.

Operator	Name	How it works
>	Greater than	**`true`** if left value is greater than right
>=	Greater or equal	**`true`** if left value is equal or greater than right
<	Less than	**`true`** if left value is less than right
<=	Less or equal	**`true`** if left value is equal or less than right
==	Equal	**`true`** if left and right values are equal
!=	Not equal	**`true`** if left and right values are not equal

Table 6.2: Relational Operators

Examples of relational operators are shown in Code Example 6.4.

```
<html>
     <head>
          <script type="text/javascript">
               var x = 1;
               var y = 2;

               alert(x > y);
               alert(x >= y);
               alert(x < y);
               alert(x <= y);
               alert(x == y);
               alert(x != y);
          </script>
     </head>
     <body>
          Relational operations example
     </body>
</html>
```

Code Example 6.4: Relational operators

Exercise 6.3

On paper, write down what Code Example 6.4 will output.

6.5. Logical Operators

Logical operators combine two Boolean values. The resulting value will be **`true`** or **`false`**.

Operator	Name	How it works
&&	And	**`true`** if both values are **`true`**
||	Or	**`true`** if one or both values are **`true`**
!	Negate	**`true`** becomes **`false`**, **`false`** becomes **`true`**

Table 6.3:Logical Operators

Examples of relational operations are show in Code Example 6.5.

```
<html>
    <head>
        <script type="text/javascript">
            var x = 1;
            var y = 2;
            var testValue = false;

            alert(x==1 && y==2);
            alert(x==1 && y==1);
            alert(x==1 || y==1);
            alert(x==0 || y==0);

            testValue = x>0;
            alert(testValue);
            alert(!testValue);
        </script>
    </head>
    <body>
        Logical operations example
    </body>
</html>
```

Code Example 6.5: Logical operators

Exercise 6.4

On paper, write down what Code Example 6.5 will output.

6.6. String Operators

The **+** operator can be used to join two strings. It can also be used to append other values (numbers or Booleans) to the end of a string.

```
<html>
    <head>
        <script type="text/javascript">
            var message = "Hello";
            var number = 5;

            message = message + " World!";
            alert(message);
            alert("Mambo No. " + number);
        </script>
    </head>
    <body>
        String operations example
    </body>
</html>
```

Code Example 6.6: String operations

Exercise 6.5

On paper, write the following program (you only need the part that goes between the **`<script>...</script>`** tags.)

1. Declare a variable called **`message`** and initialise it with the string **`"Hello"`**.
2. Add a space to the end of the string value using a **+** operation.
3. In a call to **`alert()`** output the value of message and append your name as a string in quotes.

Not in implicit curriculum

7. Abutment

Most computers can only achieve one action at a time. With modern operating systems, computers can run multiple programs at the same time, but actually these programs must take turns accessing the computer's processor to complete their next action. Within programs, only a single statement can be processed at a time. Statements are processed in order from top to bottom. It is therefore important to recognise that to achieve a certain goal or goals, the steps required to achieve this must be discovered and the order in which they are put into action must be understood.

Take, for example, the simple goal of adding two numbers for a user. We can plan the steps involved as follows.

1. Declare two variables
2. Input two numbers
3. Perform calculation
4. Output result

To complete the required goal, the steps above cannot be ordered in any other way. In a program each of the steps will be performed in order and never out of sequence. Placing these steps adjacent to each other, one after the other, is referred to as *abutment*.

If this goal were part of some larger goal, the simple plan shown above would need to be *abutted* with other plans.

Exercise 7.1

On paper, *order* the following steps to create the message "Hello XXX" for a user who's name will replace XXX.

a. Append user's name to message variable.

b. Declare a message variable initialised to "Hello "

c. Get the user's name and assign to **`userName`**.

d. Output message.

e. Declare a variable with identifier **`userName`**.

8. Debugging

An important skill in programming is to find problems on code that:

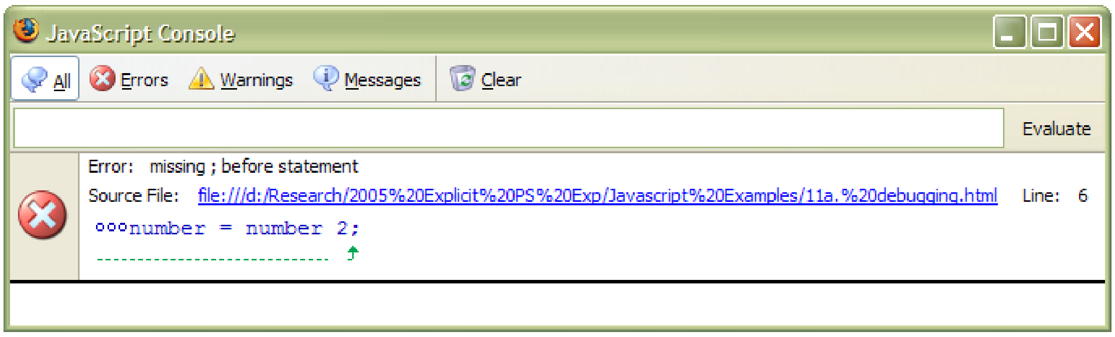
1. Stop the program from running at all, or
2. Don't stop the program running, but cause the program to perform incorrectly.

