Software Engineering II

or how to write good programs

A C Norman, Easter Term 1997

Part IA (50%)

1 Introduction

The course *Software Engineering I* presented several major issues. One was that errors in software can have serious consequences, up to and including loss of life and the collapse of businesses. Another was that the construction of large computer-related products will involve teams of programmers working to build and support software over many years, and this raises problems not apparent to an individual programmer working on a private project. A third was that formal methods can be used when defining an planning a project to build a stable base for it to grow on, and this can be a great help. The emphasis was on programming in the large, which is what the term “Software Engineering” is usually taken to have at its core. Overall the emphasis was on recognition of the full life-time costs associated with software and the management strategies that might keep these costs under control.

This course complements that perspective and looks at the job of one person or a rather small group, working on what may well be a software component or a medium sized program rather than on a mega-scale product. The intent of the course is to collect together and emphasise some of the issues that lie behind the skill of programming. Good programmers will probably use many of the techniques mentioned here without being especially aware of what they are doing, but for everybody (even the most experienced) it can be very useful to bring these ideas out into the open.

One of the major views I would like to bring to the art (or craft, or science, or engineering discipline, depending on how one like to look at it) of programming is an awareness of the value of an idea described by George Orwell in his book “1984”. This is *doublethink*, the ability to believe two contradictory ideas without becoming confused. Of course one of the original pillars of doublethink was the useful precept *Ignorance is Strength*, but collections of views specifically about the process of constructing programs. These notes will not be about the rest of the association of computers with surveillance, Newspeak or other efficiency-enhancing ideas. The potentially conflicting views about programming that I want to push relate to the prospect of a project succeeding. Try to keep in your minds both the idea *Programming is incredibly difficult and this program will never work correctly: I am going to have to spend utterly hopeless ages trying to coax it into passing even the most minimal test cases* and its optimistic other face, which claims cheerfully *In a couple of days I can crack the core of this problem, and then it will only take me another few to finish off all the details. These days even young children can all write programs*. The concise way to express this particular piece of doublethink (and please remember that you really have to believe both part, for without the first you will make a total botch of everything, while without the second you will be too paralysed ever to start actual coding), is

Writing programs is easy.

A rather closely related bit of doublethink alludes both to the joy of achievement when a program appears to partially work and the simultaneous bitter way in which work with computers persistently fail. Computers show up our imperfections and frailties, which range through unwillingness to read specifications via inability to think straight all the way to incompetence in mere mechanical typing skills. The short-hand for the pleasure that comes from discovering one of your own mistakes, and having spent many frustrating hours tracking down something that is essentially trivial comes out as

Writing programs is fun.

A further thing that will be curious about this course is that it does not present universal and provable absolute truths. It is much more in the style of collected good advice. Some of this is based on direct experience, other parts has emerged as an often-unstated collective view of those who work with computers. There are rather fewer books covering this subject than I might have expected. There is a very long reading list posted regularly on `comp.software-eng`, but most of it clearly assumes that by the time things get down to actually writing programs the reader will know from experience what to do. Despite the fact that it is more concerned with team-work rather than individual programming I want to direct you towards the Mythical Man Month[4], if only for the cover illustration of the La Brea Tar Pits[1] with the vision that programmers can become trapped just as the Ice Age mammoths and so on were. For looking at programming tasks that are fairly algorithmic in style the book by Dijkstra[7] is both challenging and a landmark. There are places where people have collected together some of the especially neat and clever **little** programs they have come across, and many of these indeed contain ideas or lessons that may be re-cyclable. Such examples have come to be referred to as "pearls"[2][3]. Once one has worked out what program needs to be written ideas (now so much in the mainstream that this book is perhaps now out of date) can be found in one of the big presentations by some of the early proponents of structured programming[11]. Stepping back and looking at the programming process with a view to estimating programmer productivity and the reliability of the end product, Halstead[8] introduced some interesting sorts of software metrics, which twenty years on are still not completely free of controversy. All these still leave me feeling that there is a gap between books that describe the specific detail of how to use one particular programming language,

[1]You may not be aware that the tar pits are in the middle of a thoroughly built-up part of Los Angeles, and when visiting them you can try to imagine some of the local school-children venturing too far and getting bogged down, thus luring their families, out for a week-end picnic, to a sticky doom.

and those concerned with large scale software engineering and project management. To date this gap has generally been filled by an apprentice system where trainee programmers are invited to work on progressively larger exercises and their output is graded and commented on by their superiors. Much like it is done here! With this course I can at least provide some background thoughts that might help the apprentices start on their progression a little more smoothly.

When I started planning this course it was not quite obvious how much there was going to be for me to say that avoided recitations of the blindingly obvious and that was also reasonably generally applicable. As I started on the notes it became clear that there are actually a lot of points to be covered. To keep within the number of lectures that I have and to restrict these notes to a manageable bulk I am therefore restricting myself (mostly) to listing points for consideration and giving as concrete and explicit recommendations as I can: I am not including worked examples or lots of anecdotes that illustrate how badly things can go wrong when people set about tasks in wrong-minded ways. But perhaps I can suggest that as you read this document you imagine it expanded into a book-length presentation with all that additional supporting material and with a few exercises at the end of each section. You can also think about all the points that I raise in the context of the various programming exercises that you are set or other practical work that you are involved with.

2 Different sorts of programming tasks

When considering software-related projects it is useful to start with a classification of possible sorts of program. There are three justifications for this:

1. Different sorts of computer systems are not all equally easy to build. For instance industrial experience has shown repeatedly that the construction of (eg) an operating system is very much harder than building a (again eg) a compiler even when the initial specifications and the amount of code to be written seem very similar. Thinking about the category into which your particular problem falls can help you to plan time-scales and predict possible areas of difficulty;

2. The way you go about a project can depend critically on some very high level aspects of the task. A fuller list of possibilities is given below, but two extreme cases might be (a) a software component for inclusion in a safety-critical part of an aero-space application, where development budget and timescale are subservient to an over-riding requirement for reliability, and (b) a small program being written for fun as a first experiment with a new programming language, where the program will be run just once and

nothing of any real important hands on the results. It would be silly to carry forward either of the above two tasks using a mind-set tuned to the other: knowing where one is on the spectrum between can help make the selection of methodology and tools more rational;

3. I will make a point in these notes that program development is not something to be done in an isolated cell. It involves discussing ideas and progress with others and becoming aware of relevant prior art. Thinking about the broad area in which your work lies can help you focus on the resources worth investigating. Often some of these will not be at all specific to the immediate description of what you are suppose to achieve but will concerned with very general areas such as rapid prototyping, formal validation, real-time responsiveness, user interfaces or whatever.

I will give my list of possible project attributes. These are in general not mutually exclusive, and in all real cases one will find that these are not yes–no choices but more like items to be scored from 1 to 10. I would like to think of them as forming an initial survey that you should conduct before starting any detailed work on your program just to set it in context. When you find one or two of these scoring 9 or 10 out of 10 for relevance you know you have identified something important that ought to influence how you approach the work. If you find a project scores highly on *lots* of these items then you might consider trying to wriggle out of having to take responsibility for it, since there is a significant chance that it will be a disaster! The list here identifies potential issues, but does not discuss ways of resolving them: in many cases the project features identified here will just tell you which of the later sections in these notes are liable to be the more important ones for your particular piece of work. The wording in each of my descriptions will be intended to give some flavour of how *severe* instances of the issue being discussed can cause trouble, so keep cheerful because usually you will not be plunging in at the really deep end of the pool.

Ill-defined One of the most common and hardest situations to face up to is when a computer project is not clearly specified. I am going to take this case to include ones where there appears to be a detailed and precise specification document but on close inspection the requirements as written down boil down to "I don't know much about computer systems but I know what I like, so write me a program that I will like, please." Clearly the first thing to do in such a case is to schedule a sub-project that has the task of obtaining a clear and concise description of what is really required, and sometimes this will of itself be a substantial challenge;

Shifting sands If either project requirements or resources can change while software development is under way then this fact needs to be allowed for. Probable only a tiny minority of real projects will be immune from this sort of distraction, since even for apparently well-specified tasks it is quite usual that experience with the working version of the program will lead to "wouldn't it be nice if . . . " ideas emerging even in the face of carefully discussed and thought out early design decisions that the options now requested would not be supportable.

Safety-critical It is presumably obvious that safety-critical applications need exceptional thought and effort put into their validation. But this need for reliability is far from an all-or-nothing one, in that the reputation of a software house (or indeed the grades obtained by a student) may depend on ensuring that systems run correctly at least most of the time, and that their failure modes appear to the user to be reasonable and soft. At the other extreme it is worth noting that in cases where robustness of code and reliability of results are not important at all (as can sometimes be the case, despite this seeming unreasonable) that fact can be exploited to give the whole project a much lighter structure and sometimes to make everything very much easier to achieve. A useful question to ask is "Does this program have to work correctly in *all* circumstances, or does it just need to work in *most* common cases, or indeed might it be sufficient to make it work in just *one* carefully chosen case?"

Large It is well established that as the size of a programming task increases the amount of work that goes into it grows much more rapidly than the number of lines of code (or whatever other simple measurement you like) does. At various levels of task size it becomes necessary to introduce project teams, extra layers of formal management and in general to move away from any pretence that any single individual will have a full understanding of the whole effort. If your task and the associated time-scales call for a team of 40 programmers and you try it on your own maybe you will have difficulty finishing it off! Estimating the exact size that a program will end up or just how long it will take to write is of course very hard, but identifying whether it can be done by one smart programmer in a month or if it is a big team project for five years is a much less difficult call to make.

