Introduction to Systematic Programming
Unit 1 - General Introduction

1.1 About the course

This course is intended as a first course in computer programming using the programming language Ada. No previous knowledge of the subject is assumed. The lecture material is presented as a series of self-contained **units**, each dealing with one or more particular aspects of the course. This first unit - a general introduction to computer programming - is intended for private study, but each subsequent unit will contain a lecture text. The units will be concerned with introducing new aspects of the course material, and will be supplemented with further notes covering case studies in program design and construction.

Although these notes will cover the essential course material, you will probably find it helpful to also acquire a text-book for reference purposes and as a source of additional examples, exercises and case studies. There are a number of suitable books available:

Ada95, Problem Solving and Program Design,
M B Feldman & E B Koffman, Addison-Wesley (2nd ed), 1996
(Price £21.95 as at April 1996)

An Introduction to Software Design and Development with Ada,
D Rudd, West Publishing Co, 1995 (Price £20.95 as at September 1995)

Ada, Problem Solving and Program Design,
M B Feldman & E B Koffman, Addison-Wesley, 1992 (Price £21.95 as at September 1993)

Although the last book is now out of print, you may be able to purchase a second-hand copy from a previous first year student.

None of these books make any assumption about previous knowledge of computing, however, they cover Ada in a slightly different order and approach to that adopted in these lecture notes. The approaches adopted in these books are a perfectly adequate ones, but a little "slower" in the early stages than that adopted in this course. Thus if you can follow the approach covered in these lecture notes you will find that you should only need one of the above books for reference purposes. Alternatively, if you have difficulty following the course material, you could try reading one of these books in addition to the lecture notes.

The latter two books are based on a version of the Ada language which was designed in 1983 (often now referred to as Ada83). A new version of the language was designed in 1995 (known as Ada95), and the first of these books is based on the new version of the language. This course will be using a few of the new features of Ada95. However the majority of Ada83 and Ada95 are the same, and we will not (in this initial course at least) be using any of the more advanced new features of Ada95. Therefore it doesn't really matter very much whether you have a textbook based on Ada83 or Ada95.

If you have done a lot of programming before, are absolutely certain about your ability to design and write programs, and you just want a reference book on Ada95 that makes no attempt at teaching you how to program, the best one available is:

Programming in Ada95, J G P Barnes, Addison-Wesley, 1995
(Price £24.95 as at September 1995)

From time to time you will also receive sets of programming problems. These will form the vitally important practical component of the course and will provide a basis for discussion in the tutorial and laboratory classes. You will also be set a number of continuous assessment assignments. Generally you will have a few weeks to complete each assignment which will be marked and returned to you.

This year the department will be operating three versions of the Ada programming course:

CS110	Comprising 6 hours/week formal contact time during Term 1. Taken by students studying: BSc Computing Science BEng Electronic Engineering and Computer Science BEng Electrical and Electronic Engineering BEng/MEng Electronic Systems Engineering

CS410	Comprising 6 hours/week formal contact time during Term 1. Taken by students studying: MSc Information Technology
CS115	Comprising 3 hours/week formal contact time during Terms 1 and 2. Taken by students studying: BSc Combined Honours (Computer Science Option) BSc Information Technology for Business

All students taking the course will be doing their practical work on Sun Sparc workstations, some of which are provided by the department and some by Information Systems (the University computing service). The practical aspects of using such computer systems, eg. logging on, editing and printing files, compiling and running programs etc, will be dealt with in a separate course.

1.1.1 Organisation of the CS110 and CS410 versions of the course

The course lasts for ten weeks (Term 1) and each week there will be three lectures, a small group tutorial and two practical (laboratory) classes. In addition the computer laboratories are available for use during free periods (provided they are not timetabled for use by other classes) and in the evenings and at weekends. Two of the weekly lectures will deal with one or more of the basic aspects of computer programming in general and the language Ada in particular, and will be covered by these lecture notes. The third lecture will be devoted to a number of case studies developing programs to solve a variety of practical problems. There will be three continuous assessment assignments in computer programming which will count 30% towards the overall assessment of the course. The remaining 70% will be assessed via a three-hour written examination paper.