When writing code, you will be initially concerned with the first of these two.

```
01 <html>
02      <head>
03           <script type="text/javascript">
04                var number = 1;
05                number = number + 1;
06                number = number 2;
07                number = number * 3;
08           </script>
09      </head>
10      <body>
11           Debugging example
12      </body>
13 </html>
```

Code Example 8.1: A program containing a bug (line 06)

The code you write in JavaScript is interpreted by the Web Browser that is displaying the page. Different Web Browsers will deal with bugs in JavaScript code in different ways. The code in Code Example 8.1 contains an error on line 06; an operator is missing between the variable identifier **`number`** and the value **`2`**. After reaching this point in the program, the Web Browser would stop and the remaining program will not be executed. Using Mozilla Firefox (v1.0) the JavaScript Console reports errors. The JavaScript Console can be accessed from the Tools menu.



```
Error:   missing ; before statement
Source File:   file:///d:/Research/2005%20Explicit%20PS%20Exp/Javascript%20Examples/11a.%20debugging.html   Line: 6
ooonumber = number 2;
---------------------------- ↑
```

Figure 8.1: The JavaScript Console from Mozilla Firefox v1.0

Try to work on one error at a time. Error messages are the Web Browsers best guess at the program author's intention. Quite often they are incorrect and often confusing. What we can determine is:

- What line the error appeared on, and
- Roughly where in the line the error was located.

Knowing where the error has occurred is a good start. Return to the source code of the program and find the location. Sometimes the error is obvious and relying on what you have learned so far, it should be possible to correct the error. If the error does not jump out at you, and you find yourself staring indefinitely, ask for help.

When you have changed the source code, save the file, go to the JavaScript console and press the "Clear" button, then go to the Web Browser and click "Refresh". On returning to the JavaScript Console, hopefully the error will be gone.

```
01 <html>
02     <head>
03         <script type="text/javascript">
04             var exampleString = a string value;
05             var number 1;
06
07             alert(exampleString " and some more");
08         </script>
09     </head>
10     <body>
11         Debugging example for exercise
12     </body>
13 </html>
```

Code Example 8.2: A program containing several bugs

Exercise 8.1

The code in Code Example 8.2 contains three errors. Before you enter the code into your computer, attempt (on paper) to identify the line numbers containing errors, give a description of the error and say how you would fix it.

Using your template, enter the code exactly as shown. Open the file in your Web browser. Open the JavaScript Console (Tools → JavaScript Console) and attempt to locate and fix the errors one at a time. If you get stuck, ask for help.

A strategy for discovering faults in a program that is running but produces incorrect results is referred to as "print-lining". As a program runs, the variables in the program change. If the end result is incorrect, the point at which the program deviated from your intended route needs to be discovered. At points in your program it is possible to add calls to the **`alert()`** function to output the value of a variable (or variables) at that point. Usually it is best to start near the beginning, moving the line containing the call to **`alert()`** to later points in the program until the place where things start to go wrong is identified.

Not in implicit curriculum

```
<html>
    <head>
        <script type="text/javascript">
            var exampleVariable;

            alert(3 + exampleVariable);
        </script>
    </head>
</html>
```

Code Example 8.3: Code contains an error, but where

Exercise 8.2

Code Example 8.3 contains an error. Use your template to create this example.

- What does this program produce as output?
- Use a call to **`alert()`** to output the value of **`exampleVariable`** before line 06.
- What is the error?
- What can be done to remedy the error?

9. Functions that Return Values

The **`alert()`** function produces output to the user, but does not gather or create any information that can be used in our program. Many functions in JavaScript perform some action, then return a value that can be used in your program.

9.1. *`prompt()`*

The function **`prompt()`** is an example of a function which returns a value. This function, as the name implies, prompts the user to enter some information. That information is captured and can then be used in the program. The function **`prompt()`** returns a string value. To use this string we can either:

- Store the value in a variable;
- Use the value in an operation; or
- Pass the value on to another function as an argument (input).

```
01  <html>
02      <head>
03          <script type="text/javascript">
04              var name = "";
05
06              name = prompt("Enter your name");
07              alert("Hello " + name);
08          </script>
09      </head>
10      <body>
11          Input example
12      </body>
    </html>
```

Code Example 9.1: Example using the `prompt()` function

In Code Example 9.1 the **`prompt()`** function is used at line 05. The user is prompted to enter their name. A text box is given to do this as shown in Figure 2.1. When complete the user presses the ENTER key or clicks the OK button.