Urgent When are you expected to get this piece of work done? How form is the deadline? If time constraints (including the way that this project will compete with other things you are supposed to do) represents a real challenge it is best to notice that early on. Note that if, while doing final testing on your code, you find that it has a bug in it there may be no guarantee that you can isolate or fix this to any pre-specified time-scale. This is because (at least for most people!) there is hardly any limit to the subtlety of bugs and the amount of time and re-work needed to remove them. If the delivery date for code is going to be rigidly enforced (as is the case with CST final year projects!) this fact may be important: even if there is plenty of the project as a whole a rigid deadline can make it suddenly become urgent at the last minute;

Unrealistic It is quite easy to write a project specification that sounds good, but is not grounded in the real world. A program that modelled the stock markets and thereby allowed you to predict how to manage your portfolio, or one to predict winning numbers in the national lottery, or one to play world-class chess... Now of course there are programs that play chess pretty well, and lots of people have made money out of the other two projects (in one case the statistics that one might apply is *much* easier than the other!), but the desirability of the finished program can astonishingly often blind one to the difficulties that would arise in trying to achieve it. In some cases a project might be achievable in principle but is beyond either today's technology or this week's budget, while in other cases the idea being considered might not even realistic given unlimited budget and time-scales. There are of course places where near-unreasonable big ideas can have a very valuable part to play: in a research laboratory a vision of one of these (currently) unrealistic goals can provide direction to the various smaller and better contained projects that each take tiny steps towards the ideal. At present my favourite example of something like this is the idea of *nanotechnology* with armies of molecular-scale robots working together to build their own heirs and successors. The standard example of a real project that many (most?) realistic observers judged to be utterly infeasible was the "Star Wars" Strategic Defence Initiative, but note that at that sort of level the political impact of even starting a project may be at least as important as delivery of a working product!

Multi-platform It is a luxury if a program only has to work on a single fixed computer system. Especially as projects become larger there is substantial extra effort required to keep them able to work smoothly on many different systems. This problem can show up with simple issues such as word-lengths, byte-arrangements in memory and compiler eccentricities, but it gets much worse as one looks at windowed user interfaces, multi-media functions, network drivers and support for special extra plug-in hardware;

Long life-time The easiest sort of program gets written one afternoon and is thrown away the next day. It does not carry any serious long-term support concerns with it. However other programs (sometimes still initially written in little more than an afternoon) end up becoming part of your life and get themselves worked and re-worked every few years. In my case the program I have that has the longest history was written in around 1972 in Fortran, based on me having seen one of that year's Diploma dissertations and having (partly unreasonably) convinced myself I could do better. The code was developed on Titan, the then University main machine. I took it to the USA with me when I spent a year there and tried to remove the last few bugs and make it look nicer. When the University moved up to an IBM mainframe I ran it on that, and at a much later stage I translated it (semi automatically) into BBC basic and ran it (very slowly) on a BBC micro. By last year I had the code in C with significant parts of the middle of it totally re-written, but with still those last few bugs to find ways of working around. If I had been able to predict when I started how long this would be of interest to me for maybe I would have tried harder to get it right first time! Note the radical changes in available hardware and sensible programming language over the lifetime of this program. The topical allusion to the problems of not thinking far enough ahead is to the worry about the year 2000 and computer-stored dates;

User interface For programs like modern word processors there is a real chance that almost all of the effort and a very large proportion of the code will go into supporting the fancy user interface, and trying to make it as helpful and intuitive as possible. Actually storing and editing the text could well be fairly straight forward. When the smoothness of a user interface is a serious priority for a project then the challenge of defining exactly what must happen is almost certainly severe too, and in significant projects will involve putting test users in special usability laboratories where their eye-movement can be tracked by cameras and their key-strokes can be timed. The fact that an interface provides lots of windows and pull-down menus does not automatically make it easy to use;

Diverse users Many commercial applications need to satisfy multiple users with diverse needs as part of a single coherent system. This can extend to new computer systems that need to interwork seamlessly with multiple existing operational procedures, including existing computer packages. Some users may be nervous of the new technology, while others may find excessive explanation an offensive waste of their time. The larger the number of interfaces needed and the wider the range of expectations the harder it will be to make a complete system deliver total satisfaction;

Speed critical Increasingly these days it makes sense to buy a faster computer if some task seems to take a little longer than is comfortable. However there remain some areas where absolute performance is a serious issue and where getting the very best out of fixed hardware resources can give a competitive edge. The case most in my mind at present is that of (high security) encryption, where the calculations needed are fairly demanding but where there is real interest in keeping some control over the extra hardware costs that user are expected to incur. If speed requirements lead to need for significant assembly code programming (or almost equivalently to the design of task-specific silicon) then the resource requirements of a project can jump dramatically. If in the other hand speed is of no importance at all for some task it may become possible to use a higher level programming system, simpler data structures and algorithms and generally save a huge amount of aggravation;

Real time Real-time responsiveness is characteristic of many control applications. It demands that certain external events be given a response within a pre-specified interval. At first this sounds like a variant on tasks that are just speed-critical, but the fine granularity at which performance is specified tends to influence the entire shape of software projects and rule out some otherwise sensible approaches. Some multi-media applications and video games will score highly in this category, as will engine management software for cars and flight control software for airports;

Memory critical A programming task can be made much harder if you are squeezed for memory. The very idea of being memory-limited can feel silly when we all know that it is easy to go out and but another 16 Mbytes for (currently) of the order of £50. But the computer in your cell-phone will have an amount of memory selected on the basis of a painful compromise between cost (measured in pennies), the power drain that the circuitry puts on the battery (and hence the expected battery life) and the set of features that can be supported. And the software developers are probably give the memory budget as a fixed quantity and invited to support as long a list of features as is at all possible within it;

Add-on A completely fresh piece of software is entitled to define its own file formats and conventions and can generally be designed and build without too much hindrance. But next year the extension to that original package is needed, or the new program is one that has to work gracefully with data from other people's programs. When building an add-on it is painfully often the case that the existing software base is not very well documented, and that the attempted new use of it reveals previously unknown bugs or limitations in the core system. Thus the effort that will need to be put into the second package may be much greater than would have been predicted based on experience from the first;

Embedded If the computer you are going to program is one in an electric egg-timer (or maybe a toy racing car, or an X-ray scanner) then testing may involve be a quite different experience from that you become used to when debugging ordinary applications that run on standard work-stations. In particular it may become necessary to become something of an expert in the hardware and electronics and also in the application area of the system within which your code will be placed;

Tool-weak environment This is a follow-on from the "embedded" heading, in that it is perhaps easiest the envisage an electric pop-up toaster where anything that slowed down or enlarged the code being run would perturb system timing enough to burn the toast, and where the target hardware is not automatically equipped with printers and flashing lights that can be used to help sense what is going on inside its CPU. For some such cases it is possible to buy or build real-time emulators or to wire in extra probes into a debuggable version of the hardware. There are other cases where either technology or budget mean that program development has to be done with a slow turn-around on testing and with only very limited ability to discover what happened when a bug surfaced. It is incredibly easy to simulate such a tool-weak environment for yourself by just avoiding the effort associated with becoming familiar with automated testing tools, debuggers and the like;

Novel One of the best and safest ways of knowing that a task is feasible is to observe that somebody else did it before, and their version was at least more or less satisfactory. The next best way is to observe that the new task is really rather similar to one that was carried out successfully in the past. This clearly leads to the obvious observation that if something is being attempted and there are no precedents to rely on then it becomes much harder to predict how well things will work out, and the chances of nasty surprises increases substantially.

There are two sort of program not listed above which deserve special mention. The first is the implementation of a known algorithm. This will usually end up as a package or a subroutine rather than a complete free-standing program, and there are plenty of algorithms that are complicated enough that programming them is a severe challenge. However the availability of a clear target and well specified direction will often make such programming tasks relatively tractable. It is however important to distinguish between programming up a complete and known algorithm (easyish) from developing and then implementing a new one, and uncomfortably often things that we informally describe as algorithms are in fact just strategies, and lots of difficult and inventive fine detail has to be filled into make them realistic.

The second special sort of program is the little throw-away one, and the recognition that such programs can be lashed together really fast and without any fuss is important, since it can allow one to automate other parts of the program development task through strategic use of such bits of jiffy code.

3 Analysis and description of the objective

Sometimes a programming task starts with you being presented with a complete, precise and coherent explanation of exactly what has to be achieved. When this is couched in language so precise that there is not possible doubt about what is required you might like to ask why you are being asked to do anything, since almost all you need to do is to transcribe the specification into the particular syntax of the (specified) programming language. Several of the Part IA tickable problems come fairly close to this pattern, and there the reason you are asked to do them is exactly so you get practical experience with the syntax of the given language and the practical details of presenting programs to the computer. But that hardly counts as serious programming!

Assuming that we are not in one of these artificial cases, it is necessary to think about what one should expect to find in a specification and what does not belong there. It is useful to discuss the sorts of language used in specifications, and to consider who will end up taking prime responsibility for everything being correct.

A place to start is with the observation that a specification should describe what is wanted, rather than how the desired effect is to be achieved. This ideal can be followed up rather rapidly by the observation that it is often amazingly difficult to know what is really wanted, and usually quite a lot of important aspects of the full list of requirements will be left implicit or as items where you have to apply your own judgement. This is where it is useful to think back to the previous section and decide what style of project was liable to be intended and where the main pressure points are liable to be.

3.1 Important Questions

I have already given a check-list that should help work out what general class of problem you are facing. The next stage is to try to identify and concentrate on areas of uncertainty in your understanding of what has to be done. Furthermore initial effort ought to go into understanding aspects of the problem that are liable to shape the whole project: there is no point in agonising over cosmetic details until the big picture has become clear. Probably the best way of sorting this out is to imagine that some magic wand has been waved and it has conjured up a body of code and documentation that (if the fairy really was a good one!) probably does everything you need. However as a hard-headed and slightly cynical person you need to check it first. Deciding what you are going to look for to see if the submitted work actually satisfied the project's needs can let you make explicit a lot of the previously slightly woolly expectations you might have. This viewpoint moves you from the initial statement "The program must achieve X" a little closer to "I must end up convinced that the program achieves X and here is the basis for how that conviction might be carried". Other things that might (or indeed might not) reveal themselves at this stage are:

1. Is user documentation needed, and if so how detailed is it expected to be? Is there any guess for how bulky the user manual will be?

2. How formal should documentation of the inner workings of the code be?

3. Was the implementation language to be used pre-specified, and if not what aspects of the problem or environment are relevant to the choice?

4. Is the initial specification a water-tight one or does the implementer have to make detailed design decisions along the way?

With regard to choice of programming language note that evidence from studies that have watched the behaviour of real programmers suggests that to a good first approximation it is possible to deliver the same number of lines of working documented code per week almost whatever language it is written in. A very striking consequence of this is that languages that are naturally concise and which provide biuilt-in support for more of the high-level things you want to do can give major boosts to productivity.