1.1.2 Organisation of the CS115 version of the course

The course lasts for twenty weeks (Terms 1 and 2) and each week there will be one lecture, a small group tutorial and a practical (laboratory) class. In addition the computer laboratories are available for use during free periods (provided they are not timetabled for use by other classes) and in the evenings and at weekends. The weekly lecture will deal with one or more of the basic aspects of computer programming in general and the language Ada in particular, and will be covered by these lecture notes. The tutorial will be used to discuss issues related to program design to support the lecture material and provide the foundation for the practical work, and will also be used to consider a number of case studies developing programs to solve a variety of practical problems. There will be four continuous assessment assignments in computer programming, and three class tests which will count 40% towards the overall assessment of the course. The remaining 60% will be assessed via a three-hour written examination paper.

1.2 Why use computers?

Computer systems manipulate data, which is usually a numerical and/or textual representation of some information such as bank balances, names and addresses, scientific observations, etc. They can store data, move data around and perform calculations on that data; so can human and mechanical systems; but computers have important advantages in three main areas:

a) **Speed** Computers can perform calculations much faster than is otherwise possible (for example, the arithmetic needed to guide a rocket in flight might take milliseconds if done by computer, but several days if done by hand - clearly not very helpful!).
b) **Volume** Computers make it feasible to handle very large collections of data, such as the many millions of items needed in national records, for example those held in connection with television, driving and vehicle licences.
c) **Accuracy** (Reliable) computer systems don't forget, make mistakes or get tired.

1.3 What sort of tasks can computers facilitate?

These generally fall into a few broad categories:

a) Filing (information storage) - eg. a library's catalogue
b) Sorting - eg. putting catalogue entries into order
c) Searching - eg. looking for a particular entry
d) Numerical calculation - eg. statistics of book borrowing
e) Report production (printing) - eg. presenting the results of (d)
f) Text manipulation (word processing) - eg. for preparing these notes
g) Process control - eg. controlling lifts, washing machines, car ignition systems etc.

1.4 How can computers be applied to these tasks?

To answer this, we can observe that most computer applications involve three basic elements:

i)	raw information (data)	- supplied as input to the computer
ii)	information processing, such as numerical calculation and/or sorting	- actions to be performed by the computer on the data
iii)	processed information (results)	- produced as output by the computer

To use a computer to solve a problem, we have to state how (ii) above is to be performed - we need to specify a method, or in other words give the computer a sequence of instructions (or a 'recipe') which define how the information processing is to be performed. Such 'recipes' are familiar; for example, supposing you wished to tell a stranger how to find the way somewhere, you might say:

"Turn left out of the door"

"Carry straight on until you come to the traffic lights"

"Turn left there"

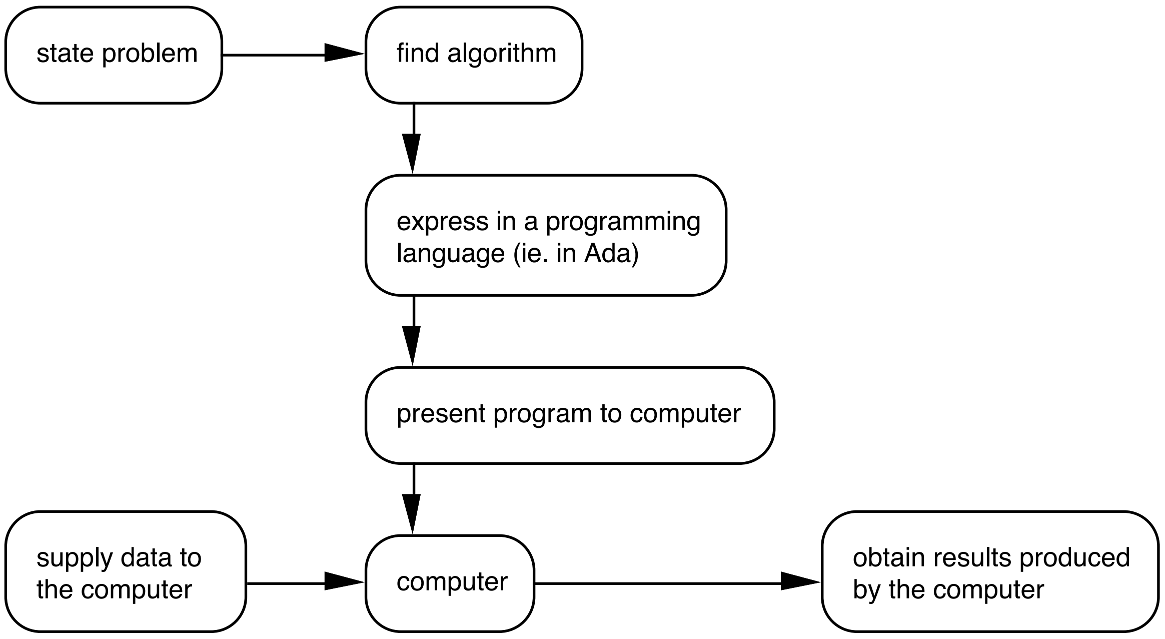
"Carry on until you come to the road on the right (and that will be where you are looking for)"