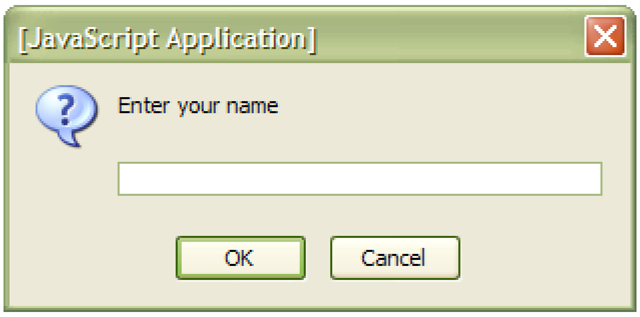


Figure 9.1: The effect of a call to `prompt()`

Still on line 05 of Code Example 9.1 the string returned from **`prompt()`** (the string entered by the user) is stored in the variable **`name`**. On the next line, the message "Hello" followed by the name the user entered is output.

Exercise 9.1

Using your template, create a program that will ask the user their age using the **`prompt()`** function. Store the age in a variable called **`age`**. Output the message "You are XXX years old" where XXX is the age entered by the user.

9.2. `parseInt()` and `parseFloat()`

The function **`prompt()`** returns a string. This is good where a string is needed, but a string cannot be used in arithmetic operations. Two functions are provided which can take a value (including a string) and turn it into a number. The function **`parseFloat()`** will return a number with a fraction expressed in decimal places. The function **`parseInt()`** will return a number without any decimal places. It should be noted that it does not round a number, it truncates it (just chops off the decimal places). So sending the value 1.9 to **`parseInt()`** would result in a value of 1.

```
<html>
     <head>
          <script type="text/javascript">
               var number = 0;

               number = prompt("Enter a number");
               alert(typeof number + " " + number);
               number = parseInt(prompt("Enter a number"));
               alert(typeof number + " " + number);
               number = parseFloat(prompt("Enter a number"));
               alert(typeof number + " " + number);
          </script>
     </head>
     <body>
          parseInt() and parseFloat() example
     </body>
</html>
```

Code Example 9.2: Using `parseInt()` and `parseFloat()` to get an numeric input

Exercise 9.2

Answer to the following on paper first, then confirm your answer by creating and testing the program. Assuming a user entered 4.56 for each input, what would the program in Code Example 9.2 output?

Exercise 9.3

Using the code you wrote in Exercise 9.1 take the result returned by the **`prompt()`** function and pass it to **`parseInt()`** to convert the user's age from a string to a number in integer form and store this in **`age`**. Increment the value of **`age`**. Output the message to say "Soon you will be XXX years old" where XXX is the incremented age.

10. Selection

In the programs we have seen so far, there has been only one path of execution through the program. Sometimes we may wish to execute some statements only when certain conditions are met. Sometimes we may wish to have two possible sets of statements of which only one will be executed according to certain conditions. Choosing whether or not to execute a body of statements is referred to as *selection*. A number of structures are provided for us to achieve selection.

10.1. The `if` Statement

The **`if`** statement can be used to execute a *body* of statements when certain conditions are met. We use a *test* to determine if these conditions have been met. The test will result in a **`true`** or **`false`** value. Relational (**`>`**, **`<`**, **`==`**...) and logical (**`&&`**, **`||`**, **`!`**) operators are often used in such a test to obtain a Boolean value.

```
if( TEST ) {
       BODY
}
```

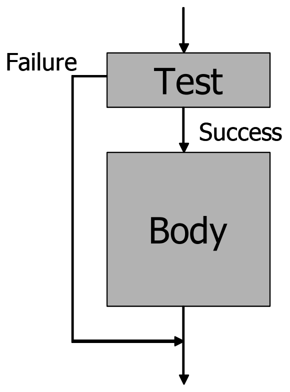


Figure 10.1: How `if` works

In the syntax description above we see an **`if`** statement starting with the word **`if`**. This is followed by the test which is always enclosed in parentheses **`()`**. The body contains statements that will be executed if the test results in a **`true`** value. The body is enclosed in curly braces **`{ }`**. If the test fails (results in a **`false`** value) the body will be skipped and the next statement after the body will be executed as shown diagrammatically in Figure 10.1.