The object of all this thought is to lead to a proper specification of the task. Depending on circumstances this may take one of a number of possible forms:

3.2 Informal specifications

Documents written in English, however pedantically phrased and however voluminous, must be viewed as informal specifications. Those who have a lot of spare time might try reading the original description of the language C[9] where Appendix A is called a reference manual and might be expected to form a useful basis for fresh implementations of the language. Superficially it looks pretty good, but it is only when you examine the careful (though still "informal" in the current context) description in the official ANSI standard[12] that it becomes clear just how much is left unsaid in the first document. Note that ANSI C is not the same language as that defined by Kernighan and Ritchie, and so the two documents just mentioned can not be compared quite directly, and also be aware that spotting and making clear places where specifications written in English are not precise is a great skill, and one that some people enjoy exercising more than others do! The description in section 19 is another rather more manageable example of an informal textual specification. When you get to it you might like to check to see what it tells you what to do about tabs and back-spaces, which are clearly characters that have an effect on horizontal layout. What? It fails to mention them? Oh dear!

3.3 Formal descriptions

One response to the fact that almost all informal specifications are riddled with holes (not all of which will be important: for instance it might be taken as understood by all that messages that are printed so that they look like sentences should be properly spelt and punctuated) has been to look for ways of using formal description languages. The ZED language (developed at Oxford[2], and sometimes written as just Z) is one such and has at times been taught in Software Engineering courses here. The group concerned with the development of the language ML were keen to use formal mathematically-styled descriptive methods to define exactly what ML ought to do in all possible circumstances. Later on in the CST there are whole lecture courses on Specification and Verification and so I am not going to give any examples here, but will content myself by observing that a good grounding in discrete mathematics is an absolute pre-requisite for anybody thinking of working this way.

3.4 Executable specifications

One group of formal specification enthusiasts went off and developed ever more powerful mathematical notations to help them describe tasks. Another group observed that sometimes a careful description of what must be achieved looks a bit like a program in a super-compact super-high-level programming language. It may not look like a realistic program, in that it may omit lots of explanation about how objectives should be achieved and especially how they should be achieved reasonably efficiently. This leads to the idea of an *executable specification*, through building an implementation of the specification language. This will permitted to run amazingly slowly, and its users will be encouraged to go all out for clarity and correctness. To give a small idea of what this might entail, consider the job of specifying a procedure to sort some data. The initial informal specification might be that the output should be a re-ordering of the input such that the values in the output be in non-descending order. An executable specification

[2]`http://www.comlab.ox.ac.uk/oucl/prg.html`

might consist of three components. The first would create a list of all the different permutations of the input. The second would be a procedure to inspect a list and check to see if its elements were in non-descending order. The final part would compose these to generate all permutations than scan through them on at a time and return the first non-descending one found. This would not be a good practical sorting algorithm, but could provide a basis for very transparent demonstrations that the process shown did achieve the desired goal! It should be remembered that an executable specification needs to be treated as such, and not as a model for how the eventual implementation will work. A danger with the technique is that it is quite easy for accidental or unimportant consequences of how the specification is written to end up as part of the project requirements.

4 Ethical Considerations

Quite early on when considering a programming project you need to take explicit stack of any moral or ethical issues that it raises. Earlier in the year you have had more complete coverage of the problems of behaving professionally, so here I will just give a quick check-list of some of the things that might give cause for concern:

1. Infringement of other people's intellectual property rights, be they patents, copyright or trade secrets. Some companies will at least try to prevent others from creating new programs that look too much like the original. When licensed software is being used the implications of the license agreement may become relevant;

2. Responsibility to your employer or institution. It may be that certain sorts of work are contrary to local policy. For instance a company might not be willing to permit its staff to politically motivated virtual reality simulations using company resources, and this University has views about the commercial use of academic systems;

3. A computing professional has a responsibility to give honest advice to their "customer" when asked about the reasonableness or feasibility of a project, and to avoid taking on work that they know they are not qualified to do;

4. It can be proper to take a considered stance against the development of systems that are liable to have a seriously negative impact on society as a whole. I have known some people who consider this reason to avoid any involvement with military or defence-funded computing, while others will object to technology that seems especially liable to make tracking, surveillance or eavesdropping easier. Those of you with lurid imaginations can no

doubt envisage plenty more applications of computers that might be seen as so undesirable that one should if necessary quit a job rather than work on them.

5 How much of the work has been done already?

The points that I have covered so far probably do not feel as if they really help you get started when faced with a hard-looking programming task, although I believe that working hard to make sure you really understand the specification you are faced with is in fact always a very valuable process. From now onwards I move closer to the concrete and visible parts of the programming task. The first question to ask here is "Do I actually have to do this or has it been done before?"

There are three notable cases where something has been done before but it is still necessary to do it again. Student exercises are one of these, and undue reliance on the efforts of your predecessors is gently discouraged. Sometimes a problem has been solved before, but a solution needs to be re-created without reference to the original version because the original is encumbered with awkward commercial[3] restrictions or is not locally available. The final cause for re-implementation is if the previous version of the program concerned was a failure and so much of a mess that any attempt to rely on it would start the new project off in wrong-minded directions.

Apart from these cases the best way to write any program at all is to adopt, adapt and improve as much existing technology as you can! This can range from making the very second program that you ever write a variation on that initial "Hello World" example you were given through to exploiting existing large software libraries. The material that can be reclaimed may be as minor as a bunch of initial comments saying who you (the author) are and including space to describe what the program does. It might just be some stylised "import" statements needed at the head of almost any program you write. If you need a tree structure in today's program do you have one written last week which gives you the data type definition and some of the basic operations on trees? Have you been provided with a collection of nice neat sample programs (or do you have a book or CD ROM with some) that can help? Many programming languages are packaged with a fairly extensive collection of chunks of sample code.

Most programming languages come with standardised libraries that (almost always) mean there is no need for you to write your own sorting procedure or code to convert floating point values into or out of textual form. In many important areas

[3]Remember that if the restriction is in the form of a patent then no amount of re-implementation frees you from obligations to the patent-owner, and in other cases you may need to be able to give a very clear demonstration that your new version really has been created completely independently.

there will be separate libraries that contain much much more extensive collections of facilities. For instance numerical libraries (eg the one from NAG) are where you should look for code to solve sets of simultaneous equations or to maximise a messy function of several variables. When you need to implement a windowed interface with pull-down menus and all that sort of stuff again look to existing library code to cope with much of the low-level detail for you. Similarly for data compression, arbitrary precision integer arithmetic, image manipulation...

Observing that there is a lot of existing support around does not make the problem of program construction go away: knowing what existing code is available is not always easy, and understanding both how to use it and what constraints must be accepted if it is used can be quite a challenge. For instance with the NAG (numerical) library it may take beginners quite a while before they discover that E04ACF (say, and not one of the other half-dozen closely related routines) is the name of the function they need to call and before they understand exactly what arguments to pass to it.

As well as existing pre-written code (either in source or library form) that can help along with a new project there are also packages that write significant bodies of code for you, basing what they do one on either a compact descriptive input file or interaction with the user through some clever interface. The well-established examples of this are the tools `yacc` and `lex` that provide a convenient and reliable way of creating parsers. Current users of Microsoft's Visual C++ system will be aware of the so-called "Wizards" that it provides that help create code to implement the user interface you want, and there are other commercial program generators in this and a variety of business application areas. To use one of these you first have to know of its availability, and then learn how to drive it: both of these may involve an investment of time, but with luck that will be re-paid with generous interest even on your first real use. In some cases the correct use of a program generation tool is to accept its output uncritically, while on other occasions the proper view is to collect what it creates, study it and eventually adjust the generated code until you can take direct responsibility for subsequent support. Before deciding which to do you need to come to a judgement about the stability and reliability of the program generator and how often you will need to adjust your code by feeding fresh input in to the very start.

Another way in which existing code can be exploited is when new code is written so that it converts whatever input it accepts into the input format for some existing package, one that solves a sufficiently related problem that this makes some sense. For instance it is quite common to make an early implementation of a new programming language work by translating the new language into some existing one and then feeding the translated version into an existing compiler. For early versions of ML the existing language was Lisp, while for Modula 3 some compilers work by converting the Modula 3 source into C. Doing this may result in a complete compiler that is slower and bulkier than might otherwise be the case, but it can greatly reduce the effort in building it.

6 What skills and knowledge are available?

A balance needs to be drawn between working through a new programming project using only the techniques and tools that you already know and pushing it forward using valuable but unfamiliar new methods. Doing something new may slow you down substantially, but an unwillingness to accept that toll may lead to a very pedestrian style of code development using only a limited range of idioms. There is a real possibility that short-term expediency can be in conflict with longer term productivity. Examples where this may feel a strain include use of formal methods, new programming languages and program generation tools. The main point to be got across here is that almost everything to do with computers changes every five years or so, and so all in the field need to invest some of their effort in continual personal re-education so that their work does not look too much as if it has been chipped out using stone axes. The good news is that although detailed technology changes the skills associated with working through a significant project should grow with experience, and the amount of existing code that an old hand will have to pillage may be quite large, and so there is a reasonable prospect for a long term future for those with skills in software design and construction. Remember that all the books on Software Engineering tell us that the competence of the people working on a project can make more difference to its success than any other single factor.

It is useful to have a conscious policy of collecting knowledge about what can be done and where to find the fine grubby details. For example the standard textbook[6] contains detailed recipes for solving all sorts of basic tasks. Only rarely will any one of these be the whole of a program you need to write, but quite often a larger task will be able to exploit one or more of them. These and many of the other topics covered in the CST are there because there is at least a chance that they may occasionally be useful! It is much more important to know what can be done than how to do it, because the *how* can always be looked up when you need it.

7 Design of methods to achieve a goal

Perhaps the biggest single decision to be made when starting the detailed design of a program is where to begin. The concrete suggestions that I include here are to some extent caricatures; in reality few real projects will follow any of them totally but all should be recognisable as strategies. The crucial issue is that it will not be possible to design or write the whole of a program at once so it is necessary to split the work into phases or chunks.