Other familiar forms of recipe are cooking recipes and knitting patterns. A recipe which, if followed exactly, is guaranteed to produce the required result after a finite number of steps is known as an **algorithm**. This is a very important concept in computing; in general, in order to successfully apply a computer to any given problem, we must first determine a suitable algorithm.

1.5 How is the computer made to obey an algorithm?

Natural languages, such as English, are capable of great complexity, but are inherently imprecise and often ambiguous; hence they are not suitable for specifying algorithms to computers; what is needed is a special (simpler), formalised language which will enable us to state our requirements with sufficient precision and detail. Such a language is known as a **computer programming language**. There are many such languages - the one used in this course is known as **Ada**, which was named after Augusta Ada Byron, Countess of Lovelace who was an assistant and patron of the computer pioneer Charles Babbage.

To specify an algorithm to a computer, we must express it as a **program** in our chosen programming language; we can now complete the process of applying a computer to solve a particular problem:


state problem
find algorithm
express in a programming language (ie. in Ada)
present program to computer
supply data to the computer
computer
obtain results produced by the computer

Ada is a modern general purpose language suitable for use in most computer applications and is a descendant of the language Pascal. It was developed in the late 1970's and early 1980's by the computer scientist Jean Ichbiah and Honeywell Bull in France, and later extended to include features of object-oriented design in 1995. It is superficially similar to Pascal, but remedies a number of its shortcomings. In particular, it has powerful features to support the development of large complex programs. In recent years it has also become popular as an initial language for teaching programming - replacing Pascal in many universities and colleges. Because Ada is a language intended for the professional development of programs it is quite large and complex - too large to cover entirely in an initial programming course. Thus the courses CS110, CS410 and CS115 will cover only part of the language. Later courses will cover further parts of the language.

1.6 A very simple example program

Consider the following problem:

Given two numbers, obtain their total.

A suitable algorithm would be:

```
Read the numbers into the computer (ie. input the two data items for the problem)
Add them together
Write out the result (ie. output the answer)
```

Let us now see how the algorithm could appear when expressed as an Ada program. (Do not worry about the details for now - they will all be explained later).

Ada Program

```
WITH CS_Int_IO;
USE  CS_Int_IO;

PROCEDURE Add IS

   First  : Integer;
   Second : Integer;
   Sum    : Integer;

BEGIN
   Get(Item => First);
   Get(Item => Second);
   Sum := First + Second;
   Put(Item => Sum);
END Add;
```