In Code Example 10.1 we see an example of an **`if`** statement starting on line 05 and ending on line 07. The test compares the string the user entered with the string **`"hi"`**. If they are the same, a **`true`** value results and the body will be executed.

```
01  <html>
02      <head>
03          <script type="text/javascript">
04              var input = "";
05
06              input = prompt("Enter a string");
07              if(input == "hi") {
08                  alert("Well hello to you too.");
09              }
10          </script>
11      </head>
12      <body>
13          if example
        </body>
    </html>
```

Code Example 10.1: Example using `if`

Exercise 10.1

Using your template create a program that will prompt the user for a number. Convert the user's input to an integer using **`parseInt()`** and store in a variable. Using an **`if`** statement test the input; if the value is greater or equal to zero, output the message "Number was positive".

10.2. The `if-else` Statement

The if-else statement is similar to if but provides a second body which is executed when the test fails.

```
if( TEST ) {
      BODY
}
else {
      BODY
}
```

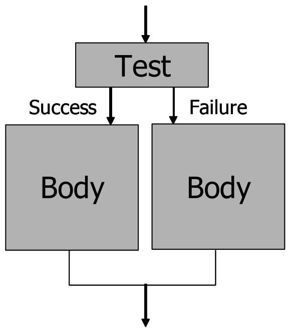


Figure 10.2: How `if-else` works

Only one body is executed as shown in Figure 10.2. After the appropriate body of statements is executed, there is a jump to the next statement after the **`if-else`** statement.

```
<html>
     <head>
         <script type="text/javascript">
              var input = "";

              input = prompt("Enter a string");
              if(input == "hi") {
                  alert("Well hello to you too.");
              }
              else {
                  alert("You entered: "+input);
              }
         </script>
     </head>
     <body>
         if-else example
     </body>
</html>
```

Code Example 10.2: Example using `if-else`

Exercise 10.2

Change the program you created for Exercise 10.1 so that if a user enters a negative number, the message "Number is negative" will be displayed.

10.3. Indenting and Formatting

In programming indenting is used to visually display structure in a program. Indenting is not required for the program to work and has no effect on how the program is executed. However it is good programming practice to use indenting so code is easily readable by humans.

The key to know where to use indenting usually lies in where curly braces **`{ }`** are placed. The content enclosed in braces should be indented one level further than the surrounding code.

```
<html>
     <head>
          <script type="text/javascript">
               var input = "";
               var number = 0;

               input = prompt("Enter a string");
               number = parseInt(prompt("Enter a number"));
               if(input == "hi") {
                    if(number > 0) {
                         alert("You are a positive person");
                    }
                    else {
                         alert("You entered: " + number);
                    }
               }
          </script>
     </head>
     <body>
          Indenting example
     </body>
</html>
```

Code Example 10.3: How indenting is used to show the structure of a program

In Code Example 10.3 an **`if`** statement is used and inside this is another **`if`** statement. The content of the outer (first) **`if`** is indented one level. Within the inner (second) **`if-else`** statement the bodies of the **`if`** and **`else`** are indented again.

Exercise 10.3

Answer the following on paper. What will happen when a user enters:

a. A string other than "hi"?

b. The string "hi" and a number greater than zero?

c. The string "hi" and a number less than zero?

d. The string "hi" and the number zero?

10.4. "Dangling `else`"

Code without indenting is harder to read. In Code Example 10.4 two **`if`** statements are shown without curly braces or indenting. This code achieves the same result as the previous example but is harder to read.

```
<html>
     <head>
          <script type="text/javascript">
               var input = "";
               var number = 0;

               input = prompt("Enter a string");
               number = parseInt(prompt("Enter a number"));
               if(input == "hi")
               if(number > 0)
               alert("You are a positive person");
               else
               alert("You entered: " + number);
          </script>
     </head>
     <body>
          Dangling else example
     </body>
</html>
```

Code Example 10.4: Without indenting code is harder to read

Exercise 10.4

Answer the following on paper.

a. What `if` does the `else` belong to?

b. What would happen `if` a statement was inserted after the second `if` and before the call to `alert()`?

Not in implicit curriculum

10.5. Guarding Division

One application of an `if` statement is to prevent code which could result in unpredictable behaviour or cause the program to crash while being executed. Previously we saw how dividing by zero can produce an unusable result. In some programming languages the effects can be even more severe. It is recommended that you always test the divisor (the second, right-hand operand) before a division operation takes place. If the divisor is zero, division should be avoided.

```
<html>
    <head>
        <script type="text/javascript">
            var number = 0;

            number = parseInt(prompt("Enter a number for division"));
            if(number != 0) {
                alert(100 / number);
            }
            else {
                alert("Dividing by zero causes problems");
            }
        </script>
    </head>
    <body>
        Guarding division example
    </body>
</html>
```

Code Example 10.5: The numerator of a division should always be tested before the division

Exercise 10.5

Using your template, create a program that will prompt the user to enter a pre-calculated *sum* of numbers and pre-calculated *count* of numbers. Calculate the *average* (the sum divided by the count). How should your program behave if the user enters zero for the count of numbers?