7.1 Top-Down Design

In Top Down Design work on a problem starts by writing a “program” that is just one line long. Its text is:

```
begin SolveMyProblem(); end;
```

where of course the detailed punctuation may be selected to match the programming language being used. At this stage it is quite reasonable to be very informal about syntax. A next step will be to find some way of partitioning the whole task into components. Just how these components will be brought into existence is at present left in the air, however if we split things up in too unreasonable a way we will run into trouble later on. For many simple programs the second stage could look rather like:

```
(* My name, today’s date, purpose of program *)
import Standard-libraries;
begin
  (* declare variables here *)
  data := ReadInData();
  results := Calculate(data);
  DisplayResults(results)
end;
```

The ideal is that the whole development of the program should take place in baby-sized steps like this. At almost every stage there will be a whole collection of worrying-looking procedures that remain undefined and not yet thought about, such as `Calculate` above. It is critical not to worry too much about these, because each time a refinement is made although the number of these unresolved problems may multiply the expected difficulty of each will reduce. Well it had better, since all the ones that you introduce should be necessary steps towards the solution of your whole original task, and it makes sense to expect parts to be simpler than the whole.

After a rather few steps in the top-down development process one should expect to have a fully syntactically correct main program that will not need any alterations later as the low level details of the procedures that it calls get sorted out. And each of the components that remain to be implemented should have a clearly understood purpose (for choice that should be written down) and each such component should be clearly separated from all the others. That is not to say that the component procedures might not call each other or rely on what they each can do, but the internal details of any one component should not matter to any other. This last point helps focus attention on interfaces. In my tiny example above the serious interfaces are represented by the variables `data` and `results` which pass information from one part of the design to the next. Working out exactly what must be captured in these interfaces would be generally need to be done fairly early on. After enough stages of elaboration the bits left over from top-down design are liable to end up small enough that you just code them up without need to worry: anything that is trivial you code up, anything that still looks murky you just apply one more expansion step to. With luck eventually the process ends.

There are two significant worries about top-down design. These are “How do I know how to split the main task up?” and “But I can’t test me code until everything is finished!”. Both of these are proper concerns.

Splitting a big problem up involves finding a strategy for solving it. Even though this can be quite hard, it is almost always easier to invent a high-level idea for how to solve a problem than it is to work through all the details, and this is what top-down programming is all about. In many cases sketching on a piece of paper what you would do if you had to solve the problem by hand (rather than by computer) can help. Quite often the partition of a problem you make may end up leading your design into some uncomfortable dead end. In that case you need to look back and see which steps in your problem refinement represented places where you had real choice and which ones were pretty much inevitable. It is then necessary to go back to one of the stages where a choice was possible and to re-think things in the light of your new understanding. To make this process sensible you should refuse to give up fleshing out one particular version of a top-down design until you are in a position to give a really clear explanation of why the route you have taken represents failure, because without this understanding you will not know how far back you need to go in the re-planning. As an example of what might go wrong, the code I sketched earlier here would end up being wrongly structured if user interaction was needed, and that interaction might be based on evaluation of partial results. To make that sort of interface possible it might be necessary to re-work the design as (say)

```
(* My name, today’s date, purpose of program *)
import Standard-libraries;
begin
  (* declare variables here *)
  (* set empty data and results *)
  while not finished do
  begin
```

```
          extra := ReadInMoreData();
          if EndOfUserInput(extra) then finished := true;
          else
          begin
             data := Combine(data, extra);
             results := UpdateResults(results, data);
             DisplaySomething(results);
          end;
       end;
       DisplayFinalResults(results);
     end;
```

which is clearly getting messier! And furthermore my earlier and shorter version looked generally valid for lots of tasks, while this one would need careful extra review depending on the exact for of user interaction required.

There is a huge amount to be said in favour of being able to test a program as it is built. Anybody who waits right to the end will have a dreadful mix of errors at all possible levels of abstraction to try to disentagle. At first sight it seems that top-down design precludes any early testing. This pessimism is not well founded. The main way out is to write *stubs* of code that fill in for all the parts of your program that have not yet been written. A stub is a short and simple piece of code that takes the place of something that will later on be much more messy. It does whatever is necessary to simulate some minimal behaviour that will make it possible to test the code around it. Sometimes a stub will just print a warning message and stop when it gets called! On other occasions one might make a stub print out its parameters and wait for human intervention: it then reads something back in, packages it up a bit and returns it as a result. The human assistant actually did all the clever work.

There are two other attitudes to take to top-down design. One of these is to limit it to **design** rather than implementation. Just use it to define a skeletal shape for your code, and then make the coding and testing a separate activity. Obviously this only makes sense when you have enough confidence that you can be sure that the chunks left to be coded will in fact work out well. The final view is to think of top-down design as an ideal to be retrofitted to any project once it is complete. Even if the real work on a project went in fits and starts with lots of false trails and confusion, there is a very real chance that it can be rationalised afterwards and explained top-down. If that is done then it is almost certain that a clear framework has been built for anybody who needs to make future changes to the program.

7.2 Bottom-Up Implementation

Perhaps you are uncertain about exactly what your program is going to do or how it will solve its central problems. Perhaps you want to make sure that every line of code you ever write is documented, tested and validated to death before you move on from it and certainly before you start relying on it. Well these concerns lead you towards a bottom-up development strategy. The idea here is to identify a collection of smallish bits of functionality that will (almost) certainly be needed as part of your complete program, and to start by implementing these. This avoids having to thing about the hard stuff for a while. For instance a compiler-writer might start by writing code to read in lines of program and discard comments, or to build up a list of all the variable names seen. Somebody starting to write a word processor might begin with pattern-matching code ready for use in search-and-replace operations. In almost all large projects there are going to be quite a few fundamental units of code that are obviously going to be useful regardless of the high level structure you end up with.

The worry with bottom-up construction is that it does not correspond to having any overall vision of the final result. That makes it all to easy to end up with a collection of ill-co-ordinated components that do not quite fit together and that do not really combine to solve the original problem. At the very least I would suggest a serious bout of top-down design effort be done before any bottom-up work to try to put an overall framework into place. There is also a clear prospect that some of the units created during bottom-up work may end up not being necessary after all so the time spend on them was wasted.

An alternative way of thinking about bottom-up programming can soften the impact of these worries. It starts by viewing a programming language not just as a collection of fragments of syntax, but as a range of ways of structuring data and of performing operations upon it. The fact that some of these operations happen to be hard-wired into the language (as integer arithmetic usually is) while others exist as collections of subroutines (floating point arithmetic on 3000-digit numbers would normally be done that way) is of secondary importance. Considered this way each time you define a new data type or write a fresh procedure you have extended and customised your programming language by giving it support for something new. Bottom-up programming can then be seen as gradually building layer upon layer of extra support into your language until it is rich in the operations most important in your problem area. Eventually one then hopes that the task that at first had seemed daunting becomes just half a dozen lines in the extended language. If some of the procedures built along the way do not happen to be used this time, they may well come in handy the next time you have to write a program in the same application area, so the work they consumed has not really been wasted after all. The language Lisp is notable for having sustained a culture based on this idea of language extension.

7.3 Data Centred Programming

Both top-down and bottom-up programming tend to focus on what your program looks like and the way in which it is structured into procedures. An alternative is to concentrate not on the actions performed by the code but on the way in which data is represented and the history of transformations that any bit of data will be subject to. These days this idea is often considered almost synonymous with an Object Oriented approach where the overall design of the class structure for a program is the most fundamental feature that it will have. Earlier (and pre-dating the widespread use of Object Oriented languages) convincing arguments for design based on the entities that a program must manipulate or model come from Jackson Structured Programming and Design[5]. More recently SSADM[1] has probably become one of the more widespread design and specification methodologies for commercial projects.

7.4 Iterative Refinement

My final strategy for organising the design of a complete program does not even expect to complete the job in one session. It starts by asking how the initial problem can be restricted or simplified to make it easier to address. And perhaps it will spot how the most globally critical design decisions for the whole program could me made in two or three different ways, with it hard to tell in advance which would work out best in the end. The idea is then to start with code for a scruffy mock-up of a watered down version of the desired program using just one of these sets of design decisions. The time and effort needed to write a program grows much faster then linearly with the size of the program: the natural (but less obvious) consequence of this is that writing a small program can be **much** quicker and easier than completing the full version. It may in some cases make sense even to write several competing first sketches of the code. When the first sketch version is working it is possible to step back and evaluate it, to see if its overall shape is sound. When it has been adjusted until it is structurally correct, effort can go into adding in missing features and generally upgrading it until it eventually gets transformed into the beautiful butterfly that was really wanted. Of all the methods that I have described this is the one that comes closest to allowing for “experimental” programming. The discipline to adhere to is that experiments are worthy of that tag if the results from them can be evaluated and if something can thus be learned from them.

7.5 Which of the above is best?

The “best” technique for getting a program written will depend on its size as well as its nature. I think that puritanical adherence to any of the above would be unreasonable, and I also believe that inspiration and experience (and good taste) have important roles to play. However if pushed into an opinion I will vote for presenting a design or a program (whether already finished or still under construction) as if it were prepared top-down, with an emphasis on the early design of what information must be represented and where it must pass from one part of the code to another.

8 How do we know it will work?

Nobody should ever write a program unless they have good reason to believe that it ought to work. It is of course proper to recognise that it will not work, because typographic errors and all sorts of oversights will ensure that. But the code should have been written so that in slightly idealised world where these accidental imperfections do not exist it would work perfectly. Blind and enthusiastic hope is not sufficient to make programs behave well, and so any proper design needs to have lurking behind it the seeds of a correctness proof. In easy-going times this can remain untended as little comments that can just remind you of your thinking. When a program starts to get troublesome it can be worth growing these comments into short essays that explain what identities are being preserved intact across regions of code, why your loops are guaranteed to terminate and what assumptions about data are important, and why. In yet more demanding circumstances it can become necessary to conduct formal validation procedures for code.