Brief explanation

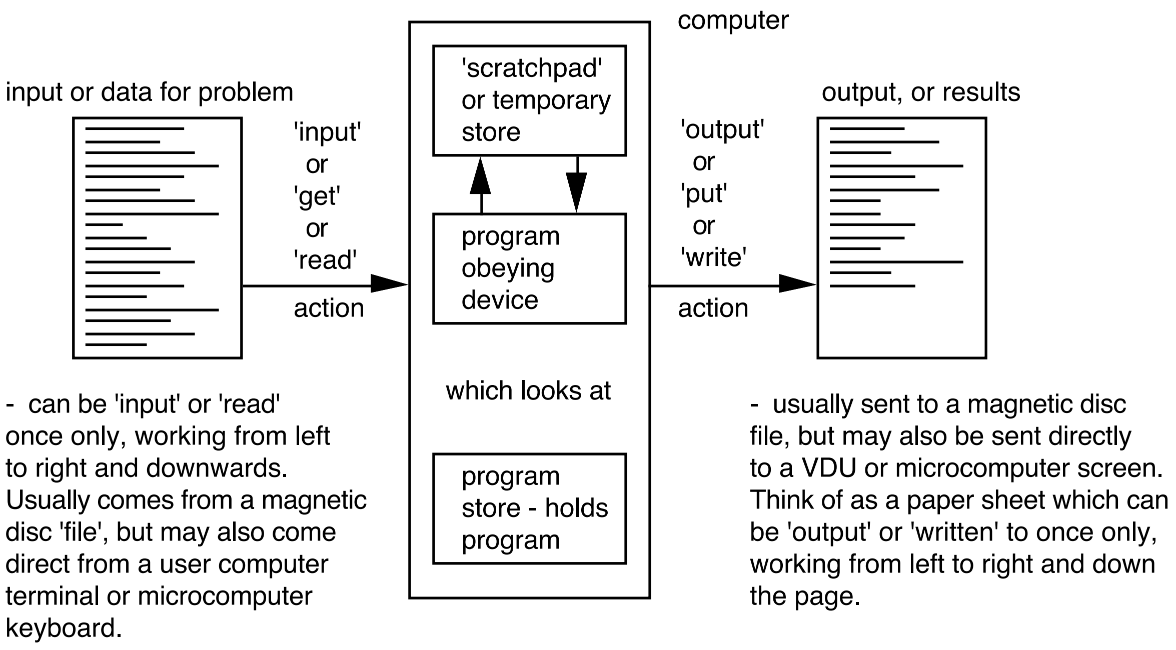
- the program uses the procedures `Get` and `Put` from the library `CS_Int_IO`
- to specify a title for the program to be performed
- reserve space to store the input data and the result in three variables named `First`, `Second` and `Sum` respectively
- to denote the start of execution of the program
- input (or 'read') the first data value
- input (or 'read') the second data value
- add these two values and store the result in `Sum`
- output (or 'write') the result
- to denote the end of the program

For clarity the Ada program is shown enclosed in a 'box'. A further convention will also be adopted in these notes, namely that program components will always (whether part of a program or when referred to within an explanation) appear in a different type font. The notes will always follow these conventions so that it is clear which parts of the notes are 'program' text and which parts are explanation.

Note also that certain program items are shown all in capital letters; these items are called **keywords** (or **reserved words**) and have special significance in program construction. An initially obscure feature of Ada programs is the use of `WITH` and `USE` steps, examples of which appear as the first two lines of the above program. This program makes use of two library procedures: `Get` and `Put` to input data and output results respectively. These procedures are found in the library `CS_Int_IO`, and the `WITH` and `USE` steps make this library available to the program.

1.7 A model for a computer

Obviously we shall wish to work on algorithms for the types of problem to which we can readily apply a computer, and therefore need to get our algorithms in a suitable form for use by a computer. As a means to this end, it is very helpful to have a clear idea of what sort of actions a computer can do. By looking at an idealised model of a computer, such as the following, we can concentrate our attention on those aspects which are relevant to us.


computer
input or data for problem
'scratchpad' or temporary store
output, or results
'input' or 'get' or 'read' action
'output' or 'put' or 'write' action
program obeying device
which looks at
program store - holds program
- can be 'input' or 'read' once only, working from left to right and downwards. Usually comes from a magnetic disc 'file', but may also come direct from a user computer terminal or microcomputer keyboard.
- usually sent to a magnetic disc file, but may also be sent directly to a VDU or microcomputer screen. Think of as a paper sheet which can be 'output' or 'written' to once only, working from left to right and down the page.