11. Repetition (Loops)

Often it is desirable to repeat the execution of statements. One way to achieve this is to have the same statements repeated in a program. This can be undesirable because:

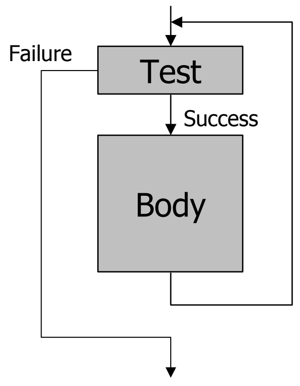
- If a change is needed, each repeated statement will need to be changed. This effort could result in errors.
- It is not possible to achieve a number of repetitions which is determined as the program is running (indefinite repetitions).

A number of structures are provided for achieving repetition.

11.1. `while` Loop

A **`while`** loop works like an **`if`** statement except the body of the loop is executed repeatedly while the test results in a **`true`** value (in other words, until it results in a **`false`** value).

```
while( TEST ) {
       BODY
}
```



When the loop starts, the test is performed; if a **`true`** value results, the body of the loop is executed, otherwise the body is skipped and the next statement after the loop is executed. When the end of the body is reached, the test is run again and this process continues.

```
<html>
     <head>
          <script type="text/javascript">
               var number = 5;

               while(number > 0) {
                    alert("Countdown: "+number);
                    number--;
               }
               alert("BLASTOFF!");
          </script>
     </head>
     <body>
          while loop example
     </body>
</html>
```

Code Example 11.1: Example of a `while` loop

Exercise 11.1

Using your template, create a program that determine if a number is divisible by 2 (**`number%2 == 0`**) and (using **`&&`**) divisible by 3 (**`number%3 == 0`**). If this is the case, output the value and add the number to a sum variable. Repeat this testing within a loop. Start testing at the number 1. Stop looping when the sum is greater or equal to 50. At the end, output the final sum value.

Not in implicit curriculum

11.2. Sentinel Controlled Loops

One application of a while loop is to repeat code until a certain value referred to as a *sentinel* is discovered. Code Example 11.2 shows a poor attempt at achieving a Sentinel Controlled Loop. The program is attempting to count inputs entered by a user before the sentinel is reached.

```
01 <html>
02      <head>
03          <script type="text/javascript">
04              var input;
05              var countOfInputs = 0;
06              var message = "Enter a number (999 to end): ";
07
08              while(input != 999) {
09                  input = parseInt(prompt(message));
10                  countOfInputs++;
11              }
12              alert("Counted "+countOfInputs+" numbers");
13          </script>
14      </head>
15      <body>
16          Bad loop example
17      </body>
18 </html>
```

Code Example 11.2: Repeating until a sentinel is found – this example will produce an incorrect result

This example is deficient because:

- The value of **`input`** is not initialised or set before it is used in the test at line 08. This could have unpredictable consequences.
- The goal of the code is to count numbers before the sentinel is encountered. In this example, when the sentinel is entered by the user it will be included in the count.

A correct solution is shown in Code Example 11.3. In this example, a value for **`input`** is gathered before the test is conducted. If the first number entered is the sentinel, the body of the loop is never executed, which is efficient. On successive inputs the value is always tested before the count is incremented. This will produce the correct answer in the most efficient fashion.

```
01 <html>
02      <head>
03          <script type="text/javascript">
04              var input = 0;
05              var countOfInputs = 0;
06              var message = "Enter a number (999 to end): ";
07
08              input = parseInt(prompt(message));
09              while(input != 999) {
10                  countOfInputs++;
11                  input = parseInt(prompt(message));
12              }
13              alert("Counted "+countOfInputs+" numbers");
14          </script>
15      </head>
16      <body>
17          Sentinel controlled loop example
18      </body>
19 </html>
```

Code Example 11.3: Repeating until a sentinel is found –allows for the sentinel in the first instance and correctly counts inputs

Exercise 11.2

First plan your solution to the following problem on paper, then implement the program using the template.

Sum floating point numbers collected from a user until they enter 999.

Consider:

a. What variables will be needed? How will they be initialised?

b. How will the loop work? How will the input be collected/converted?

c. When will the output be performed?

11.3. `for` Loop

A **`for`** loop is a loop construct with commonly used components conveniently 'built-in'. A **`for`** loop has a number of parts as show in the following syntax description. Note: parts ❶, ❷ and ❹ are separated by semicolons (`;`).

```
for( ❶ INITIALISATION ; ❷ TEST ; ❹ STEP ) {
    ❸ BODY
}
```

The parts of a **`for`** loop are executed in the following order.