The easiest advice to give here is that before you write even half a dozen lines of code you should write a short paragraph of comment that explains what the code is intended to achieve and why your method will work. The comment should usually not explain **how** it works (the code itself is all about “how”), but **why**. To try to show that I (at least sometimes!) follow this advice here is a short extract from one of my own programs…

```
/*
 * Here is a short essay on the interaction between flags and
 * properties. It is written because the issue appears to be
 * delicate, especially in the face of a scheme that I use to
 * speed things up.
 * (a) If you use FLAG, REMFLAG and FLAGP with some indicator
 *     then that indicator is known as a flag.
 * (b) If you use PUT, REMPROP and GET with an indicator then
 *     what you have is a property.
 * (c) Providing the names of flags and properties are disjoint
 *     no difficulty whatever should arise.
 * (d) If you use PLIST to gain direct access to a property list
 *     then flags are visible as pairs (tag . t) and properties
 *     as (tag . value).
 * (e) Using RPLACx operations on the result of PLIST may cause
 *     system damage.  It is to be considered illegal. Also
 *     changes made that way may not be matched in any
 *     accelerating caches that I keep.
 * (f) After (FLAG ’(id) ’tag) [when id did not previously have
 *     any flags or properties] a call (GET ’id ’tag) will
 *     return t.
 * (g) After (PUT ’id ’tag ’anything) a call (FLAGP ’id ’tag)
 *     will return t whatever the value of "anything".  A call
 *     (GET ’id ’tag) will return the saved value (which might
 *     be nil).  Thus FLAGP can be thought of as a function
 *     that tests if a given property is attached to a symbol.
 * (h) As a consequence of (g) REMPROP and REMFLAG are really
 *     the same operation.
 */

Lisp_Object get(Lisp_Object a, Lisp_Object b)
{
    Lisp_Object pl, prev, w, nil = C_nil;
    int n;
/*
 * In CSL mode plists are structured like association lists, and
 * NOT as lists with alternate tags and values.  There is also
 * a bitmap that can provide a fast test for the presence of a
 * property...
 */
    if (!symbolp(a))
    {
#ifdef RECORD_GET
        record_get(b, NO);
        errexit();
#endif
        return onevalue(nil);
    }
    ... etc etc
```

The exact details of what I am trying to do are not important here, but the evidence of mind-clearing so that there is a chance to get the code correct first time is. Note how little the comment before the procedure has to say about low-level implementation details, but how much about specifications, assumptions and limitations.

I would note here that keyboarding is generally one of the least time-consuming parts of the whole programming process, and these days disc storage is pretty cheap, and thus various reasons which in earlier days may have discouraged lay-out and explanation in code no longer apply.

Before trying code and as a further check that it ought to work it can be useful to "walk through" the code. In other words to pretend to be a computer executing it and see if you follow the paths and achieve the results that you were supposed to. While doing this it can be valuable to think about which paths through the code are common and which are not, since when you get to testing it may be that the uncommon paths do not get exercised very much unless you take special steps to cause them to be activated.

The "correctness" that you will be looking for can be at several different lev-els. A *partially correct* program is one that can never give an incorrect answer. This sounds pretty good until you recognise that there is a chance that it may just get stuck in a loop and thereby never give any answer at all! It is amazingly often much easier to justify that a program is partially correct than to go the whole hog and show it is correct, ie that not only is it partially correct but that it will always terminate. Beyond even the requirements of correctness will be performance de-mands: in some cases a program will need not only to deliver the right answers but to meet some sort of resource budget. Especially if the performance target is specified as being for performance that is good "on the average" it can be dread-fully hard to prove, and usually the only proper way to start is by designing and justifying algorithms way before any mention of actual programming arises.

A final thing to check for is the possibility that your code can be derailed by unhelpful erroneous input. For instance significant security holes in operating systems have in the past been consequences of trusted modules of code being too trusting of their input, and them getting caught out by (eg) input lines so long that internal buffers overflowed thereby corrupting adjacent data.

The proper mind-set to settle in to while designing and starting to implement code is pretty paranoid: you want the code to deliver either a correct result or a comprehensible diagnostic whenever anything imaginable goes wrong in either the data presented to it or its own internal workings. This last statement leads to a concrete suggestion: make sure that the code can test itself for sanity and correctness every so often and insert code that does just that. The assertions that you insert will form part of your argument for why the program is supposed to work, and can help you (later on) debug when it does not.

9 While you are writing the program

Please remember to get up and walk around, to stretch, drink plenty of water, sit up straight and all the other things mentioned at the Learning Day as relevant occupational health issues. My experience is that it is quite hard to do effective programming in 5 minute snippets, but that after a few hours constant work productivity decreases. A pernicious fact is that you may not notice this decrease at the time, in that the main way in which a programmer can become unproductive is by putting more bugs into a program. It is possible to keep churning out lines of code all through the night, but there is a real chance that the time you will spend afterwards trying to mend the last few of them will mean that the long session did not really justify itself.

In contrast to programming where long sessions can do real damage (because of the bugs that can be added by a tired programmer) I have sometimes found that long sessions have been the only way I can isolate bugs. Provided I can discipline myself not to try to correct anything but the very simplest bug while I am tired a long stretch can let me chase bugs in a painstakingly logical way, and this is sometimes necessary when intuitive bug-spotting fails.

Thus my general advice about the concrete programming task would be to schedule your time so you can work in bursts of around an hour per session, and that you should plan your work so that as much as possible of everything you do can be tested fairly enthusiastically while it is fresh in your mind. A natural corollary of this advice is that projects should always be started in plenty of time, and pushed forward consistently so that no last-minute panic can arise and force sub-optimal work habits.

10 Documenting a program or project

Student assessed exercises are expected to be handed in complete with a brief report describing what has been done. Larger undergraduate projects culminate in the submission of a dissertation, as do PhD studies. All commercial programming activities are liable to need two distinct layers of documentation: one for the user and one for the people who will support and modify the product in the future. All these facts serve to remind us that documentation is an intrinsic part of any program.

Two overall rules can guide the writing of good documentation. The first is to consider the intended audience, and think about what they need to know and how your document can be structured to help them find it. The second is to keep a degree of consistency and order to everything: documents with a coherent overall structure are both easier to update and to browse than sets of idiosyncratic jottings.

To help with the first of these, here are some potential styles of write-up that might be needed:

1. Comments within the code to remind yourself or somebody who is already familiar with the program exactly what is going on at each point in it;
2. An overview of the internal structure and organisation of the whole program so that somebody who does not already know it can start to find their way around;
3. Documentation intended to show how reliable a program is, concentrating on discussions of ways in which the code has been built to be resilient in the face of unusual combinations of circumstance;
4. A technical presentation of a program in a form suitable for publication in a journal or at a conference, where the audience will consist of people expert in the general field but not aware of exactly what your contribution is;
5. An introductory user manual, intended to make the program usable even by the very very nervous;
6. A user reference manual, documenting clearly and precisely all of the options and facilities that are available;
7. On-line help for browsing by the user while they are trying to use the program;
8. A description of the program suitable for presentation to the venture capitalists who are considering investing in the next stage of its development.

It seems inevitable that the above list is not exhaustive, but my guess is that most programs could be presented in any one of the given ways, and the resulting document would be quite different in each case. It is not that one or the other of these styles is inherently better or more important than another, more that if you write the wrong version you will either not serve your reader well or you will find that you have had to put much more effort into the documentation than was really justified.

A special problem about documentation is that of when it should be written. For small projects at least it will almost always be produced only after the program has been (at least nearly) finished. This can be rationalised by claiming "how can I possibly document it before it exists?"

I will argue here for two ideals. The first is that documentation ought to follow on from design and specification work, but precede detailed programming. The second is that the text of the documentation should live closely linked to the developing source code. The reasoning behind the first of these is that writing the text can really help to ensure that the specification of the code has been fully thought through, and once it is done it provides an invaluable stable reference to keep the detailed programming on track. The second point recognises some sort of realism, and that all sorts of details of just what a program does will not be resolved until quite late in the implementation process. For instance the exact wording of messages that are printed will often not be decided until then, and it will certainly be hard to prepare sample transcripts from the use of the program ahead of its completion[4]. Thus when the documentation has been written early it will need completing when some of these final details get settled and correcting when the code is corrected or extended. The most plausible way of making it feasible to keep code and description in step is to keep them together. The concept of Literate Programming[10] pursues this goal. A program is represented as a composite file that can be processed in (at least) two different ways. One way "compiles" it to create typeset-quality human readable documentation, while the other leaves just statements in some quite ordinary programming language ready to be fed into a compiler. This goes beyond just having copious comments in the code in two ways. Firstly it expects that the generated documentation should be able to exploit the full range of modern typography and that it can include pictures or diagrams where relevant. It is supposed to end up as cleanly presented and readable as any fully free-standing document could ever be. Secondly Literate Programming recognises that the ordering and layout of the program that has to be compiled may not be the same as that in the ideal manual, and so the disentangling tool needs to be able to rearrange bits of text in a fairly flexible way so that description can simultaneously be thought of as close to the code it relates to and to the section in the document where it belongs. This idea was initially developed as part of the project to implement the TEX typesetting program that is being used to prepare these lecture notes.

11 How do we know it does work?

A conceptual difficulty that many people suffer from is a confusion between whether a program should work and whether it does. A program should work if it has been designed so that there are clear and easily explained reasons why it can achieve what it should. Sometimes the term "easily explained" may conceal the mathematical proof of the correctness of an algorithm, but at least in theory it would be possible to talk anybody through the justification. As to programs that actually do work, well the reality seems to be that the only ones of these that you

[4]Even though these samples can be planned and sketched early.

will ever see will be no more than around 100 lines long: empirically any program much longer than that will remain flawed even after extensive checking. Proper Oriental rugs will always have been woven with a deliberate mistake in them, in recognition of the fact that only Allah is perfect. Experience has shown very clearly indeed that in the case of writing programs we all have enough failings that there is no great need to insert extra errors — there will be plenty inserted however hard we try to avoid them. Thus (at least at the present state of the art) there is no such thing as a (non-trivial) program that works.

If, however, a program *should* work (in the above sense) then the residual errors in it will be ones that can be corrected without disturbing the concepts behind it or its overall structure. I would like to think of such problems as "little bugs". The fact that they are little does not mean that they might not be important, in that missing commas or references to the wrong variable can cause aeroplanes to crash just as convincingly as can errors at a more conceptual level. But the big effort must have been to get to a first testable version of your code with only little bugs left in it. What is then needed is a testing strategy to help locate as many of these as possible. Note of course that testing can only ever generate evidence for the presence of a bug: in general it can not prove absence. But careful and systematic testing is something we still need whenever there has been human involvement in the program construction process[5].