The idealised model above shows the typical arrangement when the data is presented from one source, ie. a magnetic disc file, and the results output to another destination, ie. another magnetic disc file. This arrangement is common in many situations; as is the arrangement where the data is input from, and the results output to the same device; usually either a computer terminal or VDU, or microcomputer keyboard and screen. In this latter case the program is said to be **interactive**. The principle of 'read' and 'write' actions is the same in both situations. The only difference is that in the interactive case the input data and output results are interlaced with one another on the same 'paper sheet'.

From this model, and our example program given above, we see that a computer program will be able to contain instructions to:

a) Input (or read) the next data item (commonly some numeric value).

b) Output (or write) the next item of the results (usually some number or text).

c) Organise the use of the scratchpad store - eg. put this number in one location, or move this number to another location.

d) Perform arithmetic on values (which may be taken from the store) eg. add, subtract, multiply, divide.

In addition, we will see later that computer programs are also be able to contain instructions to:

e) Perform tests on values (which may be taken from the store) eg. test if two numbers are equal, test if a certain number is greater than zero, etc.

f) Control the execution of other instruction steps, in ways which the program obeying device can follow, ie. to perform one of an alternative series of instructions, or to repeat a series of instructions over and over again.

We can see that for a computer program to be usefully complete, it *must* contain one or more output steps (else how will we know what results the program has computed?). Most complete computer programs also (with the exception of certain unusual special cases) contain one or more input steps (else how is the computer to get hold of the data which the program operates on?).

1.8 Programming

The entire task of proceeding from a problem specification right up to getting results out of the computer is usually just referred to as **programming**, and the person who does this job is called a (**computer**) **programmer**. We can already see that there are three separate major activities involved in programming:

i) Given a problem, formulating an algorithm to solve that problem.

ii) Expressing the resulting algorithm in our chosen programming language (Ada).

iii) Presenting the program and data to the computer and obtaining the results produced.

Thus effective programming involves learning ***several*** skills.

Logically, problem analysis and algorithm formulation must precede the expression of an algorithm in a programming language; hence the next unit of this course looks at how algorithms can be expressed. It is obviously sensible to express algorithms in a form which will make stage (ii) as straightforward as possible, and we shall aim to do this. In fact this will involve no more than sticking to certain fundamental types of algorithm step which can be used directly in Ada programs.

Having obtained an algorithm and expressed it as a program, we must present our Ada program (often called **source code**) to the computer. This involves typing our program into the computer and then **compiling** it with the Ada compiler. The compiler translates the Ada program into another language, **machine code** (or **object code**) which is more suitable for direct execution by the computer. The (compiled) program then needs **linking** with the required libraries to form a complete executable program. We can then provide suitable data for the **running** (or **executing**) of the (compiled and linked) program to obtain some results. The actual mechanics of presenting a program to a computer, and getting output from it, will be explained in the Operating Systems Usage course (course code CS111) and in the practical classes.

1.8.1 Testing and checking

However, the programmer's task does not end at this point, for in reality errors may well have been made in stage (i) or stage (ii). Errors made in stage (ii) may cause the program to fail to compile or link correctly (**compile-time** and **link-time errors**). Errors made in either stage (i) or (ii) (or the provision of incorrectly formed data) may cause errors to occur when the program is run (**run-time errors**) which cause the program to terminate prematurely. Even if complete results are obtained, they must be checked to ensure that the program is **logically correct**, ie. that it is a correct solution to the given problem and is producing the 'expected' results. An illustration of a **logical error** would be if we had mis-typed the step:

```
        Sum := First + Second;      as:        Sum := First - Second;
```

in the example program given above. The program would still compile and link correctly, but on execution it would produce results which were incorrect.

1.8.2 Finding and correcting errors

You will certainly discover that the detection and correction of errors (commonly referred to as **debugging**) usually forms a significant component of programming work; and at least while learning to program, this must be expected and accepted. Programming is a practical skill acquired through substantial practical experience, so it's important not to be put off or demoralised by a few errors (or **bugs**); stick at it! In the tutorial and practical classes you will also be introduced to various debugging techniques and aids, such as an Ada source-level debugger, which are designed to help you locate the origin of errors in your programs.