❶ INITIALISATION	An opportunity to initialise a counter
❷ TEST	Determines if loop should continue (while **`true`**)
❸ BODY	Actions repeated each time the loop iterates
❹ STEP	An opportunity to move the counter towards the loop end

Note that:

- The initialisation (❶) is only performed once;
- If the test (❷) fails, parts ❸ and ❹ are skipped and the next statement after the loop body is executed; and
- The step (❹) is always followed by the test (❷).

The following code repeats the previous **`while`** loop example using a **`for`** loop.

```
<html>
     <head>
          <script type="text/javascript">
               var counter = 0;

               for(counter = 5; counter > 0; counter--) {
                    alert("Countdown: "+counter);
               }
               alert("BLASTOFF!");
          </script>
     </head>
     <body>
          for loop example
     </body>
</html>
```

Code Example 11.4: Example of a `for` loop

Exercise 11.3

Using your template, create a program that will use a *sum* and a *counter*. Set the counter to 1 and loop until it reaches 100 (`counter <= 100`). In each repetition add the value of counter to the sum. At the end output the value of the sum.

Not in implicit curriculum

11.4. Counter Controlled Loops

One application for a `for` loop is to repeat a body of statements a pre-set number of times. Rather than testing each time to see if the last repetition has been reached, a counter is used. For each repetition, the counter is incremented. When the counter reaches a pre-set value, repetition stops. This happens regardless of the content of the loop.

For example, if five values need to be collected, a Counter Controlled Loop could be used to achieve this. The number of repetitions and the termination of the repetition will be controlled by a counter and not by the values collected.

11.5. Finding the Maximum/Minimum

A common task is to find the maximum or minimum in a set of values. The following plan can be used to achieve a search for a maximum.

1. **Initialisation**
   A variable should be used to store the value of the maximum as the search progresses. Only a single variable is needed.. The variable should be set so when the first value is encountered it will become the new maximum. When searching for a maximum, the variable should be initialised to the minimum possible value. For example, if we were searching for a maximum positive integer (numbers zero or greater), the variable should be initialised to zero.

2. **Repetition**
   When searching a set of values of a know size, a Counter Controlled Loop is used. When the set size is unknown, a Sentinel Controlled Loop is used where the sentinel is a special value at the end of the set, or possibly the absence of any more values.

3. **Comparison**
   Each value of the set needs to be compared with the one stored in the variable. If value from the set is the new maximum it should be assigned to the variable.

Exercise 11.4

Using your template, create a program to find the maximum of 5 numbers entered by a user.

Not in implicit curriculum

11.6. Nesting and Merging

When presented with a problem, a series of goals will emerge which need to be achieved in order to solve the problem.

Abutment

Often the goals may need to be achieved in a certain order, in which case *abutment* is used as shown in Section 7. As an example, when searching for a minimum or maximum, a variable used to store the current max/min must be initialised *before* the search can start and the search must be completed before output can take place; this is abutment.

```
1. Initialise maximum variable
2. Search for maximum variable
3. Output maximum variable
```

Nesting

Sometimes sub-goals may need to be achieved to accomplish a greater goal. Sub-goals may be the body of a selection (**`if`** or **`if-else`**) statement or the body of a loop (**`while`** or **`for`**). This sub-goal is *nested* within a greater goal. In the example of finding a maximum or minimum, the comparison between each value in a set and the current max/min must happen within the repetition which gathers each value of the set. The comparison is nested in the repetition.

```
1. Initialise maximum variable
2. Counter Controlled Loop (Search in set of known size)
     a. Input
     b. Test for maximum
3. Output maximum variable
```

Merging

Often two goals can be achieved at the same time; we can *merge* the two goals. Say we were searching a set of a size unknown before the program began. We may want to count how many values are in the set, as well as determining the minimum or maximum. We could gather the same set of inputs twice, but a better solution would be to merge the counting of values with the comparison for a min/max.

```
1. Initialise maximum variable
2. Initialise counter
3. Input (prime loop)
4. Sentinel Controlled Loop (Search in set of unknown size)
     a. Test for maximum
     b. Increment counter
     c. Input
5. Output maximum variable
```

When two plans are merged, the order in which their commonly located parts are performed is usually not important. For instance, when we merge the maximum-search and count plans above, the initialisation of the variables (steps 1 and 2) could be re-ordered without affecting the outcome. Also the steps "Test for maximum" (a) and "Increment counter" (b) could be performed in the opposite order.

Exercise 11.5

Using your template, create a program to allow a user to enter positive integers until the user enters the sentinel 999. Determine the maximum value entered and the count of values (the value 999 will not be included).