The following thoughts may help in planning a test regime:

1. Even obvious errors in output can be hard to notice. Perhaps human society has been built up around a culture of interpreting slightly ambiguous input in the "sensible" way, and certainly we are all very used to seeing what we expect to see even when presented with something rather different. By the time you see this document I will have put some effort into checking its spelling, punctuation, grammar and general coherence, and I hope that you will not notice or be upset by the residual mistakes. But anybody who has tried serious proof-reading will be aware that blatant mistakes can emerge even when a document has been checked carefully several times;

2. If you are checking your own code and especially if you know you can stop work once it is finished then you have a clear incentive **not** to notice mistakes. Even if a mistake you find is not going to cause you to have to spend time fixing it it does represent you having found yet another instance of your own lack of attention, and so it may not be good for your ego;

3. It is very desirable to make a clear distinction between the job of testing a program to identify the presence of bugs and the separate activity of correct-

[5]Some see this observation as a foundation for hope for the future

ing things. It can be useful to take the time to try to spot as many mistakes as you can before changing anything at all;

4. A program can contain many more bugs and oddities than your worst nightmares would lead you to believe!

5. Testing strategies worked out as part of the initial design of a program are liable to be better than ones invented only once code has been completed;

6. It can be useful to organise explicit test cases for extreme conditions that your program may face (eg sorting data where all the numbers to be sorted have the same value), and to collect test cases that cause each path through your code to be exercised. It is easy to have quite a large barrage of test cases but still have some major body of code unvisited.

7. Regressions tests are a good thing. These are test cases that grow up during project development, and at each stage after any change is made all of them are re-run, and the output the produce is checked. When any error is detected a new item in the regression suite is prepared so that there can remain a definite verification that the error does not re-appear at some future stage. Automating the application of regression tests is a very good thing, since otherwise laziness can too easily cause one to skip running them;

8. When you find one bug you may find that its nature gives you ideas for other funny cases to check. You should try to record your thoughts so that you do not forget this insight;

9. Writing extra programs to help you test your main body of code is often a good investment in time. On especially interesting scheme is to generate pseudo-random test cases. I have done that while testing a polynomial factorising program and suffered randomly-generated tests of a C compiler I was involved with, and in each case the relentless random coverage of cases turned out to represent quite severe stress;

10. You do not know how many bugs your code has in it, so do not know when to stop looking. One theoretical way to attack this worry would be to get some fresh known bugs injected into your code before testing, and then see what proportion of the bugs found were the seeded-in ones and which had been original. That may allow you to predict the total bug level remaining.

Having detected some bugs there are several possible things to do. One is to sit tight and hope that nobody else notices! Another is to document the deficiencies at the end of your manual. The last is to try to correct some of them. The first two of these routes are more reasonable than might at first seem proper given that correcting bugs so very often introduces new ones.

In extreme cases it may be that the level of correctness that can be achieved by bug-hunting will be inadequate. Sometimes it may then be possible to attempt a formal proof of the correctness of your code. In all realistic circumstances this will involve using a large and complicated proof assistant program to help with all the very laborious details involved. Current belief is that it will be very unusual for bugs in the implementation of this tool to allow you to end up with a program that purports to be proved but which in fact still contains mistakes!

12 Is it efficient?

I have made this a separate section from the one on detecting the presence of errors because performance effects are only rarely the result of simple oversights. Let me start by stressing the distinction between a program that is expensive to run (eg the one that computes π to 20,000,000,000 decimal places) and ones that are inefficient (eg one that takes over half a second to compute π correct to four places). The point being made is that unless you have a realistic idea of how long a task ought to take it is hard to know if your program is taking a reasonable amount of time. And similarly for memory requirements, disc I/O or any other important resource. Thus as always we are thrown back to design and specification time predictions as our only guideline, and sometimes even these will be based on little more than crude intuition.

If a program runs fast enough for reasonable purposes then there may be no benefit in making it more efficient however much scope for improvement there is. In such cases avoid temptation. It is also almost always by far best to concentrate on getting code correct first and only worry about performance afterwards, taking the view that a wrong result computed faster is still wrong, and correct results may be worth waiting for.

When collecting test cases for performance measurements it may be useful to think about whether speed is needed in every single case or just in most cases when the program is run. It can also be helpful to look at how costs are expected to (and do) grow as larger and larger test cases are attempted. For most programming tasks it will be possible to make a trade between the amount of time a program takes to run and the amount of memory it uses. Frequently this shows up in a decision as to whether some value should be stored away in case it is needed later or whether any later user should re-calculate it. Recognising this potential trade-off is part of performance engineering.

For probably the majority of expensive tasks there will be one single part of the entire program that is responsible for by far the largest amount of time spent. One would have expected that it would always be easy to predict ahead of time where that would be, but it is not! For instance when an early TITAN Fortran compiler was measured in an attempt to discover how it could be speeded up it was found that over half of its entire time was spent in a very short loop of instructions that were to do with discarding trailing blanks from the end of input lines. Once the programmers knew that it was easy to do something about it, but one suspects they were expecting to find a hot-spot in some more arcane part of the code. It is thus useful to see if the languages and system you use provide instrumentation that makes it easy to collect information to reveal which parts of your code are most critical. If there are no system tools to help you you may be able to add in time-recording statements to your code so it can collect its own break-down to show what is going on. Cunning optimisation of bits of code that hardly ever get used is probably a waste of effort.

Usually the best ways to gain speed involve re-thinking data structures to provide cheap and direct support for the most common operations. This can sometimes mean replacing a very simple structure by one that has huge amounts of algorithmic complexity (there are examples of such cases in the Part IB Complexity course and the Part II one on Advanced Algorithms). In almost all circumstances a structural improvement that gives a better big-O growth rate for some critical cost is what you should seek.

In a few cases the remaining constant factor improvement in speed may still be vital. In such cases it may be necessary to re-write fragments of your code in less portable ways (including the possibility of use of machine code) or do other things that tend to risk the reliability of your package. The total effort needed to complete a program can increase dramatically as the last few percent in absolute performance gets squeezed out.

13 Identifying errors

Section 8 was concerned with spotting the presence of errors. Here I want to talk about working out which part of your code was responsible for them. The sections are kept separate to help you to recognise this, and hence to allow you to separate noticing incorrect behaviour from spotting mistakes in your code. Of course if, while browsing code, you find a mistake you can work on from it to see if it can ever cause the program to yield wrong results, and this study of code is one valid error-hunting activity. But even in quite proper programs it is possible to have errors that never cause the program to misbehave in any way that can be noticed. For instance the mistake might just have a small effect on the performance of some not too important subroutine, or it may be an illogicality that could only be triggered into causing real trouble by cases that some earlier line of code had filtered out.

You should also recognise that some visible bugs are not so much due to any single clear-cut error in a program but to an interaction between several parts of your code each of which is individually reasonable but which in combination fail. Most truly serious disasters caused by software failure arise because of complicated interactions between multiple “improbable” circumstances.

The first thing to try to locate the cause of an error is to start from the original test case that revealed it and to try to refine that down to give a minimal clear-cut demonstration of the bad behaviour. If this ends up small enough it may then be easy to trace through and work out what happened.

Pure thought and contemplation of your source code is then needed. Decide what Sherlock Holmes would have made of it! Run your compilers in whatever mode causes them to give as many warning messages as they are capable of, and see if any of those give valuable clues.

If this fails the next thought is to arrange to get a view on the execution of your code as it makes its mistake. Even when clever language-specific debuggers are available it will often be either necessary or easiest to do this by extra print statements into your code so it can display a trace of its actions. There is a great delicacy here. The trace needs to be detailed enough to allow you to spot the first line in it where trouble has arisen, but concise enough to be manageable. My belief is that one should try to judge things so that the trace output from a failing test run is about two pages long.

There are those who believe that programs will end up with the best reliability if they start off written in as fragile way as possible. Code should always make as precise a test as possible, and should frequently include extra cross checks which, if failed, cause it to give up. The argument is that this way a larger number of latent faults will emerge in early testing, and the embedded assertions can point the programmer directly to the place where an expectation failed to be satisfied, which is at least a place to start working backwards from in a hunt for the actual bug.

With many sorts of bugs it can be possible to home in on the difficulty by some sort of bisection search. Each test run should be designed to halve the range of code within which the error has been isolated.

Some horrible problems seem to vanish as soon as you enable any debugging features in your code or as soon as you insert extra print statements into it. These can be amazingly frustrating! They may represent your use of an unsafe language and code that writes beyond the limit of an array, or they could involve reliance on the unpredictable value of an un-initialised variable. Sometimes such problems turn out to be bugs in the compiler you are using, not in your own code. I believe that I have encountered trouble of some sort (often fairly minor, but trouble nevertheless) with every C compiler I have ever used, and I have absolute confidence that no other language has attained perfection in this regard. So sometimes trying your code on a different computer or with a different compiler will either give you a new diagnostic that provides the vital clue, or will behave differently thereby giving scope for debugging-by-comparison.

Getting into a panic and trying random changes to your code has no proper part to play either in locating or identifying bugs.

14 Corrections and other changes

With a number of bugs spotted and isolated the time comes to extirpate them. The ideal should be that when a bug is removed it should be removed totally and it should never ever be able to come back. Furthermore its friends and offspring should be given the same treatment at the same time, and of course no new mistakes should be allowed to creep in while the changes are being made. This last is often taken for granted, but when concentrating on one particular bug it is all too easy to lose sight of the overall pattern of code and even introduce more new bugs than were being fixed in the first case. Regression testing is at least one line of defence that one should have against this, but just taking the correction slowly and thinking through all its consequences what is mostly wanted. Small bugs (in the sense discussed earlier) that are purely local in scope give fewest problems. However sometimes testing reveals a chain of difficulties that must eventually be recognised as a sign that the initial broad design of the program had been incorrect, and that the proper correction strategy does not involve fixing the problems one at a time but calls for an almost fresh start on the whole project. I think that would be the proper policy for the program in section 19, and that is part of why the exercise there asks you to identify bugs but not to correct them.

Upgrading a program to add new features is at least as dangerous as correcting bugs, but in general any program that lasts for more than a year or so will end up with a whole raft of alterations having been made to it. These can very easily damage its structure and overall integrity, and the effect can be thought of as a form of *software rot* that causes old code to decay. Of course software rot would not arise if a program never needed correcting and never needed upgrading, but in that case the program was almost certainly not being used and was fossilised rather than rotting. NOte that for elderly programs the person who makes corrections is never the original program author (even if they have the same name and birthday, the passage of time has rendered them different). This greatly increases the prospect of a would-be correction causing damage.