However, it cannot be too strongly emphasised, even at this stage, that a methodical and systematic approach to both algorithm formulation and program expression is vital both to reduce the likelihood of error and the difficulty of detecting and correcting any errors which do 'slip through' stages (i) and (ii) above. Indeed modern programming methodology now makes it realistic, by the use of a very thorough and systematic technique, to aim towards producing programs which are regularly correct first time. This should be your ultimate objective.

1.9 A word to the wise

Many of you may have used computers before, possibly using the language Basic, and will be familiar with the 'Basic approach' to programming - ie. one just sits down at a terminal or a personal computer, and given a problem, attempts to obtain a program to solve it by a process of typing in statements as the inspiration arises, combined with much trial and error. You are warned that whilst this approach can work for ***very simple problems***, it is ***entirely inadequate*** for obtaining correct solutions to the problems which arise in University-level Computer Science (or for that matter in any 'real-world' commercial computing). Hence it is essential that you resist this 'ad hoc' approach and apply yourself to learning and using the systematic method of programming that will be presented to you.

1.9.1 And, a word to the very wise

Some of you may have head of, or read about, other programming languages. You may even have looked through the computing press and national newspapers and seen advertisements of jobs for programmers who know a particular programming language. You may be wondering why we are going to teach you the language Ada, and not one of the more widely advertised languages such as C, C++, COBOL, Coral, Fortran or Pascal.

a) Firstly, it is the function of a University course to provide an education, and not simply to train graduates for a particular employment. This education has to provide a body of knowledge and expertise which will outlive the usefulness of the average programming language. In three or four years time, when you graduate and start looking for employment, the commonly used commercial programming languages may be very different to the ones which appear in job advertisements now.

b) Secondly, there is a sense in which the actual programming language we teach does not matter a great deal - it is the principles of programming which are important, and these skills are largely transferable to different programming languages. For example, whereas it may take you the best part of a year to learn to program reasonably competently in your first programming language (Ada), it will then typically only take you a few weeks to learn another programming language. In addition, some of the commonly used commercial languages are very specialised (ie. they are used only for certain types of problem). Others languages are more general purpose programming languages (ie. they are suitable for tackling a wide variety of problems). Clearly general purpose programming languages are more likely to be better for illustrating programming principles than specialised ones, and Ada is one of these better ones. In fact it is widely agreed that in the current state of the art, one of the best of the available imperative programming languages for teaching computing is Ada. This is because it not only provides a suitable vehicle for the initial teaching of programming, but also because it supports ideas and principles which are important in large-scale programming projects (and most 'real-world' programming projects are large-scale). Thus you will find that Ada will be a suitable language for the majority of the programming which you will need to do in the subsequent years of your degree.

c) Finally, it is the function of a University education to empower change (hopefully for the better) amongst successive generations of graduates. It is interesting to note that many of the major changes which have taken place in the use of different programming languages, and the efforts which have taken place to improve the 'quality' of computer programs have largely been due to developments which have taken place within the Universities.

So we teach a systematic approach to algorithm design and program construction using Ada for very good reasons.

1.10 Finally a note about learning to program

These lecture notes introduce ideas about learning to program in a sequential manner which continually builds upon previously acquired knowledge and skills throughout the whole period of the course. Thus in Unit 2 a complete understanding of the contents of Unit 1 will be assumed; in Unit 3 a complete understanding of the contents of Unit 2 will be assumed; and so on. This means that you cannot afford to have any large gaps in your understanding of the contents of a unit. If your knowledge of the content of a unit is incorrect or incomplete then the next unit will be more difficult to comprehend , the unit after that even more difficult to follow, and so on. Thus if you find it difficult to understand the contents of a unit, you must try to resolve this lack of understanding as quickly as possible. There are many sources of assistance: textbooks, the academic staff who take your lectures and tutorials, the research students who take your practical classes, and fellow students - make use of them.