12. Arrays

Often it is necessary to store several similar values, for instance:

- 10 numbers entered by a user, or
- The counts of occurrences of each alphabet letter in some text,

...

We could create variables to store each of these values, but a better solution is to store them together in *array*.

12.1. Declaring Arrays

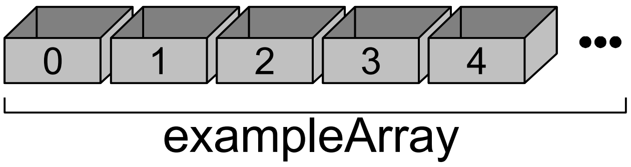
Arrays are declared in a slightly different way to a normal variable.

```
var arrayIdentifier = new Array();
```

...where **`arrayIdentifier`** would be replaced with the identifier for the array, for example...

```
var exampleArray = new Array();
```

...would create an array as follows...



12.2. Accessing Array Elements

Arrays are a collection of individual elements. Once we have created the array we can access each of the elements in an array by using an *index*. Indices are positive integers starting at `0`. The identifier of the array is followed by the index in square brackets `[]`, for example...

```
exampleArray[0]
```

...would allow us to access the first element of **`exampleArray`**. We can assign a value there as follows...

```
exampleArray[0] = 5;
```

...or read a value from that element...

```
alert(exampleArray[0]);
```

In JavaScript arrays are quite flexible.

- Arrays grow as you add to them.
- You can have more than one type of value in the same array.

12.3. Initialising Arrays

It is possible to initialise an array when it is declared. This is done by placing the initial values in the parentheses, with commas in-between each value. The following initialises an array of numbers.

`var monthArray = new Array(31,28,31,30,31,30,31,31,30,31,30,31);`

The following initialises an array of strings.

`var labelArray = new Array("Apples","Oranges","Banannas");`

You can later change the values in the array or add more.

Exercise 12.1

Using your template, declare an array initialised with the names of the days of the week stored as strings. Ask the user to enter a number between 1 and 7 (be sure to convert the input to an integer). Deduct 1 from the input value to get a value between 0 and 6. Use the decremented input as the index to the array and output the day name corresponding to the user's input.

12.4. Arrays for Values

One of the advantages of using arrays is we can perform actions on elements using a loop. Consider the goal of inputting then outputting three numbers. We could create three variables, input values into the three variables, then output the value of each. Alternately we can create an array, we can ask for input in a loop which is repeated three times, then use a loop to output the values of the array (see Code Example 12.1). Now consider what would be required if our goal were extended to 100 numbers. Using variables, this would become quite cumbersome and prone to error. With an array and loops, we merely have to increase the number of repetitions (changing the value of **`numbersToStore`** on line 05 of Code Example 12.1 would achieve this.)

```
01 <html>
02      <head>
03           <script type="text/javascript">
04                var inputArray = new Array();
05                var numbersToStore = 3;
06                var counter;
07                var message = "Please enter a number";
08
09                for(counter=0; counter<numbersToStore; counter++) {
10                     inputArray[counter] = parseInt(prompt(message));
11                }
12
13                for(counter=0; counter<numbersToStore; counter++) {
14                     alert("Input "+(counter+1)+": "+inputArray[counter]);
15                }
16           </script>
17      </head>
18      <body>
19           Array for values example
20      </body>
21 </html>
```

Code Example 12.1: Storing values in individual array elements

Exercise 12.2

Using your template, create a program will allow the user to enter 5 floating point numbers (use **`parseFloat()`**). Store each value in an array and add it to a *sum* at the same time. When input is complete, calculate the average by dividing the sum by 5. For each value in the array output the difference between the average and that value (**`average-numberArray[counter]`**). Some values may be negative and some positive.

12.5. Arrays for Categories

Another use for arrays is to count occurrences of items in a set. For instance, we could count "Apples","Oranges" and "Banannas" and store the count of each in an element of an array. The way we could do this is to refer to each item of the set using a number from 0 to 2, say 0 for Apples, 1 for Oranges and 2 for Banannas. We can then use this number as an index to an element in an array.

```
01  <html>
02      <head>
03          <script type="text/javascript">
04              var labelArray = new Array("Apples","Oranges","Banannas");
05              var numFruits = 3;
06              var message = "Please enter:\n";
07              var fruitCountArray = new Array();
08              var counter = 0;
09              var input = 0;
10
11              for(counter=0; counter<numFruits; counter++) {
12                  message = message+counter+" for "+labelArray[counter]+", ";
13              }
14              message = message + "9 to Quit";
15
16              for(counter=0; counter<numFruits; counter++) {
17                  fruitCountArray[counter] = 0;
18              }
19
20              input = parseInt(prompt(message));
21              while(input != 9) {
22                  fruitCountArray[input]++;
23                  input = parseInt(prompt(message));
24              }
25
26              for(counter=0; counter<numFruits; counter++) {
27                  alert(labelArray[counter]+": "+fruitCountArray[counter]);
28              }
29          </script>
30      </head>
31      <body>
32          Array for categories example
33      </body>
34  </html>
```

Code Example 12.2: Using an array to count occurrences of a set of elements

In Code Example 12.2 we count occurrences of fruit. The list of fruit is given in the array **`labelArray`** declared on line 04. We use an array here for the labels reasons:

1. We can declare the labels in one place and refer to them later, and
2. Declaring them in an array gives them order from 0 to 2.