All but the most frivolous code should be kept under the control of some source management tool (perhaps `rcs`) that can provide an audit trail so that changes can be tracked. In some cases a discussion of a bug that has now been removed might properly remain as a comment in the main source code, but much more often a description of what was found to be wrong and what was changed to mend it belongs in a separate project log. After all if the bug really has been removed who has any interest in being reminded of the mistake that it represented?

Whenever a change is made to a program, be it a bug-fix or an upgrade, there is a chance that some re-work will be needed in documentation, help files, sample logs and of course the comments. Once again the idea of literate programming comes to the fore in suggestion that all these can be kept together so that none of them get missed out.

15 Portability of software

Most high level languages make enthusiastic claims that programs written in them will be portable from one brand of computer to another, just as most make claims that their compilers are “highly optimising”.

In reality achieving portability for even medium sized programs is not as easy as all that. To give a gross example of a problem not addressed at all by programming language or standard library design, a Macintosh comes as standard with a mouse with a single button, while most Unix X-windows systems have three-button mice. In one sense the difference is a frivolity, but at another it invites a quite substantial re-think of user interface design. At the user interface level a design that makes good use of a screen with 640 by 480 pixels and 16 or 256 colours (as may be the best available on many slightly elderly computers) may look silly on a system with very much higher resolution and more colours.

For most programming languages you will find that implementations provided by different vendors do not quite match. Even with the most standardised languages hardly any compiler supplier will manage to hold back from providing some private extra goodies that help distinguish them from their competitors. Such extras will often be things that it is very tempting to make use of. Around Easter 1997 a good example of such a feature is “Active-X” which Microsoft is promoting. To use such a feature tends to lock you to one vendor or platform, while to ignore it means that you can not benefit from the advantages that it brings. By now you will know what my suggested response to conflicts like this will be. Yes, it is to make your decisions explicitly and consciously rather than by default, to make them in view of stated ideas about what the users of your code will need, and to include all the arguments you use to support your decision in your design portfolio.

There are frequently clever but non-portable tricks that can lead to big performance gains in code but at cost in portability. Sometimes the proper response to these is to have two versions of the program, one slow but very portable and the other that takes full advantage of every trick available on some platform that is especially important to you.

16 Team-work

Almost all of this course is about programming in the small, with a concentration on the challenges facing a lone programmer. It is still useful to think for a while how to handle the transition from this state into a large-team corporate mentality. One of the big emotional challenges in joining a team relates to the extent to which you end up "owning" the code you work on. It is very easy to get into a state where you believe (perhaps realistically) that you are the only person who can properly do anything to the code you write. It is also easy to become rather defensive about your own work. A useful bit of jargon that refers to breaking out of these thought patterns is *ego-free programming*. In this ideal you step back and consider the whole project as the thing you are contributing to, not just the part that you are visibly involved in implementing. It may also be useful to recognise that code will end up with higher quality if understanding of it is shared between several people, and that bugs can be viewed as things to be found and overcome and never as personal flaws in the individual who happened to write that fragment of code.

When trying to design code or find a difficult bug it can be very valuable to explain your thoughts to somebody else. It may be that they need not say much more than er and um, and maybe they hardly need to listen (but you probably need to believe that they are). By agreeing that you will listen to their problems at a later stage this may be a habit you can start right now with one or a group of your contemporaries.

Reading other people's code (with their permission, of course) and letting them read yours can also help you settle on a style or idiom that works well for you. It can also help get across the merits of code that is well laid out and where the comments are actually helpful to the reader.

If you get into a real group programming context, it may make sense to consider partitioning the work in terms of function, for instance system architect, programmer, test case collector, documentation expert,. . . rather than trying to distribute the management effort and split the programming into lots of little modules, but before you do anything too rash read some more books on software engineering so that once again you can make decisions in an informed and considered way.

17 Lessons learned, Conclusion

One of the oft-repeated observations about the demons of large-scale software construction is that *there is no silver bullet.* In other words we can not expect to find a single simple method that, as if by magic, washes away all our difficulties. This situation also applies for tasks that are to be carried out by an individual programmer or a very small team. No single method gives a key that makes it possible to sit down and write perfect programs without effort. The closest I can come to an idea for something that is generally valuable is experience – experience on a wide range of programming projects in several different languages and with various different styles of project. This can allow you to spot features of a new task that have some commonalty with one seen before. This is, however, obviously no quick fix. The suggestions I have been putting forward here are to try to make your analysis of what you are trying to achieve as explicit in your mind as possible. The various sections in these notes provide headings that may help you organise your thoughts, and in general I have tried to cover topics in an order that might make sense in real applications. Of course all the details and conclusions will be specific to your problem, and nothing I can possibly say here can show you how to track down your own very particular bug or confusion! I have to fall back on generalities. Keep thinking rather than trying random changes to your code. Try to work one step at a time. Accept that errors are a part of the human condition, and however careful you are your code will end up with them.

But always remember the two main slogans:

Programming is easy

and

Programming is fun.

18 Some challenges

Some of you may already consider yourselves to be seasoned programmers able to cope with even quite large and complicated tasks. In which case I do not you to feel this course is irrelevant, and so I provide here at the end of the notes some programming problems which I believe are hard enough to represent real challenges, even though the code that eventually has to be written will not be especially long. There is absolutely no expectation that anybody will actually complete any of these tasks, or even find good starting points. However these examples may help give you concrete cases to try out the analysis and design ideas I have discussed: identifying the key difficulties and working out how to partition the problems into manageable chunks. In some cases the hardest part of a proper plan would be the design of a good enough testing strategy. The tasks described here are all both reasonably compact and fairly precisely specified. I have fought most of these myself and found that producing solutions that were neat and convincing as well as correct involved thought as well as more coding skill. There are no prizes and no ticks, marks or other bean-counter's credit associated with attempting these tasks, but I would be jolly interested to see what any of you can come up with, provided it can be kept down to no more than around 4 sides of paper.

18.1 MULDIV

The requirement here is to produce a piece of code that accepts four integers and computes $(a * b + c)/d$ and also the remainder from the division. It should be assumed that the computer on which this code is to be run has 32-bit integers, and that integer arithmetic including shift and bitwise mask operations are available, but the difficulty in this exercise arises because $a * b$ will be up to 64-bits long and so it can not be computed directly. "Solutions" that use (eg) the direct 64-bit integer capabilities of a DEC Alpha workstation are not of interest!

It should be fairly simple to implement `muldiv` if efficiency where not an issue. To be specific this would amount to writing parts of a package that did double-length integer arithmetic. Here the additional expectation is that speed does matter, and so the best solution here will be one that makes the most effective possible use of the 32-bit arithmetic that is available. Note also that code of this sort can unpleasantly easily harbour bugs, for instance due to some integer overflow of an intermediate result, that only show up in very rare circumstances, and that the pressure to achieve the best possible performance pushes towards code that comes very close to the limits of the underlying 32-bit arithmetic. Thought will be needed when some or all of the input values are negative. The desired behaviour is one where the calculated quotient was rounded towards zero, whatever its sign.

18.2 Overlapping Triangles

A point in the X–Y plane can be specified by giving its co-ordinates (x, y). A triangle can then be defined by giving three points. Given two triangles a number of possibilities arise: they may not overlap at all or they may meet in a point or a line segment, or they may overlap so that the area where they overlap forms a triangle, a quadrilateral, a pentagon or a hexagon. Write code that discovers which of these cases arises, returning co-ordinates that describe the overlap (if any).

A point to note here is that any naive attempt to calculate the point where two lines intersect can lead to attempts to divide by zero if the lines are parallel.

Near-parallel lines can lead to division by very small numbers, possibly leading to subsequent numeric overflow. Such arithmetic oddities must not be allowed to arise in the calculations performed.

18.3 Matrix transposition

One way of representing an m by n matrix in a computer is to have a single vector of length mn and place the array element $a_{i,j}$ at offset $mi+j$ in the vector. Another would be to store the same element at offset $i + nj$. One of these representation means that items in the same row of the matrix live close together, the other that items in the same column are adjacent.

In some calculations it can make a significant difference to speed which of these layouts is used. This is especially true for computers with virtual memory. Sometimes one part of a calculation would call for one layout, and a later part would prefer the other.

The task here is therefore to take integers m and n and a vector of length mn, and rearrange the values stored in the vector so that if they start off in one of as one representation of a matrix they end up as the other. Because the matrix should be assumed to be quite large you are not allowed to use any significant amount of temporary workspace (you can not just allocate a fresh vector of length mn and copy the data into it in the new order — you may assume you may use extra space of around $m + n$ if that helps, but not dramatically more than that).

If the above explanation of the problem[6] feels out of touch with today's computer uses, note how the task relates to taking an m by n matrix representing a picture and shuffling the entries to get the effect of rotating the image by 90 degrees. Just that in the image processing case you may be working with data arranged in sub-word-sized bite-fields, say at 4 bits per pixel.

18.4 Sprouts

The following is a description of a game[7] to be played by two players using a piece of paper. The job of the project here is to read in a description of a position in the game and make a list of all the moves available to the next player. This would clearly be needed as part of any program that played the game against human opposition, but the work needed here does not have to consider any issues concerning the evaluation of positions or the identification of good moves.

The game starts with some number of marks made on a piece of paper, each mark in the form of a capital 'Y'. Observe that each junction has exactly three

[6]This is an almost standard classical problem and if you dig far enough back in the literature you will find explanations of a solution. If you thought to do that for yourself, well done!

[7]Due to John Conway

little edges jutting from it. A move is made by a player identifying two free edges and drawing a line between them. The line can snake around things that have been drawn before as much as the player making the move likes, but it must not cross any line that was drawn earlier. The player finishes the move by drawing a dot somewhere along the new line and putting the stub of a new edge jutting out from it in one of the two possible directions. Or put a different but equivalent way, the player draws a new ‘Y’ and joins two of its legs up to existing stubs with lines that do not cross any existing lines. The players make moves alternately and the first player unable to make a further legal move will be the loser.

A variation on the game has the initial state of the game just dots (not ‘Y’ shapes) and has each player draw a new dot on each edge they create, but still demands that no more that three edges radiate from each dot. The difference is that in one case a player can decide which side of a new line any future line must emerge from. I would be equally happy whichever version of the game was addressed by a program, provided the accompanying documentation makes it clear which has been implemented!

The challenge here clearly largely revolves around finding a way to describe the figures that get drawn. If you want to try sprouts out as a game between people before automating it, I suggest you start with five or six starting points.

18.5 ML development environment

The task here is not to write a program, but just to sketch out the specification of one. Note clearly that an implementation of the task asked about here would be quite a lot of work and I do not want to provide any encouragement to you to attempt all that!

In the Michaelmas Term you were introduced to the language ML, and invited to prepare and test various pieces of test code using a version running under Microsoft Windows. You probably used the regular Windows “notepad” as a little editor so you could change your code and then paste back corrected versions of it into the ML window. Recall that once you have defined a function or value in ML that definition remains fixed for ever, and so if it is incorrect you probably need to re-type not only it but everything you entered after it. All in all the ML environment you used was pretty crude (although I am proud of the greek letters in the output it generates), and it would become intolerable for use in medium or large-scale projects. Design a better environment, and append to your description of it a commentary about which aspects of it represent just a generically nice programmer’s work-bench and which are motivated by the special properties of ML.

19 An example from the literature

The following specification is given as a paragraph of reasonably readable English text, and there is then an associated program written in the language C. The fact that the code is in C rather than any other language may slow your understanding of it down somewhat, but decoding programs written in slightly unfamiliar languages is a valuable skill to gain! This quite small chunk of code can give you experience of bug-hunting: please do not look up the original article in CACM until you have spent some while working through the code checking how it works and finding some of the mistakes. In previous years when I have presented this material to our students they did almost as well as the professional programmers used in the original IBM study, but they still found only a pathetically small proportion of the total number of known bugs!

```
/*
 * Formatting program for text input.  Converted from PL/I to C by
 * A. C. Norman,  January 1989, for use in practical classes.
 * Original PL/I version from a paper by Glen Myers, CACM vol 21
 * no 9, 1978
 *
 * (a) This program compiles correctly: it is believed not to
 *     contain either syntax errors or abuses of the C library.
 * (b) A specification is given below.  You are to imagine that the
 *     code appended was produced by somebody who had been provided
 *     with the specification and asked to produce an
 *     implementation of the utility as described.
 * (c) Your task is one of quality control - it is to check that
 *     the code as given is in agreement with the specification.
 *     If any bugs or mis-features are discovered they should be
 *     documented but it will be up to the original programmer to
 *     correct them.
 *     If there are bugs it is desirable that they all be found.
 * (d) For the purposes of this study, a bug or a mis-feature is
 *     some bad aspect of the code that could be visible to users
 *     of the binary version of the code.  Ugly or inefficient code
 *     is deemed not to matter, but even small deviations from the
 *     letter of the specification and the things sensibly implicit
 *     in it do need detecting.
 * (e) Let me repeat point (a) again just to stress it - the code
 *     here has had its syntax carefully checked and uses the C
 *     language and library in a legal straightforward way, so
 *     searching for bugs by checking fine details of the C
 *     language specification is not expected to be productive.
 *     I have put in comments to gloss use of C library functions
 *     as an aid to those who know the syntax of C but not the
 *     names/specification of all things in said library.  I have
 *     tried to keep layout of the code neat and consistent.
 *     There are few comments "because the original programmer who
 *     wrote the code delivered it in that state".
 */
```

```
/*****************************************************************
 * Specification                                                 *
 * =============                                                 *
 *                                                               *
 * Given an input text consisting of words separated by blanks or *
 * new-line characters, the program formats it into a line-by-line*
 * form such that (1) each output line has a maximum of 30        *
 * characters, (2) a word in the input text is placed on a single *
 * output line, and (3) each output line is filled with as many   *
 * words as possible.                                             *
 *                                                               *
 * The input text is a stream of characters, where the characters *
 * are categorized as break or nonbreak characters.  A break      *
 * character is a blank, a new-line character (&), or an end of   *
 * text character (/). New-line characters have no special        *
 * significance; they ar treated as blanks by the program.  & and *
 * / should not appear in the output.                             *
 *                                                               *
 * A word is defined as a nonempty sequence of non-break          *
 * characters. A break is a sequence of one or more break         *
 * characters.  A break in the input is reduced to a single blank *
 * or start of new line in the output.                            *
 *                                                               *
 * The input text is a single line entered from a terminal similar*
 * to a BBC micro with an 80 character screen mode.  When the     *
 * program is invoked it waits for the program to provide input.  *
 * The user types the input line, followed by a / (end of text)   *
 * and a carriage return.  The program then formats the text and  *
 * types it on the terminal.                                      *
 *                                                               *
 * If the input text contains a word that is too long to fit on a *
 * single output line, an error message is typed and the program  *
 * terminates.  If the end-of-text character is missing, an error *
 * message is issued and the user is given a chance to type in a  *
 * corrected version of the input line.                           *
 *                                                               *
 * (end of specification)                                         *
 *****************************************************************/
```

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

static int gchar(void);     /* forward reference to gchar() */
static void pchar(int);

int main()
{
#define LINESIZE 31
    int k,
        bufpos,
        fill,
        maxpos = LINESIZE,
        cw,
        blank = ’ ’,
        linefeed = ’$’,
        eotext = ’/’,
        moreinput = 1;
    char buffer[LINESIZE];
    bufpos = 0;
    fill = 0;
    while (moreinput)
    {
        cw = gchar();
        if (cw == blank || cw == linefeed || cw == eotext)
        {
            if (cw == eotext) moreinput = 0;
            if ((fill + 1 + bufpos) <= maxpos)
            {
                pchar(blank);
                fill = fill + 1;
            }
            else
            {
                pchar(linefeed);
                fill = 0;
            }
            for (k = 0; k < bufpos; k++) pchar(buffer[k]);
            fill = fill + bufpos;
            bufpos = 0;
        }
        else if (bufpos == maxpos)
        {
            moreinput = 0;
            printf("Word to long\n");
        }
        else
        {
            bufpos = bufpos + 1;
            buffer[bufpos-1] = cw;
        }
    }
    pchar(linefeed);
    return 0;
}


static void getrecord(char *);

static int gchar()
{
#define ILENGTH 80
    static char buffer[ILENGTH] = {'Z'};
    /* Static array with first     */
    /* element starting off as 'Z' */
    char inbuf[ILENGTH];
    static int bcount = 1;
    /* static so persists between calls    */
    int eotext = '/';
    int c;

    if (buffer[0] == 'Z')
    {
        getrecord(inbuf);
/*
 * memchr scans the memory at inbuf to see if the byte (eotext
 * here) is present within the first ILENGTH bytes.  If not it
 * will return NULL, if it is present it returns a pointer to it.
 */
        if (memchr(inbuf, eotext, ILENGTH) == 0)
        {
            printf("No end of text mark\n");
            buffer[1] = eotext;
        }
```

```
/*
 * memcpy is used here to copy ILENGTH bytes from inbuf to buffer.
 */
        else memcpy(buffer, inbuf, ILENGTH);
    }
    c = buffer[bcount-1];
    bcount = bcount + 1;
    return c;
}

static void pchar(int c)
{
/*
 * The static array outline is filled with blanks to start with.
 * The number of blanks in the string given is exactly LINESIZE
 * so that the array is neatly filled up.  No need to count the
 * characters, there is no cheating going on here!
 */
    static char outline[LINESIZE] =
         "                              ";
    static int i = 1;
    int linefeed = '$';
    if (c == linefeed)
    {
/*
 * The format %.*s displays a row of characters where the number of
 * characters is given by a parameter (LINESIZE in this case).
 */
        printf("%.*s\n", LINESIZE, outline);
/*
 * memset is a good way of setting all the elements of an array to
 * the same value, here blank.
 */
        memset(outline, ' ', LINESIZE);
        i = 1;
    }
    else
    {
        outline[i-1] = c;
        i = i + 1;
    }
}
```

```
static void getrecord(char b[ILENGTH])
{
    int i, ch = getchar();  /* getchar() reads from the keyboard */
    for (i = 0; i < ILENGTH; i++)
    {
        if (ch == '\n') b[i] = ' ';
        else
        {   b[i] = ch;
            ch = getchar();
        }
    }
}

/* End of file */
```

20 Final Words

Do I follow my own advice? Hmmm I might have known you would ask that! Well most of what I have written about here is what I try to do, but I am not especially formal about any of it. I only really go overboard about design and making documentation precede implementation when starting some code that I expect to give me special difficulty. I have never got into the swing of literate programming, and suspect that I like the idea more than the reality. And I sometimes spend many more hours on a stretch at a keyboard than I maybe ought to. If this course and these notes help you think about the process of programming and allow you to make more conscious decisions about the style you will adopt then I guess I should be content. And if there is one very short way I would like to encapsulate the entire course, it would be the recommendation that you make all the decisions and thoughts you have about programming as open and explicit as possible.
Good luck!

References

[1] Colin Bentley. *Introducing SSADM 4+*. NCC Blackwell, and also see `http://www.blackwellpublishers.co.uk/ssadmfil.htm`, 1996.

[2] Jon Bentley. *Programming Pearls*. Addison-Wesley, 1986.

[3] Jon Bentley. *MoreProgramming Pearls*. Addison-Wesley, 1988.

[4] Fred Brookes. *The Mythical Man Month*. Addison Wesley, 2 edition, 1996.

[5] J. Cameron. *JSP and JSD: The Jackson Approach to Software Development*. IEEE Computer Society Press, 1989.

[6] Leiserson Cormen and Rivest. *An Introduction to Algorithms*. MIT and McGraw-Hill, 1990.

[7] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[8] M. H. Halstead. *Elements of Software Science*. Elsevier North Holland, 1977.

[9] Brian W. Kernighan and Dennis M. Richie. *The C programming language*. Prentice-Hall, 1978.

[10] Donald E. Knuth. *Literate Programming*. CSLI Lecture Notes and CUP, 1992.

[11] C.A.R. Hoare O.-J. Dahl, E.W. Dijkstra. *Structured Programming*. Academic Press, 1972.

[12] X3J11. *ANSI X3.159, ISO/IEC 9899:1990*. American National Standards Institute, International Standards Organisation, 1990.