On line 07 we declare the array which we will use to keep a count of the occurrences of each fruit.

On lines 11 to 14 we create a message which we can use later to prompt a user to enter the code number for a particular fruit. We could use the simpler prompt, "Enter the fruit code", but here we are giving the code numbers as well.

On lines 16 to 18, we initialise the count of fruit by setting each array element to zero.

Between lines 20 to 24 is where the action is. The loop will continue until the user enters a 9. On line 22 we see how we are using the code number specified by the user as the index to the array. If the user enters a zero they are referring to Apples so we go to the array element containing the count of Apples (**`fruitCountArray[0]`**) and increment (add one to) the value there. If the user entered 1 or 2, the appropriate **`fruitCountArray`** element would be incremented.

Finally on lines 26 to 28 we output the count of each fruit.

Exercise 12.3

Using your template, create a program will input five integers between 0 and 9. For each input increment the corresponding array element. At the end, output the occurrences of values which were input 1 or more times. For instance, if input was...

6, 2, 4, 2, 2

...output would be...

2: 3

4: 1

6: 1

Not in implicit curriculum

12.6. Counting Values in a Set

Code Example 12.2 contains the biggest JavaScript program we have seen so far. Let's look at this solution in terms of the plans used.

1. **Initialisation**
   Before we can start counting set members we need to initialise the count of each element to zero.
2. **Counter Controlled Loop**
   We know how many elements there are in **`fruitCountArray`**. We will therefore use a counter controlled loop (as opposed to a sentinel controlled loop) to initialise each array element.
3. **Input (twice)**
   We need to input fruit code numbers from a user. We do this once to prime the sentinel controlled loop and again at the end of the loop.
4. **Sentinel Controlled Loop**
   There is no limit to the number of times a user could enter a fruit code number. They could enter several code numbers, they could enter 1, or they could enter none by entering the sentinel (menu option 9) in the first instance. A sentinel controlled loop is therefore used to achieve this repetition.
5. **Set Counting**
   We are not entering the value entered by the user directly into our array. Instead we are using a code number that relates to an element in a set (the set of fruit). We are keeping a count of each fruit set member in an element of an array. For convenience we have made use of fruit code numbers (0 to 2) that are equivalent to the indices of the relevant array elements. We can therefore access the appropriate array element by using the value entered by the user. We can then increment the count in that array element using the statement (from line 22)...

```
fruitCountArray[input]++;
```

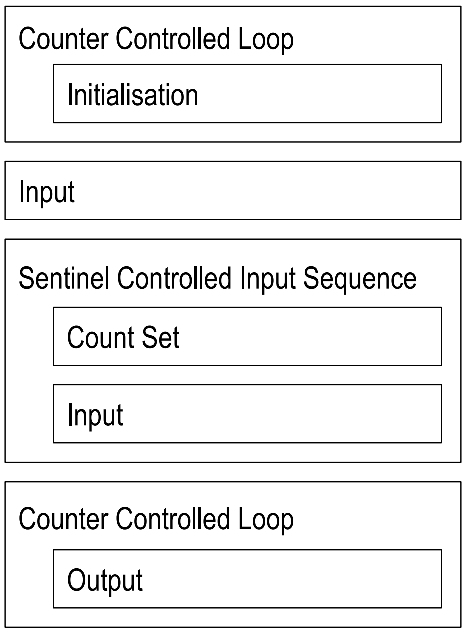
6. Output
We need to output the counts stored in the `fruitCountArray`.

7. Counter Controlled Loop
As we know how many elements there are in the `fruitCountArray` a counter controlled loop can be used to repeat the output.

Plan Integration
Abutment and nesting can be used to integrate the plans above in a way that will solve the problem.

- Initialisation (1) is nested in the first Counter Controlled Loop (2).
- Set Counting (3) and Input (5) are nested in a Sentinel Controlled Loop (4).
- Output (6) is nested in a Counter Controlled Loop (7).

These plans are abutted in the order (1 to 7) as they appear above.


Counter Controlled Loop
Initialisation
Input
Sentinel Controlled Input Sequence
Count Set
Input
Counter Controlled Loop
Output